MOVE: Mobility with Persistent Network Connections

Gong Su

Submitted in partial fulfillment of the
requirements for the degree
of Doctor of Philosophy
in the Graduate School of Arts and Sciences

COLUMBIA UNIVERSITY
2004

ABSTRACT

MOVE: Mobility with Persistent Network Connections

Gong Su

The combined force behind ubiquitous mobile computing and storage devices and universal network access has created a unique era of mobile network computing, in which computation units ranging from a single process to an entire host can move while communicating with each other across the network. A key problem therefore is how to preserve the ongoing network communication between two computation units when they move from one place to another; because current network infrastructure and protocols are designed to support stationary communication endpoints only.

We have developed MOVE, a fine-grain end-to-end connection migration architecture, to address the problem. The most distinguishing characteristic of MOVE is that MOVE achieves, *in a single system*, several essential goals of a mobile communication architecture including: (1) entirely end system only without any infrastructure demand, transport protocol independence, and backward compatibility; (2) fine-grain connection migration and unlimited mobility scope; (3) secure migration with both handoff and suspension/resumption support; and (4) very low performance overhead both before and after migration.

We first analyze the key technical problems of end-to-end network communication caused by mobility: state inconsistency, conflict, and synchronization; and we

develop a simple and elegant namespace abstraction called CELL to resolve these problems. CELL provides a virtual, private, and labeled namespace for individual connection states so that they can be transparently migrated anywhere free of the problems mentioned above. We then develop a unique handoff signaling protocol called H2O, which can handoff a connection securely in a single one-way end-to-end trip with minimal impact on the connection characteristics perceived by the transport protocols. H2O achieves this by combining the simple connection redirection mechanism afforded by the CELL abstraction with a low-overhead security mechanism, which is based on Diffie-Hellman protocol but computes session keys only at migration time. We finally integrate MOVE seamlessly with a process migration mechanism to fully exploit MOVE's fine-grain connection migration capability and enable support for new application scenarios. For example, we show how the integration can provide high service availability in proxy-based server clusters by allowing server applications and their persistent connections to be migrated during a server maintenance to avoid service disruption.

We have implemented MOVE on a commodity OS without requiring any change to the OS and applications and conducted various performance measurements, such as handoff performance, scalability, and virtualization and virtual-physical mapping overhead, etc. Our results show that MOVE handoff incurs minimal performance impact on the migrating connection, MOVE does not adversely affect system scalability, and MOVE virtualization and mapping overhead is very low. We also test MOVE with a suite of popular off-the-shelf network applications, all of which work out of the box.

Table of Contents

Table of Contents i

List of Figures iv

List of Tables viii

Acknowledgements ix

Chapter 1 Introduction 1

1.1 Background and Motivation 1
1.2 Thesis Contribution 6
1.3 Thesis Focus Area 8
1.4 Thesis Overview 10

Chapter 2 CELL Namespace Abstraction 12

2.1 Non-transparent vs Transparent Migration 12
2.2 Key Problems of Connection Migration 14
2.2.1 Inconsistency between network layer and transport layer 15
2.2.2 Conflict in transport layer 15
2.2.3 Cross address space synchronization in transport layer 17
2.3 The CELL Namespace Abstraction 22
2.3.1 Virtualize network addresses 22
2.3.2 Privatize transport identifications 25
2.3.3 Label end-to-end connections 29
2.3.4 Map between virtual and physical namespace 34
2.4 Other Architectural Issues 39
2.4.1 Host and service location 39
2.4.1.1 Host location 40
2.4.1.2 Service location 41
2.4.2 Connection-less transport protocol support 43
2.4.3 Application location-awareness 45
2.4.4 Compatibility with IPsec 49
2.5 Summary 50

Chapter 3 H2O Handoff Signaling Protocol 52

3.1 Handoff Related Issues 53
3.1.1 Layer 2 handoff vs. layer 3 handoff 53
3.1.2 Hand off detection vs. handoff execution 54
3.2 H2O Handoff Signaling Protocol 56
3.2.1 In-band vs. out-of-band signaling 58
3.2.2 H2O protocol operation 61
3.2.3 Interaction with existing network security constructs 64
3.2.3.1 SPI firewall traversal 64
3.2.3.2 VPN traversal 67
3.2.4 Migration security 68

3.2.4.1 H2O security mechanism . . . 69
3.2.4.2 DH protocol and HMAC algorithm . . . 72
3.3 H2O Protocol Analysis . . . 76
3.3.1 No advance notice . . . 77
3.3.2 Advance notice without simultaneous connectivity . . . 80
3.3.3 Advance notice with simultaneous connectivity . . . 83
3.3.4 Intra-domain handoff . . . 85
3.4 Suspension/Resumption with Migration Helpers . . . 87
3.5 Summary . . . 91
Chapter 4 High Service Availability Support . . . 93
4.1 Motivation . . . 93
4.2 Example High Service Availability Scenario . . . 96
4.3 The zPod Abstraction . . . 98
4.4 zPod Migration . . . 100
4.4.1 General server clusters . . . 101
4.4.2 Different types of proxies . . . 103
4.4.3 Single subnet of servers . . . 106
4.5 Summary . . . 109
Chapter 5 Design and Implementation . . . 110
5.1 Functional Design Overview . . . 110
5.2 Security Module . . . 113
5.3 Migration Module . . . 118
5.3.1 Handoff process . . . 119
5.3.2 Suspension and resumption process . . . 122
5.4 Mapping Module . . . 124
5.5 System Call Interception . . . 125
5.6 Transparent SRV RR Lookup Support . . . 126
5.7 Summary . . . 127
Chapter 6 Performance Measurements . . . 129
6.1 Handoff Performance . . . 130
6.1.1 Client handoff with machine migration . . . 131
6.1.1.1 Handoff on a WAN, DDT≈10ms, 200ms, and 4s . . . 134
6.1.1.2 Handoff on a LAN, DDT≈30ms and 3s . . . 143
6.1.1.3 Handoff from a WAN to LAN, DDT≈100ms and 2s . . . 148
6.1.1.4 Handoff from a LAN to WAN, DDT≈100ms and 2s . . . 154
6.1.2 Client handoff with VMware migration . . . 158
6.1.2.1 Handoff from a WAN to LAN, DDT≈8s . . . 159
6.1.2.2 Handoff from a LAN to WAN, DDT≈11s . . . 162
6.1.3 Server handoff with process migration . . . 162
6.1.3.1 Handoff with a WAN client, DDT≈2s . . . 166
6.1.3.2 Handoff with a LAN client, DDT≈2s . . . 171
6.1.4 Handoff "ping-pong" stress test . . . 174
6.1.5 Handoff for connection-less transport protocols . . . 175
6.1.6 Migrate popular real world applications . . . 177
6.2 Scalability Tests . . . 179

6.2.1 Number of simultaneous connections . . . 180
6.2.2 Rate of new connections . . . 182
6.3 Connection Virtualization and Mapping Overhead . . . 182
6.3.1 Throughput . . . 184
6.3.2 Latency . . . 185
6.3.3 CPU utilization . . . 188
6.3.4 Connection setup . . . 191
6.3.5 Overhead in proxy-based environments . . . 191
6.4 Host and Service Location Mechanism Studies . . . 196
6.4.1 Empirical DDNS studies . . . 196
6.4.2 Transparent SRV RR lookup measurements . . . 198
6.5 Summary . . . 199
Chapter 7 Related Work . . . 201
7.1 Mobile Communication Architectures . . . 201
7.1.1 Network layer solutions . . . 203
7.1.2 Transport layer solutions . . . 206
7.1.3 Application layer solutions . . . 209
7.1.4 Split connection solutions . . . 211
7.1.5 Summary . . . 213
7.2 Handoff Mechanisms . . . 214
7.2.1 Extensions to MobileIP . . . 214
7.2.2 Domain-based solutions . . . 217
7.2.3 Others . . . 220
7.3 High Service Availability Mechanisms . . . 220
7.3.1 Fault tolerance with TCP failover . . . 221
7.3.2 Performance and scalability with TCP handoff . . . 225
7.4 Process Migration Systems . . . 227
7.4.1 Special purpose OSes . . . 227
7.4.2 User-level migration . . . 228
7.4.3 Language and middleware support . . . 228
7.4.4 OS virtualization . . . 229
7.4.5 Virtual machine monitors . . . 229
Chapter 8 Conclusion . . . 231
Bibliography . . . 237

List of Figures

Figure 2-1. Inconsistency between network layer and transport layer 16
Figure 2-2. Conflict in transport layer. 17
Figure 2-3. Synchronization: from no NAT to NAT 19
Figure 2-4. Synchronization: from NAT to no NAT 20
Figure 2-5. Synchronization: from NAT to another NAT 21
Figure 2-6. CELL abstraction: virtual network addresses 24
Figure 2-7. CELL abstraction: private transport identifications 26
Figure 2-8. Conflict between VNIC and NIC. 27
Figure 2-9. CELL abstraction: labels (exchanged at connection setup time). . . 30
Figure 2-10. Labels identify connections with identical virtual tuple 31
Figure 2-11. Labels identify connections across NAT boundaries 32
Figure 2-12. Label conflict. 33
Figure 2-13. Virtual-physical namespace mapping 35
Figure 2-14. Visual representation of the CELL namespace 38
Figure 2-15. AH and ESP protection services 50
Figure 3-1. Handoff detection and execution 55
Figure 3-2. H2O protocol timeline. 62
Figure 3-3. SPI firewall traversal 65
Figure 3-4. VPN traversal 67
Figure 3-5. H2O analysis: no advance notice. 77
Figure 3-6. H2O analysis: advance notice without simultaneous connectivity 81
Figure 3-7. H2O analysis: advance notice with simultaneous connectivity . . . 84
Figure 3-8. H2O analysis: intra-domain handoff 86
Figure 4-1. High service availability in proxy-based server cluster. 96
Figure 4-2. Connection migration in proxy-based server clusters 102
Figure 4-3. Combine MOVE and layer 4-7 switches: two-way architecture . . 105
Figure 4-4. Combine MOVE and layer 4-7 switches: one-way architecture . . 106
Figure 5-1. MOVE functional design overview. 111
Figure 5-2. Security key and connection label exchange 114

Figure 5-3. DH public key and label exchange IP option format 118
Figure 5-4. Handoff process FSM and IP option format 120
Figure 5-5. Suspension and resumption process FSM 122
Figure 5-6. Connection label IP option format 124
Figure 6-1. Client handoff with machine migration testbed 132
Figure 6-2. Entire playback TCP sequence trace, DDT≈10ms << RTT≈230ms 135
Figure 6-3. Entire playback TCP throughput, DDT≈10ms << RTT≈230ms. . . 135
Figure 6-4. Zoomed TCP sequence trace, DDT≈10ms << RTT≈230ms 136
Figure 6-5. Entire playback TCP sequence trace, DDT≈200ms ≈ RTT≈235ms 138
Figure 6-6. Entire playback TCP throughput, DDT≈200ms ≈ RTT≈235ms . . . 139
Figure 6-7. Zoomed TCP sequence trace, DDT≈200ms ≈ RTT≈235ms. 139
Figure 6-8. Entire playback TCP sequence trace, DDT≈4s >> RTT≈231ms. . . 141
Figure 6-9. Entire playback TCP throughput, DDT≈4s >> RTT≈231ms 142
Figure 6-10. Zoomed TCP sequence trace, DDT≈4s >> RTT≈231ms 142
Figure 6-11. Entire playback TCP sequence trace, DDT≈30ms ≈ RTT≈33ms . . 144
Figure 6-12. Entire playback TCP throughput, DDT≈30ms ≈ RTT≈33ms 144
Figure 6-13. Zoomed TCP sequence trace, DDT≈30ms ≈ RTT≈33ms. 145
Figure 6-14. Entire playback TCP sequence trace, DDT≈3s >> RTT≈31ms. . . . 147
Figure 6-15. Entire playback TCP throughput, DDT≈3s >> RTT≈31ms 147
Figure 6-16. Zoomed TCP sequence trace, DDT≈3s >> RTT≈31ms 148
Figure 6-17. Entire download TCP sequence trace, DDT≈100ms. 149
Figure 6-18. Entire download TCP throughput, DDT≈100ms 150
Figure 6-19. Zoomed TCP sequence trace, DDT≈100ms 150
Figure 6-20. Entire download TCP sequence trace, DDT≈2s. 152
Figure 6-21. Entire download TCP throughput, DDT≈2s 153
Figure 6-22. Zoomed TCP sequence trace, DDT≈2s 153
Figure 6-23. Entire download TCP sequence trace, DDT≈100ms. 154
Figure 6-24. Entire download TCP throughput, DDT≈100ms 155
Figure 6-25. Zoomed TCP sequence trace, DDT≈100ms 155
Figure 6-26. Entire download TCP sequence trace, DDT≈2s. 157
Figure 6-27. Entire download TCP throughput, DDT≈2s 157

Figure 6-28. Zoomed TCP sequence trace, DDT≈2s 158
Figure 6-29. Client handoff with VMware migration testbed 159
Figure 6-30. Entire download TCP sequence trace, DDT≈8s. 160
Figure 6-31. Entire download TCP throughput, DDT≈8s 160
Figure 6-32. Zoomed TCP sequence trace, before handoff, DDT≈8s 161
Figure 6-33. Zoomed TCP sequence trace, after handoff, DDT≈8s 161
Figure 6-34. Entire download TCP sequence trace, DDT≈11s. 163
Figure 6-35. Entire download TCP throughput, DDT≈11s 163
Figure 6-36. Zoomed TCP sequence trace, before handoff, DDT≈11s 164
Figure 6-37. Zoomed TCP sequence trace, after handoff, DDT≈11s 164
Figure 6-38. Server handoff with process migration testbed. 165
Figure 6-39. Entire download TCP sequence trace, client-proxy, DDT≈2s 167
Figure 6-40. Entire download TCP sequence trace, proxy-server, DDT≈2s . . . 168
Figure 6-41. Entire download TCP throughput, client-proxy, DDT≈2s 168
Figure 6-42. Entire download TCP throughput, proxy-server, DDT≈2s 169
Figure 6-43. Zoomed TCP sequence trace, client-proxy, DDT≈2s 169
Figure 6-44. Zoomed TCP sequence trace, proxy-server, DDT≈2s. 170
Figure 6-45. Entire download TCP sequence trace, client-proxy, DDT≈2s 171
Figure 6-46. Entire download TCP sequence trace, proxy-server, DDT≈2s . . . 172
Figure 6-47. Entire download TCP throughput, client-proxy, DDT≈2s 172
Figure 6-48. Entire download TCP throughput, proxy-server, DDT≈2s 173
Figure 6-49. Zoomed TCP sequence trace, client-proxy, DDT≈2s 173
Figure 6-50. Zoomed TCP sequence trace, proxy-server, DDT≈2s. 174
Figure 6-51. Handoff “ping-pong” stress test on a LAN 176
Figure 6-52. Entire playback UDP byte counts, DDT≈150ms 177
Figure 6-53. Entire playback UDP throughput, DDT≈150ms 178
Figure 6-54. Zoomed UDP byte counts, DDT≈150ms 178
Figure 6-55. Throughput vs. number of connections 181
Figure 6-56. Latency vs. number of connections 181
Figure 6-57. Throughput vs. rate of connections. 183
Figure 6-58. Latency vs. rate of connections 183

Figure 6-59. MOVE virtualization and mapping overhead testbed 184
Figure 6-60. Throughput overhead, laptop system 185
Figure 6-61. Throughput overhead, server system 186
Figure 6-62. Latency overhead, laptop system 187
Figure 6-63. Latency overhead, server system 187
Figure 6-64. Throughput test CPU utilization overhead, laptop system 189
Figure 6-65. Throughput test CPU utilization overhead, server system 189
Figure 6-66. Latency test CPU utilization overhead, laptop system 190
Figure 6-67. Latency test CPU utilization overhead, server system 190
Figure 6-68. TCP connection setup overhead, laptop system 192
Figure 6-69. TCP connection setup overhead, server system 192
Figure 6-70. Throughput overhead 193
Figure 6-71. Latency overhead 194
Figure 6-72. CPU utilization overhead, throughput test 195
Figure 6-73. CPU utilization overhead, latency test 195
Figure 6-74. TCP connection setup overhead 196
Figure 7-1. Visual handoff comparison 219

List of Tables

Table 6-1. Handoff performance test cases. 131
Table 6-2. Name record TTL of some DDNS providers 197
Table 6-3. Applications used for DDNS TTL test 198
Table 6-4. Execution overhead of intercepted socket calls. 199
Table 7-1. Comparison of MOVE and other mobility solutions 202

Acknowledgements

I am in the deepest debt to my advisor, Jason Nieh, who graciously assumed and continued my sources of academic and financial support at a time when my future was uncertain. His open-mindedness and relentless pursuit for excellence without leaving out any details have inspired many key ideas of this thesis. I am constantly amazed by his ability to manage an army of students, each with a different project, yet still dumbfound me with razor-sharp questions on issues that I have been pondering myself for days. His generosity to others, calmness under stress, and tolerance for my stubbornness have taught me countless invaluable lessons towards life, not just as a researcher, but more importantly, as a person as well.

I would like to thank my former advisor, Yechiam Yemini, who had the faith in me and provided me the chance to pursue my dream of becoming a Computer Scientist, despite the fact that my early education was in a different field, Physics to be specific. I consider myself unimaginably fortunate to have the opportunity to work with a visionary like him. Yet unfortunately, I cannot even begin to count how much I learned from him to properly thank him. I can only hope that his incredible intuition, comprehension, and presentation will continue to inspire me in the years to come.

I would also like to thank my former office mate and best friend, Apostolos Dailianas, whose unreserved support, both inside and outside the office, carried me through some of the most difficult times of my graduate life. Without his constant caring, encouragement, and help, my graduate life would have ended a few years

ago and this thesis would have died before it had even started. Two other members of my thesis committee, Danilo Florissi and Bulent Yener, have also been my sources of counseling who were always ready and willing for anything I threw at them. Their understanding, encouragement, and guidance kept my course of graduation from faltering at times of stress, helplessness, and uncertainty.

This thesis has benefited from many discussions with and helps by others. In particular, Dinesh Subhraveti, Shaya Potter, and Steven Osman have provided many insights on process migration issues and implemented much of the prototype itself used in Chapter 6. Sarita Bafna helped much with the DDNS host and service location issues. Thanks also to Angelos Keromytis for his advice on security related issues and graciously agreeing to serve as my thesis committee member.

My graduate life at Columbia would have been much less enjoyable were it not for the wonderful friends and colleagues I was fortunate to meet. I cannot hope to enumerate them all. Nevertheless, let me mention a few: Sushil daSilva, Ioannis Stamos, Susan and Joe Tritto, Patricia Florissi, Andreas Prodromidis, Maria Papadopouli, Alexandros Konstantinou, Montek Singh, Martha and Erez Zadok, Ashutosh and Manu Dutta, David Olshefski, Phil Wang, Hao Huang, Vasileios Hatzivassiloglou, Michael Grossberg, Jakka Sairamesh, Tobias Höllerer, Simon Baker, Shree Nayar, and Henning Schulzrinne.

This thesis is dedicated to my grandparents and parents, who brought me up with their unreserved love, sacrifice, and patience. They are the reason for my very existence.

To my grandfather

Jishi Shi

and

the loving memory of my grandmother

Ying Jiang

and

my parents

Shaoquan Su and Hanqiao Shi

1 Introduction

1.1 Background and Motivation

The combined force behind *ubiquitous mobility* and *universal connectivity* has created a phenomenal era of distributed and networked computing in which *application mobility with persistent connectivity* is becoming a growing and pressing necessity.

Ubiquitous mobile computing is a coming reality, fueled by the proliferation of portable computing devices such as laptops, PDAs, and mobile phones, etc., and portable storage devices such as memory stick, CompactFlash, and USB pen drive, etc.; and on-demand computing and high service availability call for ways to move applications among physical resources for better resource utilization and fault tolerance. At the same time, universal network access has also become an integral part of our everyday life, driven by the immense success of World Wide Web (WWW), and advances in wireless networking technologies such as 802.11 WLAN [25], 3G/UMTS cellular [10], and Bluetooth [17], etc.

The demand for application mobility with persistent connectivity not only presents at the user frontend, but also at the server backend as well, as evidenced by the following examples:

- At the user frontend, laptop, PDA, and mobile phone users rely on net-

work applications such as email, enews, file transfer, streaming media, etc., for their daily life and business. As the users move from one network location to another, their ongoing network activities should not be interrupted. Alternatively, the users may suspend their computing devices at one network location and later resume them at another; their ongoing network sessions should also be maintained.

- Also at the user frontend, cheap, portable storage devices with capacity ranging from tens of megabyte to tens of gigabytes are readily available today. They provide new user mobility opportunities even without the mobile computing devices such as laptops or PDAs. Instead, a user computing session or even an entire virtual machine running on a desktop computer can be checkpointed, saved on the portable storage, and later restarted on another desktop computer. As with the previous case, active network connections of the user computing session or the virtual machine must be preserved.

- At the server backend, online services and businesses require five nine availability as they become an integral part of our daily life. For example, web, email, enews, messenger are now essentially commodity services; while critical business functions, such as order processing and tracking, inventory control, transaction processing, customer support, and electronic commerce, are also increasingly being conducted online. These services and businesses are supported by computing and networking facilities that must be up and running 24-7. A few minutes of downtime, scheduled or

unscheduled, translates into millions of lost dollars. Periodic maintenance of these facilities today, however, requires careful planning and usually causes lengthy service disruption. Technologies that allow maintenance without service disruption, such as by moving services off to other servers, are therefore being actively pursued.

- Also at the server backend, many commercially used network intensive and long running scientific and engineering applications, such as massive parallel graphics rendering, typically have a running time that is similar to or longer than the MTBF (mean time between failures) of their supporting hardware [119]. Therefore, there is high demand for the ability to checkpoint and restart these applications in order to: (1) provide better resilience to hardware failure; (2) enable dynamic load-balancing of the applications, either to make way for interactive jobs or to reshuffle the work load to better accommodate the cooling infrastructure in the computing center.

From these examples, which are by no means exhaustive, we can summarize the functional requirements necessary for a mobile communication architecture to adequately support the needs of these applications:

> **Easy deployment**: The architecture must be easily deployable at the Internet scale, which in turn means that several sub-requirements must be met: (1) *minimal infrastructure requirement*. The architecture should avoid mandating introduction of new network infrastructure, which history has shown to be extremely hard to deploy; (2) *transport protocol independency*. The architecture should make minimal assumptions about the operational semantics of the transport

protocols; and (3) *backward compatibility*. The architecture should interoperate with and require no modification to existing networking protocols, commodity OSes, and legacy applications.

Fine-grain and unlimited mobility: The architecture must enable fine-grain mobility at the level of individual connections as well as an entire host. In addition, the architecture should not restrict the scope within which mobile endpoints can move; nor should it restrict which endpoint of the connection can move.

Secure and flexible migration: The architecture must prevent a malicious user from hijacking a connection by exploiting migration functions, such as claiming that a connection has migrated from one machine (the victim) to another (the attacker). In addition, the architecture should support both fast online-natured handoff with minimal impact on connectivity and slow offline-natured suspension/ resumption where migrating connections must be kept alive for an extended period of time.

Low performance overhead: The architecture should incur low networking performance overhead and retain good scalability during normal communication, especially for those applications that do not yet utilize the benefit of mobility. The architecture should also support fast handoff with minimal impact on transport protocol and/or application perceived end-to-end network connectivity.

Although many approaches have been considered [34][38][58][70][89][98][103][109][122][128][132][136][139][141], achieving mobile communication functionality has been difficult in practice, especially in the realm of end-to-end connection mobility. *To date, no single architecture has met all the requirements.* More specifically, network layer solutions such as [38][70][103][132][136] require infrastructure sup-

port; transport layer solutions such as [58][122][128] require modifying existing transport protocol (TCP); application layer solutions such as [98][109][139][141] are transport protocol dependant and have high performance overhead; and split connection solutions such as [34][89] require special proxy support and limit mobility scope to client only. The lack of system support for mobile communication today is primarily due to the fact that the current *de facto* worldwide data network protocol standards suite, the network layer Internet Protocol (IP), the transport layer Transmission Control Protocol (TCP)/User Datagram Protocol (UDP), etc., were all designed with the assumption that devices attached to the network are stationary. For example, IP addresses are assigned to fixed network attachment points with implicit geographical association; TCP/UDP uses IP address and port number to identify its connection endpoints and assumes these values never change for the lifetime of a connection. As a result, one cannot move either (or both) endpoint(s) of a connection without severing the connection.

The motivation behind this thesis, therefore, is to design and implement such a mobile communication architecture that supports mobile applications with persistent network connections, an architecture that *meets all the requirements* in order to facilitate the wider deployment of mobility functions in both current and future data networks.

1.2 Thesis Contribution

At a high level, the contribution of this thesis is the design and implementation of a novel end-to-end mobile communication architecture called MOVE that, *in a*

single system, meets all the requirements outlined in the previous section, namely easy deployment, fine-grain and unlimited mobility, secure and flexible migration, and low performance overhead. More specifically, the requirements are met through the research and development of a collection of novel concepts and mechanisms, along with their design, implementation, and real world application:

- A novel namespace abstraction, called CELL (ConnEction virtuaLization and encapsuLation), that provides a virtual, private, and labeled namespace for individual connections so that they can be transparently migrated anywhere, even across address space boundaries separated by NAT (Network Address Translation)/NAPT (Network Address Port Translation) devices. CELL supporting mechanisms are independent of the transport protocol and function entirely within the two communicating endpoints without the need for a third-party entity such as a proxy or any other new network infrastructure. CELL supporting mechanisms are also compatible with and require no modification to current networking protocols, OSes, and applications.
- A unique in-band layer 3 handoff signaling protocol called H2O (Host-only HandOff) and a low-overhead security mechanism, that can migrate connections securely with a single packet in a single one-way trip from the mobile endpoint to the stationary endpoint, achieving handoff performance perceived by the transport protocol similar to (and in certain case better than) existing approaches that require very complex network infrastructure support. With a connection migration helper mechanism, H2O

supports secure connection migration through suspension/resumption where migrating connections can be kept alive for an extended period of time. Similar to CELL supporting mechanisms, H2O is also independent of the transport protocol and functions entirely within the two communicating endpoints without the need for a third-party entity such as a proxy or any other new network infrastructure. H2O is also compatible with and requires no modification to current networking protocols, OSes, and applications.

- Seamless integration with a new process migration mechanism, that is built on the same virtual private namespace concept of CELL abstraction extended to other OS resources such as PID (process ID)/GID (group ID), IPC (inter-process communication) key, memory, file system, and device, etc. We show how the integration allows MOVE's fine-grain connection migration capability to be fully exploited and enable support for new application scenarios. For example, we show how the integration can provide high service availability in proxy-based server clusters by allowing server applications and their persistent connections to be migrated during a server maintenance to avoid service disruption.

- A design and implementation of the MOVE architecture on a generic OS platform, i.e., LINUX, that requires no networking protocol, OS kernel, or application modification; and an evaluation of our MOVE implementation. We present our MOVE prototype's handoff performance in a variety of network configurations, both used standalone for moving a client host and

integrated with process migration for moving a server process; and we show that MOVE has minimal impact on the connection characteristics perceived by the transport protocols and applications. We show that our MOVE prototype does not have negative effect on the scalability of existing systems; and we also show that the virtualization overhead in terms of networking I/O such as bandwidth, delay, and CPU utilization of our unoptimized MOVE prototype is very low, which means that connections that do not migrate suffer little overhead. Once a connection is migrated and virtual-physical mapping is performed, our results show that the mapping overhead introduced by our MOVE prototype stays very low. We test our MOVE prototype with a suite of popular off-the-shelf network applications, all of which work out of the box.

1.3 Thesis Focus Area

Mobile communication is a broad area that comprises issues of many aspects. This thesis does not claim to solve all issues of mobile communication but rather focus on a few specific ones. In this section, we clarify the focus of the issues addressed by this thesis.

First, this thesis focuses on mobility of end-to-end transport *connections*, which is defined as the logical association of two communication endpoints by the transport protocols, rather than mobility of the *endpoints* themselves. Mobility of endpoints has been studied under different contexts, such as host mobility, user mobility, and session mobility, etc. Host mobility is concerned about tracking the

movement of an entire host such as a laptop, a PDA, or a mobile phone. User mobility attempts to track the movement of a person by maintaining a list of devices or applications through which the person is currently accessible. Session mobility tracks the movement of a computation session, which is usually defined as a group of related processes such as a capsule [118] or a Pod [100]. Regardless of these different types of endpoints, the common problem that must be resolved by a mobile communication architecture is to track the movement of the end-to-end connection states maintained by the transport protocol on behave of the end-points, which is the focus of this thesis. We assume that the states of the endpoints themselves are saved and restored by appropriate migration mechanisms external to our proposed MOVE architecture. In fact, we have designed our system to be independent of and can interoperate with these different migration mechanisms. For example, when an entire host is moved, the hardware BIOS suspension/ resumption functions are responsible for saving and restoring the endpoint states; when a session is moved, the particular migration mechanism such as Zap [100] is responsible for saving and restoring the endpoint states. And we have integrated MOVE with Zap to fully exploit its fine-grain connection migration capability.

Second, this thesis focuses on the issue of *tracking* a connection between mobile endpoints after it has been established, rather than the issue of *locating* a mobile endpoint before a connection can be established. It is our belief that these are two orthogonal and fundamentally different problems. Generally speaking, the former is a routing problem, while the latter is a directory problem. The requirements for systems addressing these two problems are therefore fundamentally different and

should not be mixed. For example, global directory service for locating a mobile endpoint usually implies the need for infrastructure support; but as we will show in our proposed MOVE architecture, tracking established connections can be done completely within endpoints themselves without any infrastructure requirement. Therefore, this thesis separates the tracking and locating aspects of communication mobility and focuses on the former. We leverage existing locating systems, Dynamic DNS (DDNS) [133] in particular, that are designed specifically for that purpose and conduct empirical studies on their suitability for locating mobile hosts and services in practice.

1.4 Thesis Overview

This thesis presents the design, implementation, and evaluation of the MOVE mobile communication architecture. It is organized as follows: Chapter 2 introduces the CELL namespace abstraction which is the foundation of the MOVE architecture for resolving key technical problems of transparent connection migration; Chapter 3 presents the H2O handoff signaling protocol and its security mechanism for fast and secure handoff, as well as the connection migration helper mechanism for migration through suspension/resumption; Chapter 4 describes the integration of MOVE with a new process migration mechanism to fully exploit its fine-grain connection migration capability and to provide high service availability support in a proxy-based server cluster environment; Chapter 5 presents the design and implementation of a MOVE prototype on the LINUX x86 platform; Chapter 6 evaluates our MOVE prototype performance and discuss the measurement results; Chapter 7 surveys related work in mobile communication architec-

ture, layer 3 handoff approaches, high service availability mechanisms, and process migration systems; finally, Chapter 8 concludes the thesis and discusses directions for future work.

2 CELL Namespace Abstraction

MOVE is a *transparent* connection migration system, which means that it preserves the transport connection states across migration; since having persistent connection states throughout the lifetime of a connection is the fundamental assumption made by existing transport protocols. Of course, connection migration can also be achieved without preserving the transport connection states, as we will see examples of such solutions shortly; and we call such solutions *non-transparent* connection migration systems. In this chapter, we will first argue why we believe a transparent migration system that preserves the transport connection states is a more viable solution than a non-transparent one that does not. We will then introduce the CELL namespace abstraction that is designed to solve key technical problems of transparent migration systems. We also consider a few other architectural issues such as host and service location, connection-less transport protocol support, application location-awareness, and compatibility with IPsec.

2.1 Non-transparent vs Transparent Migration

While mobility solutions have traditionally been categorized based on the layer at which they provide the mobile functionality, such as network, transport, or application layer, we look at these solutions from another angle, which focuses on the *connection states* rather than the layers, that has given us more insight on the fun-

damental problems of connection migration. We have defined an end-to-end connection as the logical association of two communication endpoints by the transport protocols. Therefore, the connection states we refer to throughout the chapter are the states maintained by the transport protocols. Note that some applications may have their own notion of a "connection"; but as we pointed out in Section 1.3 in Chapter 1, migration of application states is part of endpoint mobility mechanism orthogonal to transport connection mobility mechanism. Depending on whether or not a connection migration system preserves transport protocol connection states, it generally falls into one of the two categories: *non-transparent* or *transparent*.

Non-transparent connection migration systems do not maintain the connection states at the transport layer across migration. Connection migration is achieved either by modification to the transport protocol itself, TCP in particular, to handle the change of IP address and port number - an approach taken by traditional transport layer solutions, or by emulation above the transport layer through closing the old connection and opening a new one - an approach taken by traditional application layer solutions. These solutions, however, have a few serious drawbacks. Modifying TCP results in transport protocol dependency not only on TCP's operational semantics therefore making it incompatible with other transport protocols such as UDP and SCTP, etc., but also on the particular TCP implementation and therefore making it very hard if not impossible to deploy. Emulation at the application layer requires duplicating many of the transport protocol functions, such as double-buffering (in addition to transport protocol buffering) and go-back-N (or

similar), to account for potential packet loss due to closing the old connection. As a result, while avoiding changing the transport protocol itself, emulation at the application layer is still highly dependent upon the transport protocol operational semantics. In addition, the double-buffering must be done on the critical data path of the connections at *all* times, therefore creating substantial performance overhead, not only for migrated connections, but also for regular stationary ones as well. Furthermore, the go-back-N adds additional delay to the handoff process.

Because of these fundamental difficulties facing the non-transparent connection migration systems, we have chosen to build MOVE as a transparent connection migration system, which of course has its own set of problems that must be resolved in order to meet all the requirements of a mobile communication system we outlined in the introduction. In fact, many other transparent mobile communication systems have been proposed, such as [70][71][103][132][136], which are also commonly known as the network layer solutions. However, as we will see in the related work chapter, none of these solutions meets all the requirements. So we will now take a closer look at the key technical problems associated with transparent connection migration. For the rest of the thesis, we will use the word "migration" to always refer to transparent migration unless noted otherwise.

2.2 Key Problems of Connection Migration

Despite the seemingly large variation of mechanisms employed by existing systems, the problems of connection migration can be traced to three fundamental ones: state *inconsistency*, state *conflict*, and state *synchronization*, which we describe

in turn.

2.2.1 Inconsistency between network layer and transport layer

Preserving connection states at transport layer in a mobile environment is difficult because existing transport protocols are not designed with mobility in mind. Specifically, transport layer connection states consist of two types of names: address and port. A *tuple*, which consists of a pair of addresses and ports that correspond to the two communication endpoints, is used by the transport protocol to uniquely identify a connection. Transport protocols require that the tuple stay constant for the lifetime of the connection. This requirement, however, is violated when a connection endpoint is migrated from one host to another, or a host with open connections moves from one network to another; since the address of the endpoint has changed. Mobility therefore creates inconsistency between the transport layer tuple and the network layer address, as shown with an example in Figure 2-1. Figure 2-1a shows that when a host with an open connection `[IP2:p2, IP1:p1]` moves from `IP2` to `IP3`, the transport layer tuple `[IP2:p2, IP1:p1]` is no longer consistent with the new network layer address `IP3`. Figure 2-1b depicts the same problem in the case when an endpoint of a connection `[IP2:p2, IP1:p1]` is transparently migrated from host `IP2` to host `IP3`.

2.2.2 Conflict in transport layer

In addition to inconsistency between network layer and transport layer, other problems may arise due to mobility as well. One of these problems is that mobility creates situations where names such as address and port may be reused therefore

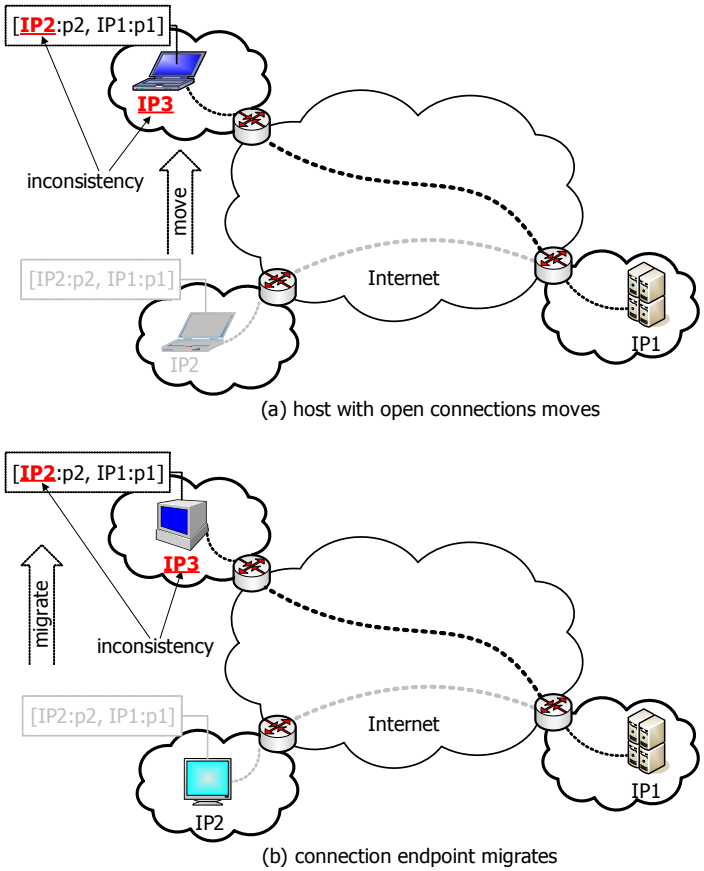


Figure 2-1. Inconsistency between network layer and transport layer

causing naming conflict in the transport layer, as illustrated in Figure 2-2. Figure 2-2a shows that a host with an open connection [IP2:p2, IP1:p1] moves from IP2 to IP3. Later, another host may reuse IP2 at the original network and a process on it may open another connection [IP2:p2, IP1:p1] to IP1:p1 using port p2. As a result, host IP1 sees two identical connections [IP1:p1, IP2:p2], which is prohibited by the transport protocols. Figure 2-2b shows the similar conflict in the case when an end-point of a connection [IP2:p2, IP1:p1] migrates from host IP2 to host IP3 and another process on IP2 reuses port p2 to open another connection [IP2:p2, IP1:p1] to the same server IP1:p1.

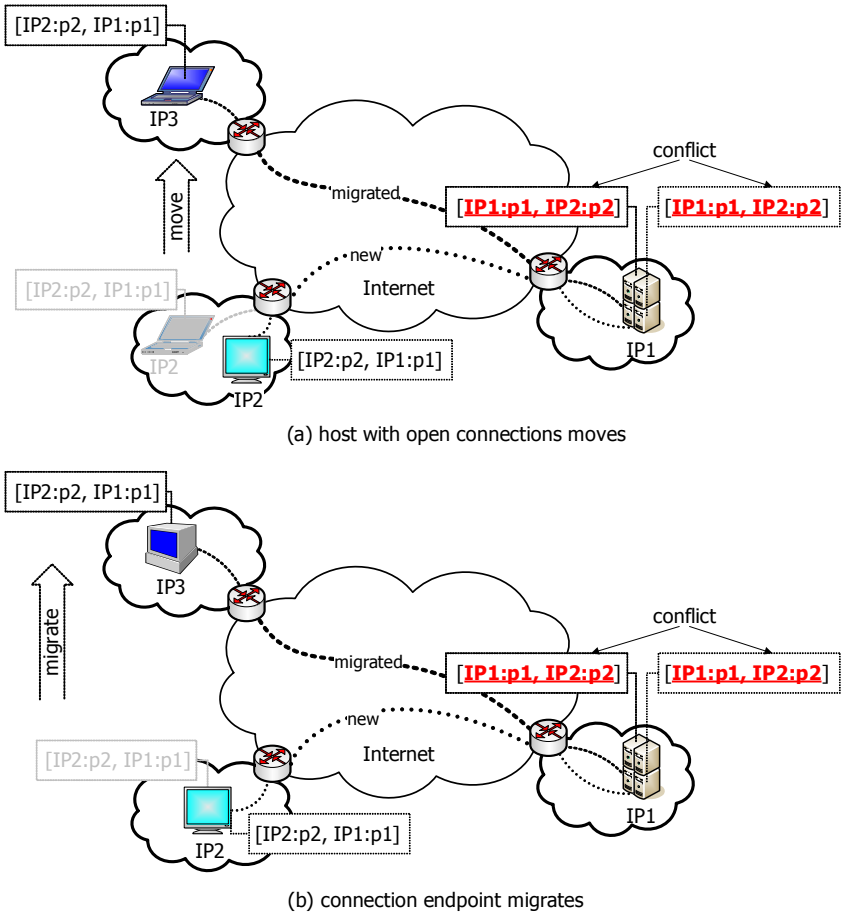


Figure 2-2. Conflict in transport layer

2.2.3 Cross address space synchronization in transport layer

Finally, we look at the third problem that can be caused by mobility. Transport protocol semantics requires that the connection states on the two communication endpoints must remain synchronized, i.e., each endpoint must have proper states to identify the same connection and the one-to-one correspondence of the states on both endpoints must be maintained for the lifetime of the connection. In traditional IP network, the entire Internet is assumed to be a single address space and host IP addresses are globally unique; therefore the synchronization can be achieved by associating with a connection the same *address:port* pair, i.e., tuple, on both end-

points.

With the introduction of NAT (Network Address Translation)/NAPT (Network Address Port Translation) devices [125] (simply referred to as NAT hereafter), however, the assumption above no longer holds. NAT separates the Internet into different address spaces and allows overlapping address spaces to coexist. As a result, a connection passing through a NAT device can no longer be identified by the same tuple on both endpoints; in addition, the one-to-one correspondence of the (different) tuples on each endpoint can only be maintained with the presence of the NAT mapping. Since NAT mapping is performed inside the network transparent to both endpoints, mobility of either endpoint can result in the loss of NAT mapping, and consequently the loss of connection state synchronization.

We illustrate the problem using the following examples covering all the possible scenarios: from no NAT to NAT, from NAT to no NAT, and from NAT to another NAT. For simplicity and without loss of generality, we assume that the stationary end is not behind a NAT. Also note that in all the scenarios we assume the case when an entire host moves from one network to another; but they apply equally to the case when an endpoint of a connection migrates from one host to another.

Figure 2-3 shows that a host with an open connection [`IP2:p2, IP1:p1`] moves from a public network without a NAT to a private network behind a NAT. At the new location, the mobile endpoint obtains a new address `IP3`. Since the transport invariant [`IP2:p2, IP1:p1`] must persist across the migration, a transparent migration system usually maintains a mapping between the transport invariant and a

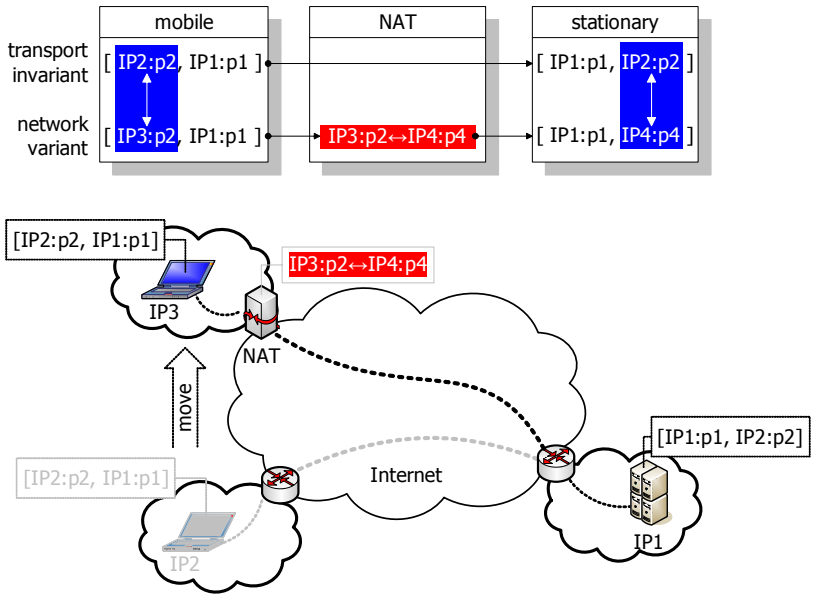


Figure 2-3. Synchronization: from no NAT to NAT

network variant [IP3:p2, IP1:p1] that changes when the mobile endpoint moves, as shown in the figure. This mapping, {IP3:p2↔IP2:p2}, must also be conveyed to the stationary endpoint so that both endpoints can remain synchronized. However, since now the connection goes through a NAT, which applies another mapping {IP3:p2↔IP4:p4}, the correct mapping for the stationary endpoint is not {IP3:p2↔IP2:p2}, but rather {IP4:p4↔IP2:p2}. In other words, due to the NAT mapping, the mobile and the stationary endpoints do not have an agreement on the network variant to identify the connection and therefore cannot synchronize their mappings for the connection.

Figure 2-4 shows the opposite scenario when a host with an open connection [IP2:p2, IP1:p1] moves from a private network behind a NAT to a public network without a NAT. We can see that the mapping for the mobile endpoint,

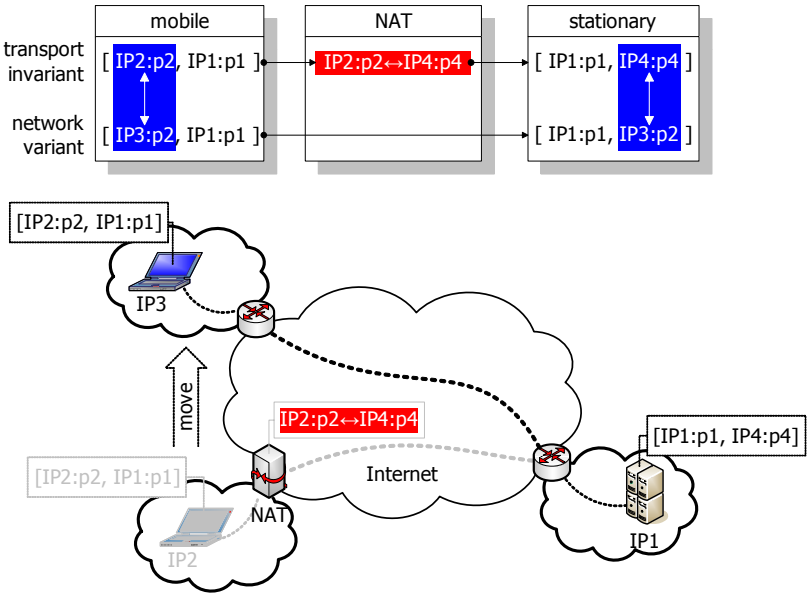


Figure 2-4. Synchronization: from NAT to no NAT

{IP3:p2↔IP2:p2}, is again wrong for the stationary endpoint, which should be {IP3:p2↔IP4:p4}. This time, however, the problem lies with the transport invariant instead of the network variant. Due to the NAT mapping {IP2:p2↔IP4:p4}, the mobile and the stationary endpoints never had an agreement on the transport invariant to uniquely identify the connection. The mobile endpoint saw the connection as [IP2:p2, IP1:p1], while the stationary endpoint saw the connection as [IP1:p1, IP4:p4]. The one-to-one correspondence between [IP2:p2, IP1:p1] and [IP1:p1, IP4:p4] for identifying the same connection can only be maintained with the presence of the NAT mapping {IP2:p2↔IP4:p4}, which is now lost.

Finally in Figure 2-5, a host with an open connection [IP2:p2, IP1:p1] moves from a private network behind a NAT to another private network behind a NAT. It's obvious that in this case the mobile and the stationary endpoints do not have an agreement on either the transport invariant or the network variant; and the correct

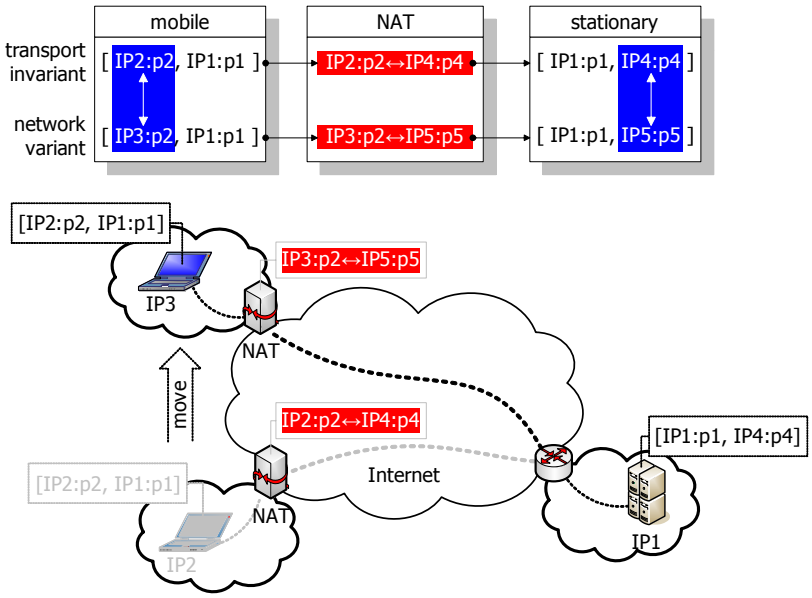


Figure 2-5. Synchronization: from NAT to another NAT

mapping needed on both endpoints are completely unrelated.

To summarize the main points of this section, we have described three fundamental problems of transparent connection migration:

- state inconsistency, because existing transport protocols are designed based on the assumption that connection endpoints are stationary and mobility breaks this assumption
- state conflict, because mobility creates situations where names used for identifying connections may be reused and therefore losing their global uniqueness
- state synchronization, because NAT breaks the end-to-end semantics and the network is no longer a globally addressable space and the one-to-one correspondence of connection states on each endpoint can no longer be

maintained without the NAT mapping, which may be lost due to mobility.

We will see in the next section how MOVE solves these problems.

2.3 The CELL Namespace Abstraction

To effectively address the key problems of fine-grain connection migration, as well as achieve the essential goals of a migration system and avoid drawbacks of existing migration systems, MOVE introduces a novel namespace abstraction, called CELL (ConnEction virtuaLization and encapsuLation). The purpose of CELL is to provide a *virtual*, *private*, and *labeled* namespace for connections of individual processes so that they can be transparently migrated anywhere free of state inconsistency, conflict, and cross address space synchronization problems.

2.3.1 Virtualize network addresses

In order to allow transport layer connection identification, i.e., the tuple, to persist across migration between different networks, CELL uses virtual addresses, which have no semantic association with any particular network locations, to provide individual connections with a constant virtual transport layer identification, regardless of where a connection is migrated. The main challenges lie in the assignment and management of the virtual address space; because virtual tuples, like their physical counterparts, must satisfy a few constraints:

- Global identification. A virtual tuple must uniquely identify a connection no matter where the connection migrates. One simple solution is to employ

a centralized system that manages a global pool of virtual addresses to guarantee that each individual virtual tuple is globally unique. This solution, however, cannot scale to Internet size. An alternative is to employ a distributed system such as DNS to manage the global pool of virtual addresses. This solution, however, requires global infrastructure support which is also undesirable.

- One-to-one correspondence. As we described in Section 2.2.3, two communication endpoints must maintain a one-to-one correspondence of the virtual tuple they have chosen to identify a connection. This implies that the two endpoints must negotiate their virtual tuples for each connection if the virtual addresses chosen by both endpoints are not mutually known beforehand. The negotiation therefore imposes extra round trip delay for connection setup. In addition, virtual tuples must be translated into physical tuples and *vice versa* even when connections are stationary, therefore creates unnecessary network I/O overhead.

CELL employs a unique virtual address assignment mechanism to answer these challenges. The mechanism, which we call *lazy assignment*, is surprisingly simple. By default, CELL selects the virtual addresses to be the current physical addresses associated with a connection. Essentially, CELL treats all physical connections as initially "implicitly" virtualized. As a host moves from one network to another or a connection migrates from one host to another, CELL maintains the virtual addresses unchanged for the migrated connection(s) and translates them into the host's current physical address to resolve inconsistency, as illustrated in Figure 2-

6. But for new connections created at the new network location, their virtual (source) addresses will be the new current physical address of the host. Therefore at any given point of time, a host/process can have multiple connections, each with a different virtual tuple that corresponds to the physical tuple of a connection created at each of the network location it has visited, as also illustrated in Figure 2-6.

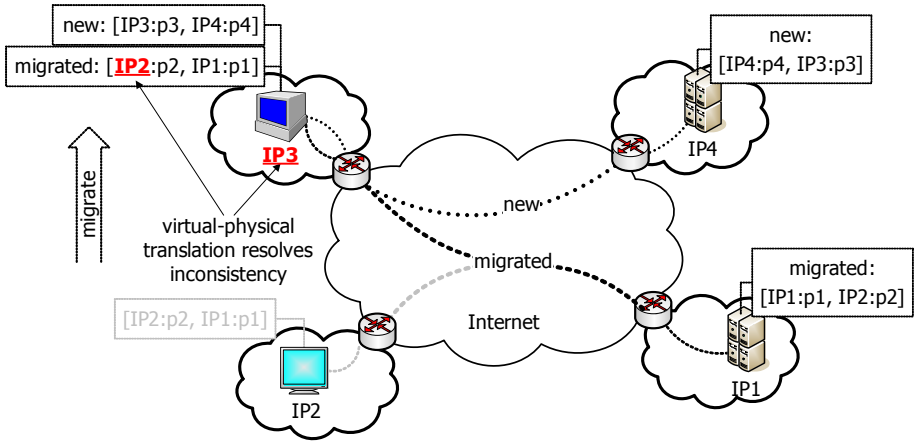


Figure 2-6. CELL abstraction: virtual network addresses

Since physical tuples are guaranteed to be globally unique (either by themselves or with proper NAT mappings), the resulting virtual tuples are also globally unique. Therefore, in the absence of migration, CELL does not need any additional mechanism to manage the virtual address space, which is simply an exact mirror of the physical address space. There is also no additional round trip delays to connection setup for exchanging the virtual addresses since they are already known. And finally, CELL does not need to perform any virtual-physical translation in the absence of migration since the virtual and physical addresses are by default the

same.

The benefits of CELL's lazy assignment, which is essentially a unmanaged approach, does come at the cost of losing one advantage that a managed approach provides: global identification of the virtual tuples in the presence of migration. Since CELL's virtual address space is a mirror of the physical address space, reusing a physical address due to mobility (recall Section 2.2.2) also results in reusing a virtual address and potential conflict in the transport layer. In the next section, we describe how CELL resolves the problem without resorting to a centralized scheme.

2.3.2 Privatize transport identifications

To resolve the conflict in the transport layer caused by reusing a physical address (and therefore reusing a virtual address) illustrated in Figure 2-2, CELL provides a private per-connection virtual address space for each individual connections. In existing transport protocols, all connection identification tuples share a single transport layer namespace, which means no two connections can have the same identification tuple. CELL, however, provides individual connections with their own private virtual tuples that are isolated and independent of each other. One can also think of this as if every connection in CELL had its own dedicated protocol stack. As a result, two identical virtual tuples can coexist side-by-side on the same host free of conflict. For example, Figure 2-7 shows the same conflict cases as those shown in Figure 2-2; but the conflicts are now resolved since the two identical virtual tuples are private to their respective connections and are independent to each

other.

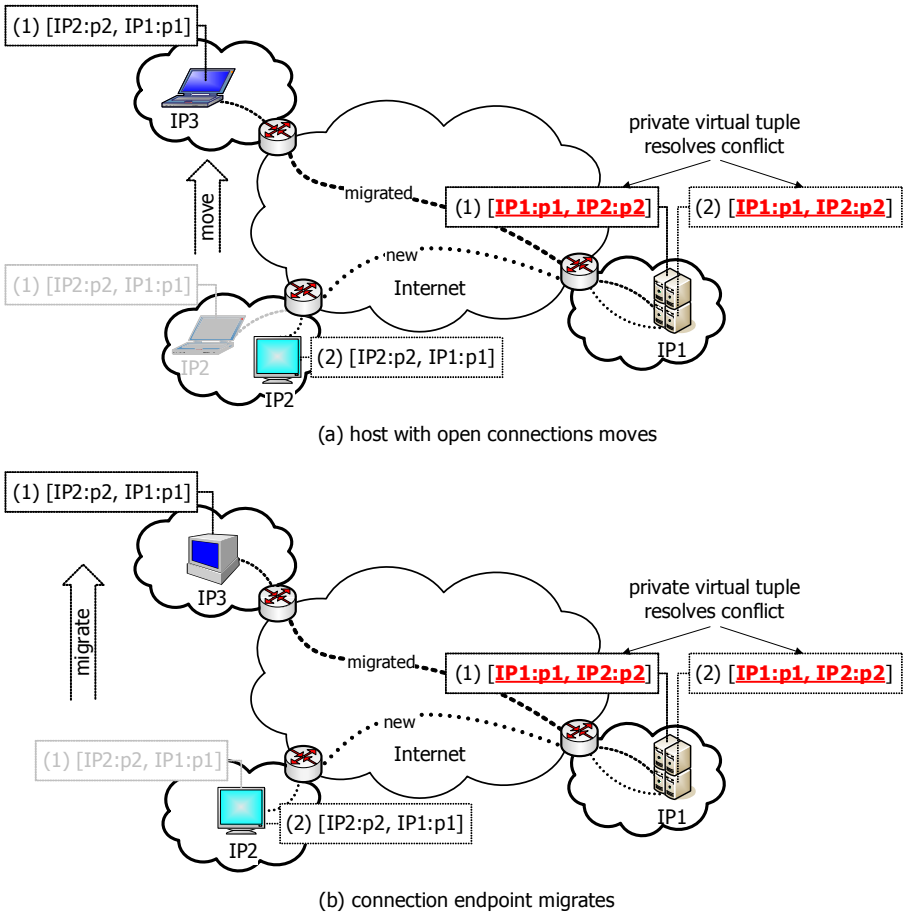


Figure 2-7. CELL abstraction: private transport identifications

Readers may note that, in the previous section, we did not provide a description of how exactly CELL virtualizes network addresses (and how exactly lazy assignment is performed). We defer it until now, along with the description of how CELL privatizes transport tuples, because CELL supports both virtualizing network addresses and privatizing transport tuples with the Virtual Network Interface Card (VNIC) mechanism. A VNIC is a software emulation of a NIC at the link layer and appears exactly the same as a NIC to network-and-above layers. Essentially,

CELL virtualizes network addresses of a connection by simply creating a VNIC with a virtual address on both endpoints and binding the connection to the VNICs. Lazy assignment simply takes the NICs and their physical addresses as the VNICs and their virtual addresses on both endpoints. To privatize transport tuples, CELL provides each connection with its own private VNIC that is not shared with and cannot be accessed by other connections. As a result, two VNICs with identical virtual address, along with two identical virtual tuples bound to the two VNICs, can coexist on the same host. The ambiguity can be resolved because the two VNICs are *different* (e.g., with different device index), even though they have identical virtual address.

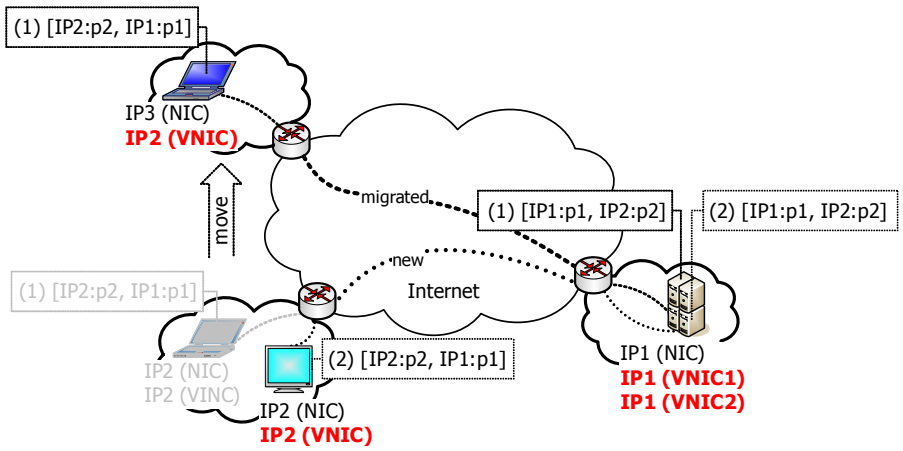


Figure 2-8. Conflict between VNIC and NIC

Because lazy assignment uses the initial physical address of a NIC as the virtual address of a VNIC, the VNIC mechanism itself may introduce additional problem as illustrated in Figure 2-8. The figure shows the conflict in transport layer in the case when a host moves from one network to another is being resolved by CELL's

private virtual tuple abstraction, which is supported by the VNIC mechanism. We can see that per-connection VNICs are created and their virtual addresses are assigned with lazy assignment to properly resolve inconsistency and conflict. However, network layer protocol semantics requires that, within a single address space, an address must uniquely identify one network interface, which is clearly violated by the VNICs in Figure 2-8. For example, the address `IP1` is now associated with three network interfaces on host `IP1`, while the address `IP2` is also associated with three network interfaces, one on host `IP3` and the other two on host `IP2`.

To remedy the problem, CELL imposes visibility constraints on the VNICs so that they are invisible in the physical network. CELL prevents a VNIC from performing any function on the physical network, such as sending and receiving packets, influencing routing decisions, participating in network layer routing protocols such as RIP, OSPF, or BGP, or participating in link layer protocols such as ARP, etc. In other words, a VNIC is only visible to the transport protocols and the connection bound to it. Therefore, network layer protocols can function unaffected. For example, the addresses `IP1` and `IP2` now uniquely identify their respective interfaces, the NIC on host `IP1` and the NIC on host `IP2`; because all the VNICs are invisible in the physical network.

Finally, in order to identify the VNIC of a connection and properly demultiplex incoming packets with identical tuple, CELL augments traditional tuple with a label to identify a connection. Since labels are location-independent, they also allow connections to be identified even in the presence of NAT devices. We

describe the details of CELL's connection label in the next section.

2.3.3 Label end-to-end connections

From previous sections we know that traditional tuple alone is no longer enough to uniquely identify a connection because of two reasons:

- Due to the conflict caused by virtual address reuse in a mobile environment, as we saw in Section 2.2.2, virtual tuples are not globally unique. While CELL uses VNIC to resolve the conflict within an endpoint, packets belonging to a connection, however, must also carry additional information beyond the traditional tuple in order for them to be properly demultiplexed to the right VNIC. We note that this is a CELL-specific requirement necessitated by lazy assignment.
- Due to the presence of NAT, as we saw in Section 2.2.3, the one-to-one correspondence of the tuples on both endpoints of a connection can no longer be maintained in a mobile environment. We note that this is a general problem applicable to any mobile communication system.

CELL addresses both problems with a single mechanism, by introducing a location-independent *label* for each connection, which can be used to uniquely identify a connection without the tuple. In fact, in addition to the traditional tuple, CELL assigns a connection *two* labels, one for each endpoint.

When a connection is setup, each endpoint independently chooses a label unique within the respective endpoint and sends the label to its peer, as shown in Figure 2-

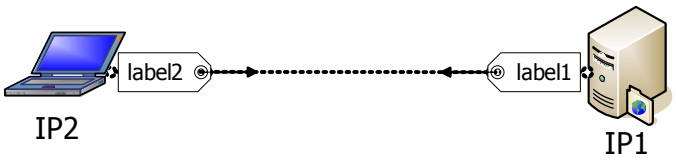


Figure 2-9. CELL abstraction: labels (exchanged at connection setup time)

9. The exchange is conducted *in-band* by piggybacking the labels onto the first data packets exchanged between the two endpoints. For example, for TCP, they are SYN and SYN-ACK packets; for UDP, they are the first data packets arriving on each endpoint. Therefore, no additional round-trip delay is introduced. Also, the piggybacking can be done transport-independently using one of several ways. For example, one way is to use IP option; another is to use encapsulation such as GRE (Generic Routing Encapsulation). In Section 5.1 in Chapter 5, we elaborate on the particular choice we made in our prototype implementation.

Since virtual tuples alone can uniquely identify a connection in the absence of migration, labels are not used in a stationary connection beyond the initial exchange. That is, after the initial exchange, no labels are attached to the data packets and the rest of the packet flow of a connection proceeds as usual, as long as the connection does not migrate. Once the connection migrates, each endpoint attaches its peer's label learned at connection setup time to allow its peer to uniquely identify the connection without relying on the tuple. It is evident that a label needs only to be host-wide unique rather than globally unique since it's only needed for demultiplexing incoming packets to their respective VNICs within an endpoint.

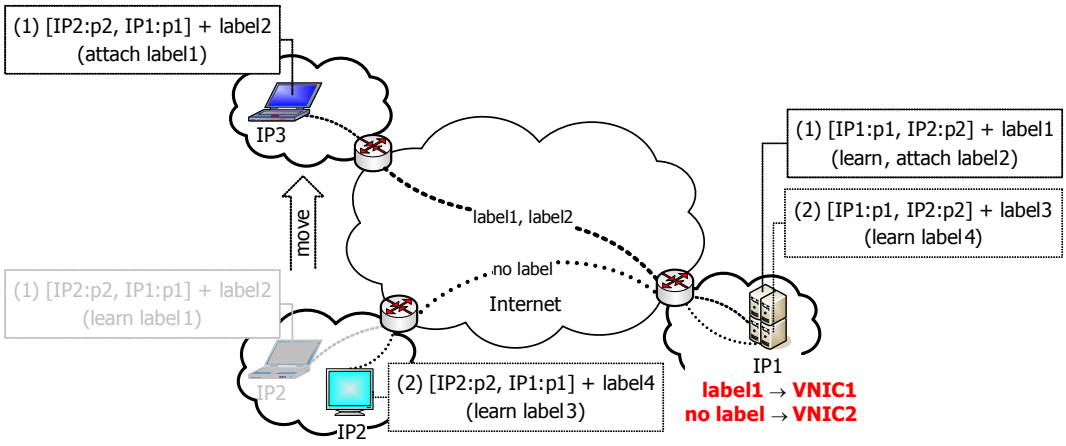


Figure 2-10. Labels identify connections with identical virtual tuple

We first illustrate in Figure 2-10 how labels allow two connections with identical virtual tuple to be properly associated with their respective VNICs. As shown in the figure, CELL assigns the first connection `label1` and `label2`, and assigns the second connection `label3` and `label4`. After the host with the first connection moves from `IP2` to `IP3`, CELL attaches `label1` to all packets of the migrated connection from `IP3` to `IP1`, and `label2` to all packets in the reverse direction. For the second connection, on the other hand, CELL attaches no labels after the initial exchange; since the connection has not migrated. Therefore in this case, the presence of `label1` or the absence of a label will allow host `IP1` to correctly identify the VNICs associated with each connection and to demultiplex packets for each connection properly. If later the second host `IP2` also moves, CELL will attach `label3` and `label4` to the second connection. In this case, the presence of `label1` or `label3`, which are guaranteed to be different since both are assigned by host `IP1`, will allow host `IP1` to distinguish the two connections.

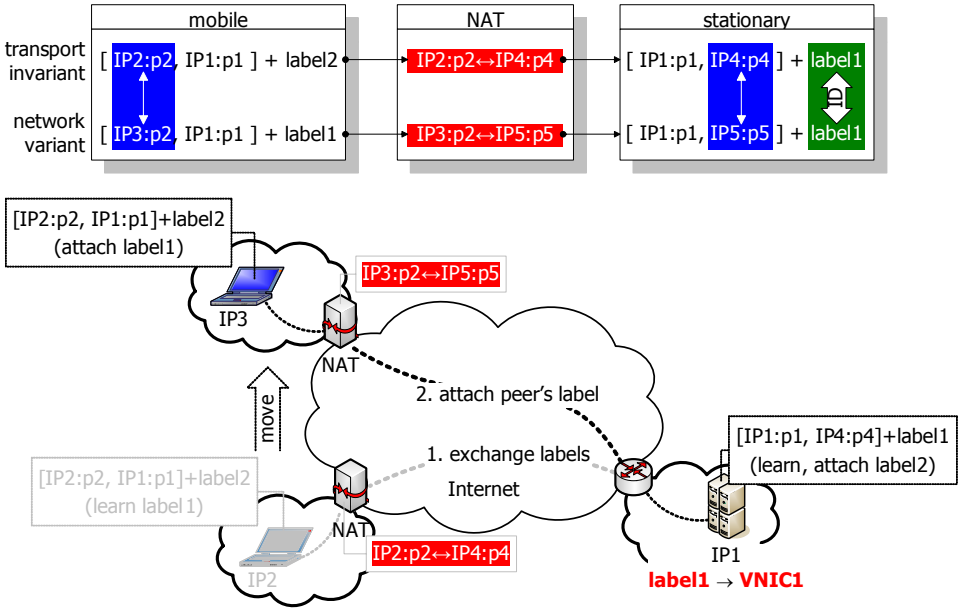


Figure 2-11. Labels identify connections across NAT boundaries

We next illustrate in Figure 2-11 how labels allow connections to be uniquely identified across NAT boundaries. We use the most generic scenario from Section 2.2.3: a host moves from a private network behind a NAT to another private network behind a NAT. As shown in the figure, CELL again assigns the connection two labels, `label1` and `label2` when the connection was setup between host `IP2` and `IP1`. Due to the NAT mapping {`IP2:p2`↔`IP4:p4`} at the original network, host `IP2` perceives the connection as [`IP2:p2`, `IP1:p1`] (with `label2`), while host `IP1` perceives the connection as [`IP1:p1`, `IP4:p4`] (with `label1`). After the host moves to `IP3`, on both host `IP3` and `IP1`, their respective virtual tuples, [`IP2:p2`, `IP1:p1`] and [`IP1:p1`, `IP4:p4`], are maintained to provide transparent migration of the connection. On host `IP3`, CELL translates [`IP2:p2`, `IP1:p1`] into [`IP3:p3`, `IP1:p1`] and attaches `label1` to all packets from `IP3` to `IP1`. Due to the NAT mapping {`IP3:p2`↔`IP5:p5`} at the new network, packets for the migrated connection appear at host `IP1` as [`IP1:p1`,

IP5:p5]. However, host IP1 can determine that the packets belong to the virtual connection [IP1:p1, IP4:p4] by using the label1 attached to the packets; and host IP1 can perform the correct translation of [IP1:p1, IP5:p5] into [IP1:p1, IP4:p4] by observing IP5:p5 from the network variant (physical packet) and IP4:p4 from the transport invariant (virtual tuple).

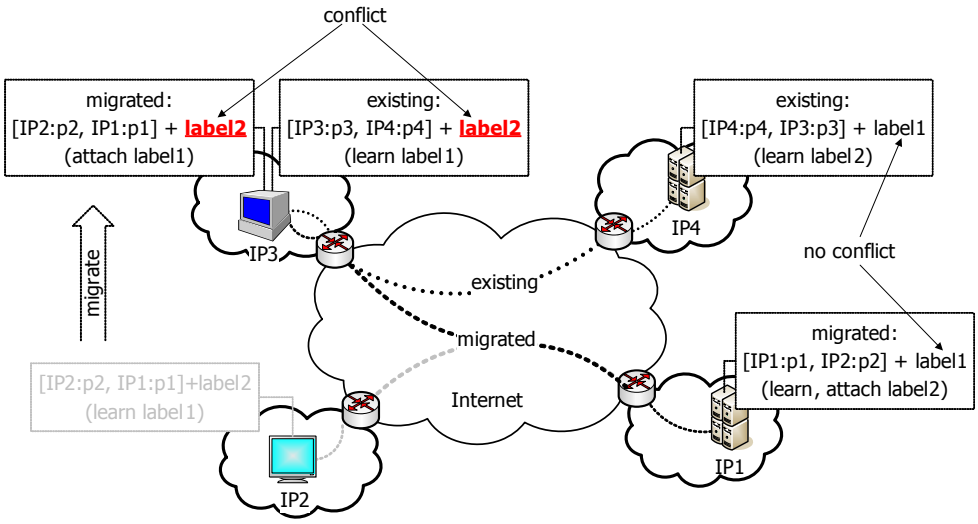


Figure 2-12. Label conflict

Keen readers will notice that since the labels are not globally unique and migrate along with their connections, they will then potentially face the same conflict problem as the virtual tuples do. While label conflict does not happen in the case when a host moves from one network to another, it does happen in the case when a connection migrates from one host to another, as shown in Figure 2-12. We can see that label conflict happens because two pairs of hosts, [IP2, IP1] and [IP3, IP4], can independently choose exactly the same label1 and label2 for the two connections between each pair. This is normally perfectly fine as long as the two connections

do not share the same endpoints. However, when an endpoint of one connection is migrated to the host where an endpoint of the other connection resides, conflict can occur. Note that `label1` used on both host `IP1` and `IP4` does *not* cause a conflict.

To resolve the label conflict, we observe a crucial difference between the label conflict and virtual tuple conflict, which is that a label *does not* have to stay constant throughout the lifetime of a connection while a virtual tuple *does* since it is the requirement of the transparent migration. Therefore, when a label conflict occurs, it can be resolved simply by replacing one of the labels with a new one. For example in Figure 2-12, the migrated connection can simply choose another `label3` and convey the new label to `IP1` during the handoff process. We will see how this is done in Section 5.3.1 in Chapter 5 when we present the design and implementation of MOVE's handoff signaling protocol.

2.3.4 Map between virtual and physical namespace

From previous sections, readers can note that CELL abstraction and its supporting mechanisms really make little change to the normal operation of a connection when it is not migrated. For example, lazy assignment makes virtual-physical translation unnecessary in the absence of migration; connection labels are also only exchanged at the beginning of a connection setup and are never used when the connection does not migrate. This is why MOVE incurs virtually zero network I/O overhead to stationary connections, which we show in our performance measurements in Chapter 6. Eventually, a connection will migrate and its CELL virtual namespace must then be mapped into the physical namespace and *vice versa*. We

describe the mechanisms that perform the mapping in this section, with an example in Figure 2-13.

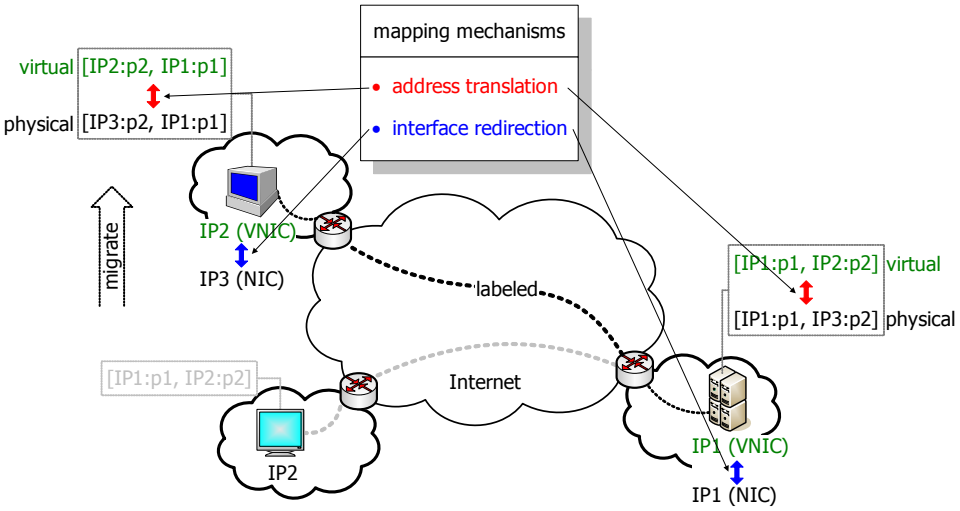


Figure 2-13. Virtual-physical namespace mapping

The figure shows the same example we have been using throughout the chapter, i.e., a connection [IP2:p2, IP1:p1] originally established between the hosts IP2 and IP1; and the endpoint of the connection on host IP2 then migrates to host IP3. We first look at the mobile endpoint of the connection on host IP3. From the figure, we can see that there are two mappings need to be performed:

- the virtual tuple [IP2:p2, IP1:p1] must be mapped into the physical tuple [IP3:p2, IP1:p1], and
- the VNIC with virtual address IP2 must be mapped into the NIC with physical address IP3.

Mapping of the tuple is done by *address translation*, which is commonly available

in modern OSes as part of the packet filtering and firewalling system. The translation is performed at the network layer therefore it is transparent to the transport-and-above layers. In our example, on host `IP3`, since the only difference between the virtual tuple and the physical tuple is the source address, i.e., `IP2` for the virtual tuple and `IP3` for the physical tuple, this is commonly called a source address translation; similarly on host `IP1`, the translation is commonly called a destination address translation. Mapping of the network interface is done by *interface redirection*, which is again commonly available in modern OSes as part of the traffic control system. In our example, we redirect all outgoing traffic of the migrated connection from the VNIC to the NIC; and we redirect all incoming traffic from the NIC to the VNIC. Note that interface redirection is also performed at the network layer and therefore it is transparent to the transport-and-above layers as well. And because both address translation and interface redirection are very common and simple operations, it explains the reason why, for migrated connections, the mapping between the CELL virtual namespace and the physical namespace performed by MOVE incurs very low overhead to the traffic of the connection, as we will also show in our performance measurements in Chapter 6.

Finally, recall that packets of a migrated connection in both directions will carry a label, as we've indicated in Figure 2-13. The label enables CELL to identify the VNIC associated with the connection and to perform the interface redirection from the NIC to the VNIC. The labels are carried in the packets the same way they were piggybacked onto the first packets of the connection when they were exchanged.

To summarize the main points of this section, we have described the CELL namespace abstraction and its supporting mechanisms that provide a clean and elegant solution to the fundamental problems of transparent migration while avoiding various drawbacks of existing solutions. The highlights of the abstraction and its supporting mechanisms are:

- *Virtual network addresses* allow transport tuple to remain constant even when and network address has changed; and *address translation* alleviates the address inconsistency problem. *Lazy assignment* avoids centralized management of virtual address space and eliminates extra round-trip connection setup delay and virtual-physical translation overhead for stationary connections.
- *Private transport identifications* resolve conflict of virtual tuples due to lazy assignment. *Software VNICs* provide each connection with its own private virtual address space that is independent of each other; therefore identical virtual tuples can coexist on the same host. *Invisibility* of the VNICs in the physical network guarantees that network layer protocol semantics are not violated.
- *Connection labels* uniquely identify migrated connections and their associated VNICs, even when the connections pass through NAT devices before and/or after migration. *Interface redirection* maps between the VNIC and NIC for outgoing and incoming traffic of the migrated connections.

We conclude this section by presenting a visual representation of the CELL virtual

namespaces and their relation to the physical namespace, as shown in Figure 2-14. From the figure we can see that each value of the CELL namespace is just one tuple taken from the physical transport namespace plus a label. However, there are two key differences between the CELL namespace and the transport namespace:

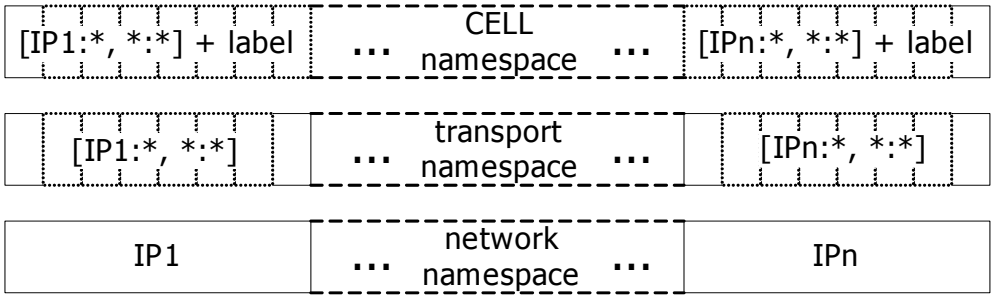


Figure 2-14. Visual representation of the CELL namespace

- Each value of the transport namespace can only appear within one host, e.g., [`IP1`:`p1`, `IP2`:`p2`] can only appear within host `IP1`; each value of the CELL namespace, on the other hand, can appear anywhere in the network, e.g., [`IP1`:`p1`, `IP2`:`p2`]+`label` can appear on any host `IPn`.
- Each value of the transport namespace can only be used once within a single host; each value of the CELL namespace, on the other hand, can be used multiple times on multiple hosts.

In essence, the first properties allows CELL values to persist across hosts. The second properties prevents identical CELL values from conflicting each other. And finally, the labels used in the CELL values maintain the one-to-one correspondence of two CELL values across address spaces.

2.4 Other Architectural Issues

We conclude this chapter with discussions of a few other issues related to general mobile communication.

2.4.1 Host and service location

An important architectural decision made by MOVE is to separate the issue of locating a mobile endpoint before a connection is established and the issue of tracking a connection after it has been established. We believe they are two fundamentally different problems; generally speaking, host locating is a directory problem while connection tracking is a routing problem. The requirements for systems addressing the two problems are fundamentally different. MOVE shows that connection tracking can be done completely within endpoints themselves without mandating new network infrastructure such as the home/foreign agents employed by MobileIP, whose main purpose is for host locating. Decoupling the two allows MOVE to take full advantage of solutions designed specifically for solving the problem of locating a mobile endpoint.

In certain application scenarios, locating a mobile endpoint may not even be an issue. For example, in the proxy-based server cluster we will consider in Chapter 4, a single static name and IP address is exposed to the rest of the world by the proxy. Mobility of the servers or services behind the proxy is purely a local matter without any special requirement on the clients other than regular DNS lookup. Nevertheless, a general mobility architecture needs to address these issues and we

discuss MOVE's approach.

2.4.1.1 Host location

MOVE leverages secure Dynamic DNS (DDNS) [133] to maintain a name-to-IP relationship to address the host locating aspect of the communication mobility problem so that a mobile host can be accessed by the same name after migration. DDNS is also used to locate mobile hosts by other approaches such as [122][136].

Since the mapping from a name to an IP address, the "A-record" in DNS, is cached by name resolvers, it is desirable to have a small caching time of a mobile host's A-record in order to minimize the time during which the mobile host is unreachable. Contrary to belief that small to zero TTL (time-to-live) values for an A-record would increase the DNS lookup traffic and latency and would cause scalability problem, studies by [75] have found that DNS scalability is not as dependent on the caching of A-records as commonly believed. This is because the NS-record, the name server record, which dictates where the DNS name lookup starts, is cache-able. [75] suggests that current trend towards more use of DDNS with low TTL for A-record is not likely to be harmful. [75] further suggests that in terms of overall scalability, eliminating all A-record caching would increase wide-area DNS traffic by at most a factor of 4 and almost none of that would involve a root server or a general top-level domain server. Even eliminating all but per-client caching would little more than double DNS traffic.

Based on these studies, we think that DDNS is a suitable mobile host locating mechanism for MOVE. We conducted additional empirical DDNS studies for its

suitability as our host locating mechanism and we will present our findings in Section 6.4.1 in Chapter 6. Although even with zero to small TTL A-records there is still a chance for the mobile host to be unreachable if it moves frequently, our hope is that higher layer such as name resolvers and application themselves will become increasingly capable of dealing with DDNS name lookups.

2.4.1.2 Service location

To support fine-grain mobility of individual services, i.e., server processes, a corresponding directory service that can locate, in addition to mobile hosts, mobile server processes is required. As a simple example, a host with a DDNS name `foo.move.cs.columbia.edu` may have an IP address `1.1.1.1` and is hosting services such as *ssh*, *pop3*, etc. Later, the *ssh* service may be migrated to another host with IP address `2.2.2.2`. Since the host `foo.move.cs.columbia.edu` did not move, its DDNS name `foo.move.cs.columbia.edu` will still resolve to IP address `1.1.1.1`; and clients trying to reach the *pop3* service on `foo.move.cs.columbia.edu` will continue to be directed to the host `1.1.1.1`. However, clients that trying to reach the *ssh* service on `foo.move.cs.columbia.edu` should be properly directed to the host `2.2.2.2` rather than `1.1.1.1`.

MOVE leverages the SRV resource record (RR) [65] defined for the DNS and dynamically updates the SRV RR to support locating mobile services. [65] defines a mapping from a symbolic {*service name, host name*} to {*port number, IP address*}. The primary intended application of the SRV RR is to allow a single domain to provide multiple instances of a service on different hosts and to allow clients to query these

instances and choose among them. We use a simple example to illustrate at a high-level how SRV RR works. A domain `move.cs.columbia.edu` can define an SRV RR as follows:

```
$ORIGIN move.cs.columbia.edu.
; format of SRV records:
; _service._protocol    SRV   priority weight port host
;
_ssh._tcp               SRV   0 1 22 foo1.move.cs.columbia.edu
                        SRV   0 2 22 foo2.move.cs.columbia.edu
; format of A records:
; hostname              A     IP address
;
foo1                    A     1.1.1.1
foo2                    A     2.2.2.2
```

In this example, the service "`ssh`" over protocol "`tcp`" for the domain `move.cs.columbia.edu` is provided by two hosts, `foo1` with IP address `1.1.1.1`, priority `0`, weight `1`, and port `22`, and `foo2` with IP address `2.2.2.2`, priority `0`, weight `2`, port `22`. A client that makes a DNS query of the SRV RR in the form of `_ssh._tcp.move.cs.columbia.edu` will receive both SRV RRs and can make a choice of the target host based on the "priority" and "weight", which are explained in [65] but not important for our discussion.

By creating per-host SRV RRs and dynamically updating them through DDNS, locating mobile services can be achieved as follows:

- When a service, e.g., `ssh`, is running on a host, e.g., `foo1` with IP address `1.1.1.1`, an SRV RR (and its accompanying A-record) can be created as:

  ```
  $ORIGIN foo1.move.cs.columbia.edu.
  _ssh._tcp   SRV   0 0 22 foo1.move.cs.columbia.edu
  foo1        A     1.1.1.1
  ```

  A client making an SRV RR lookup in the form of

_ssh._tcp.foo1.move.cs.columbia.edu will get IP address 1.1.1.1 and port number 22.

- After the ssh service on foo1 is migrated to another host foo2 with IP address 2.2.2.2, the SRV RR is updated as:

```
$ORIGIN foo1.move.cs.columbia.edu.
_ssh._tcp   SRV   0 0 22 foo2.move.cs.columbia.edu
foo2        A     2.2.2.2
```

Now a client making the same SRV RR lookup in the form of _ssh._tcp.foo1.move.cs.columbia.edu will get IP address 2.2.2.2 and port number 22 instead.

Evidently, in order for this to work, the client must support the SRV RR lookup. Unfortunately, the majority of the network applications today do not support SRV RR lookup. To connect to a service such as ssh, they only make the A-record lookup, which translates a name such as foo1.move.cs.columbia.edu into its IP address 1.1.1.1; and they use the port 22 from a standard static list such as /etc/services. Therefore, to support locating mobile services in current network applications without changing them, we have designed and implemented a mechanism to transparently support SRV RR lookup for these applications. We will present our design in Section 5.6 in Chapter 5 and we will present evaluation of the mechanism in Section 6.4.2 in Chapter 6.

2.4.2 Connection-less transport protocol support

For connection-less transport protocols such as UDP, the notion of a "connection" is undefined beyond the mere association of two communicating endpoints in the

form of a {*source IP address*:*source port number*; *destination IP address*:*destination port number*} tuple. Therefore, connection-less communication can be, in addition to point-to-point, point-to-multipoint. For example, one type of point-to-multipoint communication is *multicast* [47]. A UDP socket can be used to send a packet to a special multicast address (class D IP address ranging from 224.0.0.0 to 239.255.255.255); the packet is replicated at the network layer to any number of receivers subscribing to a multicast group denoted by the multicast address. Since there is no one-to-one correspondence at the transport protocol (in fact, the transport protocol does not even know who the receivers are), the concept of an end-to-end transport connection does not apply in multicast communication; therefore MOVE does not consider multicast communication. Another type of point-to-multipoint is to associate one source address/port pair with an arbitrary number of destination address/port pairs. In this case, the point-to-multipoint communication is maintained by the transport protocol as a group of one-to-one unicast "connections". To enable point-to-point only unicast communication, connection-less transport protocol such as UDP provides a "connected" mode operation to allow explicit binding of two communicating UDP endpoints such that the two will only accept messages from each other and no one else. Of course, this binding does not entail any other connection-oriented properties such as orderly and reliable delivery of packets.

MOVE is designed to provide transparent migration of communication endpoints without assuming any particular transport protocol semantics. To that end, a "connection" for MOVE is a mere association of two communicating endpoints in the

same manner that unicast UDP treats a "connection". Any states beyond the association maintained by the transport protocol, such as TCP's sequence number, sliding window, retransmission timer, etc., are solely the responsibility of the transport protocol itself and are opaque to MOVE. Therefore, a unicast UDP endpoint is really no different from a TCP endpoint as far as MOVE is concerned; and MOVE virtualizes, privatizes, and securely migrates unicast UDP "connections" the same way it does for TCP connections, with some minor differences:

- Since there is no explicit packet exchange for setting up a UDP "connection", MOVE implicitly derives it by tracking the data packet exchange between two UDP endpoints. When the first time a packet is sent from a UDP socket, a virtual tuple for the "connection" is inferred from the sending socket's source address/port and the outgoing packet's destination address/port.
- Since UDP "connections" are unreliable, MOVE must employ its own mechanism to reliably deliver its protocol messages for connection label exchange and handoff procedure. MOVE uses a simple finite state machine to implement this function. Details of the mechanism are presented in Chapter 5.

2.4.3 Application location-awareness

When a connection is virtualized, MOVE has the choice of exposing either the virtual addresses or physical addresses to the applications, such as when the applications call the `getsockname/getpeername` socket system calls. There are pros and

cons for both choices: exposing virtual addresses makes the movement of an endpoint completely transparent to the applications, which is required for certain legacy applications that cannot handle endpoint movement in the middle of an active connection; on the other hand, location-aware applications rely on the current physical addresses for their logic therefore cannot function properly with virtual addresses. We will use an example to illustrate the tradeoff, the choice of MOVE, and the rationale of MOVE's choice.

There are a few commonly used applications such as FTP and ICQ which are well-known to create problems with address translation schemes such as NAT. These applications typically use two separate connections for the their communication, one for control traffic and the other for data traffic. This by itself is not really a problem. The problem, though, is that the ports for the data connection are, instead of being statically allocated and well-known, dynamically negotiated through the control connection. Therefore, in order to perform NAT on the data connection, one must look into the messages exchanged over the control connection to see the ports that have been negotiated for the data connection. This task is obviously highly application dependent. Note that for MOVE, even this dynamic negotiation of ports for the data connection is *not* a problem. Because first MOVE functions entirely within an end host rather than inside the network (where one has to look at the ports used in a packet to infer the sender and/or receiver application); and second as we said in the previous section, there will be *no* mapping on the data connection due to the lazy assignment as long as the connection doesn't migrate.

The problem with FTP-like applications in the face of migration comes as the result of another characteristic of these applications, i.e., they often save the IP addresses of the two machines between which the control connection is established and use them for the data connection. We illustrate the problem using FTP as an example. FTP works as follows in active mode:

1. an FTP client on `IP1` opens a control connection to an FTP server on `IP3` at port 21; *it also saves* `IP1`.
2. to open a data connection, the client creates another socket and binds it to a dynamically chosen port `p1` and listens on this socket.
3. over the control connection, the client tells the server to connect to `IP1:p1` for the data connection; note that the client uses the `IP1` from the saved one in step 1.
4. after the client, along with the live control connection, has been migrated to `IP2`, to open a data connection, it creates another socket and binds it to a dynamically chosen port `p2` and listens on this socket.
5. but now instead of telling the server to connect to `IP2:p2`, the client will tell the server to connect to `IP1:p2` because it has saved the `IP1` in step 1.

When the server attempts to connect to `IP1:p2`, MOVE has two choices: take `IP1` as a virtual address and translate `IP1:p2` into `IP2:p2`, the same way it translates for the control connection; or take `IP1` as a physical address and perform no translation. We can see that MOVE's default lazy assignment, which is essentially the second

choice of exposing physical address `IP1` to FTP and performing no translation, is actually the wrong choice in this particular case. Because the server would have connected to the wrong client `IP1:p2`. The first choice of exposing virtual address `IP1` (remember initially both the physical and virtual addresses are `IP1`) and translating `IP1:p2` into `IP2:p2` would have been the right choice. The reasons that MOVE defaults to exposing physical addresses rather than virtual addresses are the following:

- It supports location-aware applications.
- Most legacy network applications do not require such complete endpoint movement transparency.
- It's the expected behavior of current (non-virtualized) protocol stack.
- It incurs no translation overhead for new connections created after an endpoint moves.

To support applications like FTP that require complete transparency of endpoint movement, MOVE allows, on a per application basis, the exposing of virtual addresses rather than physical addresses. Doing so implies that we must be able to special case these FTP-like applications, which fortunately are the minority. Also note that by exposing the virtual addresses, after migrating an FTP-like application and its control connection, all the new data connections will incur a translation overhead even though they haven't migrated. But as we will see in Chapter 6, the translation overhead is very small.

2.4.4 Compatibility with IPsec

Due to increasing IP network security concerns, the IETF proposed standard IPsec security architecture [78] is gaining acceptance. MOVE must work with connections protected by IPsec. IPsec is a complex suite of protocols and algorithms consisting of security protocols, cryptographic algorithms, and key management, etc. Cryptographic algorithms and key management are issues orthogonal to MOVE and their discussion is beyond the scope of this thesis. Our focus in this section is on the compatibility between MOVE and the two IPsec security protocols: Authentication Header (AH) [76] and Encapsulating Security Payload (ESP) [77]. AH offers data integrity and authentication. ESP offers, in addition to data integrity and authentication, data encryption as well. Both AH and ESP can operate in one of two modes: transport mode or tunnel mode. Transport mode is primarily intended for protecting end-to-end next higher layer protocols between hosts, while tunnel mode is primarily intended for protecting tunneled traffic between gateways. Figure 2-15 illustrates the protection services offered by AH and ESP in transport and tunnel mode.

The key to understand why MOVE is compatible with AH/ESP is that MOVE resides in endpoint only. Therefore, MOVE does not suffer from the incompatibility between AH/ESP and traditional NAT/NAPT, which operates outside endpoints. Within the endpoint, MOVE can be made transparent to AH/ESP by applying virtual-physical address mapping after AH/ESP processing for outgoing packets, and by applying virtual-physical address mapping before AH/ESP processing for incoming packets.

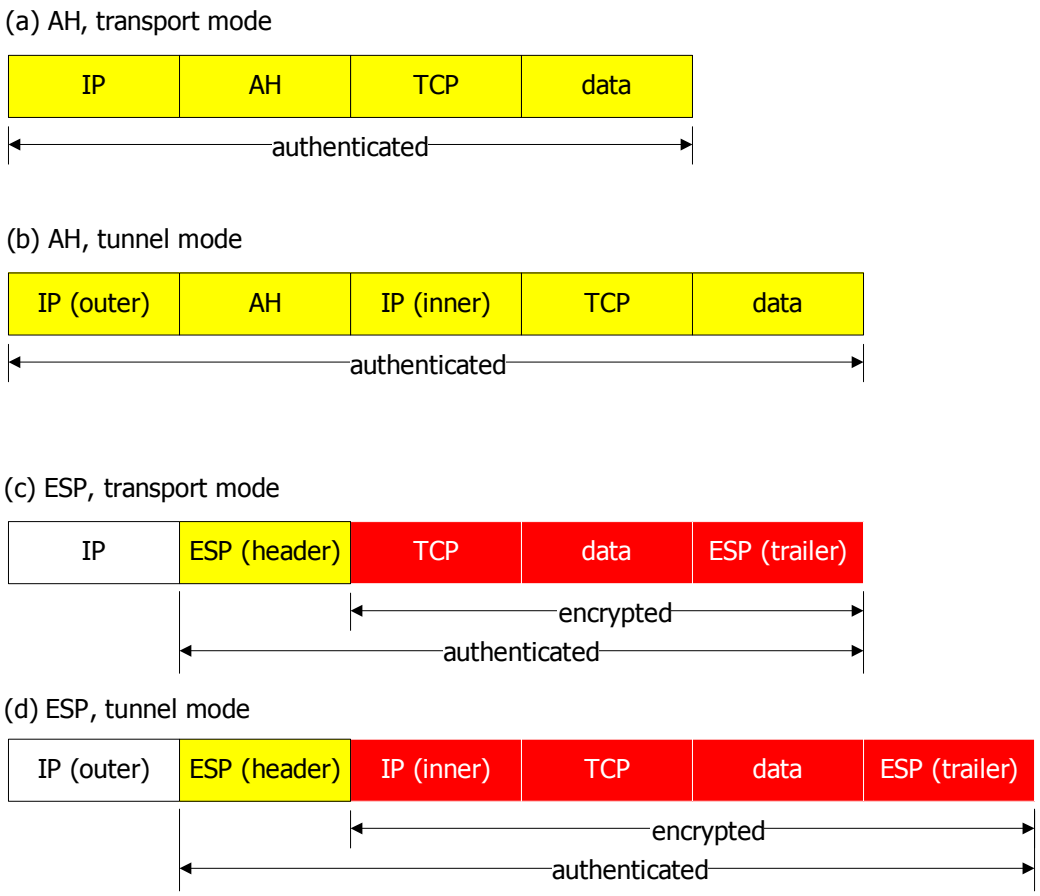


Figure 2-15. AH and ESP protection services

2.5 Summary

In this chapter, we first argued for a transparent connection migration system against a non-transparent one. We then analyzed the fundamental problems of transparent connection migration, namely state inconsistency, conflict, and synchronization. And we introduced the CELL namespace abstraction and its supporting mechanisms, which provide a virtual, private, and labeled namespace for individual connections, as a simple and elegant solution to these problems. We also discussed a few other issues related to a general mobile communication archi-

tecture, e.g., host and service location, connection-less transport protocol support, application location-awareness, and IPsec compatibility.

3 H2O Handoff Signaling Protocol

The functions of an end-to-end transport connection are supported by two distinct components: (1) states maintained by the transport protocol on two logically associated endpoints; and (2) connectivity between the two endpoints. In Chapter 2 we have shown how to preserve the connection states on two logically associated endpoints in a mobile environment and to resolve the fundamental problems of state inconsistency, conflict, and synchronization with the novel CELL namespace abstraction. In this chapter, we turn our attention to the second component: how to maintain the connectivity between two communication endpoints in a mobile environment, a mechanism commonly known as the handoff signaling protocol.

Similar to mobile communication architectures, a large body of prior art exists for handoff mechanisms. Some [51][69][82][123][137] are extensions to MobileIP while others [42][46][110] define their own micro-mobility domain with proprietary routing protocols. The common problem with these mechanisms is that they all require very complex infrastructure support. In contrast, we introduce in this chapter a novel handoff protocol called H2O (Host-only HandOff), that functions entirely within the endpoints and can handoff a connection securely in just a single one-way trip from the mobile endpoint to the stationary endpoint. We show that, through protocol analysis, H2O handoff performance is comparable to and under

certain situation better than existing handoff mechanisms. We will also describe H2O's connection migration helper mechanism to support connection migration by suspension/resumption, where a mobile entity can be disconnected from the network for a prolonged period of time.

3.1 Handoff Related Issues

The connectivity for an end-to-end transport connection is supported at two separate layers: (1) link layer (layer 2) connectivity; and (2) network layer (layer 3) connectivity. Therefore, handoff involves a few related but orthogonal issues such as layer 2 handoff vs. layer 3 handoff, and handoff detection vs. handoff execution, etc. Before diving into the details of H2O mechanisms, we will first clarify these issues and define the problem space H2O addresses.

3.1.1 Layer 2 handoff vs. layer 3 handoff

When an endpoint of a connection changes its point of network attachment, the loss of link layer connectivity may or may not result in the loss of network layer connectivity. For example, in a wired LAN, one can unplug a machine from one jack and plug it into another jack. As long as the two jacks are in the same IP subnet, the machine need not change its IP address. Therefore network layer connectivity is maintained without any special network layer signaling. Another example is in a wireless network such as WiFi, when users move from one access point (AP) to another, their network layer connectivity can also be maintained without any special signaling as long as the two APs are in the same IP subnet. Of course, when

the two jacks or two APs are in different IP subnet, which requires the mobile endpoint to change its IP address, the loss of network layer connectivity must be restored through special network layer signaling.

Maintaining connectivity for an end-to-end transport connection therefore can be achieved either by link layer (layer 2) handoff alone, or by link layer and network layer (layer 3) handoff combined. Using layer 2 handoff alone necessarily restricts the movement scope of a mobile endpoint to be within a single IP subnet. The procedure and performance characteristics of a layer 2 handoff also depend on the particular link layer technology involved, such as WiFi, TDMA, CDMA, or GSM, etc. In this thesis, we consider the more general case when the movement scope of a mobile endpoint is not restricted and therefore requires both layer 2 and layer 3 handoff support. Furthermore, we do not address issues related to layer 2 handoff but rather those related to layer 3 handoff only since the two address distinct technical issues. H2O therefore is a layer 3 handoff protocol that makes no particular assumption about the layer 2 technology in use.

3.1.2 Hand off detection vs. handoff execution

When both layer 2 and layer 3 handoff are required to maintain the connectivity of an end-to-end transport connection, the entire period during which no packets can be delivered to the mobile endpoint extends from the beginning of the layer 2 handoff, i.e., loss of link layer connectivity, to the end of the layer 3 handoff, i.e., restoration of network layer connectivity, as shown in Figure 3-1a. Therefore, a handoff process generally consists of two phases, first handoff detection, followed

by handoff execution, also shown in Figure 3-1a. Handoff detection is defined as the period between the start of layer 2 handoff, i.e., loss of layer 2 connectivity, and the start of layer 3 handoff, i.e., commencement of network layer signaling. Handoff execution is defined as the period between the start of layer 3 handoff and the end of layer 3 handoff, i.e., restoration of network layer connectivity.

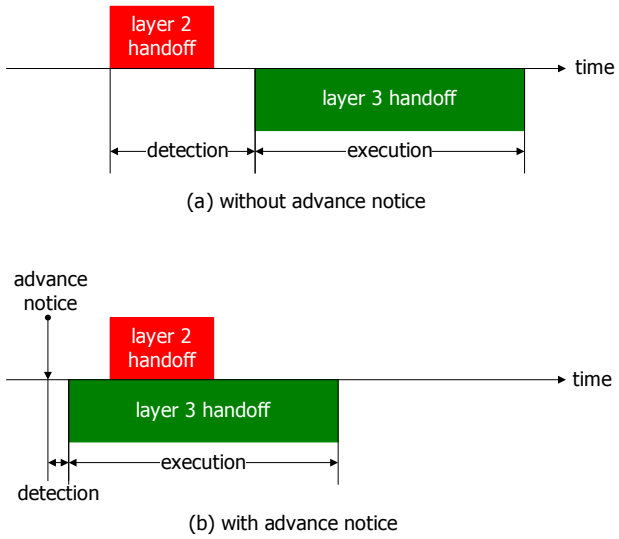


Figure 3-1. Handoff detection and execution

The length of handoff detection depends on the detection algorithm in use. Detection algorithms making use of pure network layer information such as Lazy Cell Switching (LCS), Prefix Matching (PM), and Eager Cell Switching (ECS) [104] generally have long detection delay. Algorithms making use of link layer information such as those suggested in [54][56] can generally improve detection performance. For example, certain link layer technologies, such as CDMA and TDMA, can provide an "advance notice" that the link to a device is about to be dropped. This feature provides the possibility of overlapping the layer 3 handoff with the layer 2

handoff therefore reducing the overall delay of the handoff process, as shown in Figure 3-1b. In this thesis, we do not address issues related to handoff detection phase but rather those related to handoff execution phase only. H2O therefore does not make any particular assumption about the handoff detection algorithm in use. For example, H2O does not assume the "advance notice" feature from the link layer since it's not universally available; the widely deployed WiFi network today does not have this feature. However, if the underlying link layer technology does offer this feature, H2O can take advantage of it and improve its performance.

To summarize Section 3.1.1 and Section 3.1.2, H2O in MOVE is a layer 3 (rather than layer 2) handoff protocol that addresses issues related to handoff execution (rather than handoff detection).

3.2 H2O Handoff Signaling Protocol

The function of a handoff signaling protocol is to notify certain entity (or entities) in the network, which can be the stationary endpoint (SE) itself, that the mobile endpoint (ME) has moved and traffic destined to the old location of the ME must be redirected to its new location. The requirement for the handoff protocol is to minimize the length of the handoff process and the packet loss during the handoff process so as to minimize the impact on the connectivity of the end-to-end transport connection between the ME and the SE.

A general approach to reduce handoff latency and packet loss is to introduce an entity in the network, known as the Mobility Anchor Point (MAP), that is close to

the ME so it can receive handoff signal and start buffering or redirecting packets sooner. This approach however has a couple of drawbacks:

- The shorter distance between the ME and the MAP is only beneficial when the movement of the ME does not result in a change of the MAP. In other words, the distance between the ME and the MAP determines the movement scope of the ME; the shorter the distance, the smaller the movement scope.
- It introduces complexity in the network layer and requires network infrastructure support therefore making it difficult to deploy.

The design philosophy behind H2O is based on the following key observation: for the particular problem of layer 3 handoff, the cost of introducing additional complexity in the network layer to reduce packet loss does not necessarily translate into end-to-end transport layer benefit; because transport protocols and/or applications already have their own way of handling packet loss. For example, TCP's timeout and retransmission mechanism does not distinguish between delayed and lost packets, therefore a layer 3 handoff system that reduces packet loss but not delay (by simply buffering packets) will provide no additional benefit. Even for unreliable transport protocols such as UDP, the benefit of the reduced packet loss is also questionable; because applications using UDP are generally more concerned about the timely delivery rather than the loss of packets.

Therefore, H2O is an end-to-end handoff signaling protocol that functions entirely within the ME and SE themselves without requiring any network infrastructure

support. We have developed solutions for a few technical problems so that H2O can perform handoff securely with just one packet in a single one-way trip from the ME to the SE.

3.2.1 In-band vs. out-of-band signaling

The first design choice for H2O we made is to use an in-band rather than an out-of-band signaling protocol. The choice is based on a few advantages of an in-band protocol:

- H2O protocol messages must be delivered from the ME to the SE reliably. Setting up a reliable out-of-band connection for the signaling protocol incurs extra round-trip delay therefore adding to the handoff latency and packet loss. An in-band signaling protocol, on the other hand, can reuse the existing connection without extra connection setup overhead and can take advantage of the reliable delivery of packets already provided to the existing connection by the transport protocol, such as TCP.
- An in-band protocol messages can be "authenticated" by whatever transport protocol security mechanism already in place, e.g., either plain TCP's sequence number or IPsec, etc. This is another major advantage for an in-band signaling protocol over an out-of-band one. If an out-of-band signaling protocol were used, the ME and the SE will need to conduct separate authentication process, which will increase the complexity and delay of the signaling process.
- An in-band protocol message can serve as a "trigger" for the transport pro-

tocol to immediately restart transmitting packets without waiting for a timeout. Because the protocol message is seen by the transport protocol just as a regular data packet. This feature obviously depends on the internal operational semantics of the transport protocol. We emphasize that H2O does not rely on this feature for its functions but rather it comes "for free" because the protocol messages are carried in-band.

For connection-less transport protocols such as UDP, these benefits of an in-band handoff protocol do not apply since, as we pointed out in Section 2.4.2 in Chapter 2, connection-less transport protocols do not maintain any connection states beyond the logical association of two communicating endpoints in the form of a {*source IP address*:*source port number*; *destination IP address*:*destination port number*} tuple. In other words, the difference between an in-band and an out-of-band handoff protocol for connection-less transport protocols is moot, with the only advantage of an in-band handoff protocol being that it does not require the creation of another endpoint (socket).

An in-band signaling protocol does have its drawbacks though. Generally, the protocol is implemented by interposing "special" protocol messages within the data stream. This introduces two problems:

1. There needs to be a way to differentiate these "special" protocol messages from the regular data messages.
2. Since the protocol messages are treated by the transport protocol as normal data messages, they can be buffered in a queue behind other data messages

which results in the delay of delivering the protocol messages.

The first problem can be handled by techniques such as bit stuffing, commonly used to frame continuous bit streams. However, it requires inspection of *every message body* (instead of just the message header) by the signaling protocol (instead of just the applications). And potentially every message body has to be modified to avoid a normal data message that happens to be the same as the special protocol message being misinterpreted. This will considerably increase the complexity of the signaling protocol and the overhead of processing each message.

Certain transport protocols such as TCP provide an "urgent data" mechanism to allow certain messages to skip to the head of the data message queue, which can be used to deal with the second problem. However, the solution has a couple of drawbacks of its own. First, it obvious depends on the particular transport protocol in use. Second, it may interfere with certain applications such as *telnet*, which make explicit use of such urgent data for their normal operation.

Because of the limitations of these solutions, we have developed another mechanism for H2O to resolve the problems. The mechanism is surprisingly simple. Instead of interposing a "special" protocol message within the data stream, we carry the H2O protocol messages inside the message header rather than the message body. This is possible since H2O protocol messages are all very simple and small. Because the message header is separate from the message body, there is never the problem of misinterpreting data messages as protocol messages. Also since a message header is applied onto all data messages, by putting the protocol message

inside the message header of the data message at the head of the sending queue, H2O can circumvent the problem of protocol messages being blocked by other data messages in the queue without relying on any special feature of the transport protocol. Note that H2O does not incur any processing overhead for the data stream during normal operation. Because data messages with header carrying H2O protocol messages are sent only during a handoff.

3.2.2 H2O protocol operation

Figure 3-2 shows the operation of H2O signaling protocol. We consider three cases corresponding to the different types of link layer technologies:

- no advance notice: link layer provides no warning of the imminent loss of connectivity. WiFi is an example of such network. Note that one can also consider unplug/plug in an Ethernet an example of such network.
- advance notice without simultaneous connectivity: link layer provides warning of the imminent loss of connectivity but does not allow simultaneous connectivity to both the old and new APs. TDMA is an example of such network.
- advance notice with simultaneous connectivity: link layer provides warning of the imminent loss of connectivity and allows simultaneous connectivity to both the old and new APs. CDMA is an example of such network.

We can see that the primary protocol message of H2O is the HANDOFF message. A second protocol message SUSPEND is also defined for the case when the link layer provides advance notice but not simultaneous connectivity. The SUSPEND

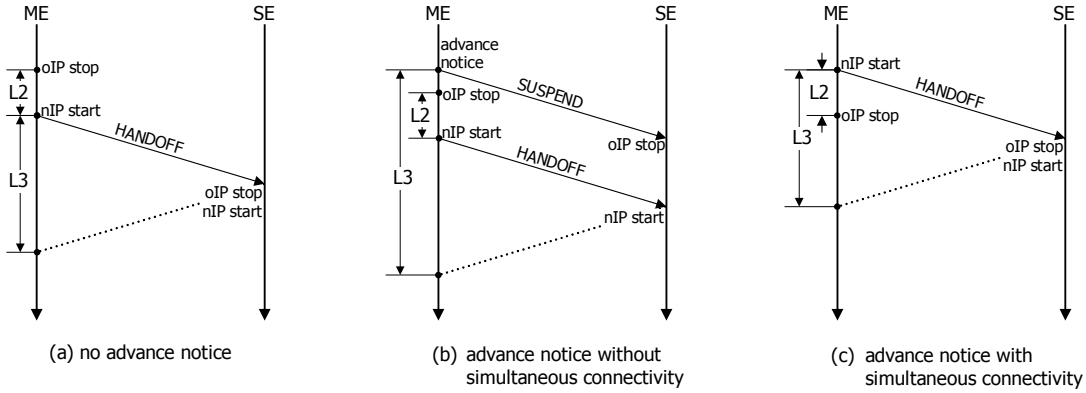


Figure 3-2. H2O protocol timeline

message is also used by H2O when connections are migrated by suspension/ resumption, which we discuss in Section 3.4. All H2O protocol messages are protected by its security mechanism presented in Section 3.2.4.

For the case of no advance notice, when the ME detects a layer 3 handoff, e.g., when it crosses network boundary and acquires a new IP address, it updates its virtual-physical address mapping for the migrated connections and sends a HANDOFF message to the SE. When the SE receives the HANDOFF message, it authenticates the messages, updates its virtual-physical address mapping for the migrated connections, and redirects traffic to the new location of the ME.

For the case of advance notice without simultaneous connectivity, when the ME receives the advance notice, since it does not yet know where it will move to, it cannot update its virtual-physical address mapping. Therefore it sends a SUSPEND message to the SE. When the SE receives the SUSPEND message, nor can it update its virtual-physical address mapping yet. The SE will instead, after authen-

ticating the message, block the owner processes of the migrating connections from sending more messages to reduce message loss during the handoff process. When the ME regains network layer connectivity, it updates its virtual-physical address mapping and sends a HANDOFF message to the SE. When the SE receives the HANDOFF message, it performs the same tasks as those in the case of no advance notice, i.e., authenticates the message, updates its virtual-physical address mapping, and redirects traffic to the new location of the ME. In addition, it unblocks the owner processes of the migrated connections from sending more messages.

For the case of advance notice with simultaneous connectivity, when the ME receives the advance notice, it can start the layer 3 handoff and acquire new network connectivity before losing the current one. As soon as it acquires a new IP address, the ME will update its virtual-physical address mapping and sends a HANDOFF message to the ME. After sending the HANDOFF message, the ME may continue to receive messages from the old IP address, which it will deliver as usual. From the perspective of the SE, there is no difference between the case of advance notice with simultaneous connectivity and the case of no advance notice. The SE performs the same tasks of authenticating the message, updating its virtual-physical address mapping, and redirecting traffic to the new location of the ME when it receives the HANDOFF message. Note that in this case, there is really no loss of layer 2 and layer 3 connectivity at the ME since the new connectivity can be acquired before the old one is lost.

3.2.3 Interaction with existing network security constructs

Maintaining the connectivity between two communication endpoints in a mobile environment is more difficult with the presence of certain network security constructs that are designed to limit network access, such as stateful packet inspection (SPI) firewalls and virtual private networks (VPN). We offer our view on the interplay between mobility and these constructs.

3.2.3.1 SPI firewall traversal

We first clarify that SPI firewall traversal is different from NAT traversal, even though the two functions are commonly performed by one single device. The most important difference between the two is that NAT is stateless *packet* filtering based solely on the IP address and port number, while SPI firewall is stateful *session* filtering based on, in addition to IP address and port number, higher layer protocol (transport and above) information such as TCP SYN flag and FTP PORT command, etc. Furthermore, the operations of SPI firewalls are governed by their security policies.

In Section 2.2.3 in Chapter 2, we have seen how NAT causes the lose of state synchronization on the ME and SE due to the movement of the ME; and we have designed the connection label mechanism to solve the problem. Apart from breaking the end-to-end semantics, NAT does not otherwise hinder the handoff process. That is, the H2O HANDOFF message will pass through the (new) NAT device even though the messages are generated in-band in the middle of a connection. With SPI firewalls, however, the situation is reversed. SPI firewalls do not break

the end-to-end semantics since they generally do not modify packets but rather filter them according to their states and security policies. However, the session states maintained by the SPI firewalls are generally created at the beginning of a session by observing specific high level protocol information carried inside a packet, such as TCP SYN flag and FTP PORT command, etc. [2][18][124] As shown in Figure 3-3, after the migration, the H2O HANDOFF message, which appears to be a packet from the middle of a migrated connection, will not be able to pass a SPI firewall at the new location since the SPI firewall never saw the beginning of the connection. Note that if the migrating "connection" is UDP, the H2O HANDOFF message will be able to pass the SPI firewall.

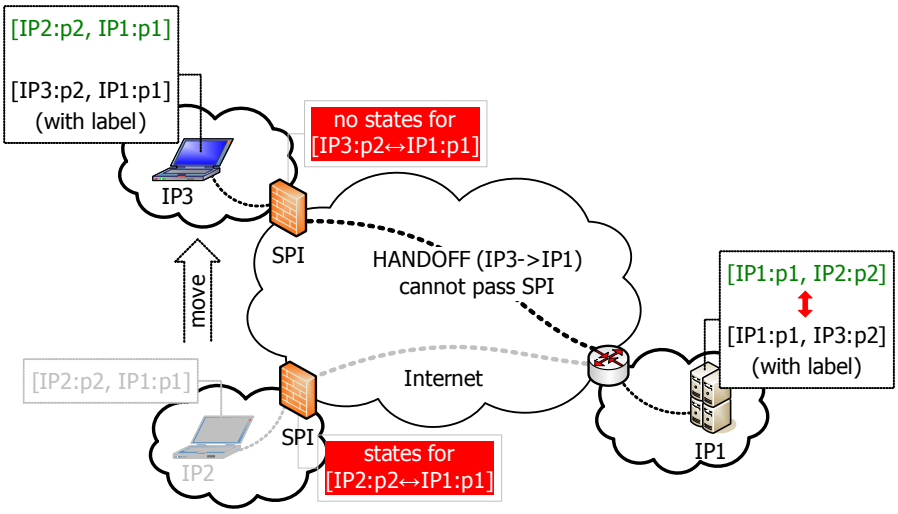


Figure 3-3. SPI firewall traversal

To resolve this problem for TCP, H2O HANDOFF message will have its SYN bit turned on and its ACK bit turned off so that it will appear to the SPI firewall as a SYN packet initiating a new connection and will be allowed to pass through. On

the SE, the reverse is done before the message is delivered to TCP. The first return packet from the SE to the ME will also have its SYN bit turned on, which appears to the SPI firewall as a SYN-ACK packet. Of course, the SYN bit is turned off on the ME before the packet is delivered to TCP. As a result, states for the migrated connection will be created on the SPI firewall at the new location and the rest of the connection traffic can continue. States for the connection on the SPI firewall at the old location will eventually timeout.

We still have to remember that the operations of SPI firewalls are under the control of their security policies. Therefore, the above mechanism will work only to the extent that is permitted by the security policies of the SPI firewall at the new location. For example in Figure 3-3, if the security policy of the SPI firewall at the new location is such that no outgoing connections to the particular site IP1 from IP3 is allowed, then the migration will fail no matter what. This also serves to show that in order to provide a network service, mobility in this case, not only technology but also any other aspects involved such as security policies, etc., must go hand-in-hand. Mobility in particular does not go along well with security. Current SPI firewall security policies are mostly based on the same assumption that is made by transport protocols: IP address and port number do not change for the lifetime of a connection, which is broken by mobility. In this thesis, we address mobility issues in the context of technology; we believe the issues must be addressed in the context of policy as well. However, that is beyond the scope of this thesis.

3.2.3.2 VPN traversal

VPNs are secure private networks constructed on top of insecure public networks through the use of crytography. VPNs are typically used to protect accesses to private corporate resources, which are allowed only by users from within the VPN. There are a few different types of VPN, such as virtual private routed networks, virtual private LAN segment, and virtual private dial networks, etc. [59], but they don't affect our discussion. We will use the virtual private dial networks, also known as remote access VPNs, as the example to illustrate how H2O handoff interoperates with a VPN.

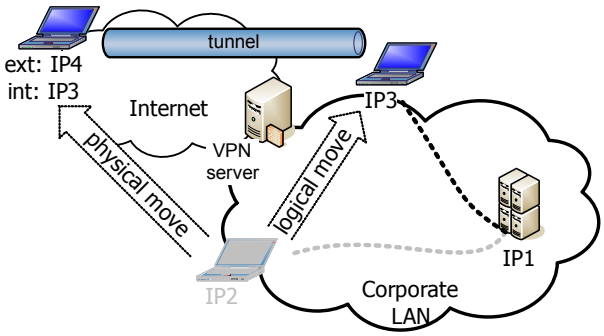


Figure 3-4. VPN traversal

Figure 3-4 shows a user `IP2` accessing a server `IP1` from within the corporate LAN. Later the user moves out of the corporate LAN and connects to the public Internet at `IP4`, e.g., the user goes from his/her office to home. Since the server `IP1` is only accessible through the corporate LAN, the user "dials" his/her corporate VPN server and sets up a secure tunnel; the tunnel logically puts him/her on the corporate LAN at `IP3`. Therefore, as far as H2O is concerned, the connection [`IP2`, `IP1`] simply migrates to [`IP3`, `IP1`]. More specifically, H2O treats the tunnel interface `IP3`

as if it were the physical interface. Tunneling `IP3` inside `IP4` is part of the VPN function and therefore is transparent to H2O.

3.2.4 Migration security

By using an in-band signaling protocol, H2O can take advantage of existing security protection already afforded to the migrating connection. For example, H2O protocol messages are "automatically" protected if the migrating connection is already protected by security mechanisms such as the AH/ESP service provided by IPsec. Unfortunately, IPsec is not yet widely deployed even though it's gaining wider acceptance. Therefore, for unprotected TCP/UDP connections H2O must provide its own security mechanism to guarantee that no additional attack can be carried out by exploiting the added migration functions. H2O does not claim to make unprotected TCP/UDP connections more secure; rather our goal is to make H2O no worse than plain TCP/UDP against existing attacks.

Currently, there are two types of attacks that can be mounted against TCP connections: one is an attacker who is on the path of the TCP connection and can observe and modify packets of the connection; the other is an attacker who is not on the path of the TCP connection and can only guess the packets of the connection. Currently, there is essentially no protection for UDP but H2O protects the migration of UDP connections the same way it protects the migration of TCP connections.

For the first type of TCP attack, which is also known as the man-in-the-middle attack, H2O does not add any additional benefit to the attacker. Because the attacker already has full control of the connection. For the second type of TCP

attack, however, the blind attacker may carry out two additional attacks by exploiting H2O's migration functions. First, the attacker can try to guess the TCP sequence number of a connection and send a fake SUSPEND message to cause the connection to be suspended. This attack, however, is only of marginal benefit to the attacker since he/she can also send a RST or FIN message to cause the connection to be closed. Second, the attacker can try to guess the TCP sequence number of a connection and send a fake HANDOFF message to cause the connection to be redirected to himself. This is a rather serious potential security vulnerability of H2O. Since by managing to redirect a connection to himself, the blind attacker effectively "upgrades" himself to a man-in-the-middle attacker. In other words, by exploiting the connection handoff function of H2O, a blind attacker can potentially conduct a man-in-the-middle attack without actually being on the path of a connection.

3.2.4.1 H2O security mechanism

To prevent these potential exploits by a blind attacker, H2O provides its own security mechanism to protect migration of plain TCP/UDP connections. The mechanism is to use a shared secret key to protect the SUSPEND and HANDOFF messages. The shared secret key is established at the connection setup time through the well-known Diffie-Hellman (DH) key exchange [48]. And the SUSPEND and HANDOFF messages are protected by computing and verifying the Keyed-hashing Message Authentication Code (HMAC) [83] of the protocol messages using the shared secret key. The main challenges are (1) since generating DH key is computationally expensive due to the requirement of computing modular

exponentiation with large prime numbers and H2O must protect migration of individual connections, we must be able to conduct the key computation and exchange very efficiently in order to avoid excessive connection setup overhead; and (2) we must be able to perform the authentication in just one single trip with one message.

The security mechanism employed by H2O is based on two main observations: (1) the computation of the DH public key does not have to be performed on a per connection basis; it can be performed on a per host basis and therefore can be precomputed; and (2) the computation of the shared secret key does not have to be performed at the connection setup time; it can be deferred until the time when a connection migrates.

Specifically, the shared secret key for a connection is established as follows:

1. For a given prime modulus p and generator g, each machine *precomputes* its public key $PK=(g^x \bmod p)$, where x is a randomly chosen private key.
2. When a machine A with PK_A opens a connection to a machine B with PK_B, PK_A and PK_B are exchanged by piggybacking them onto the first a few packets exchanged between A and B. For example, for TCP, PK_A and PK_B are exchanged during the 3-way handshake; for UDP, they are exchanged during the first a few data packets arriving one each machine. The rest of the dataflow of the connection proceeds unaffected as usual.
3. Periodically, A and B refresh their private keys and recompute their public

keys for use with later connections between them.

At the time when a connection on machine *A* is about to be migrated, either by handoff or by suspension/resumption, *A* computes the shared secret key *SK* and then computes a HMAC by applying *SK* to the message header that carries the HANDOFF or SUSPEND protocol messages. When *B* receives the HANDOFF or SUSPEND protocol messages, it too computes the same shared secret key *SK* and verifies the HMAC carried inside the message header. Note that even the shared secret key *SK* only has to be computed once for a given pair of hosts. Once computed, the *SK* can be found simply by looking up a table $(PK_A, PK_B) \Rightarrow SK$ without having to do a modular exponentiation again until either PK_A or PK_B is recomputed.

We see that the two observations of H2O mitigates the cost of computing the public and shared secret keys of the DH protocol. By precomputing the public keys per host rather than on-the-fly computing the public keys per connection, H2O dramatically reduces the connection setup overhead required for DH key exchange protocol. The only connection setup overhead is piggybacking the precomputed public keys onto the first a few packets exchanged, which we show in Chapter 6 is minimal. By deferring the computation of the shared secret key until the time when a connection migrates, H2O eliminates the shared secret key computation overhead altogether for connections that never migrate. By periodically refreshing the private keys and recomputing the public keys on each machine, H2O can reduce the size of the keys used therefore reduce the key computation

overhead when connections migrate. Combining the key size with the refreshing frequency, one can fine tune the migration overhead while maintaining desired security strength.

Since the DH protocol and the HMAC algorithm are well-known security mechanisms, the security strength of H2O's security mechanism is also well understood. The only concern is the denial of service (DoS) attack since a blind attacker may use the HANDOFF or SUSPEND messages to cause H2O to repeatedly perform the expensive shared secret key computation. We note however that DoS attack is not specific to H2O; it's a general attack that is applicable to other security systems such as IPsec as well. And currently there is no panacea for the DoS attack. The general rule of thumb for deterring DoS attack is to defer expensive computations as late as possible when processing incoming messages. This rule is used by H2O for the processing of HANDOFF and SUSPEND protocol messages. Since the HANDOFF and SUSPEND protocol messages are sent in-band through an existing connection, the messages must pass a few preliminary screening by the transport protocol first before the computation of the shared secret key occurs.

3.2.4.2 DH protocol and HMAC algorithm

We conclude this section with a brief overview of the DH key exchange protocol and HMAC algorithm for those readers who are not familiar with these technologies. For more detailed information on these topics, the readers are referred to the respective references [48] and [83].

The DH key exchange protocol is a well-known method developed by Diffie and

Hellman in the landmark paper [48] in 1976. The protocol allows two users to establish a shared secret key over an insecure medium without any prior secrets. It is based on the computationally infeasible discrete logarithm problem and works as follows:

1. For a given prime p (> 2), usually called the prime modulus, and an integer g (< p), usually called the generator, the two parties, A and B, wishing to communicate each individually chooses a private key x and y (< p-1), respectively.
2. A and B compute their public keys as follows: $PK_A=(g^x \bmod p)$ and $PK_B=(g^y \bmod p)$. A and B then exchange their public keys over an insecure medium.
3. When A and B receive each other's public keys, they compute the shared secret key as follows: $SK_{AB}=((PK_B)^x \bmod p)=((g^y \bmod p)^x \bmod p)=(g^{yx} \bmod p)$ and $SK_{BA}=((PK_A)^y \bmod p)=((g^x \bmod p)^y \bmod p)=(g^{xy} \bmod p)$. Note that $SK_{AB}=SK_{BA}$.

The assumption is that it is computationally infeasible to compute the shared secret key $SK = (g^{xy} \bmod p)$ given the two public keys $(g^x \bmod p)$ and $(g^y \bmod p)$ when the prime p is sufficiently large. [91] has shown that breaking the DH protocol is equivalent to computing discrete logarithms under certain assumptions. Note that the DH protocol *is* vulnerable to a man-in-the-middle attack. An adversary who can intercept all traffic between A and B can substitute the public keys of A and B with his/her own during the step 2 above and can establish two separate shared secret keys with A and B. In other words, the adversary can impersonate A to B and

impersonate *B* to *A* and relay all traffic between *A* and *B*.

While the DH protocol is mathematically simple and elegant, in practice, however, its use often still requires specialized hardware when performance is of primary concern. Since the DH protocol relies on computing modular exponentiation with very large prime numbers, it's computationally intensive even though many fast algorithms have been developed to compute modular exponentiation [61] and CPUs are becoming faster and faster. In fact, the increase in CPU speed does not necessarily benefit the DH protocol. Because in order to maintain the same level of security, one has to use larger key size.

The HMAC is an algorithm for computing Message Authentication Code (MAC) using cryptographic hash functions such as MD5 [115] and SHA-1 [50]. MAC is a common way to protect the integrity of data sent over insecure medium by attaching to the data an *authentication tag* that is computed by the MAC algorithm as a function of the data and the shared secret key. The receiver accepts the data only if the recomputed authentication tag matches the one attached to the data. MACs have used to be commonly constructed from block ciphers such as DES [5]. Recently, however, there has been a lot of interest in constructing MACs from cryptographic hash functions such as MD5 and SHA-1. The main reason is that computing cryptographic hash functions is much faster than computing block ciphers, at least in software implementation. However, cryptographic hash functions were not originally designed for computing MACs and therefore their use as MAC algorithms lacks sound security analysis. HMAC intends to fill this gap by

specifying ways to utilize the speed of cryptographic hash functions for computing MACs while offering rigorous analysis of their security properties.

The inputs to the HMAC algorithm are:

- *H*, the cryptographic hash function,
- *K*, the shared secret key,
- *ipad*, a string of 64 bytes filled with the octet 0x36,
- *opad*, a string of 64 bytes filled with the octet 0x5C, and
- *text*, the data to be protected.

The output of the HMAC algorithm is a string of variable length, depending on the particular hash function in use. For example, for MD5, the length is 16 bytes; for SHA-1, the length is 20 bytes.

To obtain the MAC, one computes:

$$output = H(K \; xor \; opad, H(K \; xor \; ipad, text))$$

specifically,

1. if the length of *K* is fewer than 64 bytes, pad it with zeros
2. bitwise exclusive-OR (*xor*) the (padded) *K* created in step 1 with *ipad*
3. append *text* to the result of step 2 and apply *H* to the whole stream
4. bitwise exclusive-OR (*xor*) the (padded) *K* created in step 1 with *opad*
5. append the result from step 3 to the result from step 4 and apply *H* again to

the whole stream

The result of step 5 is the final output of the HMAC algorithm.

3.3 H2O Protocol Analysis

We now present a qualitative analysis of the H2O handoff signaling protocol compared to the signaling protocols used in other handoff architectures such as Hierarchical MobileIP [123] (with or without Fast Handover [82]) and domain-based systems (e.g., HAWAII [110], Cellular IP [42], and EMA [46]). These architectures are all based on the same principle, i.e., using a crossover router (XR) located close to the ME to redirect traffic after the handoff and they differ only in the details of how the new route from the XR to the ME is setup and the exact handoff procedure. Therefore, for the purpose of this analysis, we do not differentiate among these approaches. Rather, to simplify the analysis, we consider the best case scenario for all these architectures; we assume that it takes just one single trip from the ME to the XR to effect the redirection. For the rest of the section, we categorically refer to all these architectures as IBH, infrastructure based handoff. We use TCP as the transport protocol in our analysis since it is the predominant connection-oriented transport protocol in use today. We also consider the scenario that data from the SE are continuously arriving at the MH during the handoff, which is typical of a non-interactive client-server communication (where the ME is the client and the SE is the server) such as file downloading or media streaming.

The purpose of this analysis is to show that while under certain strong assump-

tions, such as advance notice with simultaneous connectivity that lasts long enough, IBH may perform better than H2O, for the most common case of today's data network, the performance difference between H2O and IBH seen at the transport protocol will be negligible.

3.3.1 No advance notice

Handoff without advance notice is the most common case today with data networks. For example, widely used WiFi networks do not provide advance notice for layer 2 handoff and do not provide simultaneous connectivity to both old and new APs during the layer 2 handoff.

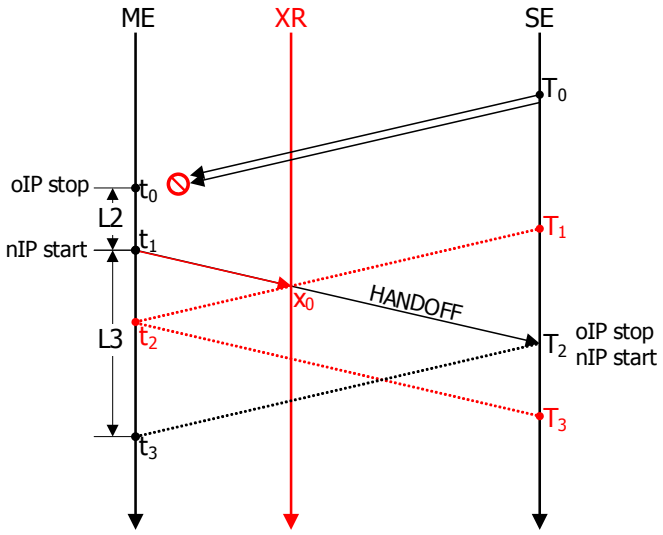


Figure 3-5. H2O analysis: no advance notice

The handoff timeline for both H2O and IBH is shown in Figure 3-5:

- At t_0, the ME starts the layer 2 handoff and therefore losing connectivity to its current AP. Packets sent by the SE at T_0, which would have arrived at

the MH right after t_0, will be lost. Note that with the continuous arrival of data and TCP's sliding window protocol, it is almost certain that at t_0 when the ME loses its connectivity to the AP, either all arrived data have not been acknowledged, or all buffered data in TCP have not been consumed by the application on the ME, or both. In other words, the next acknowledgement from the ME to the SE, which will happen after the layer 2 handoff at t_1, will either acknowledge more data, or open up the window size, or both. In any case, the acknowledgement will allow the SE to send more data to the ME.

- At t_1, ME finishes the layer 2 handoff and regains connectivity to a new AP and a new IP address. This is the earliest time the ME can send and receive packets. For H2O, the ME sends a HANDOFF message to the SE, which will reach the SE at T_2. For IBH, the MH sends a control message to the XR, which will reach the XR at x_0. For both H2O and IBH, if the MH has data to send, it can do so at t_1 and the data will reach the SE at T_2. These data will carry an acknowledgement that allows the SE to send more data to the ME (see discussion of previous bullet at t_0). However, since all data sent by the SE after T_0 are lost, the acknowledgement can only acknowledge the last packet received by the ME before t_0 (i.e., last packet sent by the SE before T_0). Assuming that the layer 2 handoff period [t_0, t_1] is small compared to the RTT (Round Trip Time) between the ME and the SE, the arrival of the acknowledgement at T_2 will cause the SE to resend the lost packets it sent after T_0 without going into slow start due to a timeout.

- If the ME has no data to send at t_1 after regaining connectivity with a new AP (which is the more common case), then the earliest time when the ME can receive a packet is t_3 for H2O and t_2 for IBH, respectively. From Figure 3-5, t_3 corresponds to the times when a packet from the SE was redirected at T_2 after the SE has received the HANDOFF; so [t_1, t_3] is the layer 3 handoff period for H2O during which packets for the ME are lost. Similarly, t_2 corresponds to the time when a packet sent from the SE at T_1 was directed by the XR at x_0 after the XR received a control message from the ME; so [t_1, t_2] is the layer 3 handoff period for IBH. We can see that for H2O all packets sent by the SE between [T_0, T_2] are lost while for IBH all packets sent by the SE between [T_0, T_1] are lost. However, the earliest time when the SE can receive an acknowledgement is T_2 for H2O and T_3 for IBH, respectively. Therefore, although H2O may lose more packets between [T_1, T_2], the SE can start resending lost packets earlier at T_2 and the performance difference between H2O and IBH seen at the transport protocol layer will be negligible. We also note that we have assumed the best case layer 3 handoff period [t_1, t_2] for IBH. More complex interaction between the ME and the XR may prolong [t_1, t_2] and consequently delay the time of arrival of the acknowledgement, T_3. If the delay were long enough to cause a timeout for the lost packets sent after T_0, TCP would go into slow start and the performance of IBH would suffer more.

To summarize, for the most common case with today's data network where is no advance notice for the layer 2 handoff, the layer 3 handoff performance between

H2O and IBH from the transport protocol's perspective is rather negligible.

3.3.2 Advance notice without simultaneous connectivity

With advance notice, packets can still be lost for IBH as shown in Figure 3-6a. This is because the advance notice at t_{-1} must come early enough such that it can reach the XR at time x_0 to allow the XR to start buffering packets sent by the SE after T_0, which would have arrived at the ME after t_0 and been lost since the ME starts the layer 2 handoff at t_0, as shown in Figure 3-6b. In other words, the time between the advance notice and the start of the layer 2 handoff, (t_0-t_{-1}), must be longer than RTT_{MX}, the RTT between the MH and the XR, to prevent packet loss for IBH. Note that if (t_0-t_{-1}) were longer than RTT_{MS}, the RTT between the ME and the SE, then there would be no packet loss for H2O either. But we assume this is unlikely to happen, if we assume that RTT_{MS} is much longer than RTT_{MX}. Therefore we assume there will always be packet loss for H2O. We first consider the case when there is packet loss for IBH; we then consider the case when there is no packet loss for IBH. From Figure 3-6a:

- At t_{-1}, the advance notice causes H2O to send a SUSPEND message to the SE. This message signals the SE to stop sending more packets but does not allow the SE to perform redirection since the ME does not yet know where it is moving to. For IBH, a control message is sent to the XR so the XR can start buffering packets.
- At t_0, the ME starts layer 2 handoff and loses connectivity to its current AP. Packets sent by the SE at T_0, which would have arrived at the ME right

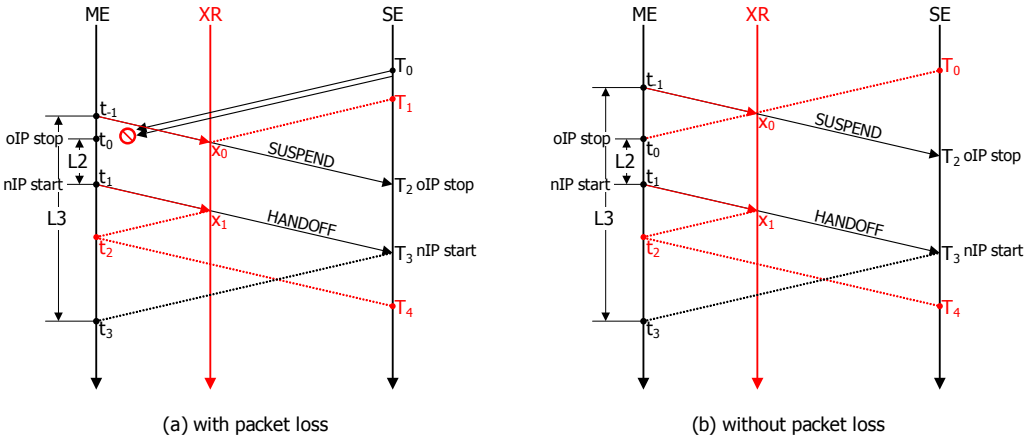


Figure 3-6. H2O analysis: advance notice without simultaneous connectivity

after t_0, will be lost. For H2O, packet loss continues until T_2, which is the time when the SE receives the SUSPEND message and stops sending more packets. For IBH, packet loss continues until T_1, which corresponds to the time x_0 when the XR receives the control message and starts buffering packets.

- At t_1, the ME finishes the layer 2 handoff and regains connectivity to a new AP and a new IP address. This is the earliest time the ME can send and receive packets. For H2O, a HANDOFF message is sent to the SE; and for IBH, a control message is sent to the XR. And the rest of the comparison will be similar to the case with no advance notice. For H2O, at T_3, the HANDOFF message will cause the SE to resend the packets lost between [T_0, T_2]. For IBH, packets sent by the SE after T_1, which are buffered by the XR, will be sent by the XR at x_1 and arrive at the ME at t_2; and the acknowledgements from the ME for these packets will arrive at the SE at T_4. These acknowledgements will carry the acknowledgement for the last packet

sent by the SE before T_0 and signal the SE to resend the packets lost between [T_0, T_1]. Therefore, while H2O may lose more packets, the SE can also start retransmission earlier. Again note that the time T_1 and T_4 in IBH can be delayed due to more complex interaction between the ME and the XR.

In the case of Figure 3-6b when there is no packet loss for IBH:

- For H2O, packets continue to be lost during [T_0, T_2]. However, the earliest time when the ME can send and receive packets is still t_1 for both H2O and IBH.
- For H2O, it behaves similarly to the previous case with packet loss. The SE can start resending packets lost between [T_0, T_2] at T_3 although packet loss is fewer in this case since [T_0, T_2] is shorter than the previous case. For IBH, since there is no packet loss, at x_1 the XR can start delivering buffered packets (those sent by the SE after T_0) to the ME, which will reach the ME at t_2. Acknowledgements for these packets will reach the SE at T_4 and it can start sending again more packets without any retransmission. The net effect is a delay of (t_2-t_0) for packets sent by the SE after T_0.

To summarize, with advance notice but no simultaneous connectivity, if there is packet loss, then the performance difference between H2O and IBH seen by the transport protocol will be negligible. If the advance notice comes early enough for IBH to avoid packet loss, it can perform the handoff with packet delay but no retransmission. Again depending on the difference between T_3, the time for H2O

when the SE can start resending lost packets, and T_4, the time for IBH when the SE can start sending more packets, the case of no packet loss may or may be advantageous for the IBH. In addition, the condition under which packet loss can be avoided is a rather strong assumption and not generally available in today's data network.

3.3.3 Advance notice with simultaneous connectivity

Finally, even with advance notice and simultaneous connectivity, there still can be packet loss for IBH as shown in Figure 3-7a. Because the connectivity to the old AP, which ends at t_0, must last long enough until t_1, which corresponds to the time x_0 when the XR has been notified and started redirecting packets to the new AP, as shown in Figure 3-7b. In other words, the period of simultaneous connectivity, (t_0-t_{-1}), must be longer than RTT_{MX}. Again note that if (t_0-t_{-1}) were longer than RTT_{MS}, there would be no packet loss for H2O either; but we again do not make such assumption. First we consider the case when there is packet loss for IBH in Figure 3-7a:

- At t_{-1}, the ME gains connectivity to the new AP while retaining connectivity to the old AP. For H2O, a HANDOFF message is sent to the SE; For IBH, a control message is sent to the XR. Note that although the ME can send from the new AP as early as t_{-1}, the earliest time it can receive packets from the new AP is t_1, which corresponds to the time x_0 when the XR has been notified and started redirecting packets. During [t_{-1}, t_0], packets from the SE continue to arrive from the old AP.

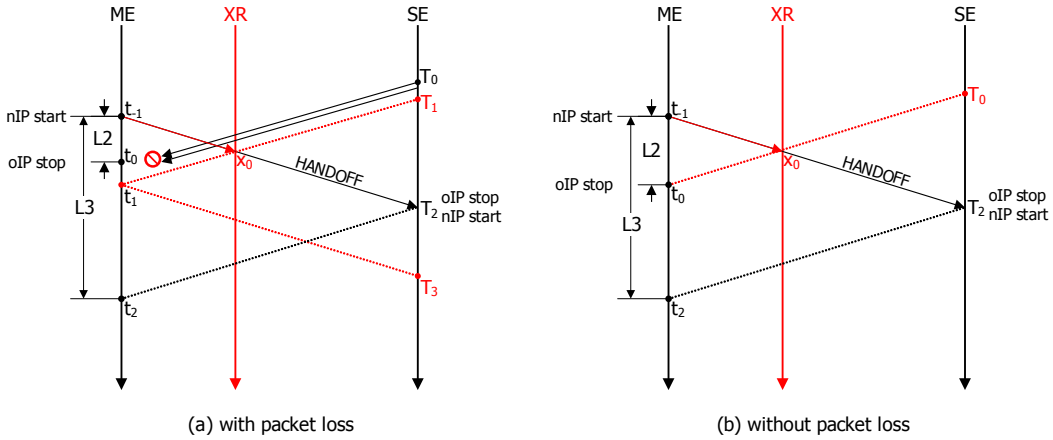


Figure 3-7. H2O analysis: advance notice with simultaneous connectivity

- At t_0, the ME loses connectivity to the old AP. Packets sent from the SE at T_0, which would have arrived right after t_0, are lost. For H2O, the HANDOFF message will reach the SE at T_2 and allow the SE to start resending the packets lost between [T_0, T_2]. For IBH, the earliest time when the SE can receive acknowledgements from the ME is T_3, which allows the SE to resend packets lost between [T_0, T_1]. So once more, this case is similar to the case of no advance notice and the case of advance notice without simultaneous connectivity (with packet loss). H2O may lose more packets but the SE can start resending the lost packets earlier.

In the case of Figure 3-7b when there is no packet loss for IBH:

- For H2O, again there is really no difference between this case and previous case with packet loss. Packets sent from the SE are lost between [T_0, T_2], where T_0 corresponds to the time t_0 when the ME has lost its connectivity to the old AP and T_2 is the time when the SE has received the HANDOFF

message and starts redirecting traffic to the new location of the ME. T_2 is also the time when the SE starts to resend the packets lost between [T_0, T_2].

- For IBH, this is the case when it can achieve truly seamless handoff without packet loss and delay. Before t_0, all packets (sent before T_0 from the SE) are received from the old AP; after t_0, all packets are redirected to the new AP by the XR, which has received the control message at x_0.

To summarize, again with advance notice and simultaneous connectivity, if there is packet loss, then the performance difference between H2O and IBH seen by the transport protocol will be negligible. If the simultaneous connectivity lasts long enough (at least RTT_{MX}) for the IBH to avoid packet loss, it can achieve truly seamless handoff without packet loss and delay. However, this is an even stronger assumption than that for the case of advance notice without simultaneous connectivity in today's data network.

3.3.4 Intra-domain handoff

Although our analysis of H2O and IBH protocols has assumed the case where the ME and the SE are "far away" from each other across the Internet, one should not forget the case where the ME and the SE are "very close" to each other in the same network. As pointed out in [112], this type of traffic, termed as intra-domain traffic, constitutes a large part of today's wireless traffic yet lacks support in existing handoff architectures. For example, architectures such as Hierarchical MobileIP and Cellular IP always route traffic towards the XR even if the ME and the SE are directly connected in the same network, while HAWAII did not specify how intra-

domain traffic is handled. Only Fast Handoff and EMA has provision for handling intra-domain traffic more efficiently. In contrast, since H2O functions entirely within the endpoints, it doesn't care whether the ME and the SE are far away from or close to each other. The H2O signaling protocol always takes one single trip from the ME to the SE; therefore H2O "automatically" takes advantage of the closer distance between the ME and the SE if they are directly connected.

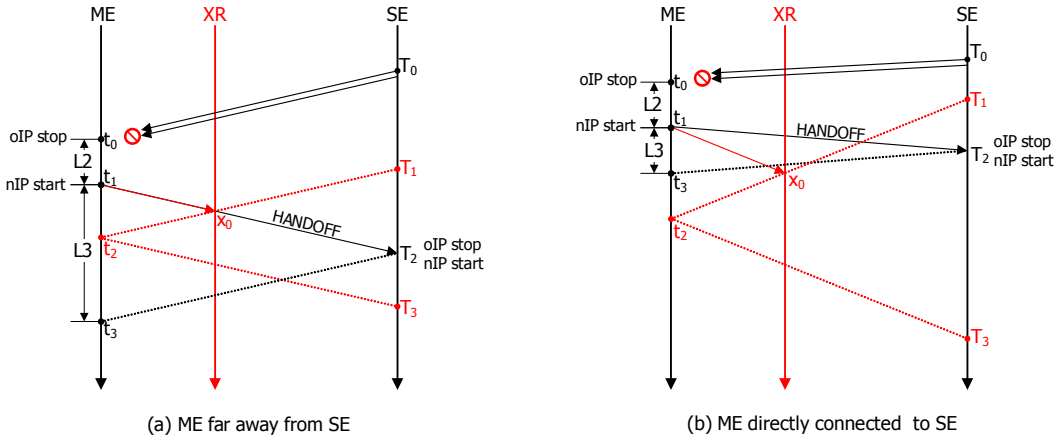


Figure 3-8. H2O analysis: intra-domain handoff

In Figure 3-8 we illustrate the difference between the two cases when there is no advance notice. Figure 3-8a is duplicated from Figure 3-5 as a convenience. As one can see from Figure 3-8b, the time for H2O's HANDOFF protocol message to arrive at the SE, T_2, is now sooner than (or comparable to) the time for IBH's control message to arrive at the XR, x_0. Also, the layer 3 handoff period for H2O, [t_1, t_3], during which packets for the ME are lost, is now shorter than (or comparable to) that for IBH, [t_1, t_2]. More importantly, however, is that the earliest time when the SE can receive an acknowledgement from the ME and start resending lost

packets is much earlier for H2O. For H2O, the SE can start resending packets lost between [T_0, T_2] at T_2; while for IBH, the SE can only start resending packets lost between [T_0, T_1] at T_3. For IBH, TCP in this case would almost certainly have timed out and gone into slow start.

3.4 Suspension/Resumption with Migration Helpers

While existing mobility architectures have all assumed handoff as the way for communication mobility, we recognize that there is another commonly used way for communication mobility, namely through suspension/resumption. For example, laptop users today regularly suspend their laptop at one place such as office and resume it at another place such as home. In this section, we discuss how MOVE supports this type of mobility with the same H2O signaling protocol and addresses some of the issues specific to suspension/resumption.

There are mainly two differences between handoff and suspension/resumption:

- connections are not abruptly dropped as in the case of handoff without advance notice; rather there is a phase where the machine can receive a suspension event and perform necessary preparation prior to the suspension.
- the suspended machine may stay unconnected for a prolonged period of time.

If one recalls the handoff case when there is advance notice but no simultaneous

connectivity (Figure 3-6 in Section 3.3.2), one can see that there is really not much difference between the advance notice and the suspension event. In fact, as far as H2O is concerned, the two events are treated exactly the same way on the mobile endpoint: they both result in a SUSPEND message being sent to the stationary endpoint.

The real issue concerning suspension/resumption is the fact that the mobile endpoint can stay unconnected for a prolonged period of time. Because during this time the suspended connections may have been timed out on the stationary endpoint due to various timeout mechanisms employed either by the transport protocols or by the applications. For example, TCP provides a keepalive mechanism that, when enabled, will send a probe to the peer if a connection has been idle for 2 hours (the timeout is configurable but is recommended and defaults to be 2 hours). Some applications such as *telnet* server use this feature to detect dead client (usually the timeout is changed to 15 minutes). Many other applications instrument their own timeout mechanism. For example, an FTP server will close an idle connection after a preconfigured timeout period. Therefore, one must disable these timeout mechanisms in order to keep the suspended connections on the stationary endpoint alive beyond the timeout limits of these mechanisms while the mobile endpoint stays unconnected. However, due to the transport protocol and application dependent nature of these timeout mechanisms, we recognize that satisfactory solution to this problem is also likely to be transport protocol and application dependent.

In order to maintain transport protocol and application independence of the core MOVE architecture while still being able to deal with transport protocol and application specific issues, we introduce a *connection migration helper* interface. A connection migration helper is an optional function that can be defined by the user and registered with the MOVE system through a well-defined interface. The helper is activated for a connection on the stationary endpoint when the connection is suspended by the mobile endpoint and is deactivated when the connection is resumed. The helper can monitor potential outgoing traffic on the suspended connection and can buffer and/or respond to the traffic in any transport protocol and/or application specific manner. While the focus of MOVE is not on providing a comprehensive suite of migration helpers to address the timeout problem of all applications, we have studied the behavior of several popular servers and developed several application-independent helper that we believe is sufficient for many of today's servers. We emphasize again that the use of these migration helpers is completely optional and we do not claim to handle all applications. One can always elect to migrate a connection using suspension/resumption under the timeout constraints of the transport protocol and/or the applications involved.

The first helper is to disable the transport protocol timeout mechanism such as the TCP keepalive timer on the stationary endpoint. This can be done very easily on a per-connection basis.

The second helper is to block the server process on the stationary endpoint from sending messages on a suspended connection. The majority of the servers today

use either blocking or nonblocking *sockets* for their network I/O. And they either use *select* on the *sockets* or use *read* and *write* (or their variants) directly on the *sockets*. When a connection is suspended, the helper takes over the *select* function associated with the *socket* of the suspended connection on the stationary endpoint. The helper will make it appear that the *socket* is never ready to be written therefore the server process that uses *select* to check the readability and writability of a *socket* will never attempt to send messages through the *socket*. In addition, the helper takes over the *write* function (and its variants) associated with the *socket* and, when the process tries to send a message from the *socket*, either blocks the process if the *socket* is in blocking mode or returns *-EAGAIN* if the *socket* is in non-blocking mode. These functions are restored to their original ones when the connection is resumed. Our experience indicates that this helper works very well as many of today's well-known servers are written in such a "standard" way.

While blocking the server process from sending messages through the suspended connections worked quite well for many servers, there are still certain applications that do not work with the helper. One notable example is the popular FTP servers such as *wu.ftpd* and *in.ftpd*. The timeout mechanism for the FTP servers does not periodically send probe to detect dead client. Rather, the server registers with the OS a timer which will fire if a connection idles beyond the timeout limit of the timer. Fortunately, timers are always registered with a *delta* which specifies how long in the future from now they should fire rather than with absolute time. This allows us to develop a third application-independent helper, which is to "freeze" the timer registered by the server. Specifically, at the time when a connection is

suspended, the helper checks to see if a timer has been registered by the server of the connection. If yes, it finds out the *delta* of the timer, i.e., how long in the future the timer is scheduled to fire. At the time when the connection is resumed, the helper will modify the timer and adds *delta* to the current time. This effectively "freezes" the timer for the period when the connection is suspended.

Finally, we mention an example of applications that none of our migration helper will be able to handle and an application-specific helper is needed. The Internet Relay Chat (IRC) server is such an application. An IRC server usually has its own periodical "ping" mechanism to detect dead clients. Unfortunately, we cannot use the second helper to simply block the server from sending the "ping" when one of its clients suspends and moves. Because the IRC server is a single threaded process that handles all its connections within a single process. If we blocked the server process from sending message through a suspended connection, it would block the entire server process. In this case, an IRC-specific connection migration helper would be needed when an IRC client is suspended and moved to monitor the suspended connection and to respond to the IRC server's "ping" probe until the connection is resumed.

3.5 Summary

We presented in this chapter a layer 3 handoff signaling protocol, called H2O, that is employed by MOVE for handoff execution. H2O has the following features: end-point only, single one-way trip handoff, self-secure, and suspension/resumption

support. The most distinguishing characteristic of H2O is its end-to-end nature, a clear departure from traditional handoff mechanisms that introduce complexity in the network infrastructure. We have shown, through qualitative analysis, that handoff performance difference between H2O and traditional handoff mechanisms is essentially indistinguishable by the transport protocols, due to the fact that transport protocols often have their own packet loss handling mechanisms - a key observation of H2O. We will show with performance measurements in Chapter 6 that H2O incurs minimal impact on the end-to-end transport connection characteristics.

4 High Service Availability Support

In previous Chapter 2 and Chapter 3, we described the fundamental mechanisms of MOVE architecture for supporting transparent migration of fine-grain end-to-end network connections. The endpoint-only nature of MOVE makes it easily applicable, besides general client mobility, to a variety of mobility application scenarios. In this chapter, we describe how we integrate MOVE with a process migration mechanism to fully exploit MOVE's fine-grain connection migration capability and to enable new system support for high service availability. In particular, we show how the integration can provide high service availability in proxy-based server clusters by allowing server applications and their persistent connections to be migrated during a server maintenance to avoid service disruption. We start with our motivation for this particular application scenario.

4.1 Motivation

Online services and businesses are becoming an integral part of our daily life. For example, web, email, enews, messenger are now essentially commodity services; while critical business functions, such as order processing and tracking, inventory control, transaction processing, customer support, and electronic commerce, are also increasingly being conducted online. These services and businesses are sup-

ported by computing and networking facilities that are typically organized as a server farm behind a firewall/proxy and must be up and running 24-7. A few minutes of downtime, scheduled or unscheduled, translates into millions of lost dollars.

Server clusters are one of the most popular ways to meet the stringent demands for service scalability and availability in businesses today. Since server clusters can be built with cheap off-the-shelf hardware components; and free or commercial cluster software are also readily available. Locally distributed web server systems [43] are the most widely used server clusters today. And a lot of research [31][81][101][102][120][130] have been conducted on mechanisms such as TCP handoff to support dynamic content-aware request distribution and to improve the performance and scalability of the web servers. Web server clusters also improve the service availability by allowing requests to be redirected when a server fails or is being serviced. A limitation of these server clusters though is that they require the services to be stateless. That is, each request from a client can be serviced independently by different servers; or in other words, no server application states beyond those trivially replicated ones such as static web pages are needed to serve a request. However, many applications are transactional and stateful, such as database servers and application servers, etc. Providing high service availability for these applications require replicating and transferring both connection states and application states.

Server fault tolerant systems [26][29][80][90][99][138][140] are another way to pro-

vide high service availability in the events of unexpected failure by maintaining a mirror backup server of the primary server at all times. While the backup server replicates all states of the primary server, maintaining the exact mirror of the primary server states has proven to be very difficult. Consequently, current fault tolerant systems require the server applications to be "deterministic", i.e., the server states are completely determined by the network stream between the client and the server. Due to the cost of backup hardware and software, fault tolerant systems are dedicated for a few critical servers rather than loosely coupled server clusters. Most fault tolerant systems also require server transport protocol (TCP) and/or server application change.

Therefore, providing high service availability for arbitrary stateful applications in server clusters remains a unsolved problem. Particularly, events such as scheduled server maintenance today require careful planning and cause lengthy service disruption. For example, typically an announcement is made well in advance in hope to steer clients away from the particular server involved and to reduce potential unsatisfactory factor. Still, any active sessions on the server are discarded at the time of the maintenance. Ironically, the busier the server, the more likely it is to require more frequent maintenance; and the better maintained a system is, the less likely it is to fail. In other words, maintenance today causes conflict in goals. On one hand, it is desirable to perform frequent maintenance in order to minimize the chance of failure; but on the other hand, the service disruption due to the maintenance itself goes up. Therefore, avoiding service disruption due to server maintenance entails great benefits.

4.2 Example High Service Availability Scenario

The application environment we focus on is a proxy-based architecture commonly deployed by business service providers today, as shown in Figure 4-1. The proxy is a single address frontend that admits service requests from clients across the Internet, and dispatches the requests to the appropriate backend application servers. The proxy can operate at either layer 4 or layer 7 and can employ any suitable scheduling rule and load balancing policy for dispatching the requests.

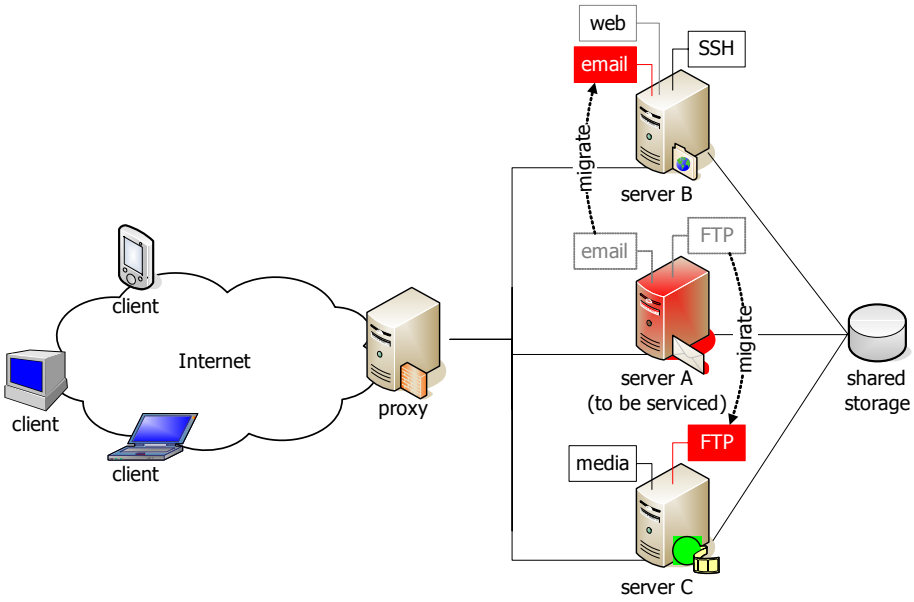


Figure 4-1. High service availability in proxy-based server cluster

We instrument both the proxy and the servers to provide zero service disruption server maintenance without touching the clients. The following steps are taken at maintenance time:

Stop dispatching new requests to a server *A* that is about to be serviced.

However, requests for existing application sessions on server *A* continue to be forwarded.

Relocate existing sessions on server *A*, along with their open network connections, to other servers such as *B* and/or *C*. We allow sessions to be distributed to several other servers rather than just one because (1) it avoids overloading a running server; (2) one particular running server may not have all the software/hardware configuration necessary to support all the active sessions on server *A*.

Resume dispatching new requests to server *A* after maintenance. Migrated sessions of server *A* may continue to finish on server *B* and/or *C*, or be migrated back to server *A*.

We assume that there can be any number of services on each server, but only one instance of a service presents on each server (assuming no other virtualization such as VMware [22] is used). We also assume that a given service can be served by two or more servers in the cluster, which is generally the very purpose of a cluster. Stopping and resuming new requests while continuing to service existing sessions are common functions available on modern proxies such as Resonate Central Dispatch [16] and Foundry ServerIron [7]. So the main challenges are:

- migrate stateful sessions and their open persistent connections from one server to another
- allow easy deployment with minimal cluster configuration and management, no server OS or application change

- incur minimal server performance overhead and provide fast connection handoff while retaining scalability

We meet these requirements by integrate MOVE with the Zap [100] process migration mechanism. We introduce a new process and connection abstraction, called zPod, which combines Zap's Pod abstraction for process states with MOVE's CELL abstraction for connection states. zPod therefore provides a virtual and private namespace for process states as well as transport connection states.

4.3 The zPod Abstraction

Migrating stateful application with open connections remains a difficult problem. Existing migration abstractions have generally required assigning each migratable unit with its own routable IP address, making the unit a host-like entity in terms of network communication. However, this requirement raises network configuration, management, and compatibility issues.

First, since the migratable units are volatile and created dynamically on-demand, assigning them with static name/IP is infeasible; additional mechanisms such as DHCP are needed, which add possible sources of failure. Furthermore, dynamic server name/IP conflicts with current server configuration and management practice as existing server clusters are mostly configured with static server name/IP for easier management and better control. A few examples are:

- Cluster components such as loader balancers require configuration of vir-

tual-physical IP address mapping, which can be very difficult to do with dynamic server IP. Most monitoring software also requires static server name/IP.

- Certain server properties such as SSL certificate are assigned on a per-name basis; and many server software are licensed on a per-IP basis. These can be very hard to manage with dynamic server name/IP.
- Due to security concerns, static server name/IP is used for better control. For example, IBM Global Service configures everything static to have better control on routing and address allocation.

Besides adding network configuration and management complexities, individual name/IP for each migration unit is incompatible with certain existing networking constructs, RPC (remote procedure call) being an example. RPC port mapper protocol allows a server written in RPC to register its listening port, transport protocol, program number, etc. But the implicit assumption is that the server will be at the same IP address where the port mapper is. Therefore, a server written in RPC must use the machine's IP address which the port mapper uses.

zPod, similar to Zap's pod, provides a group of processes with a virtual and private namespace for a complete set of the underlying OS resources, including transport connection states; and zPod allows these processes, along with their open connections, to be transparently migrated across hosts. zPod is a VM-like entity but without a guest OS; therefore zPod is very light-weight and requires no configuration. zPod simply maps between its virtual namespace and the underlying

physical OS resources. Particularly, zPod exposes to its encapsulated applications a single virtual interface, which just mirrors the physical interface of the host where the zPod is initially created. All zPods on a host share the host's interface and are reachable only through the different port number of the service they encapsulate, just like regular server processes. When a zPod migrates, its virtual interface stay intact and is mapped to the physical interface of the new host. Therefore, zPod is completely transparent to the host and requires zero network configuration and management, and is also compatible with network constructs such as RPC.

Migrating zPod requires MOVE's fine-grain connection migration; since migrated and non-migrated connections share the same host interface and they are not distinguishable by traditional host mobility solutions such as MobileIP. Although other mobility solutions such as transport layer and application layer approaches do provide fine-grain connection migration capability, these solutions, besides not being designed with process migration integration, all have drawbacks (see Chapter 7 related work) and do not meet deployability and performance requirements.

4.4 zPod Migration

Migrating a zPod amounts to packaging its process and connection states on one machine, and transporting and restoring the states on another machine. This thesis focuses on the connection migration aspect of zPod migration; therefore we refer

readers who are interested in the process migration aspect of zPod migration to the original Zap paper [100]. We reexamine the connection migration problems in the context of such proxy-based server clusters and see how the CELL abstraction and MOVE mechanisms can be equally applied to solve these problems. Note that for the rest of the chapter we concentrate on the proxy and the server part of the system since process and connection migration are completely transparent to the clients.

4.4.1 General server clusters

Figure 4-2 depicts problems of connection migration in a general proxy-based server cluster where servers can reside across different subnets. Obviously, connection migration in the server cluster does not have to deal with cross address space synchronization problem due to NAT described in Section 2.2.3 in Chapter 2. However, the problems of inconsistency between network layer and transport layer and conflict in transport layer, described in Section 2.2.1 and Section 2.2.2 in Chapter 2 respectively, still arise. As illustrated in Figure 4-2, when a connection [`IP20`:`p2`, `IP10`:`p1`] is migrated from server *A* (`IP20`) to *C* (`IP30`), the same inconsistency problem as that described in Figure 2-1b in Chapter 2 occurs. Also as illustrated in Figure 4-2, if another process on server *A* (`IP20`) reuses port `p2` to connect to the proxy (`IP10`) on port `p1` after the first connection [`IP20`:`p2`, `IP10`:`p1`] is migrated to server *C* (`IP30`), we see that the same conflict problem as that described in Figure 2-2b in Chapter 2 will occur.

Without repeating the content of Chapter 2 and Chapter 3, readers should be able

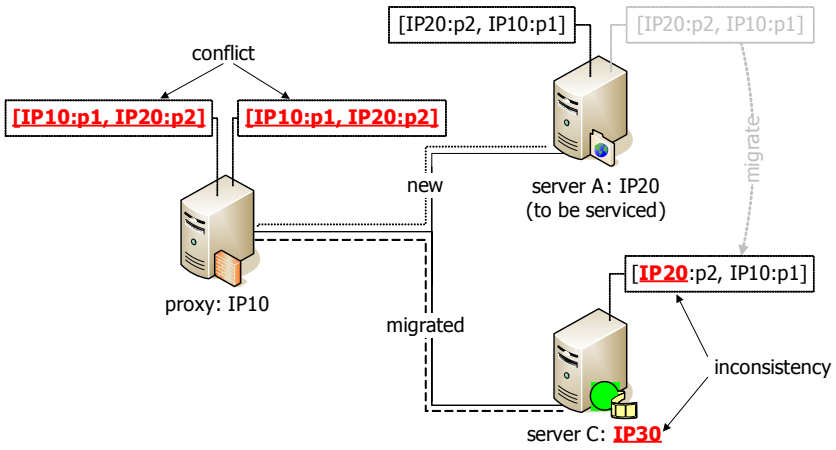


Figure 4-2. Connection migration in proxy-based server clusters

to convince themselves that supporting connection migration across different subnets of servers is a straightforward process of enabling MOVE on both the proxy and the servers because MOVE does not differentiate the two communication endpoints of a connection. By taking advantage of the proxy as the "anchor" point, MOVE can provide migration of the server end of a connection without touching the client end, as shown in Figure 4-1.

One issue worth some discussion is security. Since server clusters are generally protected by the proxy which also acts as a firewall, connection migration security in a server cluster is not as pressing as it is in a public network. However, firewalls are not one hundred percent safe; they can be broken into and when they are the servers behind the firewalls are just as open and unprotected as those on the public networks. For example, when a firewall is broken in and a server is compromised, an attacker can potentially carry out the same attacks that we described in Section 3.2.4 in Chapter 3. For example, the attacker can try to send a fake HAND-

OFF message to the proxy and cause a connection belonging to another server to be redirected to himself. Therefore, we believe that migration in server clusters must be protected the same way as they are in public networks. MOVE's security mechanism therefore applies equally in the server clusters as well.

In addition, to be compatible with IPsec, the proxy must terminate the IPsec security association with the client. Because the proxy can work either at layer 4-7, in which case the termination is natural, or at layer 3, in which case the termination is necessary for its address translation functions that are inherently incompatible with IPsec. In either case, an IPsec connection between the client and the server consists of two security associations: one between the client and the proxy, and the other between the proxy and the server. Each association is an end-to-end association. Therefore, MOVE functions performed on the proxy and the server are compatible with the proxy-server part of the IPsec security association; as they are compatible with any end-to-end IPsec security association we described in Section 2.4.4 in Chapter 2.

4.4.2 Different types of proxies

Another aspect of connection migration in the proxy-based server clusters concerns with different types of proxies, depending on at which layer the proxy operates and whether the proxy maintains full transport connection states. Application Level Gateways (ALG) are proxies operating at application layer and maintaining full transport connection states. A connection between the client and the server going through an ALG is in fact two separate connections, one between the client

and the ALG and the other between the ALG and the server, "spliced" together transparently by the ALG at the application level. Layer 4-7 switches are proxies operating at network layer without maintaining full transport connection states. Instead, they use layer 4-7 information of packets for making dispatching decisions and relay traffic between the client and the server by rewriting transport tuple in the packet header once the decision is made. Both types of proxies have their own pros and cons and are widely used in the real world: ALGs offer great flexibility in supporting various application layer protocols such as FTP, SSL, etc., and are simple to implement, while layer 4-7 switches have better performance.

Recall in Section 2.3.4 in Chapter 2, the mapping mechanisms of MOVE, address translation and interface redirection, function at network layer. Therefore, MOVE is transparent to ALGs. In fact, to MOVE, ALGs are exactly the same as any end host client or server applications. Layer 4-7 switches, on the other hand, operate also at network layer and their packet header rewrite function is rather similar to the address translation of MOVE. While MOVE can also work transparently without requiring any change to the switch, because of the similarity between the functions of the two, we discuss how they can be easily combined to simplify the switch functions and improve its performance. Layer 4-7 switches often come in two flavors, depending on how incoming traffic to the servers and outgoing traffic back to the clients are routed. The main difference between the two is in the traffic from the servers back to the clients. In *two-way architectures*, both incoming and outgoing traffic pass through the switch, while in *one-way architectures* only the incoming traffic passes through the switch. We discuss each architecture in turn.

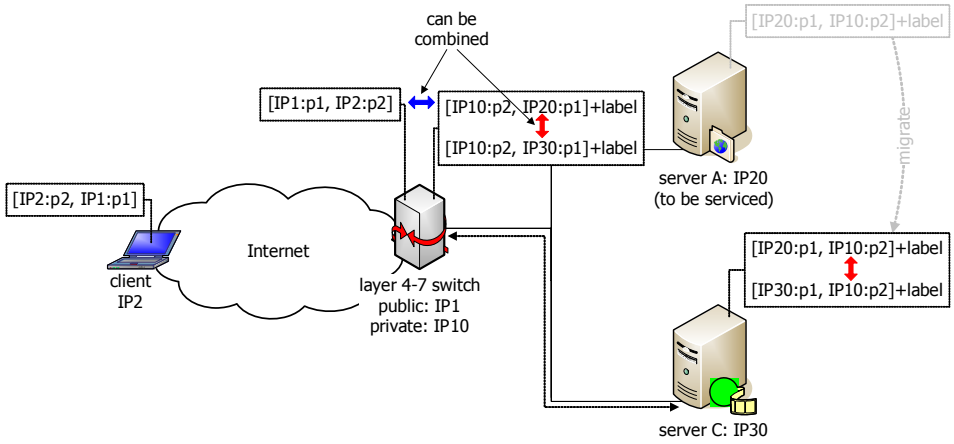


Figure 4-3. Combine MOVE and layer 4-7 switches: two-way architecture

Figure 4-3 shows the two-way layer 4-7 switch architecture. The figure shows a client opens a connection [IP2:p2, IP1:p1] to the public address IP1 of the switch, which is rewritten by the switch as [IP10:p2, IP20:p1] (with MOVE connection label) and forwarded to the server IP20. Return traffic from the server to the client is also translated properly by the switch. When the connection is migrated to the server IP30, two translations are performed on the connection by the switch, one is the original translation between [IP1:p1, IP2:p2] and [IP10:p2, IP20:p1] and the other is MOVE translation between [IP10:p2, IP20:p1] and [IP10:p2, IP30:p1]. The two can be easily combined as a single translation between [IP1:p1, IP2:p2] and [IP10:p2, IP30:p1].

Figure 4-4 shows the one-way layer 4-7 switch architecture. The figure shows a client opens a connection [IP2:p2, IP1:p1] to the public address IP1 of the switch, which is rewritten by the proxy as [IP2:p2, IP20:p1] (with MOVE connection label) and forwarded to the server IP20. Note that the source address IP2 is not changed by the switch and as a result the server IP20 perceives the connection as coming

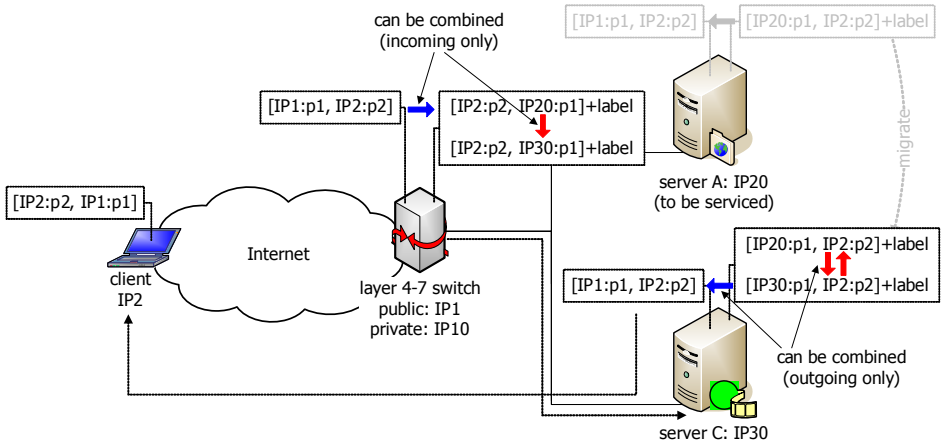


Figure 4-4. Combine MOVE and layer 4-7 switches: one-way architecture

directly from the client. The return traffic goes directly from the server back to the client without passing through the switch. This, of course, requires another connection for the return traffic; in addition, before sending packets back to the client, the server will translate its own private address `IP20` back into the public address `IP1` of the switch, as shown in Figure 4-4. When the connection is migrated to server `IP30`, we can see that both the switch and the server `IP30` perform two translation on the connection. On the switch, both original translation and MOVE translation are performed for the incoming traffic. On the server `IP30`, both original translation and MOVE translation are performed for the outgoing traffic; in addition, MOVE translation is performed for the incoming traffic. From Figure 4-4, we can see that the original translation and MOVE translation on both the switch and the server `IP30` can be combined for their respective traffic direction, similar to the way they are combined in the previous case of two-way architecture.

4.4.3 Single subnet of servers

Because the server cluster is under complete control of its owner, one potential

solution to deal with connection migration when processes are migrated between servers is to connect all servers in a single flat subnet. A subnet is defined as a network segment that is solely connected through layer 2 (switches and hubs) and below (repeaters) elements. This way, when connections are migrated along with their zPods from server to server, their layer 3 IP addresses need not change. Therefore, all connections can be kept intact from the point of view of layer 3 and above. Specifically, the mechanisms would work as follows:

- Assign IP addresses on a per-zPod base rather than a per host base. Recall from Section 4.3 that dynamically assigning per-zPod IP address is against the general practice of static configuration of server clusters. However, there may be situations where statically assigning per-zPod IP addresses is feasible, for example, when the number of zPods are relatively small and static (they only host pre-defined well-known services).
- Divide the entire IP address space of the single subnet into two parts, one for all the servers, and the other for all the zPods on the servers. For example, assume the IP address space for the subnet is the 16-bit private address block `192.168/16` (defined in [113]), `192.168.0.1` - `192.168.0.255` can be reserved for the (255) servers, while the rest `192.168.1.0` - `192.168.255.254` can be reserved for the (255*256-1=65279) zPods.
- Each time a zPod is created on a server, it's assigned a unused IP address from the pool above, which is used for all network communication from and to the zPod. The IP address for the zPod is created as an *alias* to the server's NIC. For example, the server NIC can have an IP address

`192.168.0.1`, while an IP address `192.168.1.1` can be assigned to the NIC as an alias for the zPod.

- When the zPod is migrated to another server, e.g., `192.168.0.2`, its IP address `192.168.1.1` need not change since the entire network is a single subnet `192.168/16`. Rather, its IP address is simply (re)created as an alias to the NIC of the new server. ARP (Address Resolution Protocol) will map the zPod's IP address `192.168.1.1` to the MAC address of the current server's NIC.

It should be evident that since the IP address of a zPod never changes and is never reused within the single subnet, the inconsistency and conflict problems due to migration can be avoided. Note that ARP caching may cause the zPod migration to be "invisible" to the proxy until its ARP cache times out, which is typically a few minutes. This can be easily addressed by having the migrated zPod send a "gratuitous" ARP request asking the MAC address of its own IP address, which allows the proxy to immediately invalidate its ARP cache for the zPod's IP address in question. "Gratuitous" ARP requests are commonly used to detect duplicate IP address, and to allow a backup server's NIC to take over a primary server's NIC.

A single flat subnet, however, has a few drawbacks that limit its scalability, such as broadcast storm, switch address table overflow, and spanning tree loop, etc. In addition, physical limitations of the media make it difficult to expand the network even across buildings. For example, 100Mbits ethernet has a hard limit of 100 meters between a transmitter and a receiver. Some of the problems such as broad-

cast storm can be addressed by techniques such as VLAN [9]. But these solutions are often vendor specific and introduce their own management complexities. Therefore, in practice any nontrivial size LANs are almost always divided into different subnets.

4.5 Summary

In this chapter, we described how we integrated MOVE with the Zap process migration mechanism to fully exploit MOVE's fine-grain connection migration capability. We developed the zPod abstraction to unify the migration of process states as well as connection states. We demonstrated how zPod enables zero disruption service availability for arbitrary stateful applications during server maintenance, without introducing additional server cluster configuration and management complexity. We showed that our solution meets the requirements of preserving application and networking sessions, complete transparency to the clients, no modification to server OS and applications, and compatibility with different proxy and server cluster configurations. In Chapter 6, we will show the server handoff performance of MOVE integrated with Zap in such proxy-based server clusters.

5 Design and Implementation

To demonstrate the viability of the CELL concept and MOVE architecture mechanisms introduced in Chapter 2 and Chapter 3, we have implemented a prototype MOVE system on the LINUX 2.4 operating system running on Intel x86 family processors. We have implemented all of the MOVE functions as a kernel module, which can be dynamically loaded and unloaded at any time without modifying, recompiling, or rebooting the kernel. The entire system has fewer than 500 lines of C code, which serves as another testimonial to the simplicity and elegance of the CELL concept and its supporting mechanisms. In the following sections, we first present an overview of the MOVE system functions and we then look at each functional component in more detail.

5.1 Functional Design Overview

MOVE consists of three major functional components, corresponding to the three points of time during a connection's lifetime when these functions are performed. They are shown in Figure 5-1a. Figure 5-1b shows where these three components reside inside the OS kernel relative to the standard protocol stack and how they interact through a set of tables.

Part of the security module, which resides at the boundary of transport and network protocol layers, implements the exchange of a per-connection Diffie-Hell-

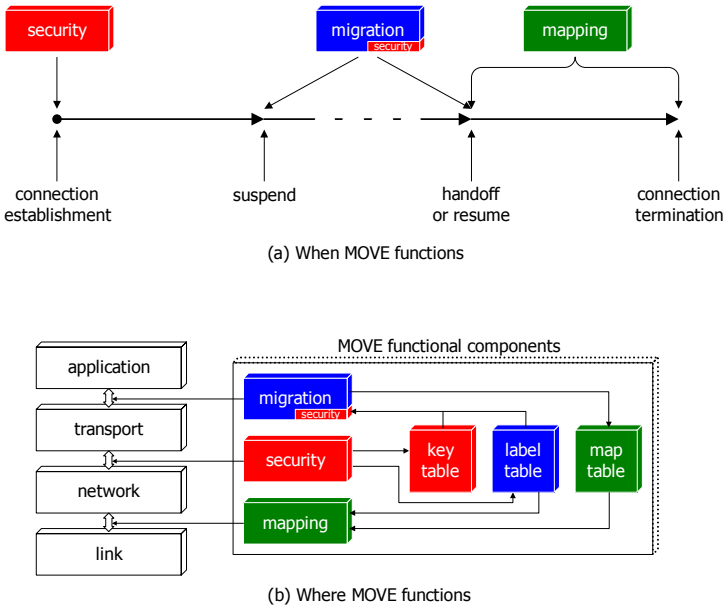


Figure 5-1. MOVE functional design overview

man public key and connection label at connection establishment time, which are recorded in the key table and label table. The other part of the security module, which is part of the resides inside the migration module at the boundary of the application layer and transport protocol layer, consults the key table and label table to compute and verify HMAC for the migration module at connection migration time.

The migration module implements the H2O signaling protocol for connection migration either through handoff or through suspension/resumption. It updates the map table for a migrated connection after the H2O signaling protocol messages are authenticated.

Once a connection migrates, the mapping module, which resides at the boundary of the network and data link protocol layers, uses the map table to perform the

namespace mapping through address translation and interface redirection for the rest of the connection's lifetime.

In addition, there is a common functional requirement for all three modules, which is not shown in Figure 5-1. As we pointed out in earlier sections, the DH public key and connection label are exchanged by piggybacking them onto the first a few packets exchanged between the two machines; the H2O signaling protocol uses in-band data packets to carry its protocol messages; and finally, for a migrated connection, a connection label is carried along with each packet. All of these require carrying certain information in the packet header. And since MOVE is transport protocol independent, this must also be done in a transport protocol independent fashion. There are different ways this can be accomplished. For example, one way is to use an IP option. The processing of IP options is well-defined by standards in routers and end hosts [33][107]. However, some routers do not conform to standards and may drop packets with (unknown) MOVE IP options. Another way is to use encapsulation, such as GRE (Generic Routing Encapsulation) or IPIP (IP in IP). The drawback with encapsulation is that it has slightly higher packet processing overhead due to the encapsulation and decapsulation. In our prototype MOVE implementation, we have chosen to use the IP option approach since it's simple and easy to implement.

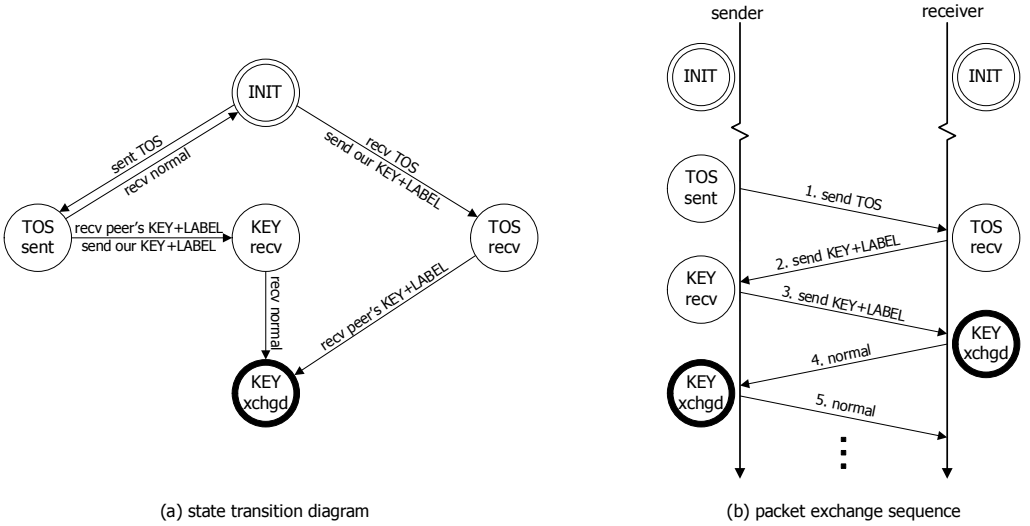
One potential problem with carrying the handoff protocol messages in the packet header is that a packet of (or very close to) size MTU (Maximum Transmission Unit) will be fragmented due to the increased header size. For connection-less

transport protocols such as UDP, this is generally not a problem since the protocol doesn't buffer packets and our experience shows that applications rarely send MTU sized UDP packets constantly since applications have no concept of MTU. For connection-oriented transport protocols such as TCP, this can be a problem since TCP buffers application data and attempts to stream packets to fill up the MTU. As a result, once a connection is migrated and a connection label is attached to the packet header, every MTU-sized packet may have to be re-fragmented, which can cause serious performance degradation. Fortunately, this problem can be solved relatively easily by properly reducing the MTU of the VNIC that the migrated connection is bound to. For example, before migration the MTU of the VNIC is the same as the NIC, i.e., 1500 bytes for ethernet; after migration the MTU of the VNIC is reduced to 1500-8=1492 bytes to account for the 8 bytes of connection label IP option. This will allow streaming transport protocols such as TCP to properly adjust their MSS (Maximum Segment Size) when building outgoing packets to avoid fragmentation. Note that right after the migration, there may be a few outgoing packets in the sending queue which are built using the original MTU size of 1500 bytes. These packets must be re-fragmented using the new MTU size of 1492 bytes. However, once these packets are cleared, all subsequent packets will be built with the proper MSS to avoid fragmentation.

5.2 Security Module

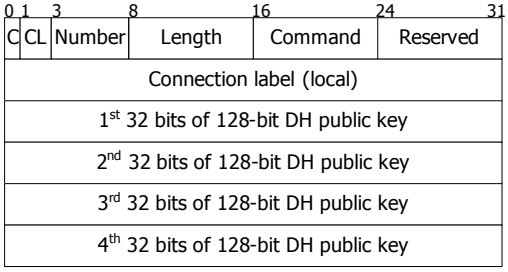
Because handoff or suspension/resumption can occur at any time during the lifetime of a connection, the security key necessary for protecting the H2O signaling

protocol messages and the connection label necessary for setting up proper namespace mapping must be in place right from the start of the connection. The DH public key and connection label are exchanged with a simple Finite State Machine (FSM) shown in Figure 5-2a. Figure 5-2b shows the sequence of packets exchanged between the two machines and Figure 5-2c shows the IP option format used to carry the key and label.



(a) state transition diagram

(b) packet exchange sequence



(c) IP option format

Figure 5-2. Security key and connection label exchange

1. The sender starts by sending a packet (packet #1) that has one unused bit in the TOS byte of the IP header set and goes into the "TOS sent" state. The bit used by MOVE is the 5th bit (from the most significant bit, numbered as 0), which has a meaning of "maximize reliability" by the original TOS definition [28]. However, currently the diffserv [97] working group has redefined the TOS byte in the IP header and the 5th bit is unused. The reason for this special TOS packet is to probe whether MOVE is present on the peer in order to interoperate with machines that do not have MOVE installed. Remember that the security key and connection label have to be carried either inside an IP option or an encapsulated header. In the case of using an IP option, this special packet is not really necessary and the sender can put the key and label in the IP option of the very first packet it sends. Because if the receiver is not MOVE-enabled, the unknown IP option will just be ignored. However, in the case of using an encapsulated header, the sender cannot just blindly send the key and label in the very first packet. Since unless the receiver is MOVE-enabled, it wouldn't know how to decapsulate the packet and will simply drop it.

2. When the receiver receives the special TOS packet, if it's not MOVE-enabled, it will ignore the TOS bit and reply with a normal packet. When the sender sees the normal packet, it knows that the receiver is not MOVE-enabled and will go back to normal state and no further action will be taken by either side. Otherwise, the receiver will respond by sending its key and label (packet #2) and goes into the "TOS recv" state. In this state,

the receiver may continue to see the special TOS packet from the sender because the packet containing its key and label may be lost. Or the sender may have received the receiver's key and label but the packet containing the sender's key and label (packet #3) may be lost. Therefore, the receiver continues to send it key and label until it can transit into the next state, "KEY xchgd", which can only happen when it has received sender's key and label.

3. When the sender receives the packet carrying the receiver's key and label, it knows the peer is MOVE-enabled. Therefore, it saves the receiver's key and label, responds by sending its own key and label (packet #3), and goes into the "KEY recv" state. Because the packet carrying sender's key and label can be lost, the sender continues to send it's key and label until it knows that the receiver has received its key and label, which is indicated by a normal packet (packet #4) from the receiver.

4. When the receiver receives the sender's key and label, it knows the peer has received its key and label; since otherwise it would be getting more special TOS packets from the sender. The receiver concludes its part of the exchange state transition by saving the sender's key and label and goes into the "KEY xchgd" state. From now on, all packets sent by the receiver will be normal packets.

5. When the sender receives a normal packet from the receiver while in "KEY recv" state, it knows the receiver has received its key and label; since otherwise it would be getting more packets carrying the receiver's key and

label. The sender now concludes its part of the exchange state transition by going into the "KEY xchgd" state. From now on, all packets sent by the sender will be normal packets.

It is interesting to note that the packet sequence for the key and label exchange can be mapped directly to existing transport protocols, which means that the exchange can be performed in-band through piggybacking rather than use a separate control connection. For example, if the transport protocol in use is TCP, packet #1, #2, and #3 in the packet exchange sequence would be mapped directly to and piggybacked on the SYN, SYN, and SYN-ACK packets of the 3-way handshake; if the transport protocol in use is UDP, packet #1, #2, and #3 would then be mapped to and piggybacked on the first three data packets exchanged between the two machines. Note that for UDP, the traffic may be unidirectional, i.e., only the sender sends packets to the receiver and the receiver never sends any packets to the sender. In this case, packet #2 would be generated by the security module and dropped by the sender. The use of in-band rather than out-of-band key and label exchange results in very low overhead for connection establishment, which is shown in Chapter 6.

The definition of the fields in the IP option, which is duplicated in Figure 5-3 from Figure 5-2(c), is as follows:

- *C*: 1 bit, copy flag, set to 0: do not copy into fragments
- *CL*: 2 bits, class, set to 3: reserved option
- *Number*: 5 bits, set to 31: option number for MOVE

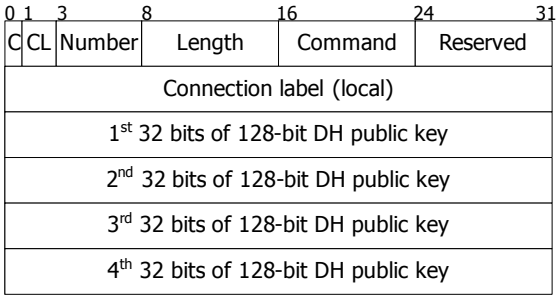


Figure 5-3. DH public key and label exchange IP option format

- *Length*: 8 bits, total length of the option in bytes
- *Command*: 8 bits, protocol message type, set to IPOPT_MOVE_DHKEYEX
- *Reserved*: 8 bits, reserved for future use
- *Connection label*: 32 bits, the connection label chosen by the machine for its incoming messages, recall Section 2.3.3 in Chapter 2
- *128-bit DH public key*: 128 bits, Diffie-Hellman public key, recall Section 3.2.4.2 in Chapter 3

Note that the layout of the first two bytes, i.e., the *C*, *CL*, *Number*, and *Length* fields, is standard for all IP options.

5.3 Migration Module

A connection can be migrated either through handoff or through suspension/ resumption. Also one can either migrate individual connections through a process migration mechanism, or migrate an entire machine by simply unplug/plug the network cable or by suspending/resuming the machine. When a connection is

being migrated by a process migration mechanism, it's the responsibility of the process migration mechanism to inform MOVE's migration module about the handoff or suspension/resumption events. When an entire machine is migrated, the migration module can receive the handoff and suspension/resumption events through various system services. For example, the migration module registers with the physical NIC device driver to receive interface up and down events. By comparing the new IP address assigned to the NIC with its previous IP address, the migration module can infer if a layer 3 handoff has occurred. The migration module also registers with the system's power management services such as APM (Advanced Power Management) or ACPI (Advanced Configuration and Power Interface) to receive machine suspension/resumption events. In any case, when these events occur, the migration module sends the peer machine H2O signaling protocol messages as appropriate. On the peer (stationary) machine, the migration module authenticates incoming H2O signaling protocol messages and takes appropriate actions. For example, when a HANDOFF messages is received, the migration module updates the map table so the mapping module can perform the necessary address translation and interface redirection functions; when a SUSPEND message is received, the migration module will block the owner process of the connection from sending more packets until the connection is resumed by a HANDOFF message.

5.3.1 Handoff process

Similar to the security module, the migration module also performs its functions according to a simple FSM, which records the current state of a connection and the

actions need to be taken when a H2O signaling protocol message is received. Figure 5-4 shows the FSM for connection migration through handoff. Also shown in the figure are the packet exchange sequence and the IP option format used for the protocol message.

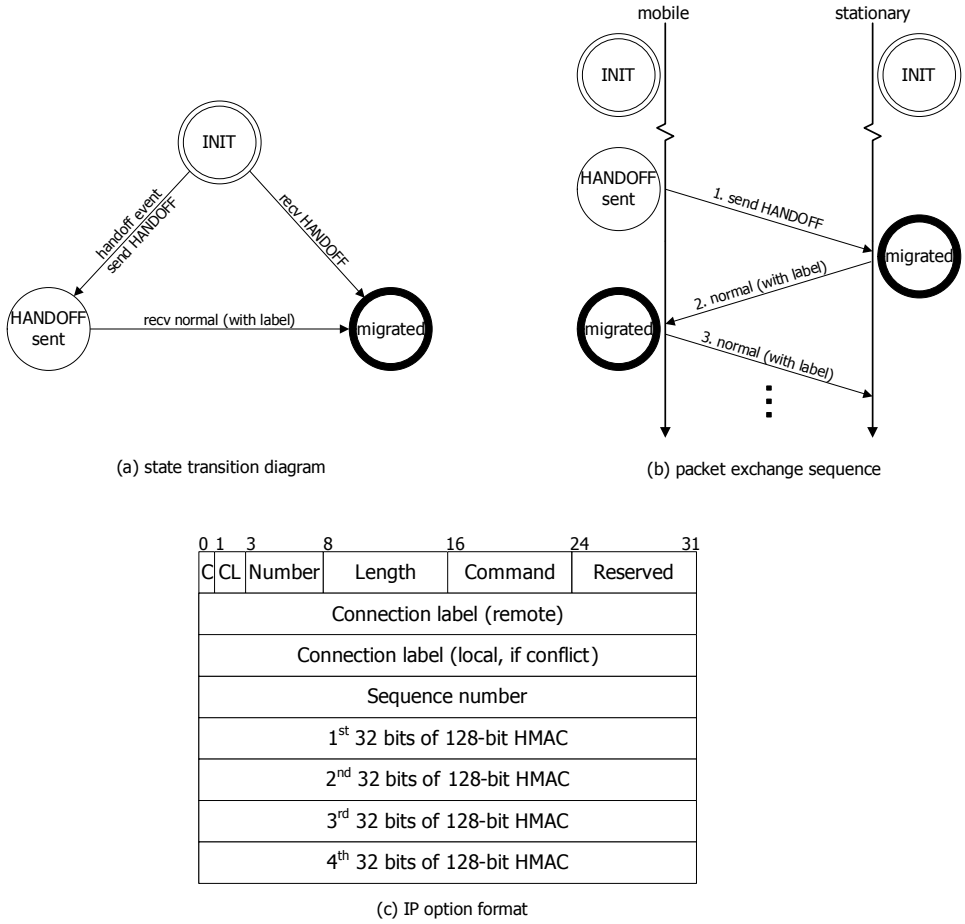


Figure 5-4. Handoff process FSM and IP option format

When the migration module on the mobile machine receives or detects a layer 3 handoff event, it updates the mapping table, sends a HANDOFF protocol message to the stationary machine, and goes into "HANDOFF sent" state. In this state, the migration module continues to send the HANDOFF protocol message because the

protocol message may be lost. When the HANDOFF protocol message reaches the stationary machine, the migration module authenticates the protocol message using the HMAC carried inside the message, updates the mapping table, and completes its handoff process by going into the "migrated" state. Now all the packets from the stationary machine can be mapped properly and sent to the new location of the mobile machine. When the mobile machine see traffic coming from the stationary machine, it knows that the HANDOFF protocol message has reached the stationary machine. It therefore stops sending the HANDOFF protocol message and concludes its handoff process by going into the "migrated" state.

The format of the IP option carrying the HANDOFF protocol message is shown in Figure 5-4c. Apart from the standard fields, other relevant fields are:

- *Command*: set to IPOPT_MOVE_HANDOFF
- *Connection label (remote)*: peer's connection label learned at the connection establishment time
- *Connection label (local, if conflict)*: new connection label for the connection if a conflict is detected on the local end, recall Section 2.3.3 in Chapter 2
- *Sequence number*: 32-bit monotonically increasing anti-replay sequence number
- *128-bit HMAC*: computed over the protocol message up to the *Sequence number* field using the shared secret key derived from the local private key and the remote public key.

5.3.2 Suspension and resumption process

The suspension/resumption process is slightly more involving, requiring two more states to handle the suspension part. The resumption part is just the same as the handoff process, as show in Figure 5-5. The format of the IP option carrying the SUSPEND protocol message, which is omitted from the figure, is exactly the same as that of the IP option carrying the HANDOFF protocol message except the *Command* field is set to IPOPT_MOVE_SUSPEND.

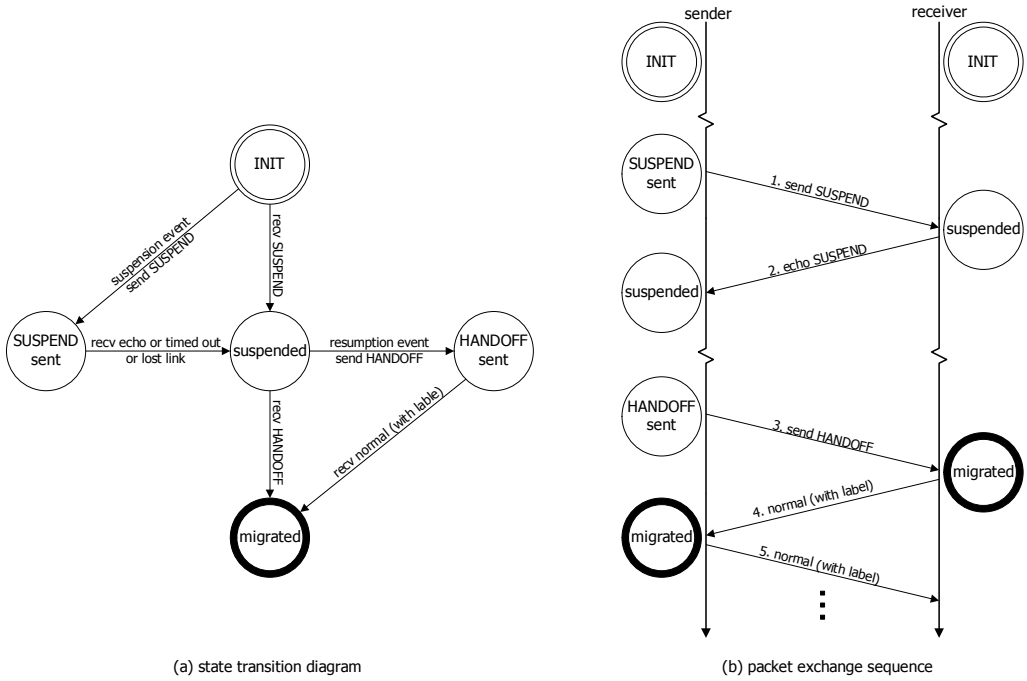


Figure 5-5. Suspension and resumption process FSM

When the migration module on the mobile machine receives or detects a suspension event, it sends a SUSPEND protocol message to the stationary machine and goes into the "SUSPEND sent" state. In this state, the migration module continues to send the SUSPEND protocol message until one of the following three events

occurs and then moves into the “suspended” state:

- received an echo of the SUSPEND protocol message. This indicates the stationary machine has received the SUSPEND protocol message and moved into the “suspended” state.
- timed out. The migration module on the mobile machine never received an echo of the SUSPEND protocol message for a certain period of time. Either the protocol message may be lost, or the echo of the protocol message may be lost. So the stationary machine may or may not be in “suspended” state.
- lost layer 2 link connectivity. This case does not actually have much to do with the suspension process. It actually happens with the handoff process when there is advance notice (but no simultaneous connectivity, recall Section 3.3.2 in Chapter 3). The advance notice is treated the same as a suspension event by the migration module. Similar to the timed out case, if no echo of the SUSPEND protocol message is seen before the mobile machine loses its layer 2 link connectivity, the stationary machine may or may not be in the “suspended” state.

When the stationary machine receives the SUSPEND protocol message, it blocks the owner process of the connection from sending more packets (receiving is still allowed), echoes the protocol message, and goes into the “suspended” state. If the stationary machine never receives the SUSPEND protocol message, the suspension/resumption process basically degenerates into a (prolonged) handoff process when the stationary machine receives a HANDOFF protocol message. It is not dif-

ficult to see that the resumption process (with a successful suspension process) is basically the same as the handoff process except that the mobile and stationary machines start with the "suspended" state instead of the "INIT" state.

5.4 Mapping Module

The mapping module is the simplest among the three modules. And it is rightly so because its functions, i.e., address translation and interface redirection, must be performed for every packet for the rest of the lifetime of a connection once the connection migrates. Every packet of a migrated connection carries a connection label in the IP option with the following format:

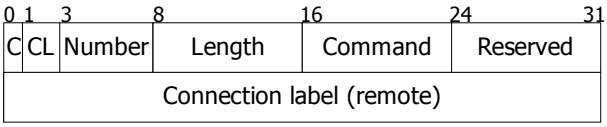


Figure 5-6. Connection label IP option format

The *command* field is set to IPOPT_MOVE_VIRTUAL and the *connection label* field is the 32-bit label obtained from the peer at connection establishment time. All the mapping module needs to do is to use the connection label as a hash key to look up the map table associated with the connection and binds the connection to a dynamically created VNIC so that all traffic of the connection will now pass through the VNIC. Inside the VNIC, address translation is performed to map between the CELL namespace and the physical namespace according to the map table. For example, for outgoing traffic, the virtual IP address in the CELL namespace is translated to the physical IP address in the physical namespace

before a packet is passed on to the physical NIC. For incoming traffic, the reverse translation is done before a packet is passed up to the higher layer. In Chapter 6, we will present measurement results that demonstrate these operations are very simple and incur very low overhead on the traffic of migrated connections.

5.5 System Call Interception

MOVE virtualization and privatization are implemented below the transport layer. MOVE therefore is transparent to transport-and-above layers and do not generally interact with the application layer directly. There is only one exception: MOVE intercepts the `getsockname`/`getpeername` socket system calls in order to support location-aware applications.

The `getsockname`/`getpeername` socket system calls are used by applications to query the current physical IP addresses of the local/peer host, which is obtained from the connection states maintained by the transport protocols (note that even with connection-less transport protocols, the minimal "connection" states, i.e., the {*source IP address*:*source port number*; *destination IP address*:*destination port number*} tuple, provide the support for these calls). Since MOVE virtualizes the transport layer, the IP addresses returned by `getsockname`/`getpeername` will be the virtual ones. However, recall in Section 2.4.3 in Chapter 2 that MOVE's lazy assignment by default exposes the physical IP addresses of the current local/peer host in order to support location-aware applications and avoid unnecessary virtual-physical translations. Therefore, MOVE by default intercepts the `getsockname`/`getpeername` socket system calls and returns the physical IP addresses of the current local/peer

host instead. For legacy applications such as FTP where complete transparency of the migration is required, MOVE allows `getsockname`/`getpeername` to obtain the virtual IP addresses directly from the transport protocols. Choosing between one of the two behaviors is done on a per-application basis through the *proc* virtual file system [79].

5.6 Transparent SRV RR Lookup Support

While the focus of this thesis is not on locating mobile endpoints, we nevertheless have described a way for supporting host location with DDNS and service location with SRV RR in Section 2.4.1 in Chapter 2. Because existing network applications do not yet support SRV RR lookup, we have designed and implemented a mechanism for transparently supporting it without changing the applications. The mechanism is a simple socket library wrapper that intercepts the following socket related calls: `gethostbyname`/`getaddrinfo` (`getaddrinfo` is a new name resolver function that is supposed to supersede `gethostbyname`; but many applications still use `gethostbyname`) and `connect`. They work as follows:

- `gethostbyname`/`getaddrinfo`: our version of the functions simply call the original version but saves the reverse IP address to host name mapping, which is needed for constructing the SRV RR request. We could also use `gethostbyaddr` to obtain the mapping but that requires an additional trip to the DNS server.
- `connect`: when a `connect(IP_address, port_number)` is called, we lookup

the host name corresponding to the IP address using the information saved by our `gethostbyname/getaddrinfo`, we lookup the service name by calling `getservbyport(port_number)`, and we make an SRV RR lookup by calling `res_querydomain(_service._protocol.hostname)`. The SRV RR lookup will return the current host name (which may be different from the host-name we supplied) and port number where the service can be reached. We then call `gethostbyname(current_hostname)` to find out the IP address of the current host name. And finally, we call the original *connect* with `connect(current_IP_address, port_number)`.

5.7 Summary

In this chapter, we have demonstrated, with a prototype design and implementation of MOVE system, that the CELL abstraction, H2O protocol, and their supporting mechanisms lend themselves readily to efficient real world utilization. The key abstraction and simple mechanisms employed by MOVE are also the reason why MOVE can meet all (performance will be presented in the next chapter) the functional requirements of a mobile communication architecture we outline in the introduction of this thesis, which we repeat here:

- easy deployment: MOVE resides and functions completely within end machines without requiring any infrastructure support inside the network; MOVE does not require modification or recompilation of existing OSes and applications, is compatible and can interoperate with legacy OSes and applications that are not MOVE-enabled; MOVE does not presume or rely

on any particular transport protocol operational semantics.

- fine-grain and unlimited mobility: MOVE supports migration of a single connection, a group of connections, or all the connections of an entire host; either endpoint of a connection can migrate anywhere in the network.
- secure and flexible migration: MOVE provides, in the absence of other security mechanisms such as IPsec, a low overhead self-securing mechanism to protect its migration functions; connections can be migrated either through "on-line" natured handoff or through "off-line" natured suspension/resumption.

In the next chapter, we will complete our work by presenting a detailed evaluation of various performance measurements of our prototype MOVE system.

6 Performance Measurements

We have implemented a MOVE prototype on the LINUX x86 platform with 2.4 series kernel. The entire functions of MOVE are implemented as a kernel module that can be dynamically loaded into the running kernel at any time without kernel recompilation or rebooting. This chapter presents various performance measurements of our MOVE prototype. We present three main categories of tests: (1) handoff performance in Section 6.1; (2) scalability measurements in proxy-based environments in Section 6.2; and (3) connection virtualization and mapping overhead in Section 6.3. We also present mobile host and service location mechanism studies in Section 6.4.

The handoff performance tests in Section 6.1 are to demonstrate the viability of MOVE with a variety of applications, endpoint migration mechanisms, network connectivity configurations, and transport layer protocols. Specifically, these tests show that MOVE handoff:

- works with a variety of off-the-shelf applications unchanged, such as *mplayer/wget/lftp* clients and *apache/vsftpd/Darwin* servers, etc.
- is compatible with different types of endpoint migration mechanisms, such as moving a physical machine, a virtual machine, or a process.
- works with different types of networks such as 10/100/1000Mbits ethernet

and 11Mbits WiFi. It also works across these different networks.

- incurs minimal impact on the connection characteristics perceived by the transport protocols and applications with a variety of duration of disconnection times (DDT) relative to the RTT, such as DDT<<RTT, DDT≈RTT, and DDT>>RTT.
- performs very well under stress with increasing rate of handoff.
- is independent of and supports both connection-oriented (TCP) and connection-less (UDP) transport protocols.

The scalability tests in Section 6.2 show that MOVE does not adversely affect the scalability of the existing system, especially when the existing system has certain hot spots such as proxies. The virtualization and mapping overhead measurements in Section 6.3 show that MOVE adds very small network I/O performance overhead to the base system, regardless of whether the base system is low-end or high-end. Finally, Section 6.4 demonstrates the feasibility of using DDNS as the host and service location mechanism.

6.1 Handoff Performance

The extensive tests of MOVE handoff performance are broken down to five categories at the top level. The first three correspond to the three different types of endpoints that are migrated: (1) a client physical machine, (2) a client VMware virtual machine, and (3) a server process. Within each category, a variety of network configurations and a variety of DDTs relative to the RTT are tested. Table 6-1 shows a

summary of the test cases and the sections in which they are described. We choose different DDT relative to the RTT because the operational semantics of reliable transport protocols such as TCP are heavily dependent upon the RTT. What matters most is not the absolute length of the DDT, but rather the relation of the DDT to the RTT. For example, as long as the DDT is sufficiently short and does not cause TCP to timeout, the exact length of the DDT makes little difference to TCP's behavior.

	within WAN	**within LAN**	**WAN to LAN**	**LAN to WAN**
Client Machine	DDT≈10ms, 200ms, 4s ***mplayer*** - *apache* (Section 6.1.1.1)	DDT≈30ms, 3s ***mplayer*** - *apache* (Section 6.1.1.2)	DDT≈100ms, 2s ***wget*** - *apache* (Section 6.1.1.3)	DDT≈100ms, 2s ***wget*** - *apache* (Section 6.1.1.4)
Client VM	**-**	**-**	DDT≈8s ***lftp*** - *vsftpd* (Section 6.1.2.1)	DDT≈11s ***lftp*** - *vsftpd* (Section 6.1.2.2)
Server Process	DDT≈2s *mplayer* - ***apache*** (Section 6.1.3.1)	DDT≈2s *mplayer* - ***apache*** (Section 6.1.3.2)	**-**	**-**

Table 6-1. Handoff performance test cases

The fourth in Section 6.1.4 is a stress test in which a connection is "ping-pong" migrated between two interfaces with a varying interval between each migration. The fifth in Section 6.1.5 is a test for migrating connection-less transport protocols in which a RTSP streaming video session using RTP over UDP is migrated. Finally in Section 6.1.6, we also present a list of popular off-the-shelf network applications that we've tested to work with MOVE right out of the box.

6.1.1 Client handoff with machine migration

The client handoff performance is measured between an IBM T22 ThinkPad laptop

computer with 1GHz Pentium III CPU, 512MB RAM, 100Mbits Intel Pro/100 SP ethernet NIC, and 11Mbits Orinoco Gold WiFi PCCard, and an IBM 4500R rack mounted server computer, with dual 933MHz Pentium III CPU, 512MB RAM, and 100Mbits AMD LANCE ethernet NIC. All machines are running LINUX kernel version 2.4.20. We study a variety of network connectivity scenarios illustrated in Figure 6-1, specifically:

- handoff on a campus WAN with wired connection
- handoff on an office LAN with wired connection
- handoff between a campus WAN with WiFi wireless connection and an office LAN with wired connection.

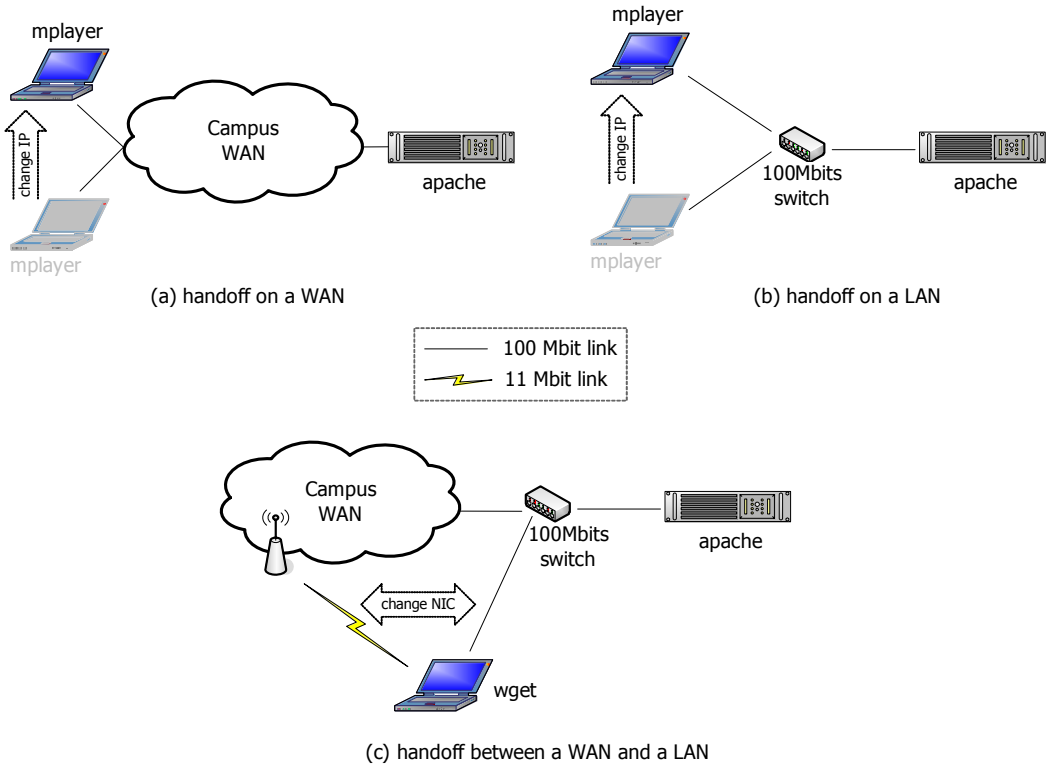


Figure 6-1. Client handoff with machine migration testbed

For the first two cases, a RealVideo 8 encoded media clip is streamed through HTTP from the server running *apache* [1] version 2.0.40 to the laptop running *mplayer* [14] version 1.0pre4; the server and the laptop are connected through their 100Mbits ethernet NIC. The handoff is effected by disconnecting the laptop's 100Mbits ethernet NIC after roughly 30 seconds of playback, waiting for a variable amount of time, and then reconnecting it with another address and continuing with roughly another 30 seconds of playback.

For the third case, the laptop is connected to the server from both its 100Mbits ethernet NIC and its 11Mbits WiFi PCCard. The 100Mbits ethernet NIC is connected directly to the office LAN where the server is connected; and the 11Mbits WiFi PCCard is connected to the campus WAN through which the server is also reachable. The laptop uses *wget* [8] version 1.8.2 to fetch a file of roughly 50MB size through HTTP from the server. The connection is first made through one of the two interfaces on the laptop and then handed off to the other during the download. The handoff off is effected by bring down the first interface through which the connection is established and then bring up the other interface. For the two cases of switching from WAN to LAN and *vise versa*, the handoff point is where roughly 80% of the file is going through the LAN and 20% of the file is going through the WAN.

For all three cases, the playback or download network session is captured on the server using *tcpdump* [19] version 3.7.2 and analyzed using *tcptrace* [20] version 6.6.1.

6.1.1.1 Handoff on a WAN, DDT≈10ms, 200ms, and 4s

The WAN used in the test is a campus network connecting offices and dormitories with no artificial elements involved. We study three cases of different DDTs: DDT≈10ms (<<RTT), DDT≈200ms (≈RTT), and DDT≈4s (>>RTT).

We first present the case when DDT≈10ms and show the TCP sequence trace and throughput of the entire playback session in Figure 6-2 and Figure 6-3. The TCP sequence trace graph (Figure 6-2) serves as a visual presentation that TCP is able to recover and playback at the same throughput after the handoff as that before the handoff. This is indicated by the slope of the sequence trace being unchanged before and after the handoff. The TCP throughput graph (Figure 6-3) is to quantitatively verify the TCP throughput before and after the handoff. This is done by using *tcptrace* to compute the TCP throughput from the *tcpdump* data. Note that the throughput before and after handoff are computed independently by *tcptrace* as two separate connections due to the change of the client IP address. We present the throughput results for both averaging over last 20 packets (the *tcptrace* default of averaging 10 packets results in too much fluctuation to be useful) and averaging over all packets seen so far. A small box in the sequence trace graph (Figure 6-2) indicates where the handoff takes place and we will be looking into the events inside the box next. Throughout the rest of Section 6.1, for each of the handoff performance measurements, this is how we will be presenting our results: we first show the TCP sequence trace and throughput for the entire network session; we then zoom into the small box in the sequence trace graph and explain in detail the handoff events.

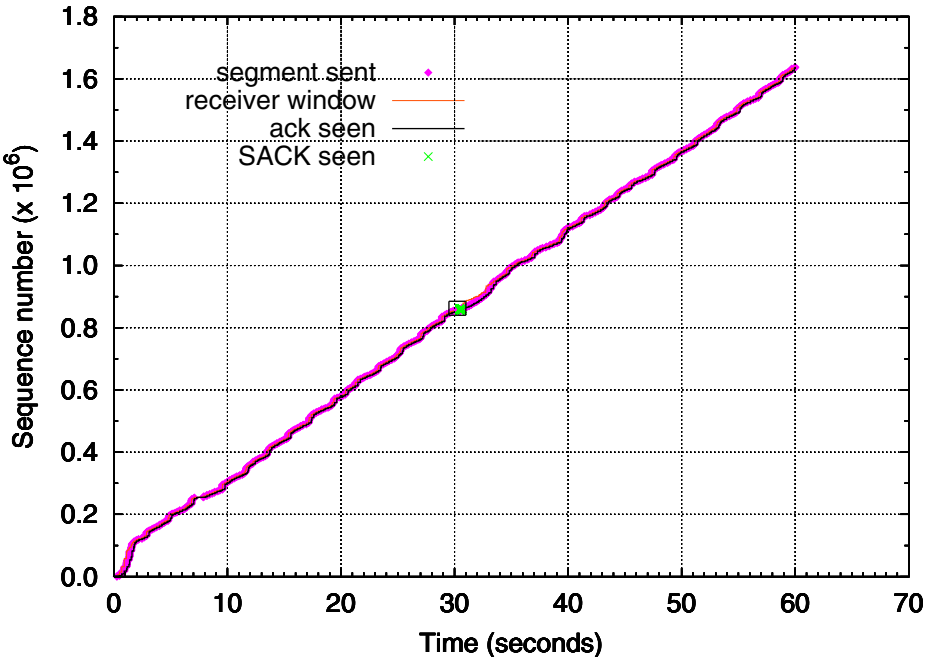


Figure 6-2. Entire playback TCP sequence trace, DDT≈10ms << RTT≈230ms

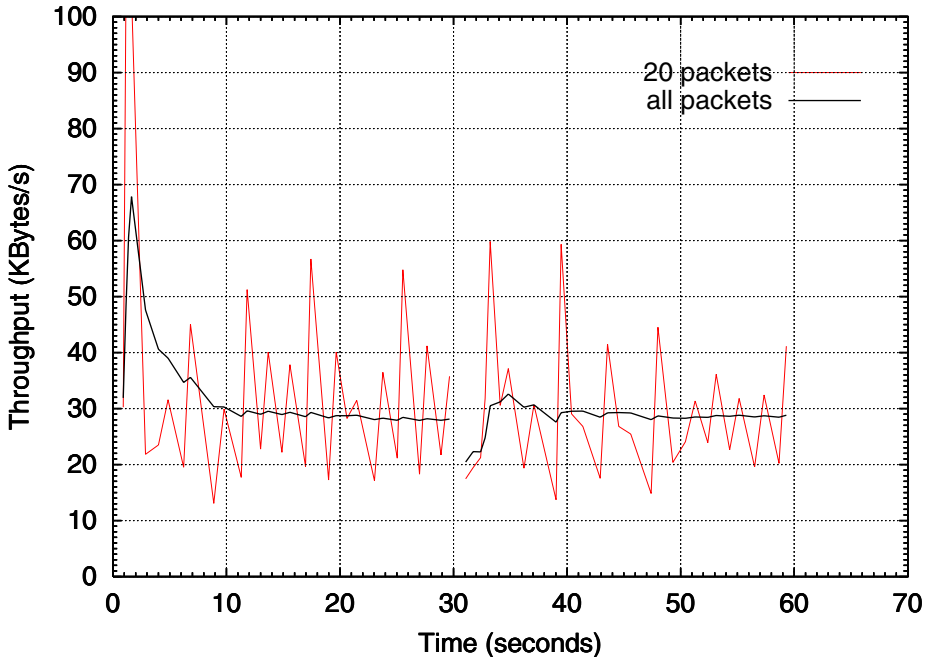


Figure 6-3. Entire playback TCP throughput, DDT≈10ms << RTT≈230ms

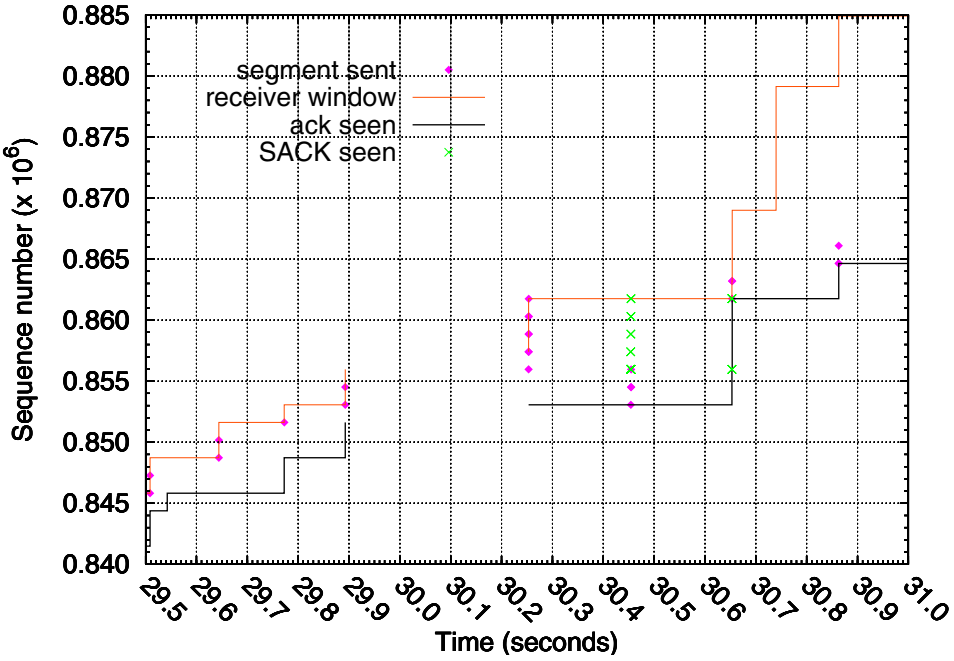


Figure 6-4. Zoomed TCP sequence trace, DDT≈10ms << RTT≈230ms

Figure 6-4 shows the zoomed TCP sequence trace graph for the case when DDT≈10ms. While not shown in the figure, we used *tcptrace* to compute the average RTT and its standard deviation observed from the captured packet trace, which are 230.1ms and 27.7ms respectively. At 29.89s, the server sees the last ack from the laptop before it is disconnected. At 30.25s, the first ack from the laptop, which carries H2O HANDOFF message, arrives after it is reconnected. The lapse of 360ms may seem a little strange at first; since we only disconnected for 10ms and the single trip for H2O HANDOFF message takes about 115ms so the ack should have arrived at roughly 125ms after 29.89s, i.e., 30.015s. This is because disconnection and reconnection cannot happen instantaneously. There is a minimum delay of about 200ms before a reconnected interface is fully operational again; this includes the need to reinstate the default gateway route. This is also evident in the

other two cases. But this does not affect our discussion. The point of this case is to show that if the handoff at the laptop happens very quickly, H2O's HANDOFF message can arrive at the server soon enough to prevent TCP from timeout. The interesting points to note in Figure 6-4 are:

1. When the first ack after reconnection arrives at 30.25s, it carries a higher seq number, roughly 0.853, than that carried by the last ack before disconnection at 29.89s, roughly 0.852. This means that right before disconnection, the laptop has received the packets between seq number 0.852 and 0.853 but didn't have a chance to ack them.
2. The ack at 30.25 has a lower seq number than the last packet sent by the server, which has a seq number roughly 0.854. This means that packets between seq number 0.853 and 0.854 are lost during the handoff.
3. The ack at 30.25s also advertises a bigger receiver window than the ack at 29.9s, which means that some packets received on the laptop right before disconnection has been delivered to the application during the disconnection.
4. After receiving the ack at 30.25, TCP on the server immediately sends several packets, between 0.856 and 0.862, to fill up the receiver window. This indicates that TCP on the server is still going at full throttle and never perceived the handoff.

We can see that at 30.45s, the lost packets between 0.853 and 0.854 are being retransmitted, while new packets sent at 30.25s between 0.856 and 0.862 are being

SACKed. At 30.65s, the lost and retransmitted packets are acked, which fill the gap below the SACKed packets; therefore, the ack jumps from 0.853 to 0.862, which indicates the conclusion of the recovery period.

We next look at the case when DDT≈200ms. Similar to the case when DDT≈10ms, we first show the TCP sequence trace and throughput for the entire playback session in Figure 6-5 and Figure 6-6. We then zoom into the small box in Figure 6-5 to see the details of the handoff events, which are shown in Figure 6-7.

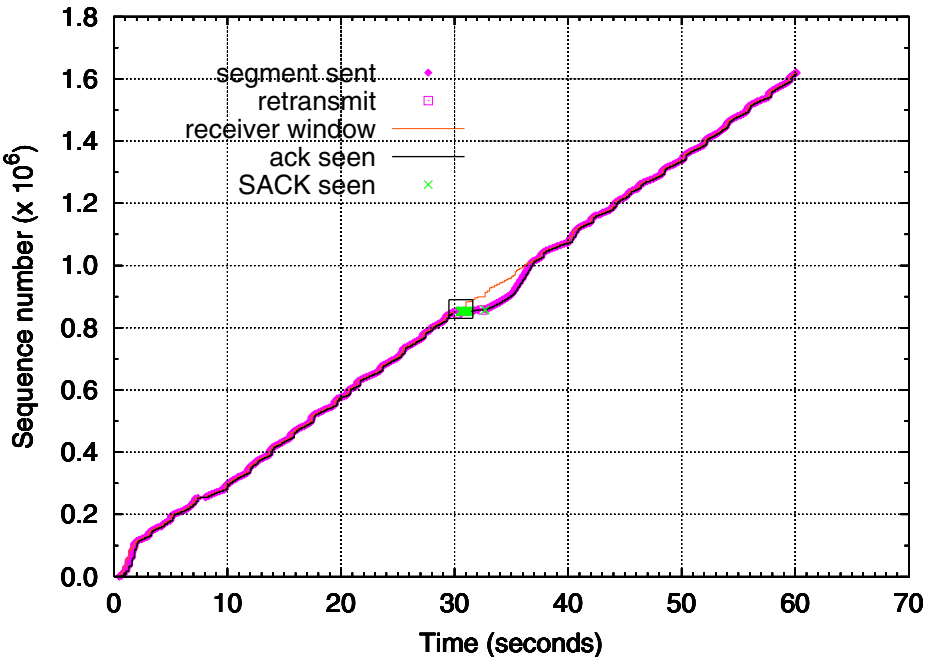


Figure 6-5. Entire playback TCP sequence trace, DDT≈200ms ≈ RTT≈235ms

For this test, *tcptrace* computes the average RTT and its standard deviation as 234.7ms and 28.7ms respectively. The last ack from the laptop before disconnection arrives at the server at 29.81s. The first ack carrying H2O HANDOFF message after reconnection arrives at 30.32s. The interesting points to note in Figure 6-7 are:

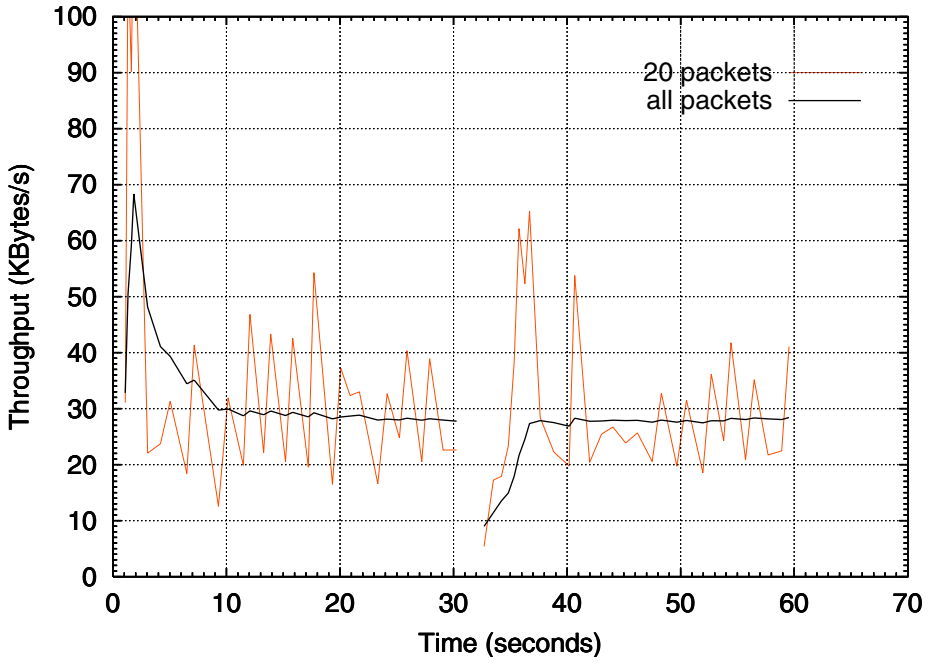


Figure 6-6. Entire playback TCP throughput, DDT≈200ms ≈ RTT≈235ms

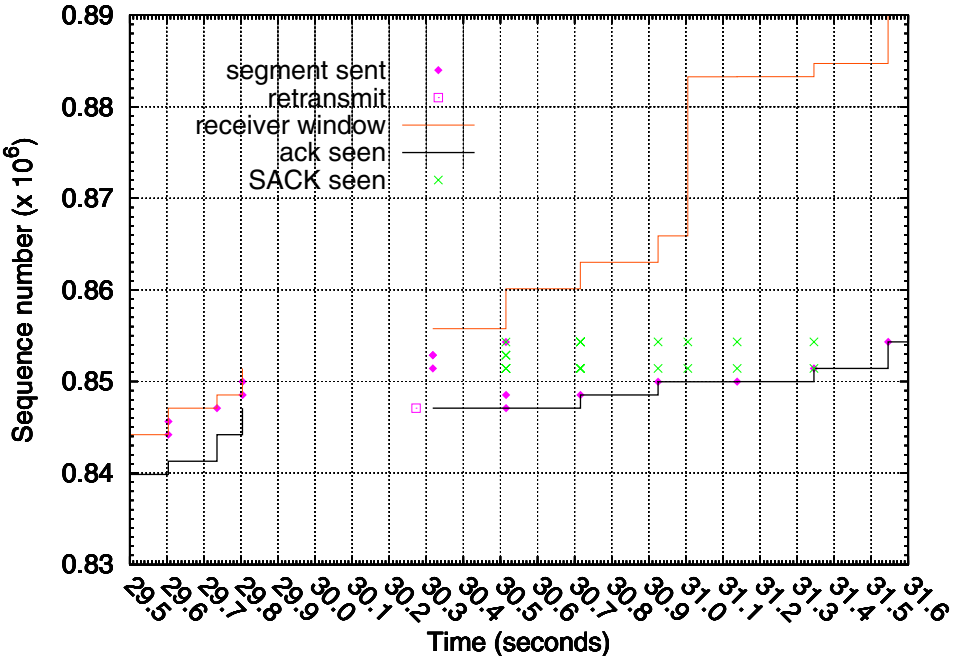


Figure 6-7. Zoomed TCP sequence trace, DDT≈200ms ≈ RTT≈235ms

1. A timeout and retransmission occurs at 30.27s for packet 0.847, which was originally sent at 29.73s.
2. Due to the delay of interface becoming full operational after reconnection, the first H2O HANDOFF message after reconnection arrives at 200 (DDT)+200 (reconnection delay)+117 (single trip)=517ms after 29.81s, i.e., 30.327s, rather than 200 (DDT)+117 (single trip)=305ms after 29.81s, i.e., 30.115s.
3. Different from previous case, the ack at 30.32s is not carrying a higher seq number than the ack at 29.81s. This means that all data received by the laptop right before disconnection are acked. But same as previous case, the ack at 30.32s advertises a bigger receiver window than the ack at 29.81s. This explains why, even though TCP on the server has timed out shortly before 30.32s and gone into slow start, it immediately starts transmission again and goes into recovery mode as soon as it receives the H2O HANDOFF message without waiting for another timeout. This also shows the advantage of an in-band signaling protocol.

We can see that, similar to the previous case, at 30.52s, lost packets between 0.847 and 0.85 are being retransmitted, while new packets sent at 30.32s between 0.8512 and 0.853 are being SACKed. In addition, at 30.52s, a new packet at 0.854 is sent. This packet, along with those sent at 30.32, are being SACKed throughout the retransmission period until 31.55s, at which point all lost packets are acked and the ack number jumps to 0.854.

We finally look at the case when DDT≈4s. We again first present the TCP sequence trace and throughput for the entire playback session in Figure 6-8 and Figure 6-9. We then present in Figure 6-10 the zoomed view of the small box shown in Figure 6-8. For this test, *tcptrace* computes the average RTT and its standard deviation as 230.8ms and 33.4ms respectively. This is a rather common TCP slow start recovery after several timeouts, where all data after disconnection are lost and retransmitted. Nevertheless, it's again interesting to note in Figure 6-10 that:

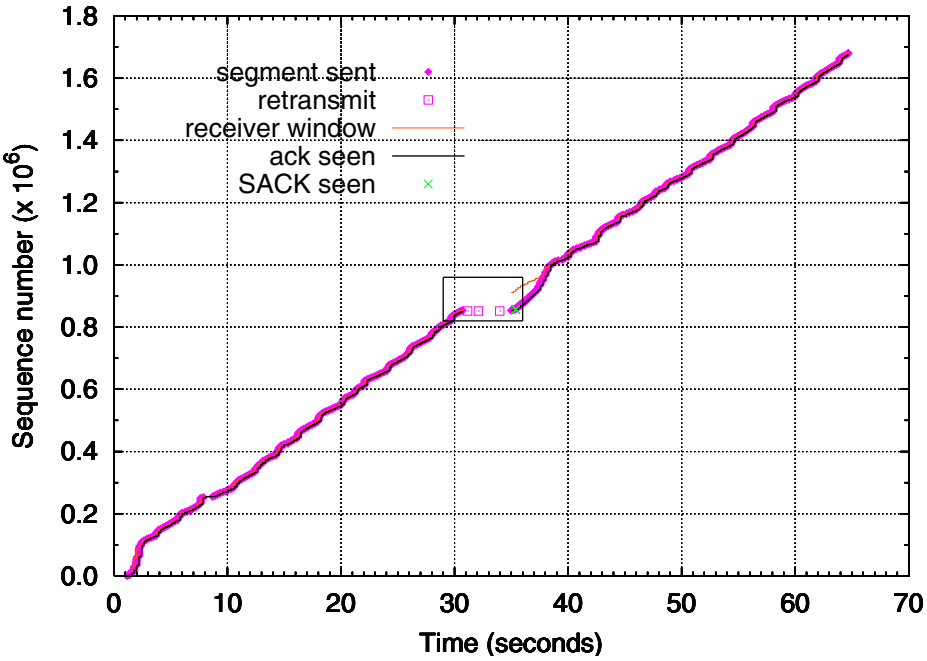


Figure 6-8. Entire playback TCP sequence trace, DDT≈4s >> RTT≈231ms

1. The first timeout occurs at around 31.2s, which is the retransmission of the packet 0.852 originally sent at around 30.55s.

2. The elapsed time between the first ack after reconnection at 35s and the last ack before disconnection at 30.7s is 4.3s, which roughly corresponds to 4s

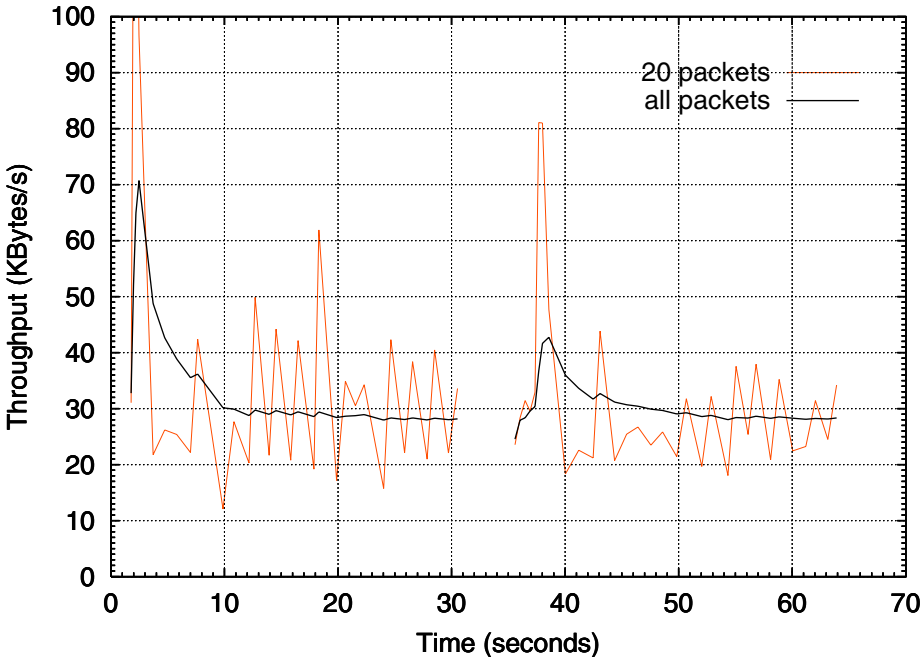


Figure 6-9. Entire playback TCP throughput, DDT≈4s >> RTT≈231ms

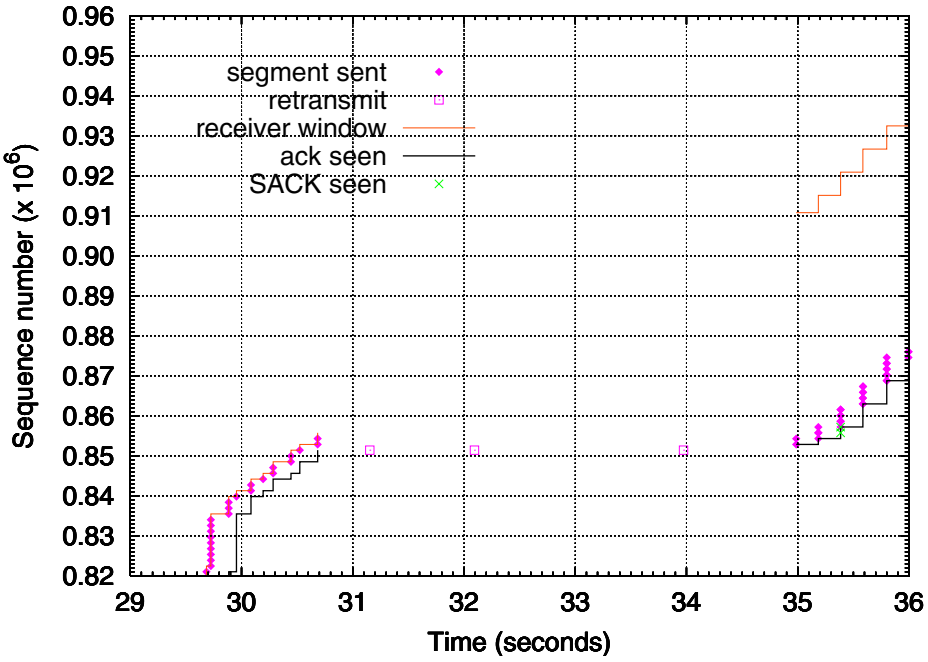


Figure 6-10. Zoomed TCP sequence trace, DDT≈4s >> RTT≈231ms

(DDT)+200ms (reconnection delay)+115ms (single trip)=4.315s.

3. The recovery process again starts immediately at 35s without waiting for further retransmission timeouts because the ack at 35s acks more data than the one at 30.7s. The 4th retransmission would have occurred roughly 8*600ms after the 3rd one, at 38.8s. Note that even with a big receiver window size at 35s, TCP does not attempt to fill up the window immediately since it's in slow start.

6.1.1.2 Handoff on a LAN, DDT≈30ms and 3s

In this test, the laptop computer and the server computer are connected directly through a 100Mbits switch as shown in Figure 6-1b. The RTT perceived by TCP is around 30ms with a standard deviation of about 10ms. Recall from Section 6.1.1.1 that there is a minimum delay of about 200ms before a reconnected interface is fully operational again. In this case, however, we were able to reduce the delay to about 90ms since the two computers are on the same subnet therefore there is no need to reinstate the default gateway route. So for handoff across a LAN, we can only test two cases: DDT≈30ms (≈ RTT), and DDT≈3s (>> RTT) because the RTT on a LAN is already very small.

We first present the TCP sequence trace and throughput of the entire playback session for the case when DDT≈30ms in Figure 6-11 and Figure 6-12. These graphs show that TCP is able to playback at the same throughput after the handoff as that before the handoff. We then zoom into the small boxes presented in Figure 6-11 to see the details of the handoff events, shown in Figure 6-13.

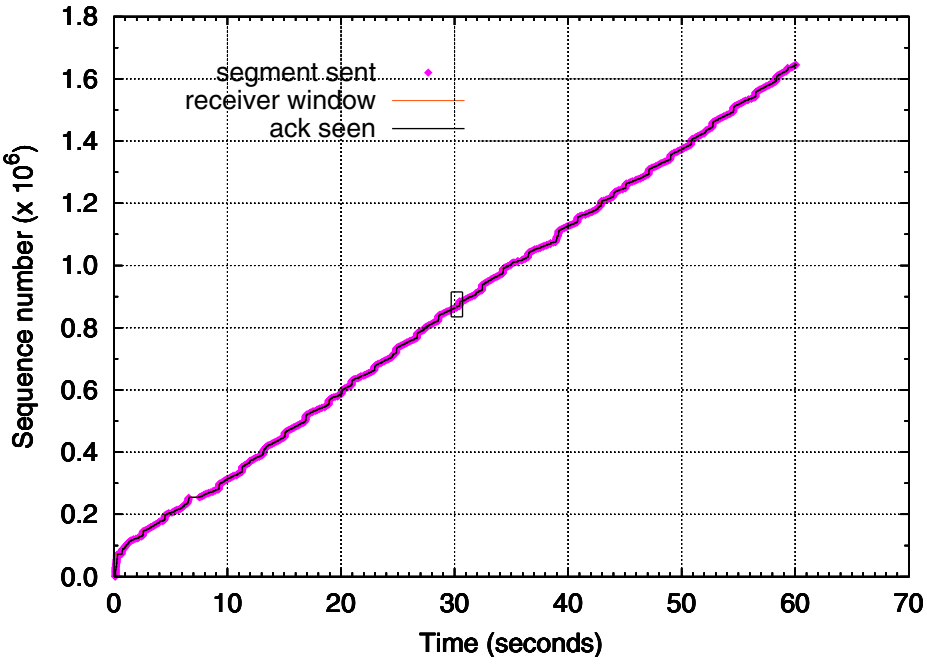


Figure 6-11. Entire playback TCP sequence trace, DDT≈30ms ≈ RTT≈33ms

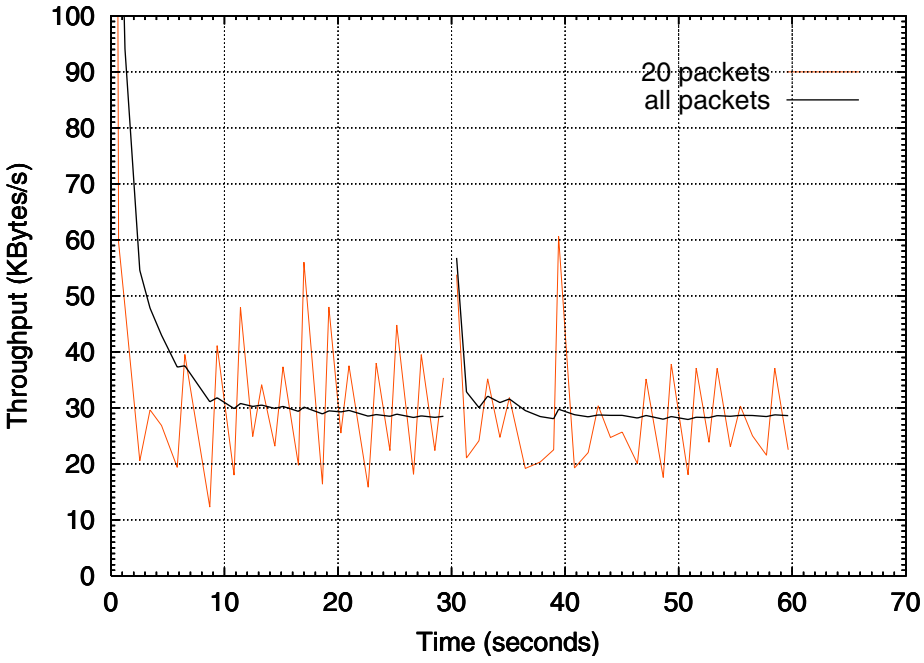


Figure 6-12. Entire playback TCP throughput, DDT≈30ms ≈ RTT≈33ms

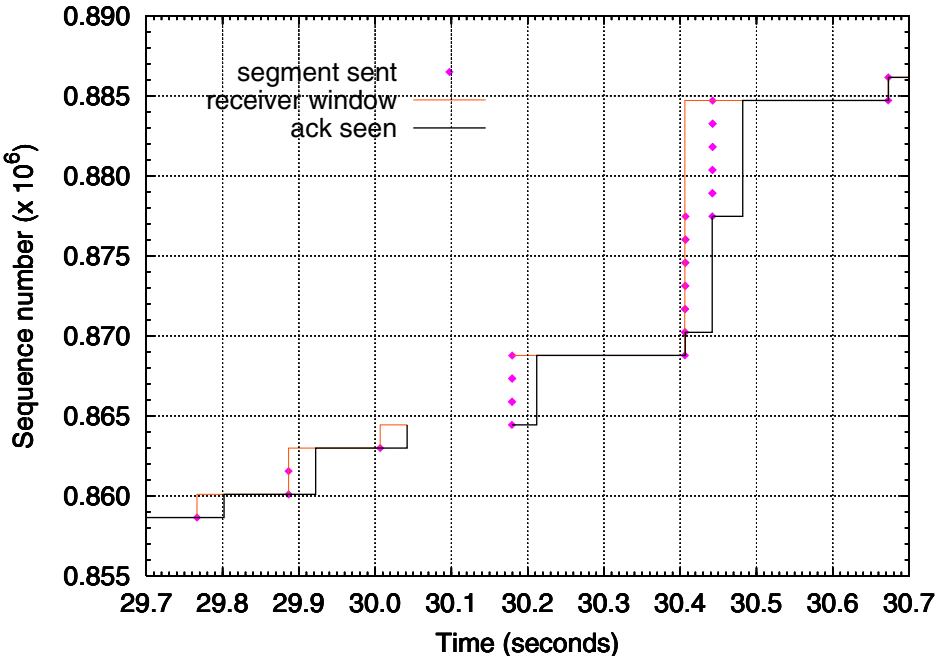


Figure 6-13. Zoomed TCP sequence trace, DDT≈30ms ≈ RTT≈33ms

As shown in Figure 6-13, the last ack from the laptop before disconnection arrives at the server at about 30.04s. The first ack carrying H2O HANDOFF message after reconnection arrives shortly after 30.18s. The interesting points to note in Figure 6-13 are:

1. The last ack before disconnection at 30.04s carries a zero window size. This means that TCP receiver queue on the laptop is full. This is because in the 100Mbits LAN packets arrive must faster than the rate at which they are played out by the application. As a result, during the handoff, no packets are sent by the server, i.e., no packets are lost.

2. The average RTT and its standard deviation D observed from the packet trace are 32.5ms and 11.4ms, respectively. The first ack after reconnection at

30.18s is 140ms after the last ack before disconnection, which roughly corresponds to 90 (reconnection delay)+30 (DDT)+16 (single trip)=136ms. Since TCP computes its RTO=RTT+4*D, this would have turned out to be 32.5+4*11.4=78ms. However, there is no retransmission at 30.04+0.078=30.118s. This is because LINUX has a minimum RTO=200ms.

3. The first ack after reconnection at 30.18s does not ack more data since no data are sent during the handoff and all data sent before the handoff have been acked. However, it carries a non-zero window therefore TCP on the server is able to continue sending immediately.

We next present the TCP sequence trace and throughput of the entire playback session for the case when DDT≈3s in Figure 6-14 and Figure 6-15. And we zoom into the small boxes presented in Figure 6-14 to see the details of the handoff events, which are shown in Figure 6-16. The last ack before disconnection arrives at about 30.1s, and the first ack after reconnection arrives roughly 3s later, at 33.1s. While this case looks similar to the case when DDT≈4s in the WAN test in Figure 6-10, there are a few interesting differences:

1. Even though the RTT and its standard deviation D are 30.7ms and 10.8ms respectively from the packet trace, which suggests the RTO=RTT+4*D=30.7+4*10.8=73.9ms, the first "retransmission" occurs at 30.3s, about 200ms after the last ack before disconnection at 30.1s. Again this is due to the minimum RTO of 200ms in LINUX.

2. The last ack before disconnection at 30.1s carries a zero window size,

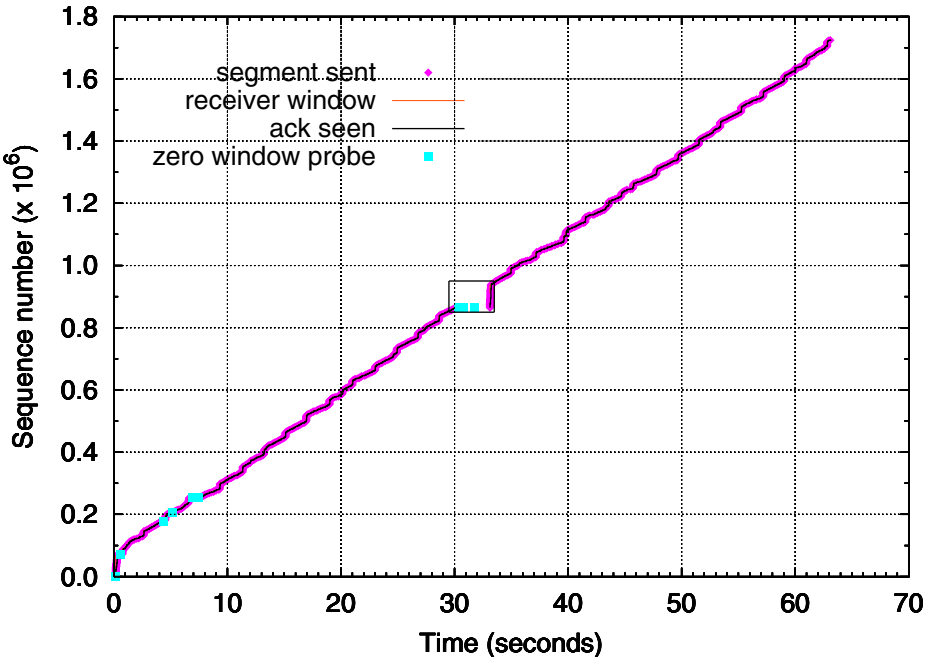


Figure 6-14. Entire playback TCP sequence trace, DDT≈3s >> RTT≈31ms

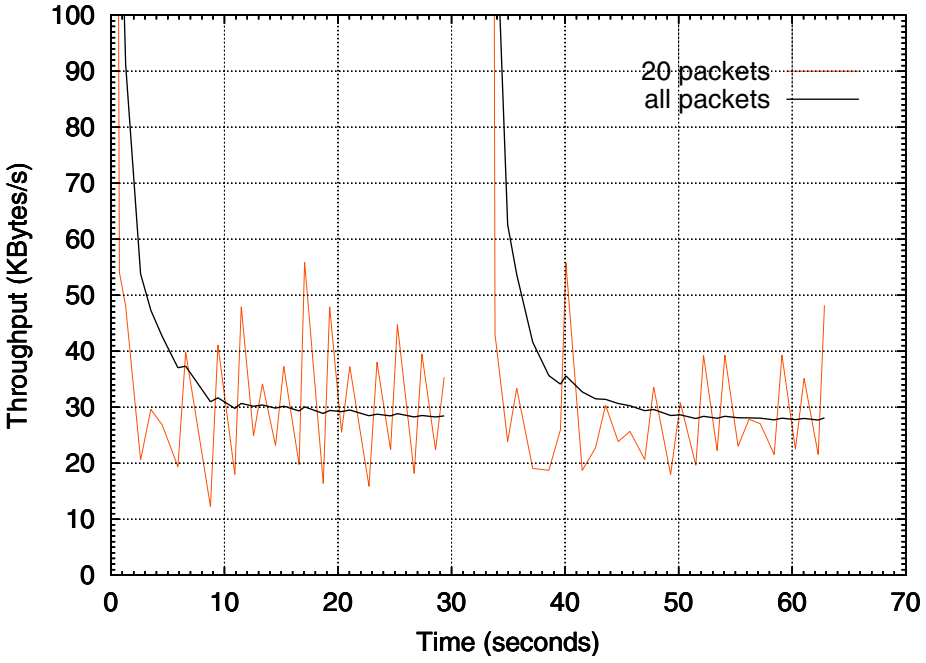


Figure 6-15. Entire playback TCP throughput, DDT≈3s >> RTT≈31ms

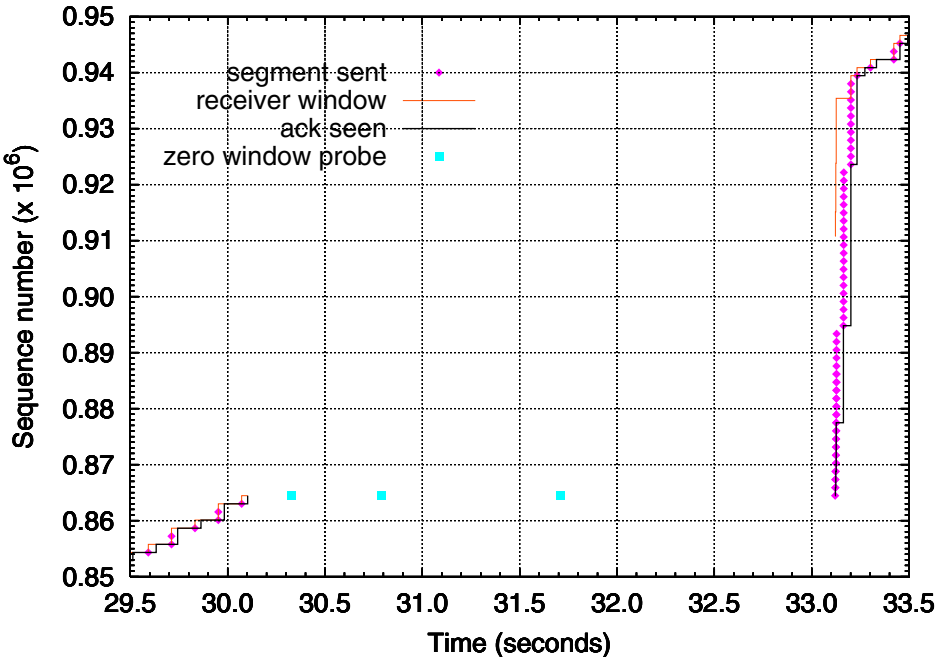


Figure 6-16. Zoomed TCP sequence trace, DDT≈3s >> RTT≈31ms

which means that TCP receiver queue on the laptop is again full, same as the previous case when DDT≈30ms. Therefore, no packets are sent during the 3s disconnection. The "retransmissions" are actually zero window probes by the TCP sender on the server, which follow the same exponential backoff rule governing the retransmissions.

3. Since no packets are lost during the handoff, when the first ack after reconnection arrives at 33.1s carrying a non-zero window, TCP immediately sends as much as it can to fill up the window instead of performing slow start.

6.1.1.3 Handoff from a WAN to LAN, DDT≈100ms and 2s

Figure 6-17 and Figure 6-18 show the TCP sequence trace and throughput of the

entire download session for the case of handoff from WAN to LAN when DDT≈100ms. We can see that throughput on the LAN connection is roughly 8MBytes/second and throughput on the WAN connection is roughly 500KBytes/second. The handoff happens at roughly 19s and TCP throughput adapts to the higher throughput of the wired LAN after the handoff. We also note that there is a "plateau" of roughly 2s for the LAN connection at roughly 23s, with 3 zero window probes sent by the server. This is the period when the buffer of *wget* on the laptop is full and is being written out to the disk so *wget* stops reading from the *socket*. Also note that since LINUX has a minimum RTO of 200ms, there is no timeout and retransmission in this case.

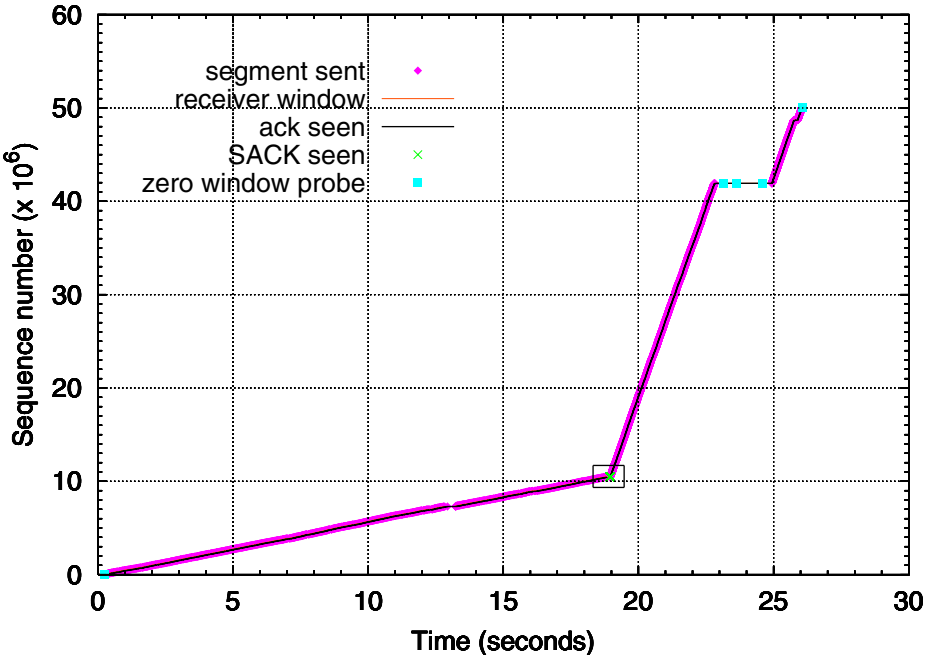


Figure 6-17. Entire download TCP sequence trace, DDT≈100ms

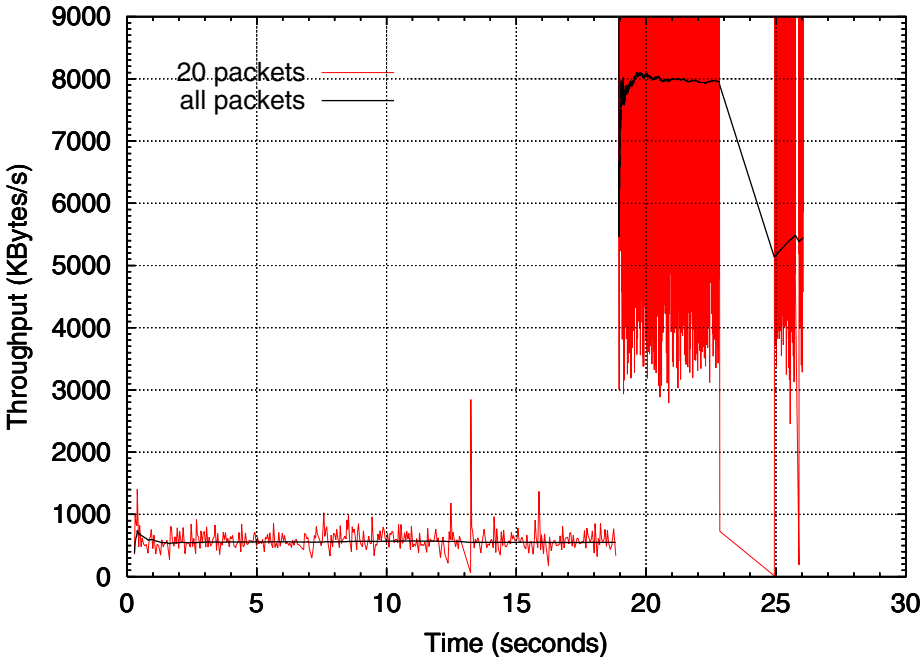


Figure 6-18. Entire download TCP throughput, DDT≈100ms

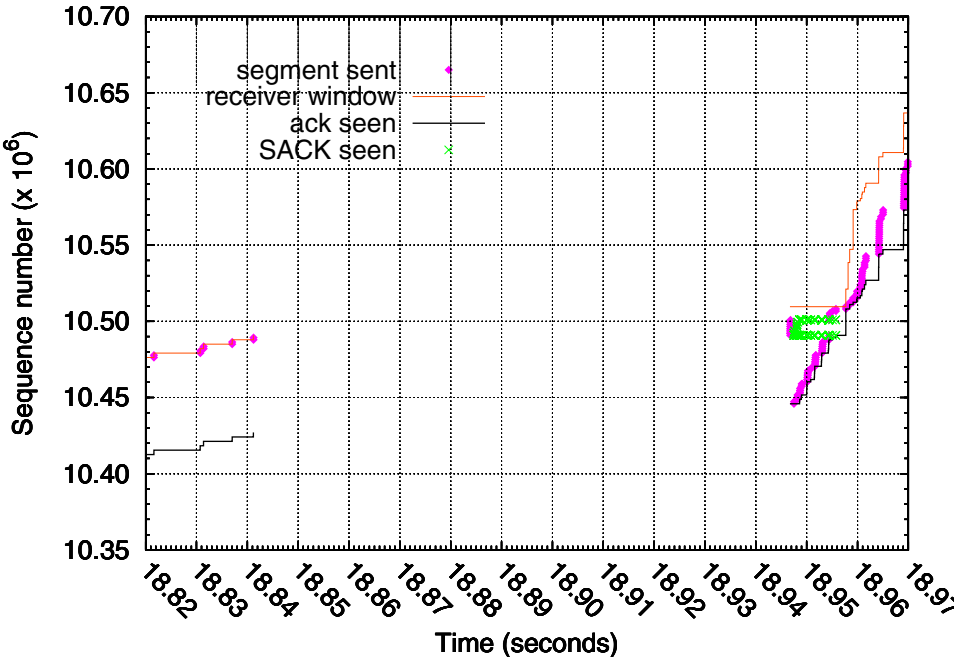


Figure 6-19. Zoomed TCP sequence trace, DDT≈100ms

Figure 6-19 shows the zoomed view for the handoff case when DDT≈100ms. It shows that:

- Shortly after 18.84s, the server sees the last ack, seq numbered 10.425, from the client, and it sends a couple of packets to the client, seq numbered 10.49.
- At roughly 18.946s, the first ack which carries H2O HANDOFF message is received by the server. This ack advertises a bigger window so the server immediately sends a few packets, seq numbered between 10.49 and 10.5. However, the ack only carries a seq number 10.445, which means that during the handoff, packets between 10.425 and 10.445 are received but all packets between 10.445 and 10.49 are lost.
- The lost packets seq numbered between 10.445 and 10.49 are being retransmitted during 18.947s and 18.955s, while new packets sent at 18.946s are being SACKed.
- At roughly 18.958s, the lost and retransmitted packets fill the gap below the SACKed packets and we see a jump of the ack to seq number 10.51 from the receiver which indicates the end of recovery period. From that point on, the server transmits at the higher throughput allowed by the new wired 100Mbits connection. The recovery period takes roughly 11ms, from 18.947s to 18.958s.

Figure 6-20 and Figure 6-21 show the TCP sequence trace and throughput of the entire download session for the case of handoff from WAN to LAN when DDT≈2s.

We can see that, similar to the case when DDT≈100ms, throughput on the LAN connection is roughly 8MBytes/second and throughput on the WAN connection is roughly 500KBytes/second. The handoff happens at roughly 18s and the TCP throughput also adapts to the higher throughput of the wired LAN after the hand-off. We also note that there is a "plateau" of roughly 2s with 2 zero window probes for the LAN connection at roughly 22s, similar to that of the case when DDT≈100ms.

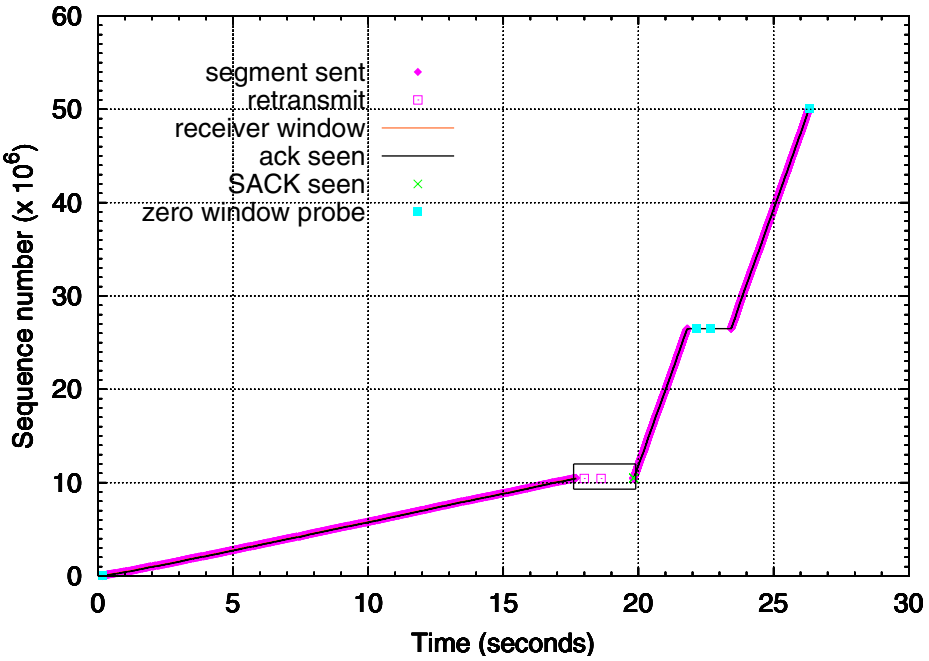


Figure 6-20. Entire download TCP sequence trace, DDT≈2s

Figure 6-22 shows the zoomed view for the handoff case when DDT≈2s. After the timeout and retransmission, the server goes into slow start. When the first ack that carries H2O HANDOFF message arrives at around 19.8s, no new packets are sent. Instead, lost packets are retransmitted and the server adapts to the new wired LAN

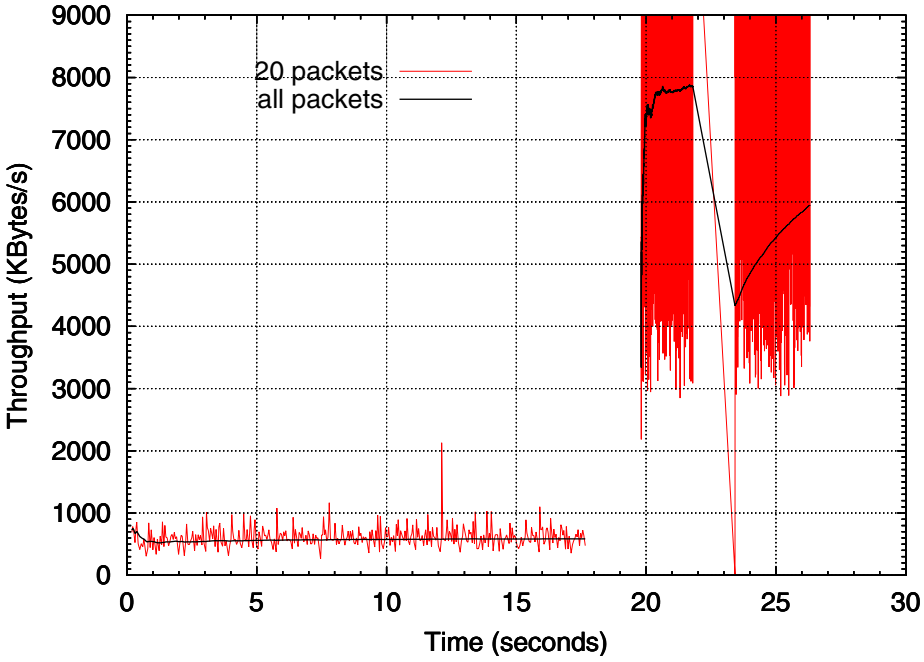


Figure 6-21. Entire download TCP throughput, DDT≈2s

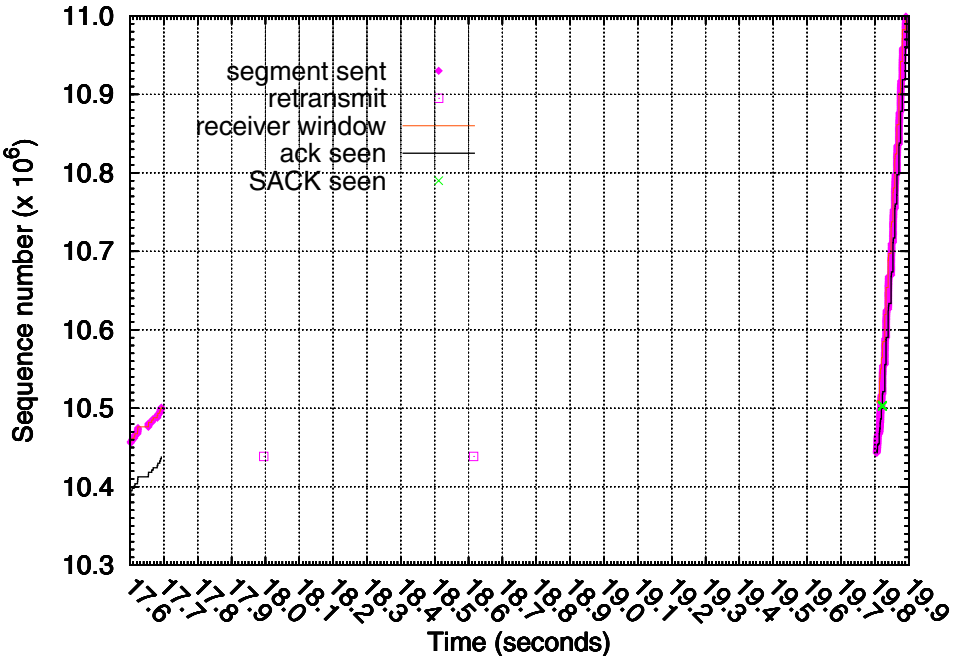


Figure 6-22. Zoomed TCP sequence trace, DDT≈2s

throughput as if the connection were just established.

6.1.1.4 Handoff from a LAN to WAN, DDT≈100ms and 2s

Figure 6-23 and Figure 6-24 show the TCP sequence trace and throughput of the entire download session for the handoff case from LAN to WAN when DDT≈100ms. We can see that, apart from the reversed throughput before and after the handoff, the characteristics of the download session are rather similar to those of the handoff case from WAN to LAN in Figure 6-17 and Figure 6-18. Note again the "plateau" for the LAN connection at roughly 3s with 3 zero window probes.

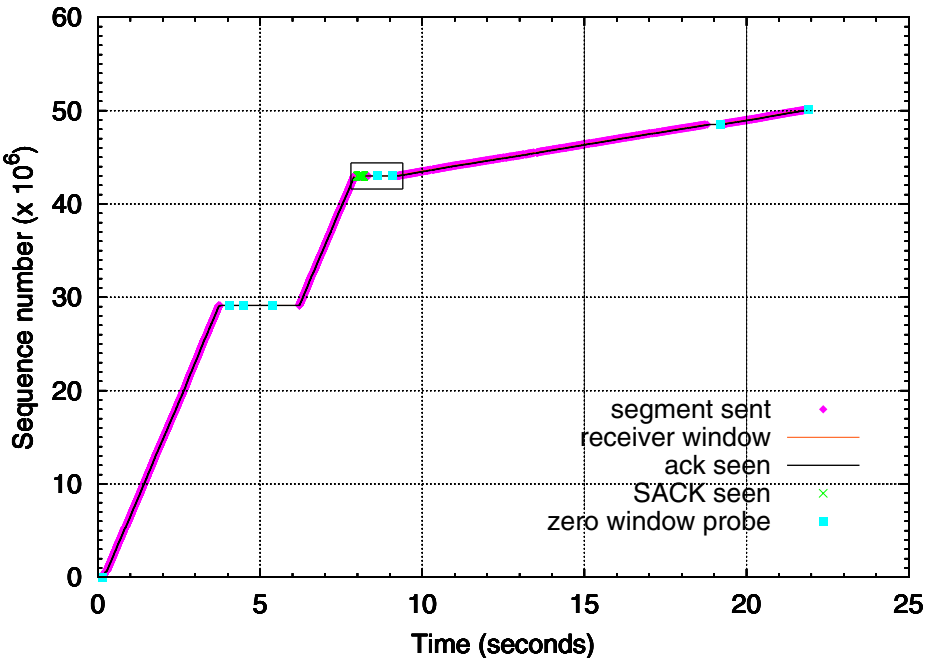


Figure 6-23. Entire download TCP sequence trace, DDT≈100ms

Figure 6-25 shows the zoomed view for the handoff case when DDT≈100ms. We can see that it has a recovery period similar to that of the handoff case from WAN to LAN when DDT≈100ms in Figure 6-19. During the recovery period, lost packets

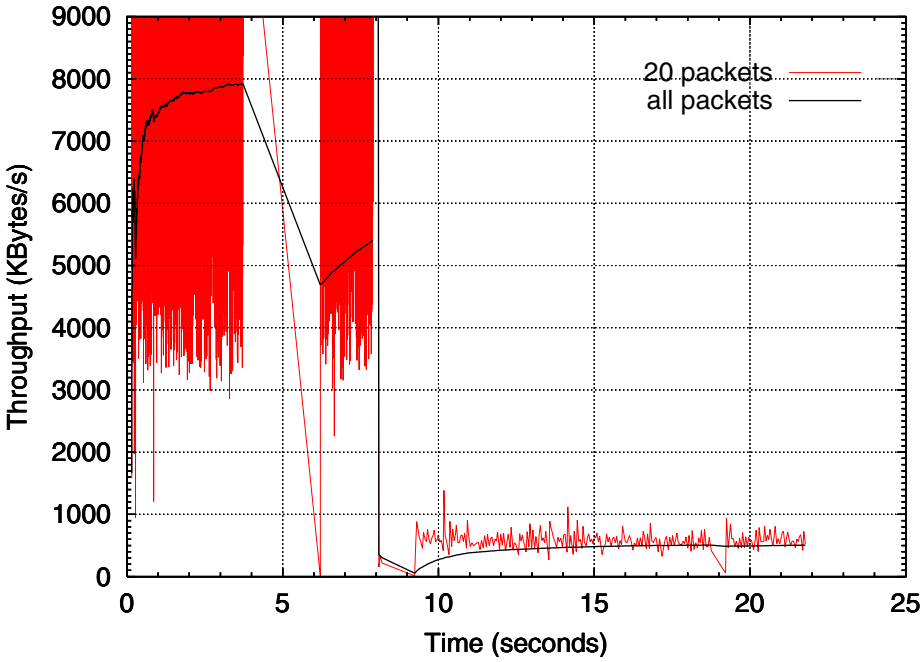


Figure 6-24. Entire download TCP throughput, DDT≈100ms

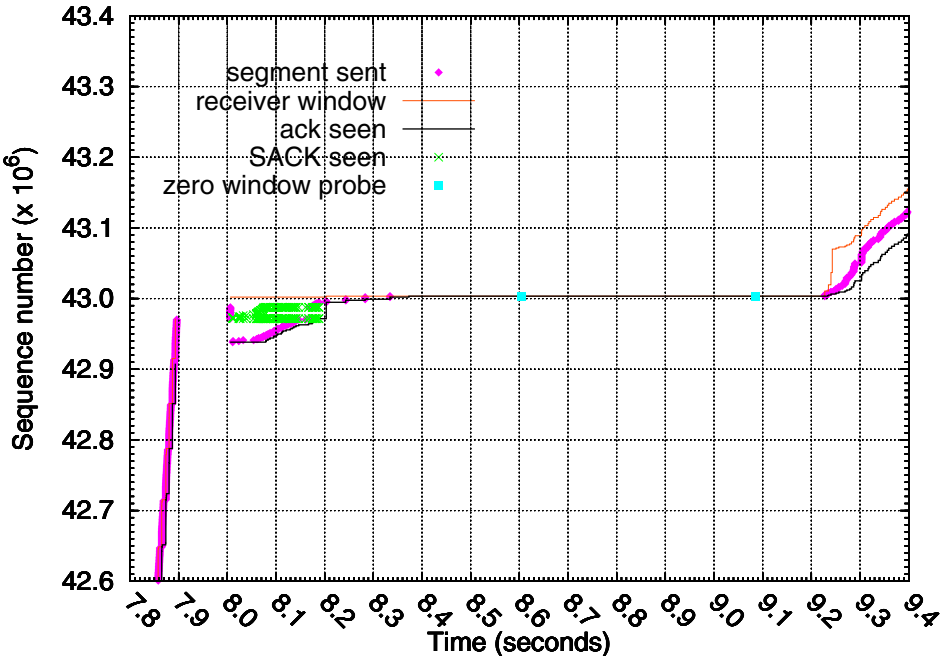


Figure 6-25. Zoomed TCP sequence trace, DDT≈100ms

seq numbered between 42.94 and 42.97 are being retransmitted while new packets sent at 8.01s seq numbered between 42.97 and 42.99 are being SACKed. The recovery period, which lasts about 200ms from 8.0s to 8.2s, is much longer than that in Figure 6-19 due to the slower WiFi connection. Also different from the WAN to LAN case, at the end of the recovery period, roughly 8.2s, receiver window on the laptop only increases slightly, therefore the server can only send a few packets from 8.2s to 8.33s, at which point the receiver window becomes 0. After two zero windows probes from the server, one at 8.61s and the other at 9.08s, the laptop finally advertises a nonzero receiver window at roughly 9.23s and increases the window dramatically at 9.24s. From that point on, the server transmits freely but at a lower throughput restricted by the new WiFi 11Mbits connection.

Figure 6-26 and Figure 6-27 show the TCP sequence trace and throughput of the entire download session for the handoff case from LAN to WAN when DDT≈2s. Once more, apart from the reversed throughput before and after the handoff, the characteristics of the download session are rather similar to those of the handoff case from WAN to LAN in Figure 6-20 and Figure 6-21. The "plateau" with 3 zero window probes for the LAN connection also presents at roughly 3s.

Figure 6-28 show the zoomed view for the handoff case when DDT≈2s. Similar to the handoff case from WAN to LAN when DDT≈2s in Figure 6-22, after the timeout and retransmission, the server goes into slow start. When the first ack that carries H2O HANDOFF message arrives, no new packets are sent. Instead, lost packets are retransmitted and the server adapts to the new connection throughput

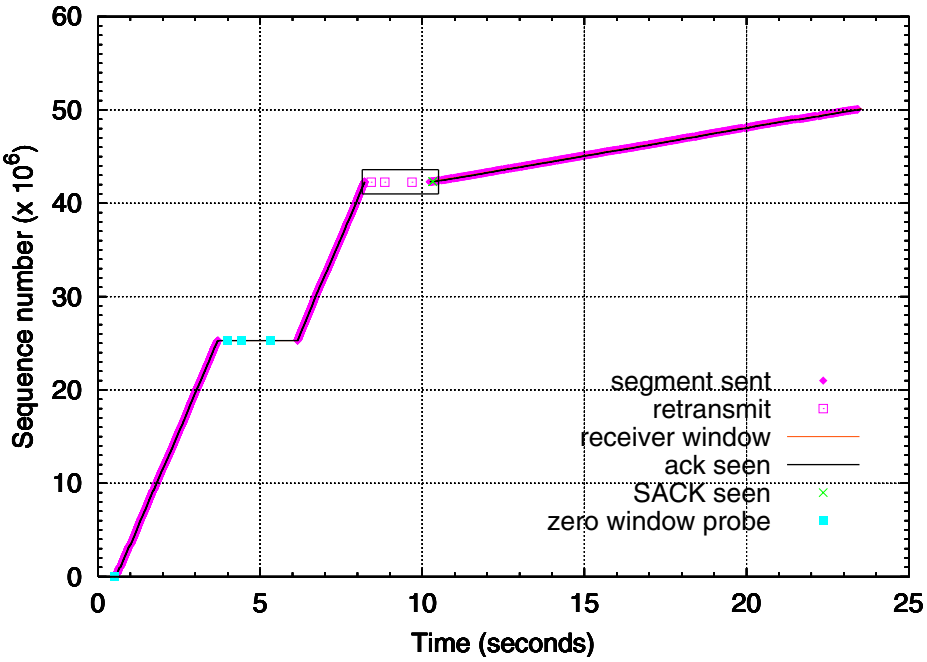


Figure 6-26. Entire download TCP sequence trace, DDT≈2s

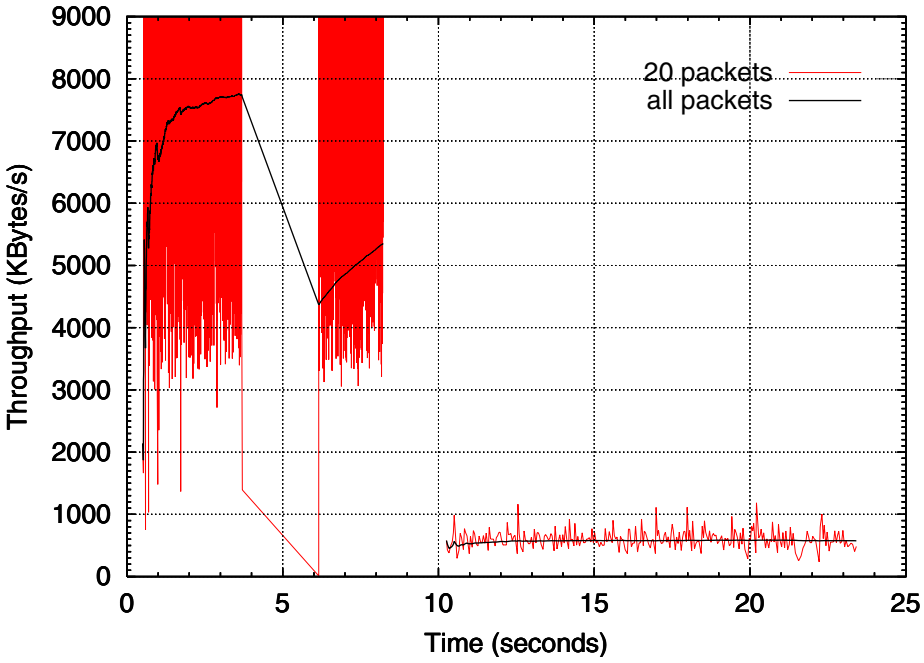


Figure 6-27. Entire download TCP throughput, DDT≈2s

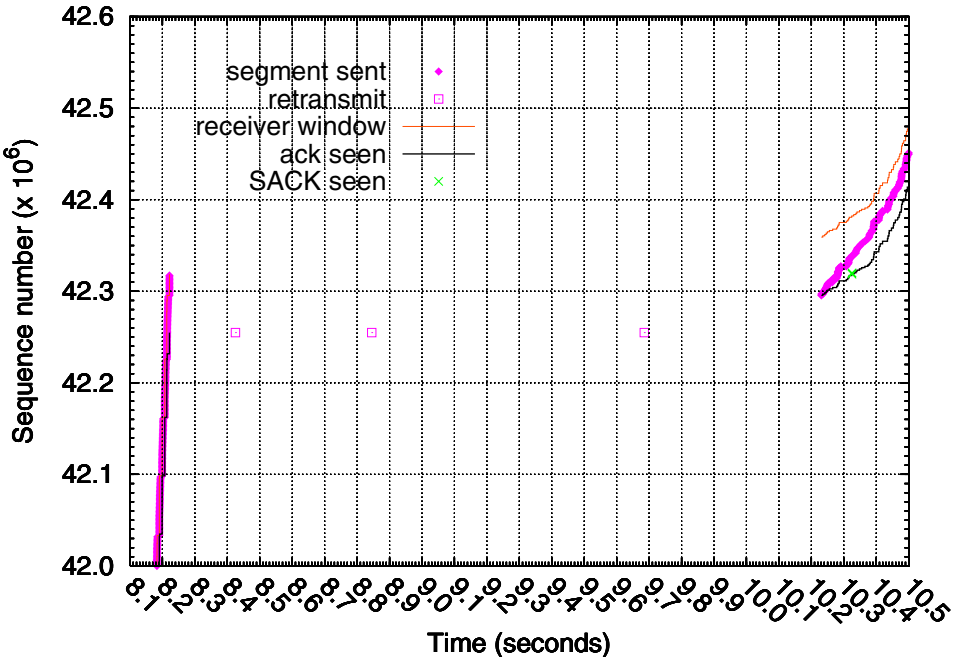


Figure 6-28. Zoomed TCP sequence trace, DDT≈2s

as if the connection were just established.

6.1.2 Client handoff with VMware migration

This test is the same as the third case of the client handoff with machine migration test, except that instead of migrating the laptop by switching between its 100Mbits ethernet NIC and 11Mbits WiFi PCCard, a VMware version 4.5.2 virtual machine configured with 64MB RAM is suspended and resumed between two IBM T22 ThinkPad laptop computers with 1GHz Pentium III CPU and 512MB RAM, one has an 100Mbits ethernet NIC, and the other an 11Mbits WiFi PCCard. Both laptops as well as the VMware VM run LINUX kernel version 2.4.20. The testbed is depicted in Figure 6-29.

Also, instead of *wget*, we use *lftp* [12] version 2.6.3 running inside a VMware VM

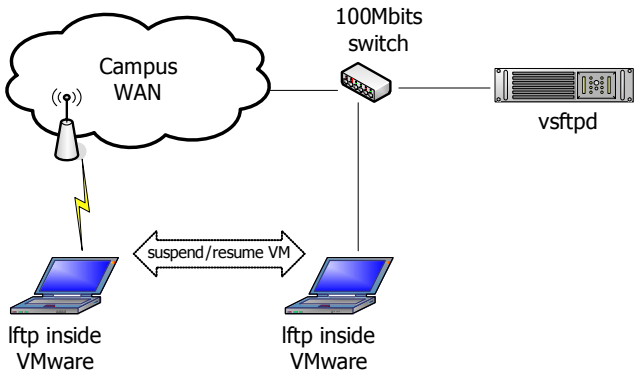


Figure 6-29. Client handoff with VMware migration testbed

on the laptop to fetch the same 50MB file from the server running *vsftpd* [24] version 1.1.3. The handoff is effected by suspending the VM on one laptop and resuming it on the other. For the two cases of switching from WAN to LAN and *vise versa*, the handoff point again is where roughly 80% of the file is going through the LAN and 20% of the file is going through the WAN. The download session is captured on the server using *tcpdump* and analyzed using *tcptrace*.

6.1.2.1 Handoff from a WAN to LAN, DDT≈8s

Figure 6-30 and Figure 6-31 show the TCP sequence trace and throughput of the entire download session for the handoff case from WAN to LAN with DDT≈8s. We can see that, apart from the longer gap due to the handoff, the characteristics of the download session are rather similar to those in Figure 6-20 and Figure 6-21 when a client machine is migrated from a WAN to LAN.

Figure 6-32 and Figure 6-33 show the zoomed view of the handoff, divided into two figures to avoid the large gap during the handoff. Since it takes about 8 seconds to suspend and resume the VM, there is no way to avoid TCP timeout and

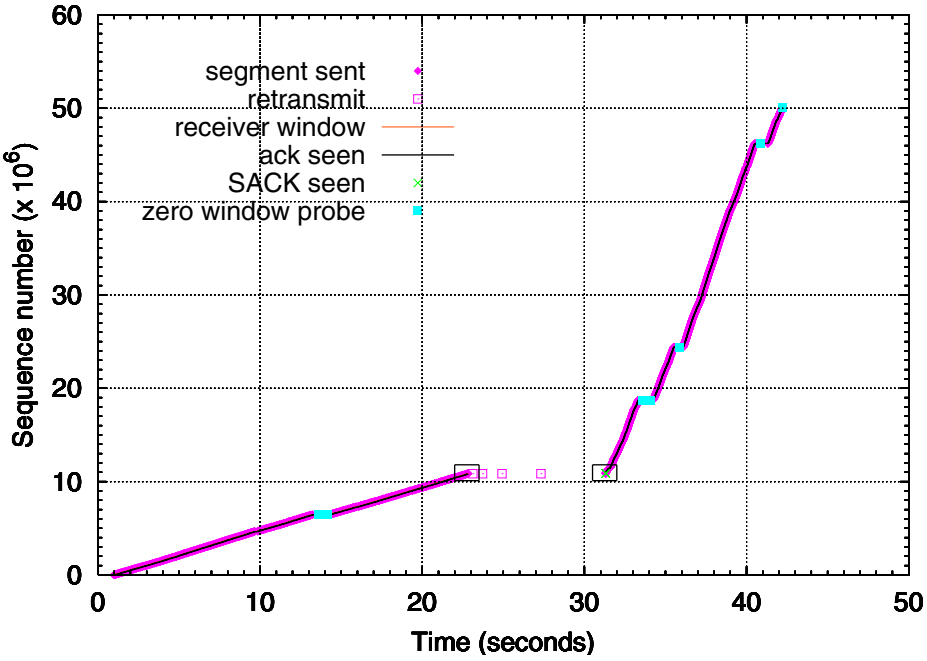


Figure 6-30. Entire download TCP sequence trace, DDT≈8s

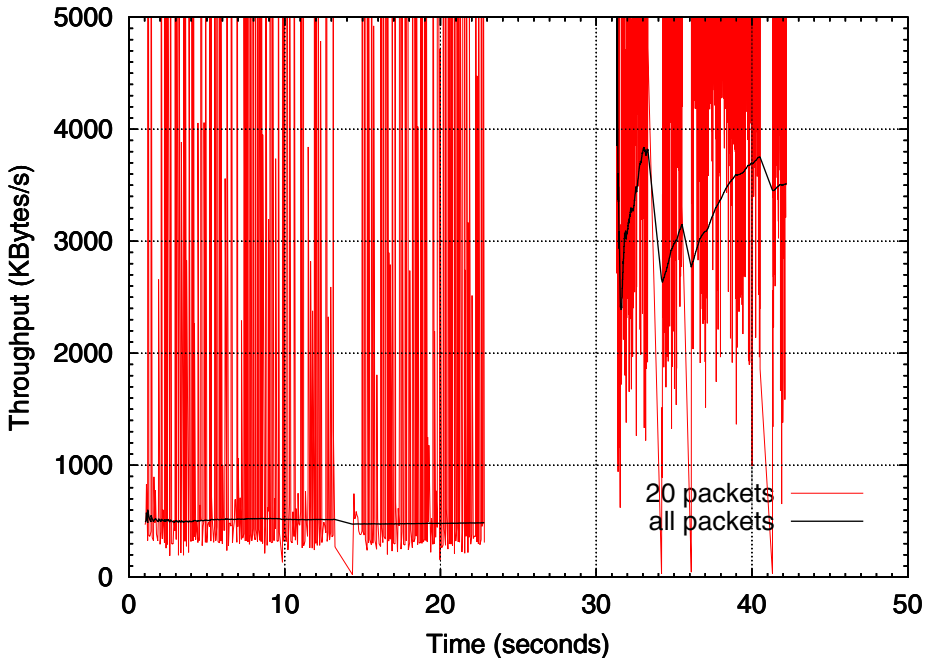


Figure 6-31. Entire download TCP throughput, DDT≈8s

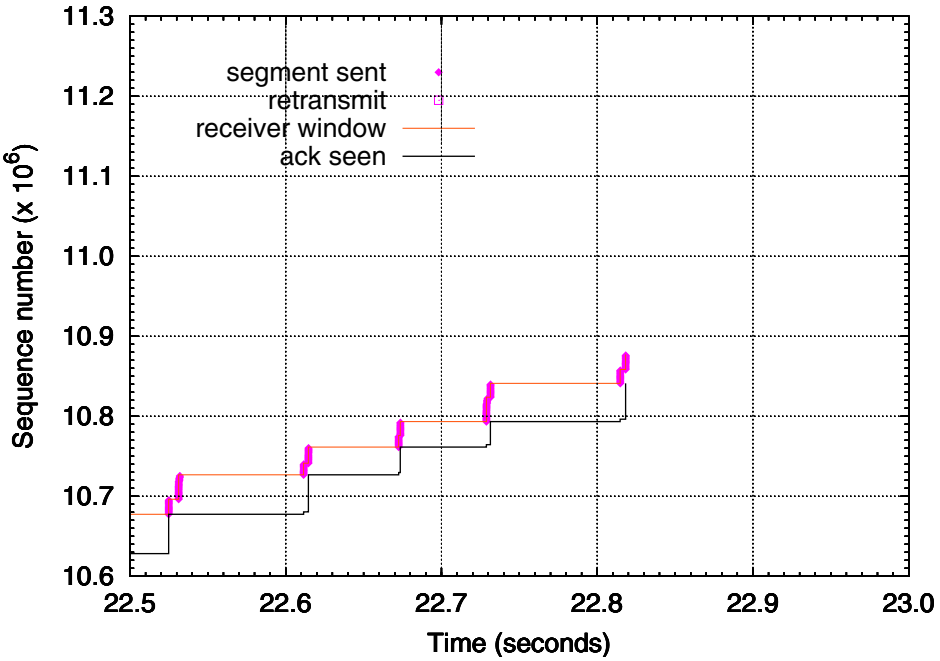


Figure 6-32. Zoomed TCP sequence trace, before handoff, DDT≈8s

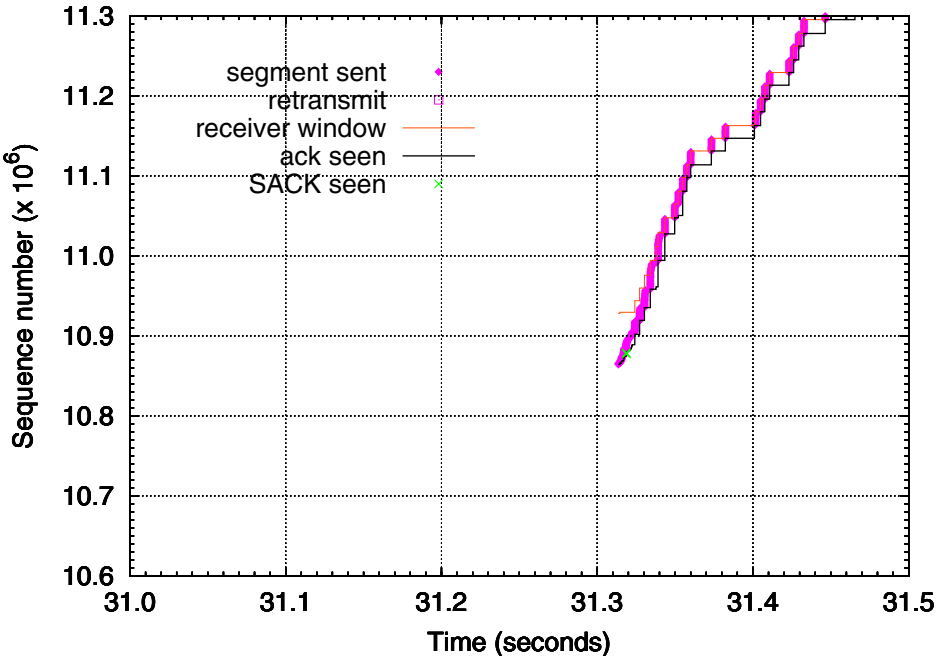


Figure 6-33. Zoomed TCP sequence trace, after handoff, DDT≈8s

retransmission. Therefore after the handoff, TCP on the server only sends one packet even though the receiver window has opened up. Apart from the longer handoff time, the characteristics of Figure 6-32 and Figure 6-33 are also rather similar to those of Figure 6-22 when a client machine is migrated from a WAN to LAN.

6.1.2.2 Handoff from a LAN to WAN, DDT≈11s

Figure 6-34 and Figure 6-35 show the TCP sequence trace and throughput of the entire download session for the handoff case from LAN to WAN with DDT≈11s. Figure 6-36 and Figure 6-37 show the zoomed view for the handoff, again divided into two figures to avoid the large gap during the handoff. We can observe that apart from the reversed TCP throughput before and after the handoff, the characteristics of the handoff are rather similar to those in Figure 6-30 through Figure 6-33.

Since it takes about 11 seconds to suspend and resume the VM, there is also no way to avoid TCP timeout and retransmission in this case. Also apart from a longer handoff time, the characteristics of Figure 6-34 and Figure 6-35 are rather similar to those of Figure 6-26 and Figure 6-27 when a client machine is migrated from a LAN to WAN; and the characteristics of Figure 6-36 and Figure 6-37 are rather similar to those of Figure 6-28 when a client machine is migrated from a LAN to WAN.

6.1.3 Server handoff with process migration

This section presents handoff performance measurements of MOVE integrated with the Zap process migration mechanism and used in a proxy-based environ-

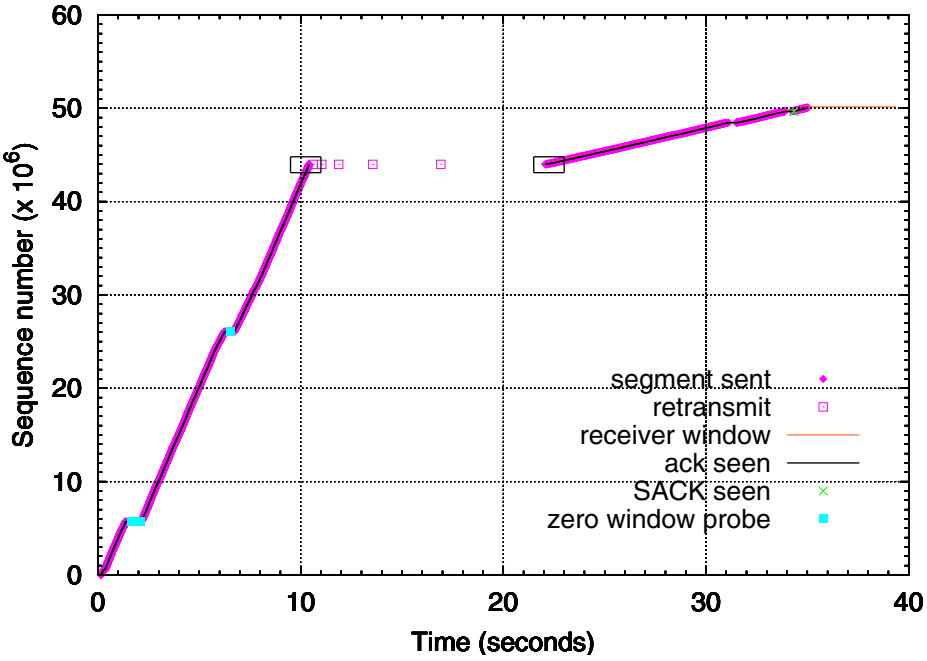


Figure 6-34. Entire download TCP sequence trace, DDT≈11s

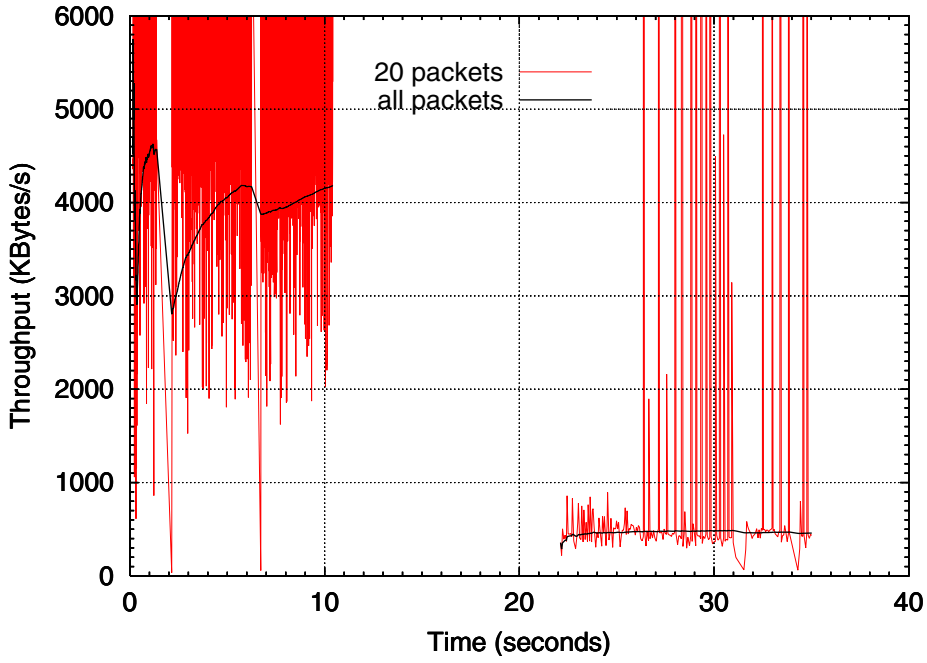


Figure 6-35. Entire download TCP throughput, DDT≈11s

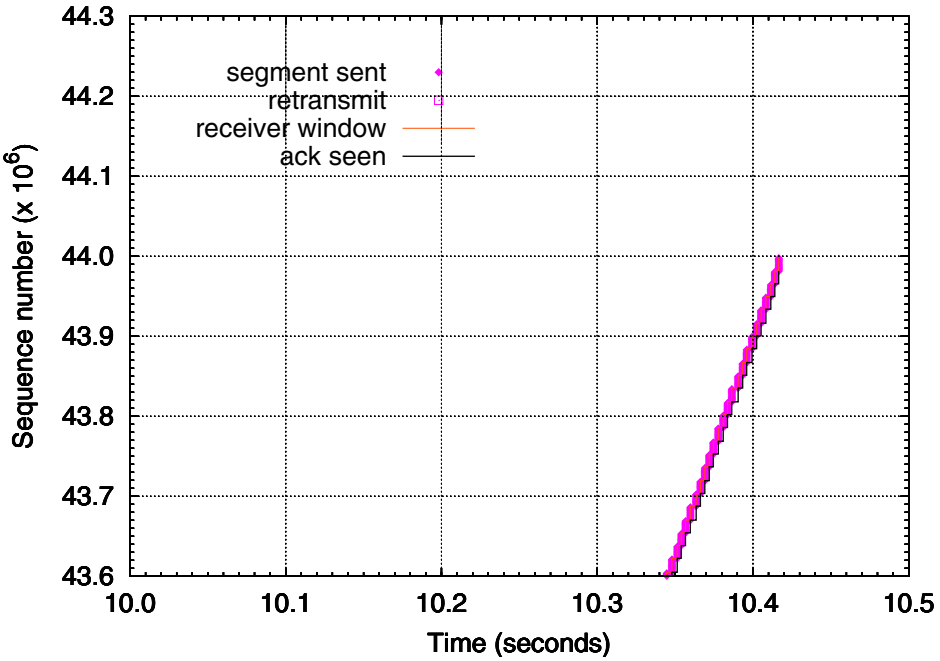


Figure 6-36. Zoomed TCP sequence trace, before handoff, DDT≈11s

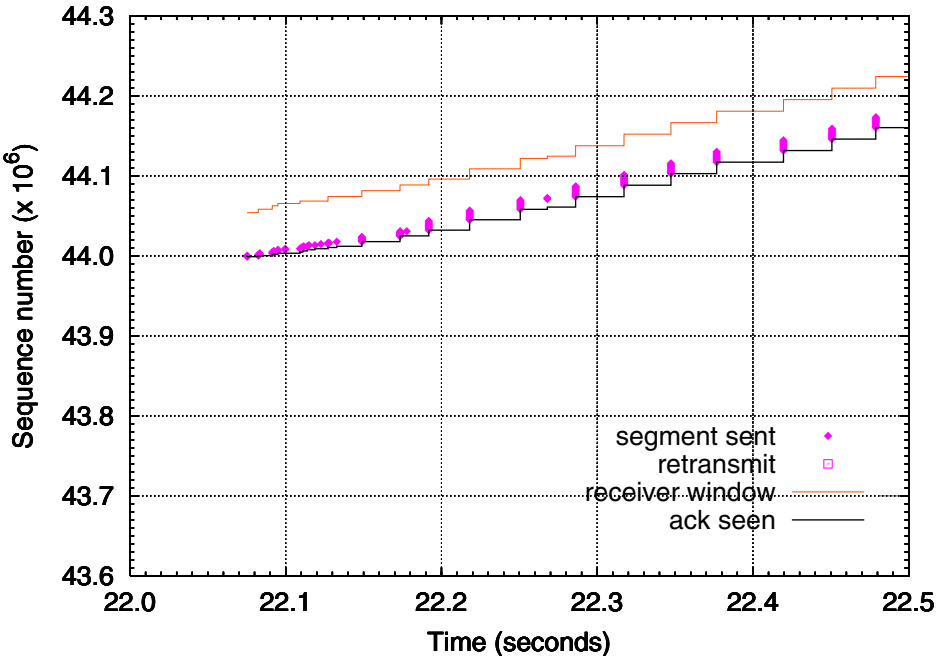


Figure 6-37. Zoomed TCP sequence trace, after handoff, DDT≈11s

ment, such as a server cluster described in Chapter 4. The testbed for these measurements consists of an IBM T22 ThinkPad laptop with 1GHz Pentium III CPU, 512MB RAM, and 100Mbits Intel Pro/100 SP ethernet NIC, four IBM rack mounted Netfinity 4500R servers, each with dual 933MHz Pentium III CPU, 512MB RAM, and 1Gbits Intel Pro/1000MT ethernet NIC, as illustrated in Figure 6-38. All machines are running LINUX kernel version 2.4.20.

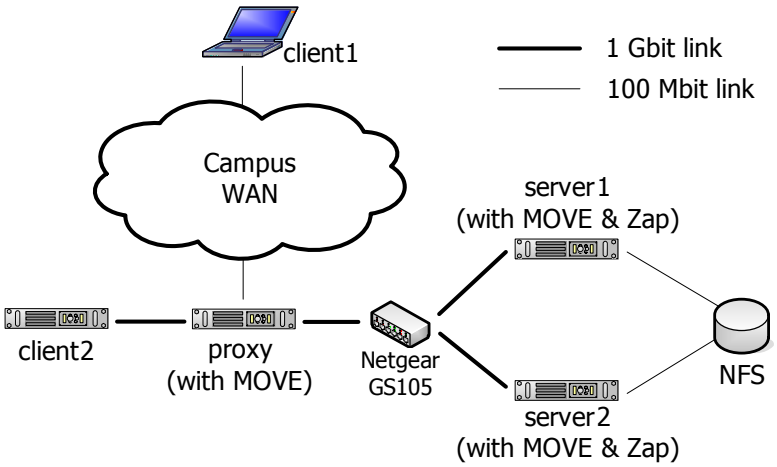


Figure 6-38. Server handoff with process migration testbed

We present the handoff performance of our integrated system by migrating an *apache* web server from server1 to server2 while it's streaming an RealVideo 8 clip to an *mplayer* application on a client machine, either client1 or client2, through *delegate* [4] version 8.9.2, a popular general purpose application level proxy, on the proxy machine.

The same applications and RealVideo clip as those we used in client machine handoff in Section 6.1.1 are used in the server handoff performance tests. The differences are:

- The *mplayer* client is instructed to connect to the *delegate* proxy, which in turn forwards the connection to the *apache* web server.
- The *apache* web server is migrated by checkpointing it to an NFS mounted storage on server1 and restarting it on server2. Note that in this case we do not have precise control on how long the server is "disconnected" from the network. Instead, we restart the server process on server2 immediately after it is checkpointed on server1. The entire procedure takes roughly 2 seconds, counting checkpointing, restarting, and paging through NFS.

We also study two cases: (1) the *mplayer* client1 is connected to the *delegate* proxy through a WAN as the one we tested in Section 6.1.1.1; (2) the *mplayer* client2 is connected directly to the *delegate* proxy through a 1Gbit link. In both cases, the proxy and the servers are connected through 1Gbit links, as shown in Figure 6-38. Packet traces for both the client-proxy connection and the proxy-server connection are captured on the proxy using *tcpdump* and analyzed using *tcptrace*.

6.1.3.1 Handoff with a WAN client, DDT≈2s

We again first present, in Figure 6-39 through Figure 6-42, the TCP sequence trace and throughput for the entire trace of the playback for both the client-proxy connection and the proxy-server connection. Note that both connections playback at roughly the same rate. The client-proxy playback has a delay of about 3-4 seconds relative to the proxy-server playback. This is due to the initial startup time (e.g., loading codec, initializing window system, etc.) of the *mplayer* client. Also note that the server handoff is completely transparent to the client-proxy connection, as

indicated by its throughput graph. The proxy-server connection, on the other hand, is perceived by *tcptrace* as two separate connection before and after migration due to the change of server IP address.

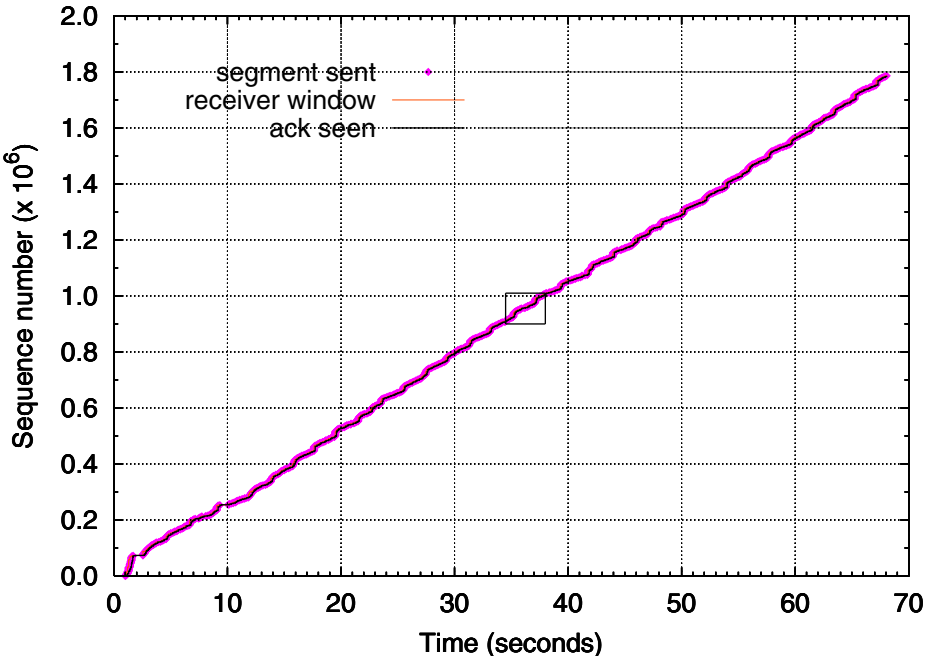


Figure 6-39. Entire download TCP sequence trace, client-proxy, DDT≈2s

We now zoom into the small boxes indicated in the TCP sequence trace graphs (Figure 6-39 and Figure 6-40) for both connections. From Figure 6-43, we can see that the client-proxy connection never perceived the handoff due to buffering at the proxy; we also did not notice any visual disruption in the playback on the client. For the proxy-server connection shown in Figure 6-44, the interesting points to note are:

- At about 35.2s, the proxy receives a zero-window probe from the server since the ack sent by the proxy at around 35s advertises a zero window.

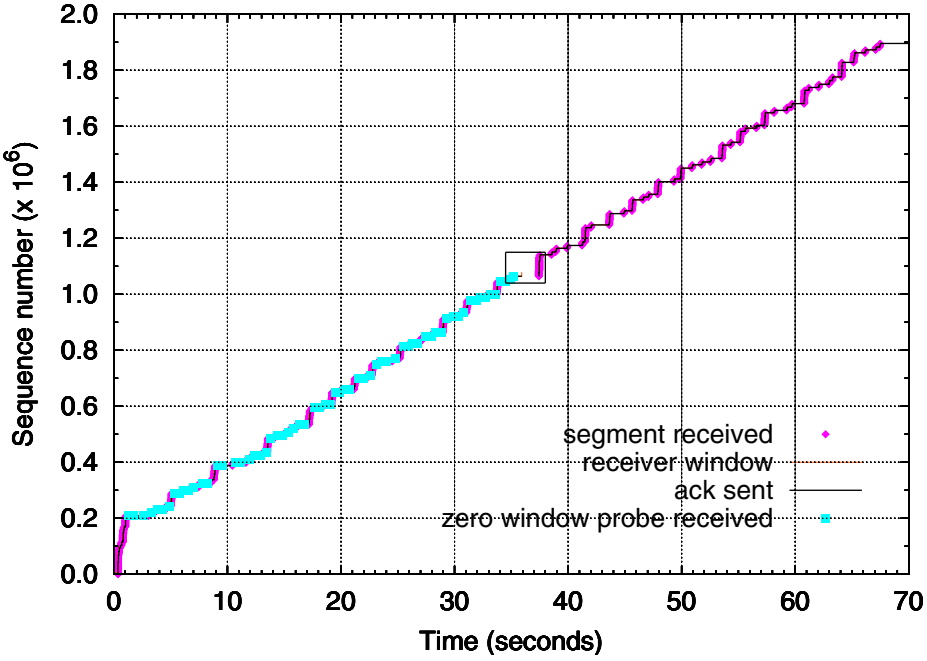


Figure 6-40. Entire download TCP sequence trace, proxy-server, DDT≈2s

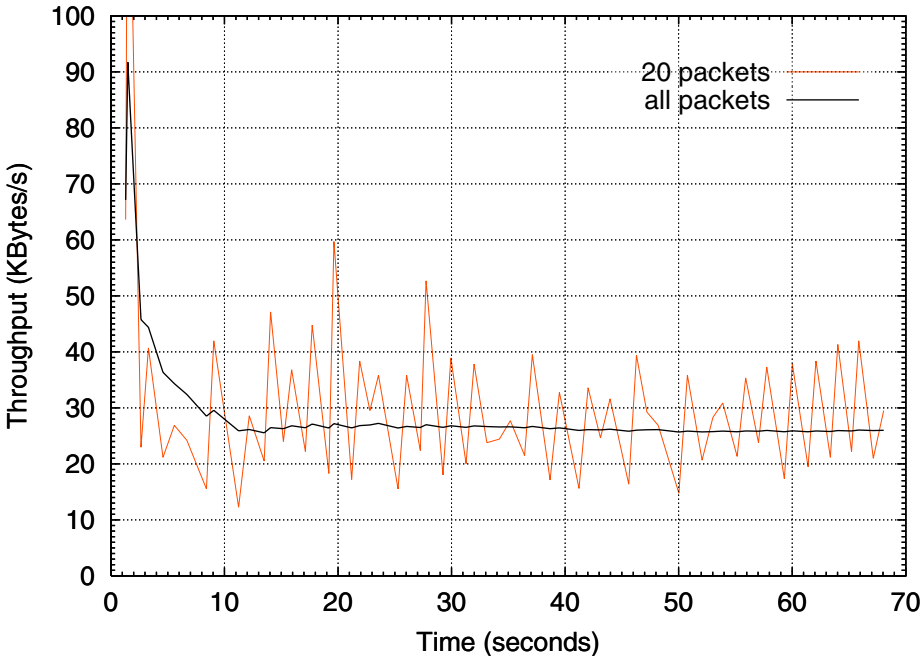


Figure 6-41. Entire download TCP throughput, client-proxy, DDT≈2s

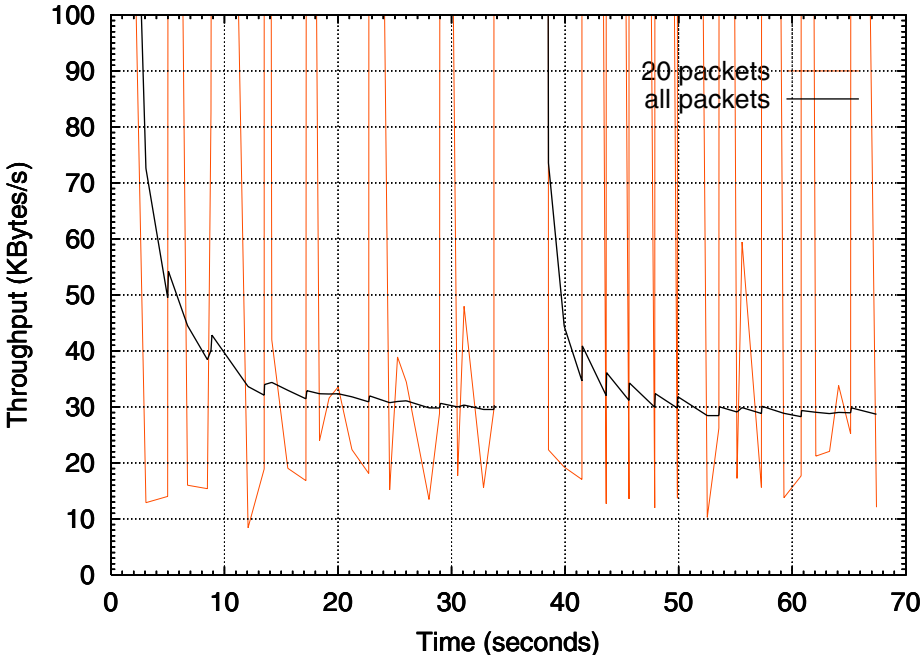


Figure 6-42. Entire download TCP throughput, proxy-server, DDT≈2s

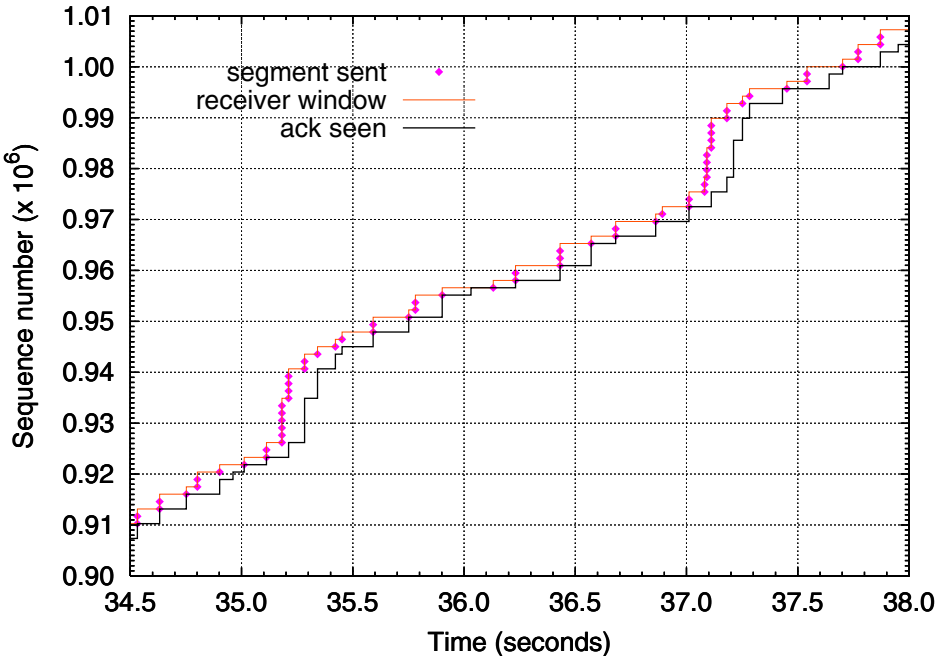


Figure 6-43. Zoomed TCP sequence trace, client-proxy, DDT≈2s

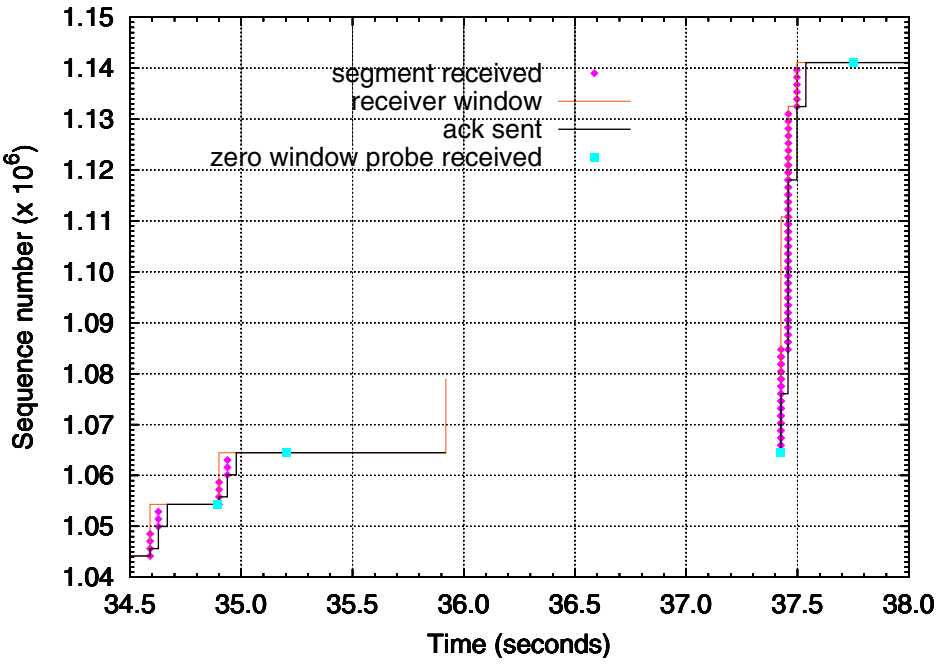


Figure 6-44. Zoomed TCP sequence trace, proxy-server, DDT≈2s

This is also the last packet the proxy receives from the server before the server is checkpointed. While invisible from the figure but appeared in the raw packet trace, at 35.2s immediately after receiving the zero-window probe, the proxy responds with an ack. However, that ack still advertises a zero window. Later, at 35.9s, the proxy sends another ack to report a non-zero window; but at this time the server is already checkpointed so the ack is lost. Note that TCP acks are *not* sent reliably.

- At 37.4, the server is restarted and sends another zero window probe, which carries H2O's HANDOFF message. The proxy responds with a much bigger window than the one advertised at 35.9s since more data has been delivered to the client during the time the server is being checkpointed and restarted. And the server immediate starts to send as much as

it can to fill up the window.

6.1.3.2 Handoff with a LAN client, DDT≈2s

Figure 6-45 through Figure 6-48 present the TCP sequence trace and throughput of the entire playback session for both the client-proxy connection and the proxy-server connection in the all gigabit LAN testbed. The characteristics of the handoff behavior are rather similar to those in the WAN test presented in the previous section, except that the delay of client-proxy playback relative to proxy-server playback is roughly 2-3 seconds. The shorter delay is due to the faster initial startup time of the *mplayer* on the client2 machine.

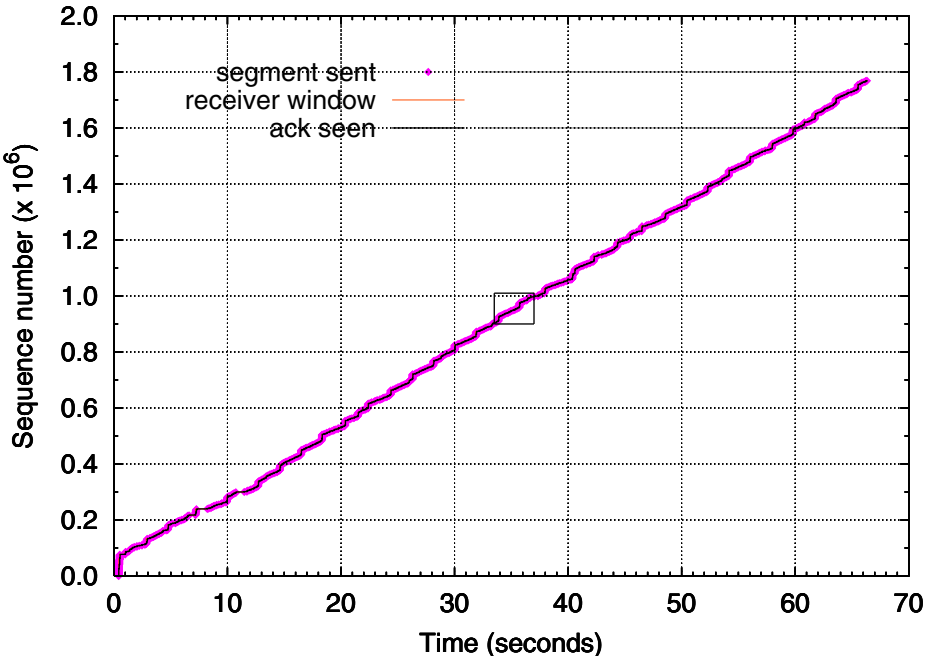


Figure 6-45. Entire download TCP sequence trace, client-proxy, DDT≈2s

Figure 6-49 and Figure 6-50 present the zoomed view of the handoff TCP sequence trace of the client-proxy connection and proxy-server connection, respectively.

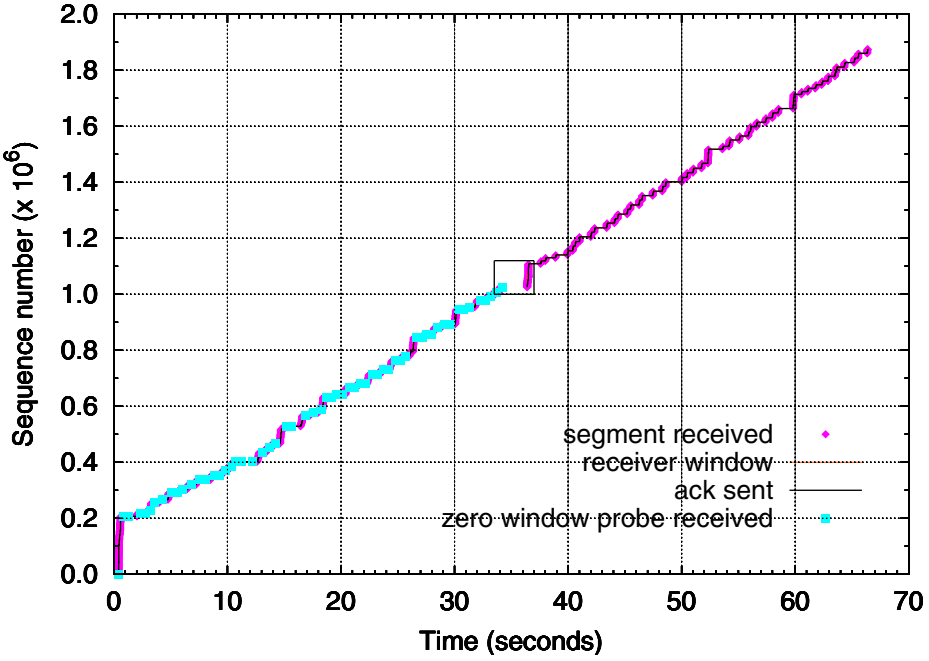


Figure 6-46. Entire download TCP sequence trace, proxy-server, DDT≈2s

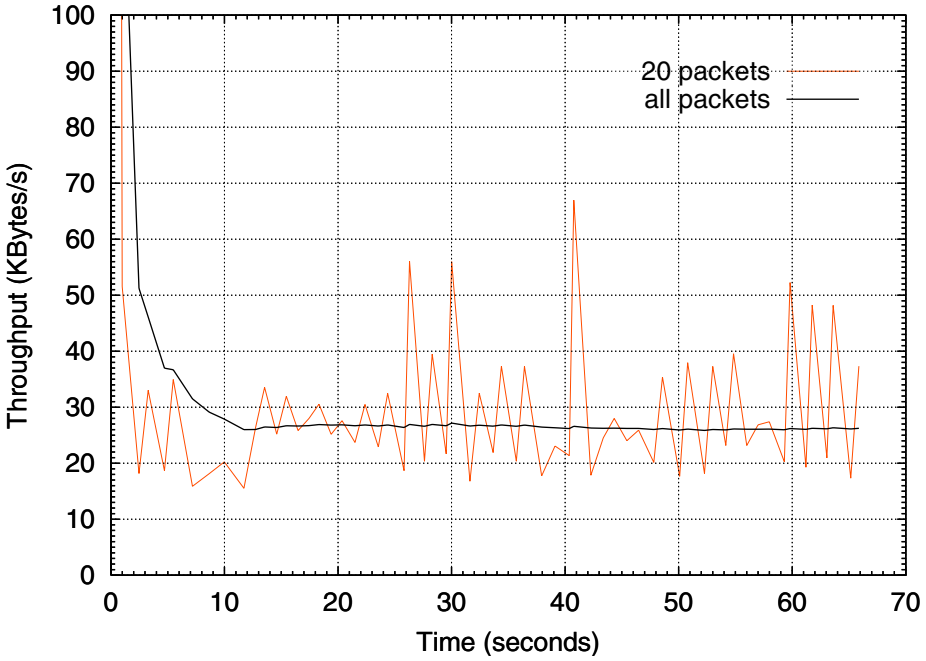


Figure 6-47. Entire download TCP throughput, client-proxy, DDT≈2s

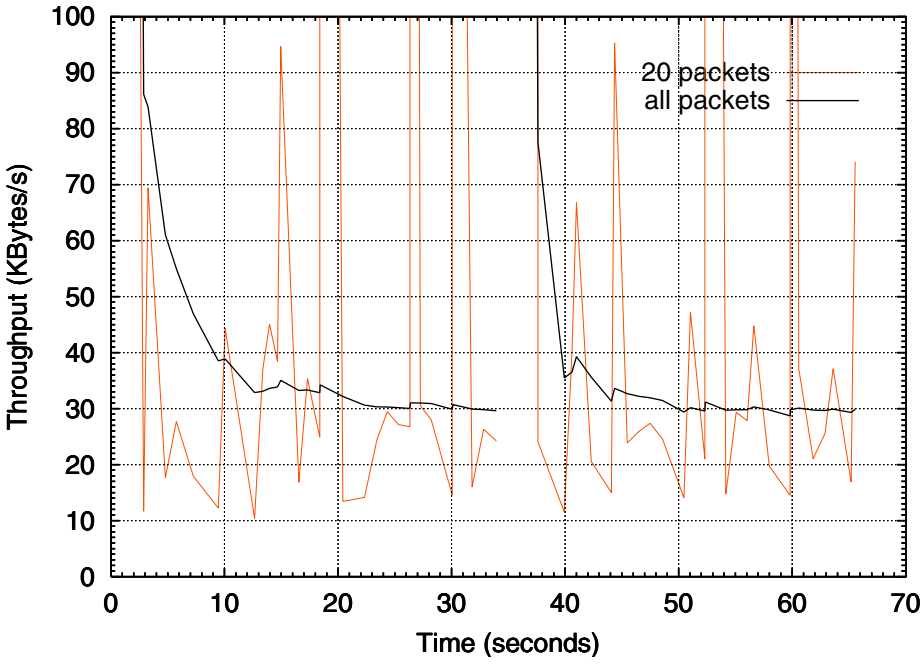


Figure 6-48. Entire download TCP throughput, proxy-server, DDT≈2s

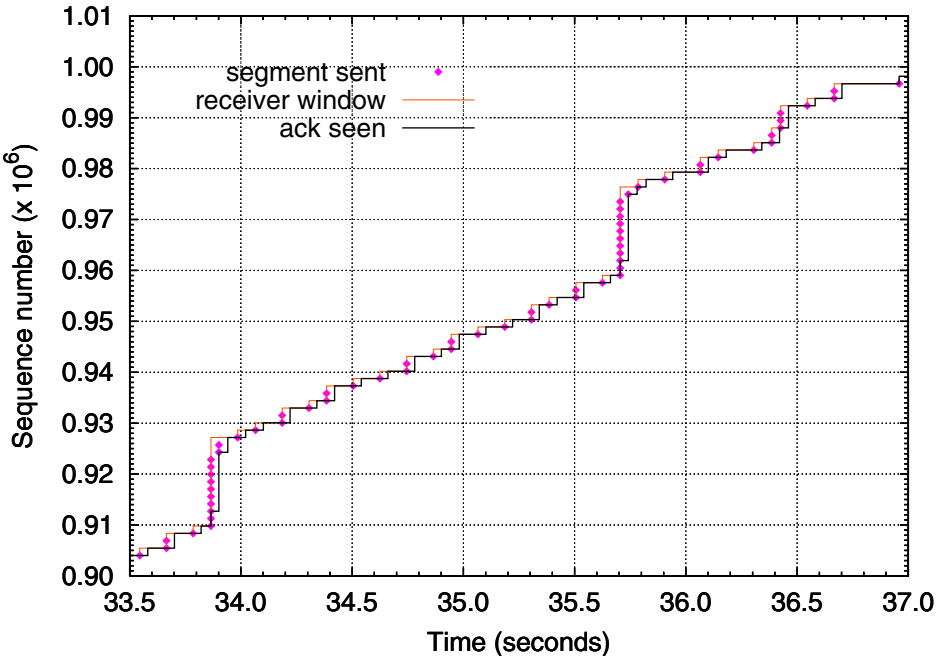


Figure 6-49. Zoomed TCP sequence trace, client-proxy, DDT≈2s

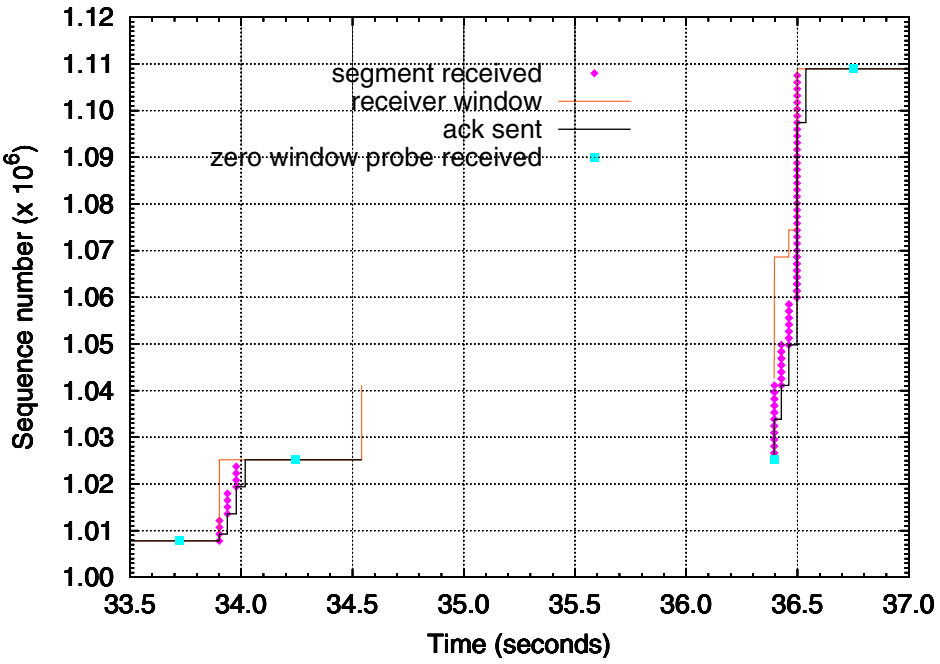


Figure 6-50. Zoomed TCP sequence trace, proxy-server, DDT≈2s

Again, the behavior is very similar to that in the WAN test so we do not repeat the discussion. This comes as no real surprise since the dynamic part of the test, migration of the server process behind the proxy, is the same for both cases.

6.1.4 Handoff "ping-pong" stress test

The handoff "ping-pong" stress test is conducted on the same testbed as that for the client machine migration on a LAN shown in Figure 6-1b. However, instead of using *mplayer*, we use *wget* to fetch a very large file from the *apache* server and the "ping-pong" migration is conducted as follows:

- *wget* on the client machine with IP1 starts downloading the large file
- after 5 seconds, the interface IP address is changed to IP2

- for the next 60 or so seconds, the interface IP address is switched between IP1 and IP2 with a varying interval of 10 seconds, 5 seconds, and 1 second between each switch. In other words, with 10 seconds interval, 6 switches are made, with 5 seconds interval, 12 switches are made, and with 1 second interval, 60 switches are made
- at the end of the "ping-pong" switches, the download is continued for another 5 seconds before it is stopped

The test results are shown in Figure 6-51. Note that since it takes roughly 20ms to change the IP address of client's NIC (during which time the client is disconnected from the network), the total elapsed time of a test increases as the rate of switch increases. But for all tests, the total *connected time* of the client should be roughly the same 5+60+5=70 seconds.

As shown in the figure, at a switch rate of once every 10 seconds, the impact of the handoffs is virtually invisible. Even at a switch rate of once every second, the impact of the handoffs is very small, with a decrease of about 1MB in the total bytes transferred at the end of the test comparing to the case of no handoff.

6.1.5 Handoff for connection-less transport protocols

Finally, we test MOVE's handoff support for connection-less transport protocols, UDP in this case. We again use the same testbed as that for client machine handoff on a LAN shown in Figure 6-1b. However, instead of using *mplayer* client and *apache* server, we use *openRTSP* [15] version 2004.06.02 client to "play" an MPEG-4 encoded video from *Darwin* streaming server [3] version 4.1.3 over RTP. Similar

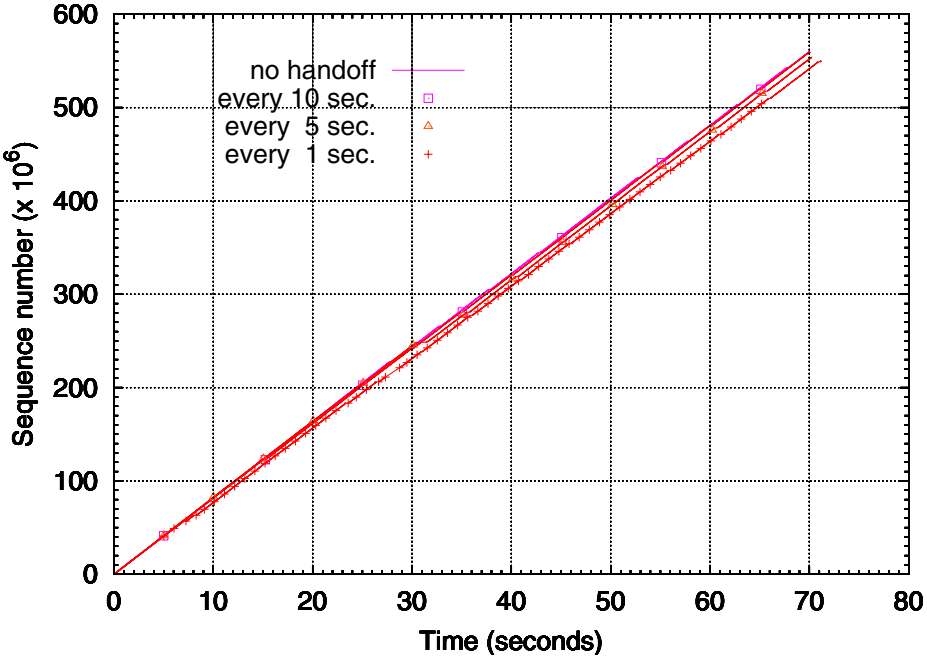


Figure 6-51. Handoff "ping-pong" stress test on a LAN

to the HTTP video playback test for client machine handoff described in Section 6.1.1, we "play" the video for about 30 seconds, change the IP address of the client's NIC, and then let the playback continue for about another 30 seconds. Note that a RTP session actually consists of two connections, one is a control connection over TCP known as RTCP (RTP Control Protocol), the other is a data "connection" over UDP. Both are migrated simultaneously when the client machine is migrated but our focus in this section is on the UDP "connection".

Since UDP packets do not have sequence numbers, we cannot plot sequence number trace like we did for TCP packets. Instead, we plot the cumulative bytes received at the client for the entire playback session, which is shown in Figure 6-52. The UDP throughput is shown in Figure 6-53. The handoff shows no visible

impact on the playback.

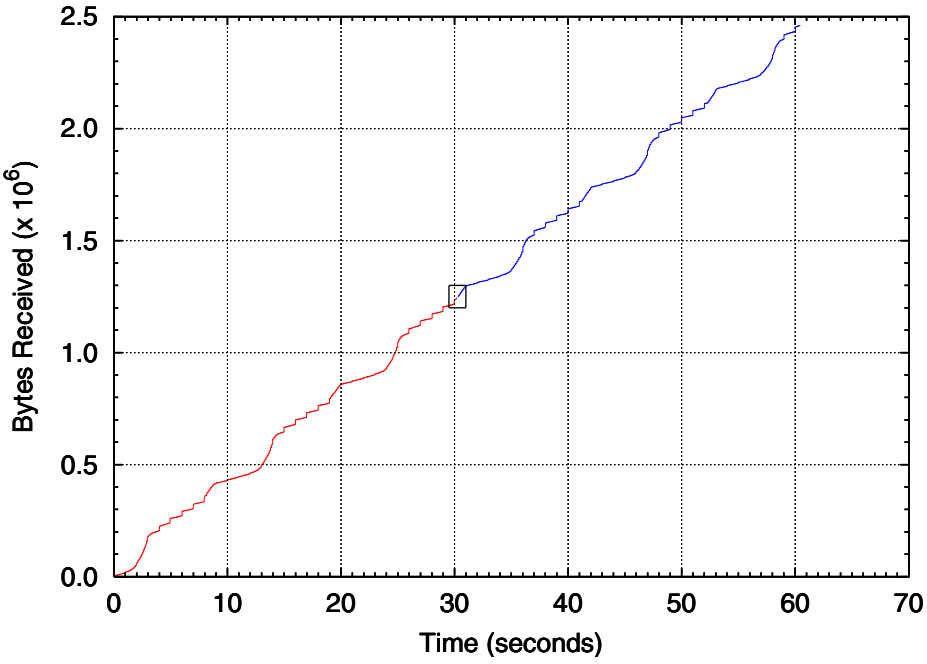


Figure 6-52. Entire playback UDP byte counts, DDT≈150ms

Figure 6-54 shows the zoomed view of the handoff indicated by the small box in Figure 6-52. We see that the handoff lasts about 150ms, starting at around 30.17s when the laptop is disconnected from the network and ending at around 30.32s when the laptop is reconnected to the network. The first message from the client to the server at 30.32s carries the MOVE HANDOFF protocol message. Note that since we are counting cumulative bytes received at the client, those lost packets during the 150ms handoff are not reflected in the figure.

6.1.6 Migrate popular real world applications

To conclude the handoff performance measurement section, we test the migration capability of our MOVE system with a suite of popular real world LINUX applica-

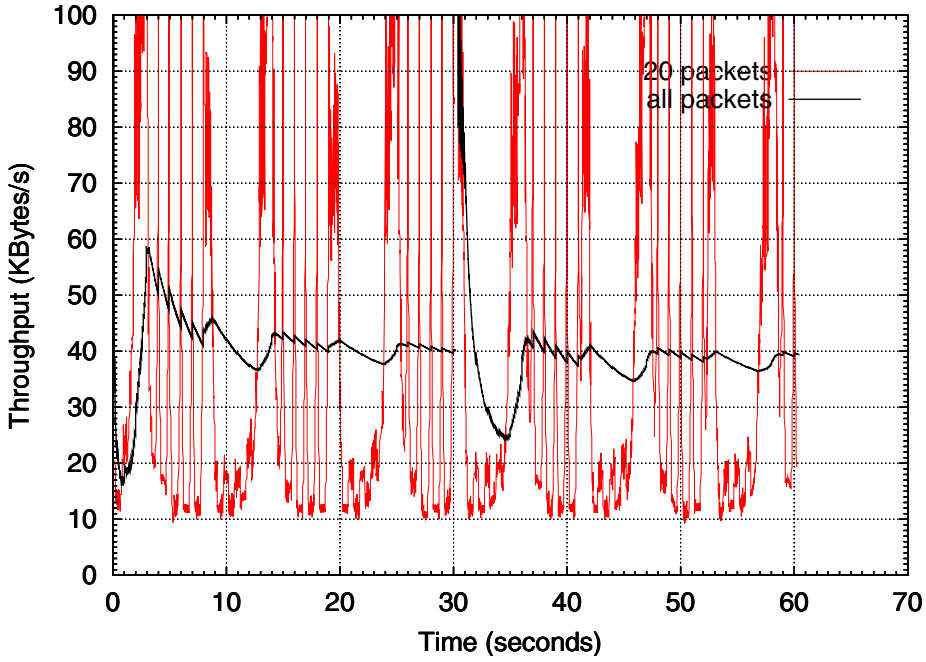


Figure 6-53. Entire playback UDP throughput, DDT≈150ms

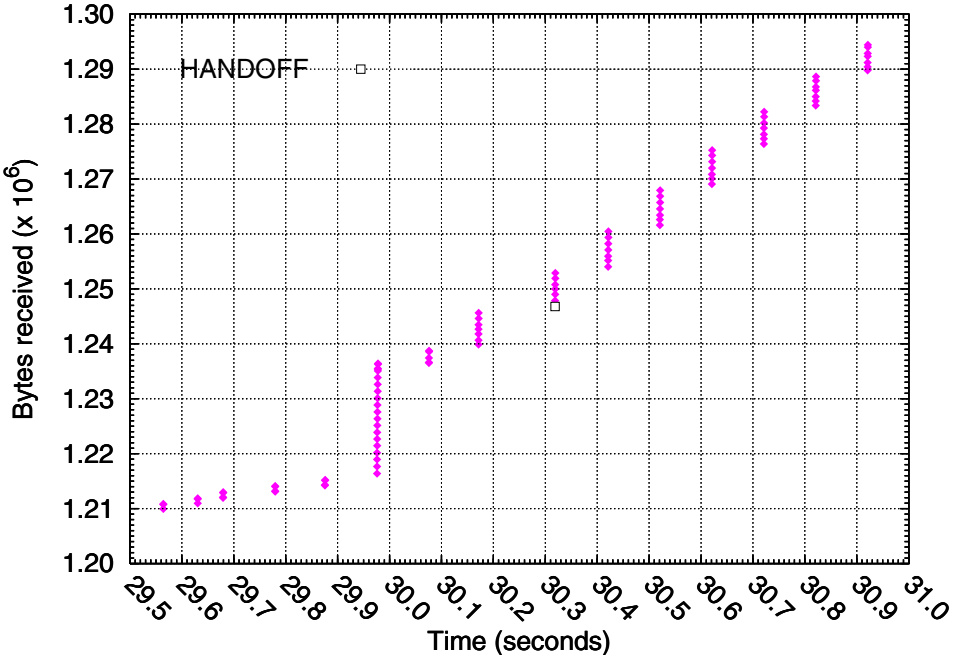


Figure 6-54. Zoomed UDP byte counts, DDT≈150ms

tions, including:

- *telnet* client and server (standalone and *xinetd*)
- *ftp* client and server (standalone and *xinetd*), both active and passive modes
- *ssh* client and server
- *mozilla/netscape/opera* client and *apache/zeus* server
- Ximian *evolution* client and *qpopper/sendmail* server
- *slrn* client and *innd* server
- *VNC* thin client and *VNC* server
- remote *X* client and *X* server
- *mplayer/realplay* client and *Darwin/Helix* server

All the above applications work over a virtualized connection right out of the box. We are able to migrate live connections created by all the above applications and the connections stay alive as if nothing happened. None of the applications, except *ftp*, requires us to completely hide the migration by exposing the virtual address. We are glad to see that these representative applications behave as we have expected. That is, rather than relying on transport connection properties for their application logic, they use the transport protocol solely for the purpose of transporting data.

6.2 Scalability Tests

We perform scalability tests in a proxy-base server cluster environment using *autobench* [92] version 2.1.1, a Perl script wrapper for *httperf* [94] version 0.8, with the

same testbed in Figure 6-38 that we used for server process handoff tests. We run *autobench* client on the client2 machine and *apache* web server on the server1 and server2 machines. We run *delegate* on the proxy machine to evenly distribute the connections from the *autobench* client to the two *apache* servers. The client1 machine is not used. We perform two scalability tests, one is scalability relative to the number of simultaneous connections and the other is scalability relative to the rate of new connections, for three different test configurations: *Vanilla*, *MOVE1*, and *MOVE2*. The *Vanilla* configuration is a stock LINUX system without MOVE loaded into the kernel. The *MOVE1* and *MOVE2* are configurations with MOVE loaded. On *MOVE1*, no connections are migrated and hence only connection virtualization is performed; on *MOVE2*, all connections are migrated and hence both connection virtualization and virtual-physical mapping are performed.

6.2.1 Number of simultaneous connections

For scalability test relative to the number of simultaneous connections, a number of connections are opened and the same number of requests are sent through each connection. Each request asks for a file of size 4KBytes which is locally available on each server, i.e., no NFS is involved. The total number of requests over all connections remains a constant of 1048576 (2^{20}) for each test run. For our testbed, this roughly translates into 5 minutes for each test run. We used HTTP 1.1 persistent connection in this test in order to be able to measure migrated connections for the *MOVE2* configuration.

Figure 6-55 shows the throughput test. From the figure, *MOVE1* has about at most

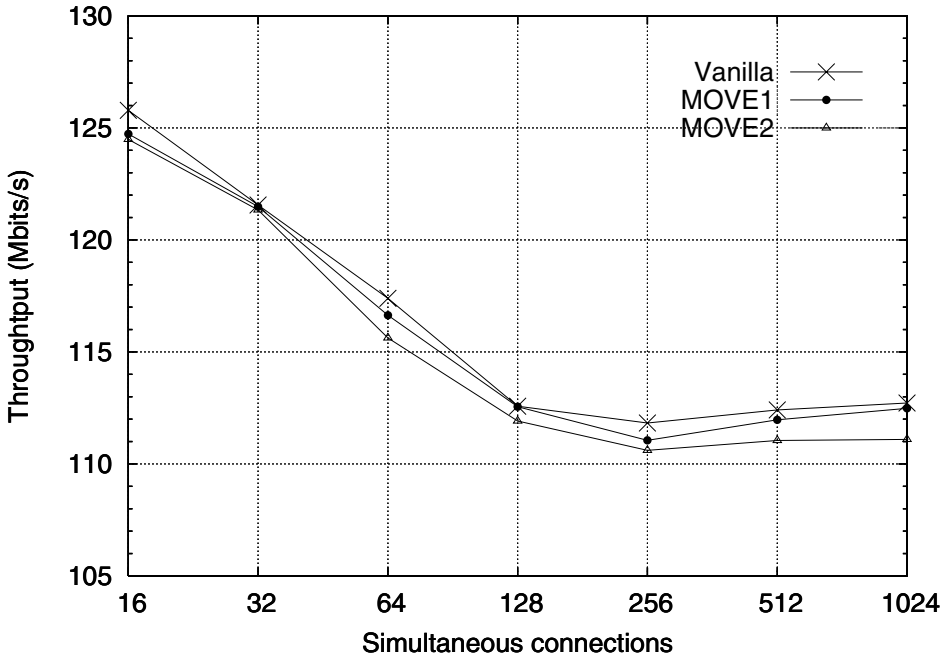


Figure 6-55. Throughput vs. number of connections

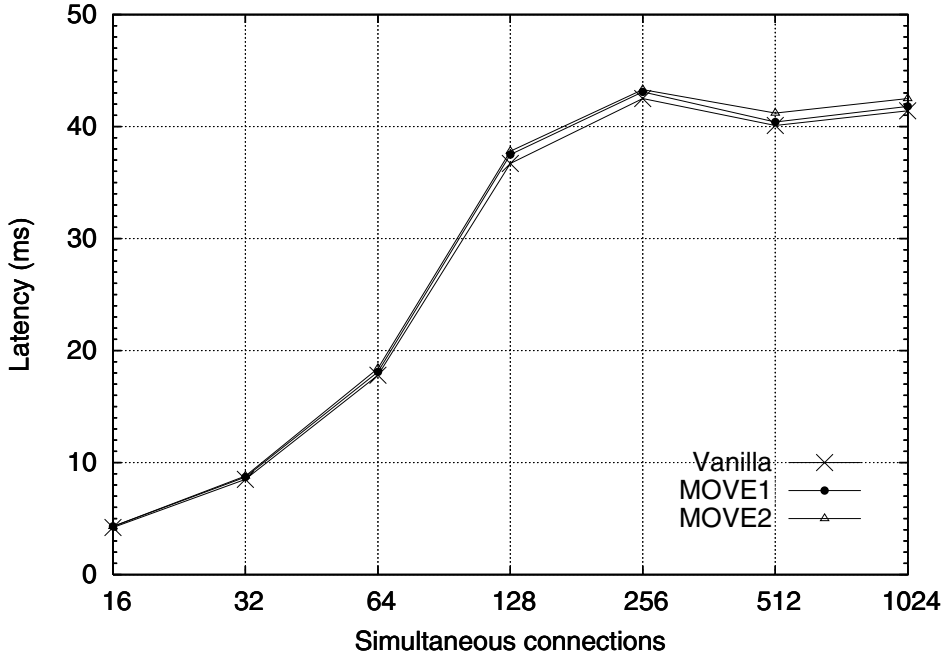


Figure 6-56. Latency vs. number of connections

1.1Mbits/second overhead over *Vanilla*, and *MOVE2* has about at most 1.8Mbits/second overhead over *Vanilla*. Figure 6-56 shows the latency test, in which *MOVE1* has about at most 0.8ms overhead over *Vanilla* and *MOVE2* has about at most 1.1ms overhead over *Vanilla*. However, the important thing to note is that, for both *MOVE1* and *MOVE2,* the throughput and latency overhead do not increase after the proxy has been overloaded at around 256 simultaneous connections.

6.2.2 Rate of new connections

For scalability relative to the rate of new connections, connections are generated at certain rate and the rate remains constant for 5 minutes. Each connection sends 300 requests, each asking for a file of size 4Kbytes available locally on each server. So again no NFS is involved. Note that for this test, we do not measure for the *MOVE2* configuration since each connection doesn't last long enough to be migrated.

Figure 6-57 shows that the throughput overhead of *MOVE1* over *Vanilla* is at most about 5Mbits/second. Figure 6-58 shows that the latency overhead of *MOVE1* over *Vanilla* is at most about 4.4ms. Again note that both throughput and latency overhead do not increase after the proxy has been overloaded at the rate of around 128 connections per second.

6.3 Connection Virtualization and Mapping Overhead

We measure the virtualization and mapping overhead of the CELL abstraction employed by MOVE. The overhead is measured using a micro benchmark pro-

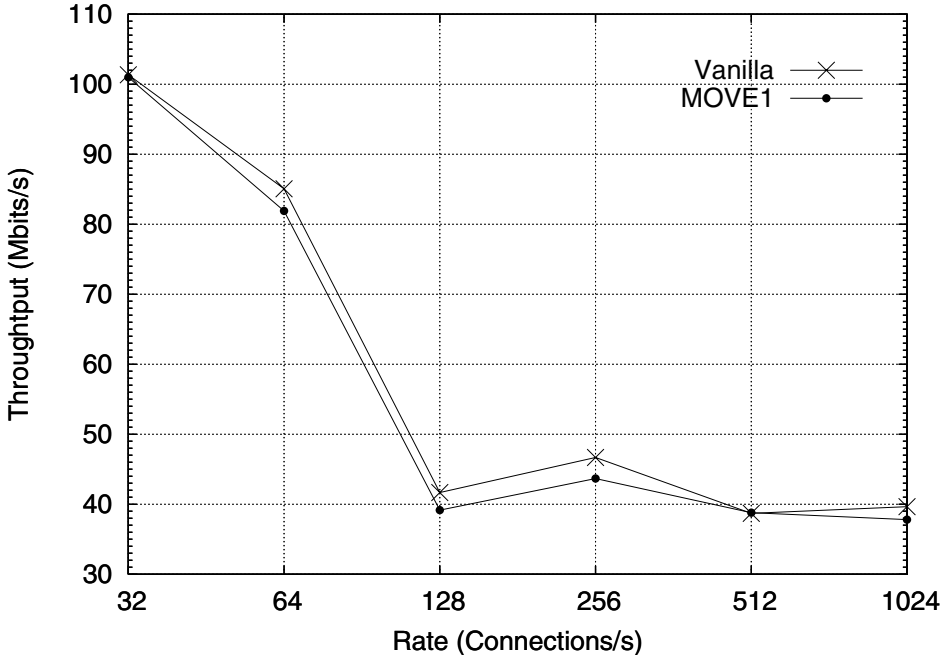


Figure 6-57. Throughput vs. rate of connections

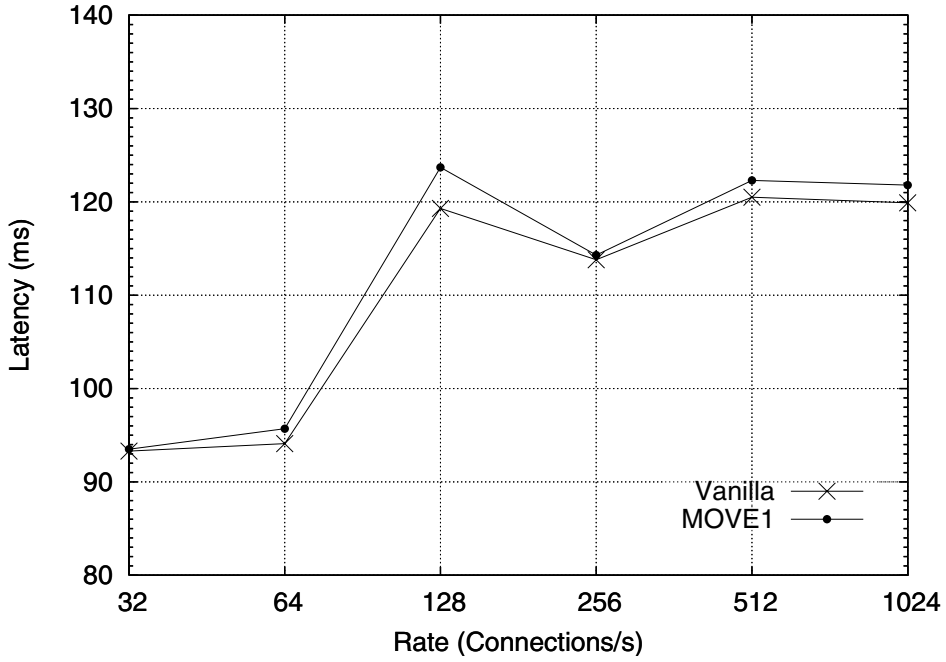


Figure 6-58. Latency vs. rate of connections

gram *netperf* [72] version 2.2pl4 on two types of systems. One system, which we call the *laptop* system, consists of a pair of IBM ThinkPad 770 laptops, each with a 233MHz Pentium CPU, 160MB RAM, and an 100Mbits LINKSYS PCM200 ethernet PCCard. The *laptop* system represents a system with low computing power. The other system, which we call the *server* system, consists of a pair of rack mounted IBM Netfinity 4500R servers, each with dual 933MHz Pentium III CPU, 512MB RAM, and an 1Gbits Intel Pro/1000MT ethernet NIC. The *server* system represents a system with high computing power. All machines are running LINUX kernel version 2.4.20. The two systems are illustrated in Figure 6-59. For both systems, we run *netperf* client and server and measure network I/O in terms of throughput, latency, CPU utilization, and connection setup for the same three different configurations, *Vanilla*, *MOVE1*, and *MOVE2*, that we used in the scalability tests in Section 6.2.

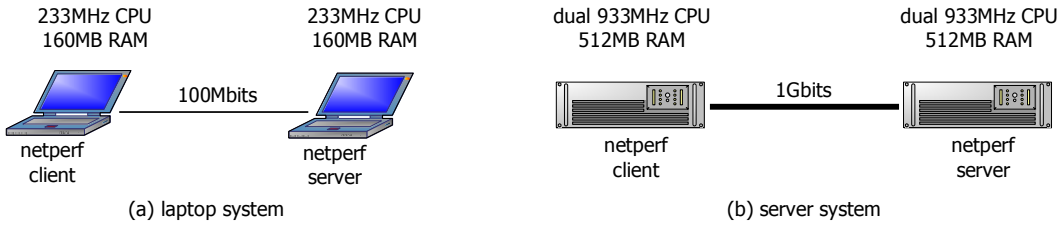


Figure 6-59. MOVE virtualization and mapping overhead testbed

6.3.1 Throughput

The throughput experiment simply measures the throughput achieved when sending messages of varying sizes as fast as possible from the client to the server. Figure 6-60 and Figure 6-61 show the throughput overhead for the two systems we

tested, each with three configurations. We can see that in both the *laptop* and *server* systems, *MOVE1* performs nearly identically to *Vanilla*. In the *laptop* system, throughput overhead is about 0.45Mbits/second, while in the *server* system, the overhead is about 1.3Mbits/second. This also shows that the overhead due to the exchanging of Diffie-Hellman public key and connection label is rather insignificant. *MOVE2* shows the throughput overhead due to the virtual-physical mapping, i.e., address translation and interface redirection, which is at most around 3.3Mbits/second in the *laptop* system, and 11.8Mbits/second in the *server* system.

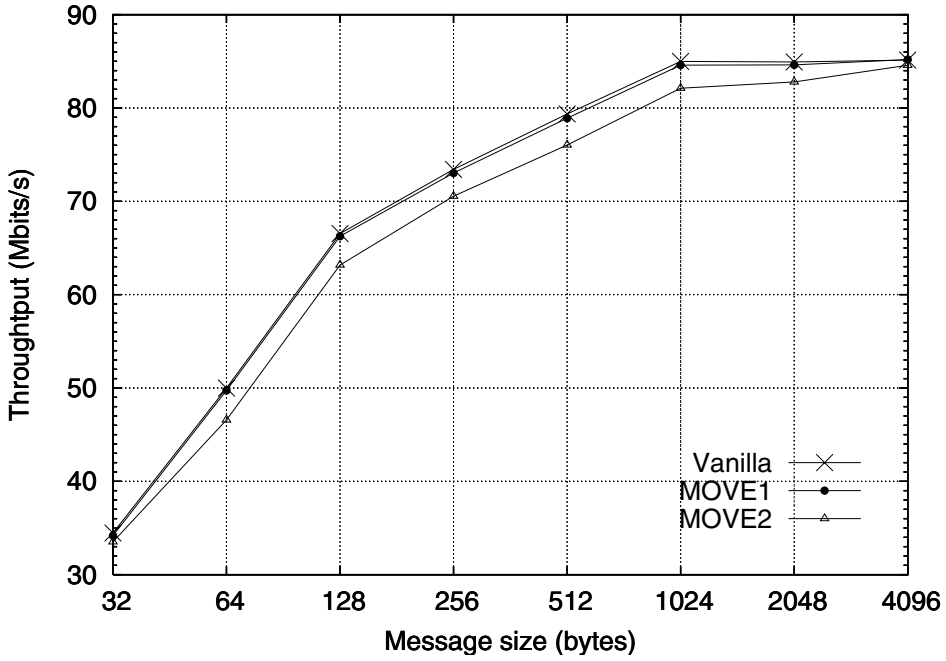


Figure 6-60. Throughput overhead, laptop system

6.3.2 Latency

The latency experiment measures the inverse of the transaction rate, where a transaction is the exchange of a request message of size 128 bytes and a reply message

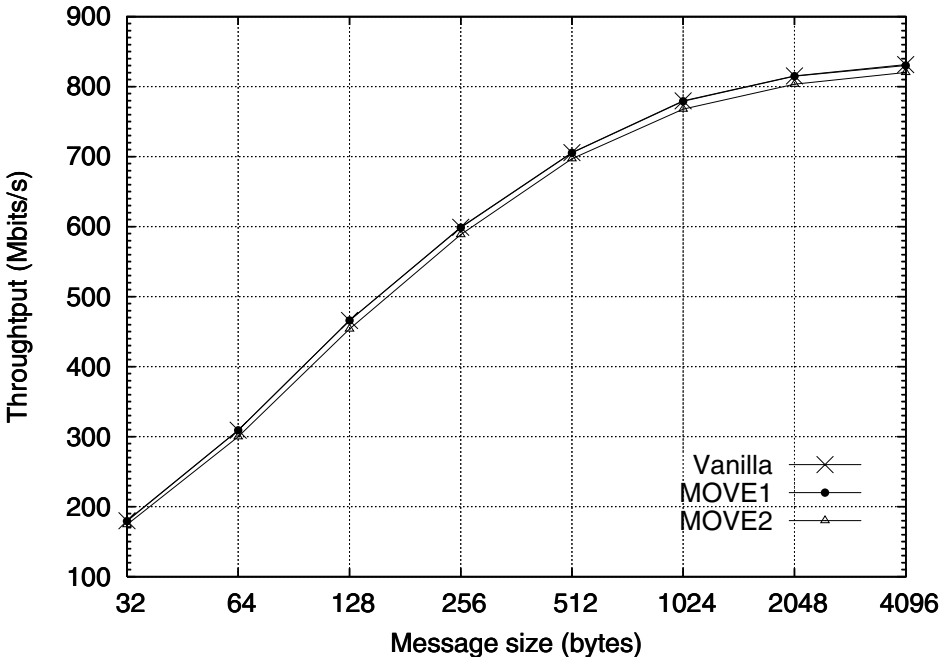


Figure 6-61. Throughput overhead, server system

of varying sizes between the client and the server over a single connection. Figure 6-62 and Figure 6-63 show the latency overhead for the two systems we tested, each with three configurations. The results bear the same characteristic as that for the throughput overhead. Performances for *Vanilla* and *MOVE1* are again rather indistinguishable, with a latency overhead of about 7.4 microseconds in the *laptop* system and 6.7 microseconds in the *server* system. Latency overhead due to the virtual-physical mapping in *MOVE2* can be observed to be at most around 34.1 microseconds in the *laptop* system and 32.5 microseconds in the *server* system.

Note that, in the *server* system, there is a strange drop of latency above the reply message size of 128 bytes. We determine that this unusual behavior is due to a problem with the LINUX device driver for the Intel Pro/1000MT NIC that was

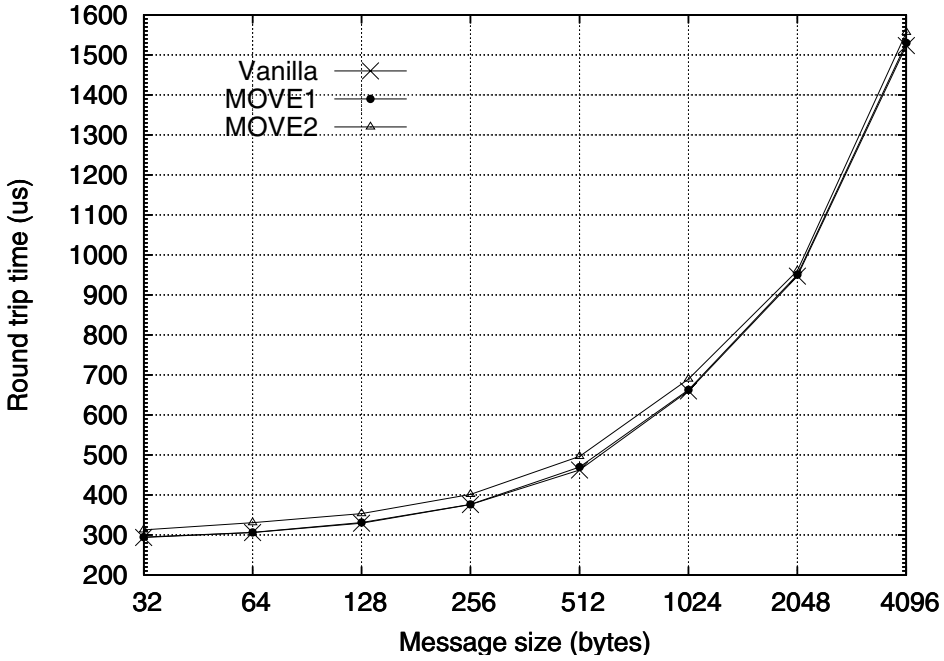


Figure 6-62. Latency overhead, laptop system

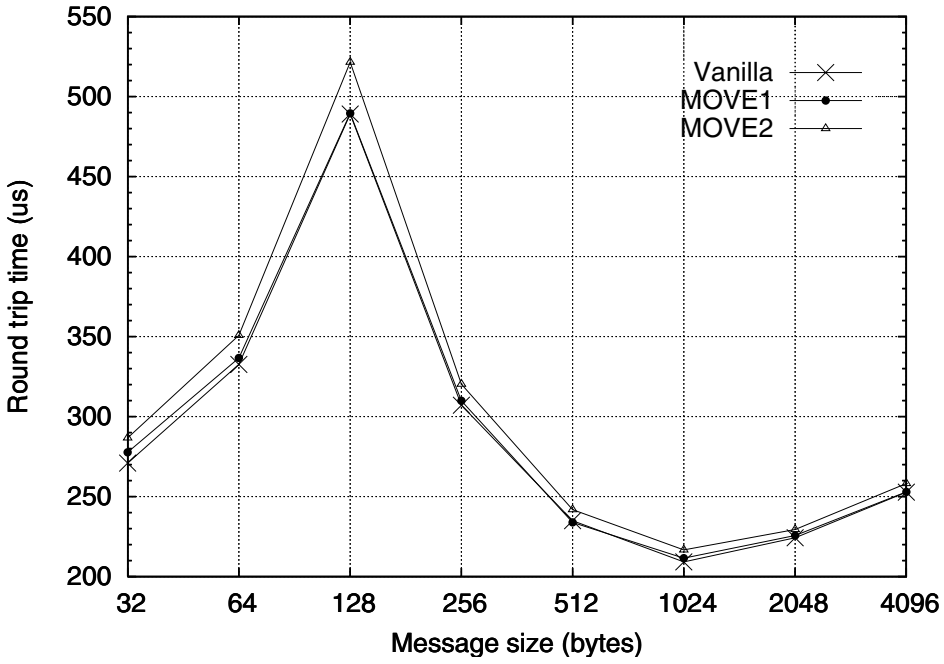


Figure 6-63. Latency overhead, server system

used. While the behavior is unusual, it does not affect the key result shown, which is the small relative performance difference between using stock LINUX and MOVE.

6.3.3 CPU utilization

Figure 6-64 and Figure 6-65 show the CPU utilization overhead measured from the throughput test on the server; results on the client are similar and omitted. We can see that for both the *laptop* and the *server* systems, *MOVE1* performs virtually the same as *Vanilla*, with a CPU utilization overhead of about 331.2 microseconds/Mbits in the *laptop* system and 39.8 microseconds/Mbits in the *server* system. The CPU utilization overhead due to virtual-physical mapping in *MOVE2* is at most around 1814.4 microseconds/Mbits in the *laptop* system and 145 microseconds/Mbits in the *server* system.

Figure 6-66 and Figure 6-67 show the CPU utilization overhead measured from the latency test on the server; results on the client are similar and omitted. For both the *laptop* and the *server* systems, *MOVE1* again performs virtually the same as *Vanilla*, with a CPU utilization overhead of about 4.7 microseconds per transaction in the *laptop* system and 0.5 microseconds per transaction in the *server* system. The CPU utilization overhead due to virtual-physical mapping in *MOVE2* is at most around 17.1 microseconds per transaction in the *laptop* system and 2.1 microseconds per transaction in the *server* system.

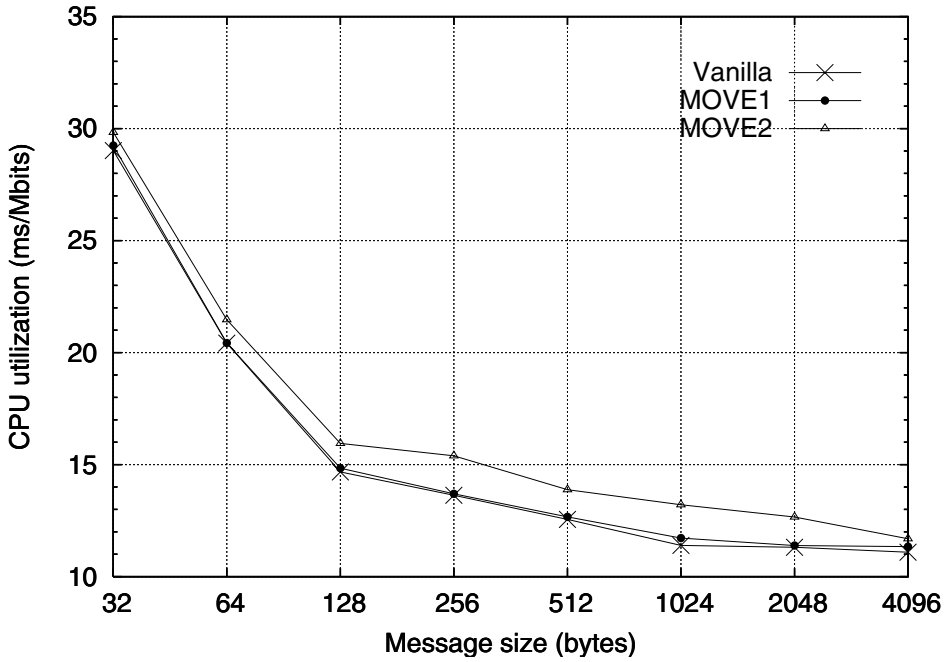


Figure 6-64. Throughput test CPU utilization overhead, laptop system

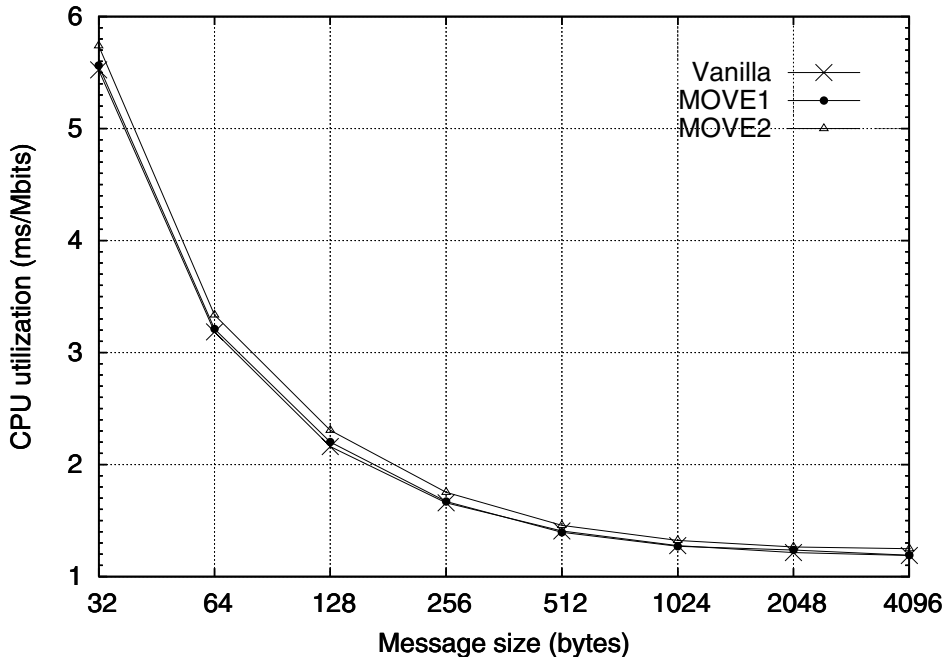


Figure 6-65. Throughput test CPU utilization overhead, server system

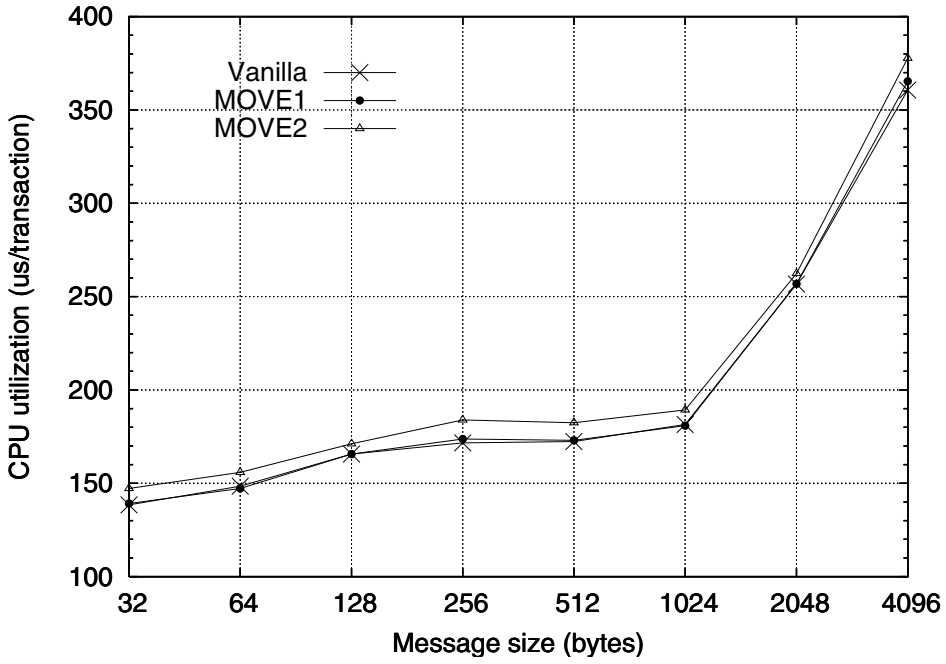


Figure 6-66. Latency test CPU utilization overhead, laptop system

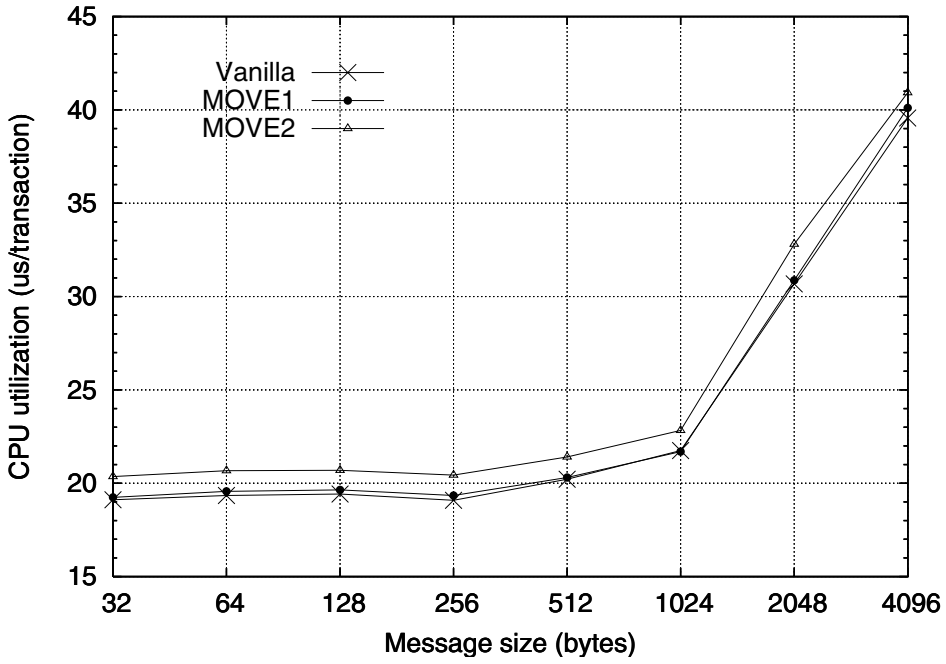


Figure 6-67. Latency test CPU utilization overhead, server system

6.3.4 Connection setup

The connection setup experiment is the same as the latency experiment except that a new connection is used for every request/response transaction. This experiment simulates the interaction between a client and a server in which many short-lived connections are opened and closed. Figure 6-68 and Figure 6-69 show the TCP connection setup overhead for *Vanilla* and *MOVE1* in the two systems we tested. Note that since connection setup occurs before migration, there is no mapping overhead associated with connection setup, therefore this measurement is not applicable to *MOVE2*. This is the test that measures the overhead of exchanging Diffie-Hellman public key and connection label. From the figure we can see that the connection setup overhead is at most around 26.2 transactions per second in the *laptop* system and 19 transactions per second in the *server* system. In the *server* system, due to the same Intel Pro/1000MT NIC LINUX driver problem as that in the latency test, we also see a strange increase of connection rate above the reply message size of 128 bytes.

6.3.5 Overhead in proxy-based environments

We finally repeat the micro benchmark for measuring virtualization and mapping overhead for proxy-based environments using the same testbed in Figure 6-38 that we used for server process handoff tests. We run *netperf* client on the client2 machine and *netperf* server on the server2 machine. We run *delegate* on the proxy machine and it forwards all connections from the *netperf* client to the *netperf* server. Client1 and server1 machines are not used in this test.

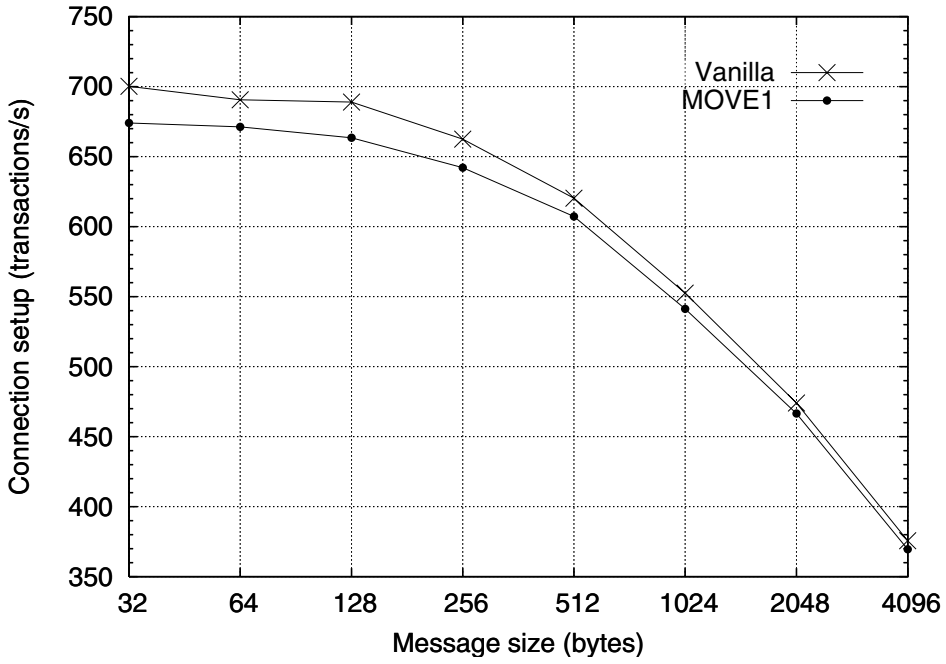


Figure 6-68. TCP connection setup overhead, laptop system

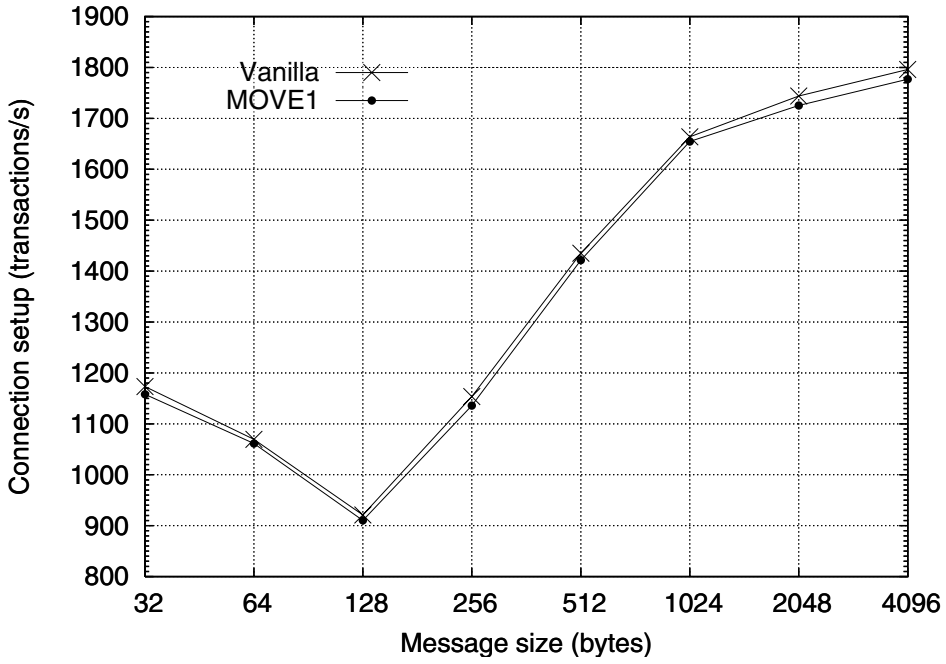


Figure 6-69. TCP connection setup overhead, server system

Figure 6-70 shows the throughput overhead for the three configurations, *Vanilla*, *MOVE1*, and *MOVE2*. We can see that *MOVE1* performs very close to *Vanilla*, with an overhead of about 1.4Mbits/second. *MOVE2* shows the throughput overhead due to the virtual-physical mapping, which is around 10Mbits/second.

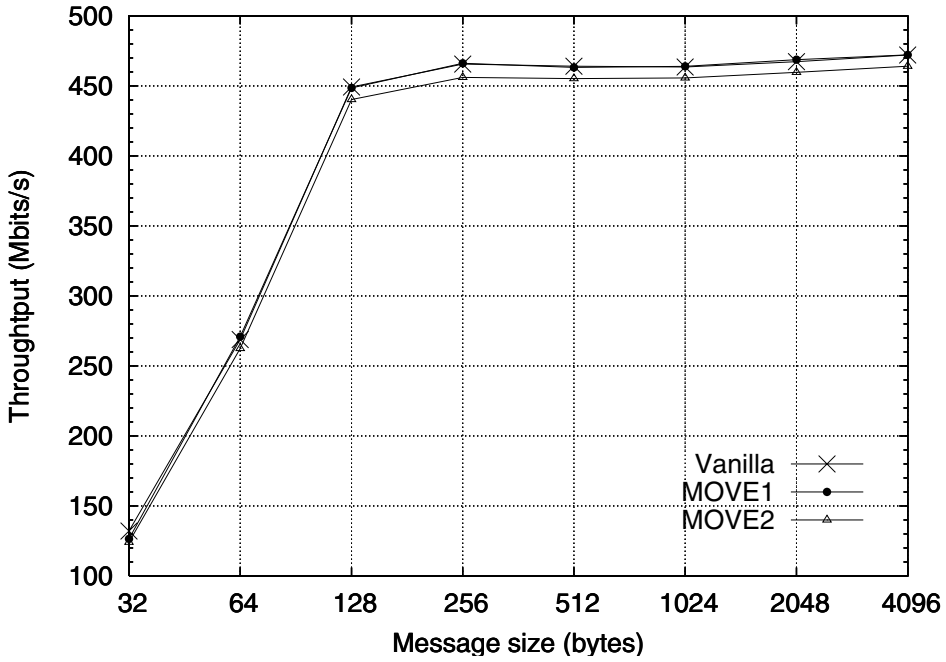


Figure 6-70. Throughput overhead

Figure 6-71 shows the latency overhead for the three configurations. The results bear the same characteristic as that for the throughput overhead. Performance difference between *Vanilla* and *MOVE1* is about 9.4 microseconds; while latency due to the virtual-physical mapping in *MOVE2* can be observed to be around 40 microseconds. Note the similar drop of latency above the reply message size of 128 bytes we also observed in the *server* system in Section 6.3.2.

Figure 6-72 and Figure 6-73 show the CPU utilization overhead measured from

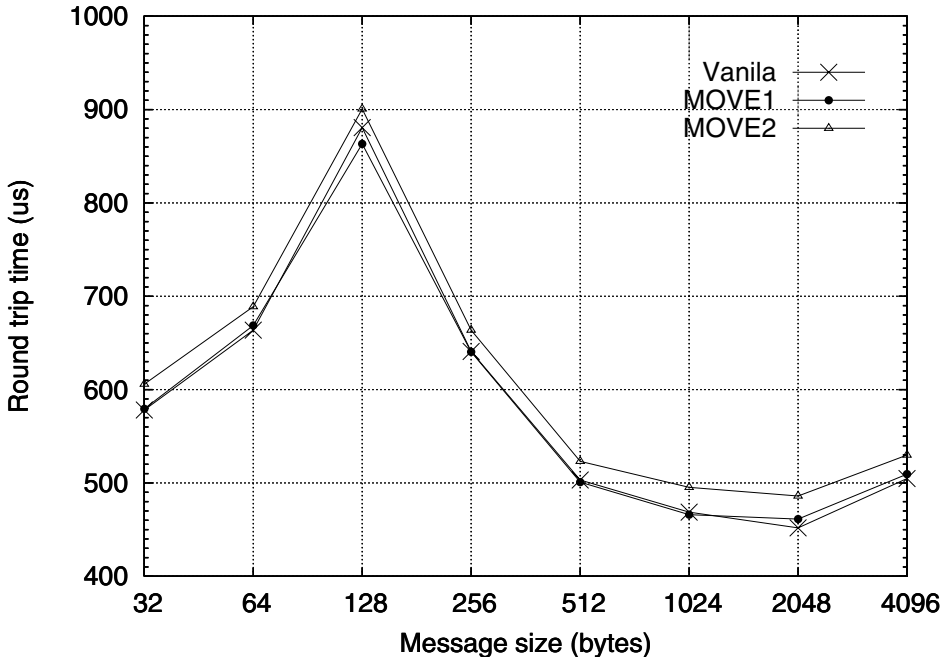


Figure 6-71. Latency overhead

both the throughput and the latency tests. From the throughput test, we can see that *MOVE1* again performed very close to *Vanilla*, with an overhead of about 44 microseconds/Mbits; and the CPU utilization overhead due to virtual-physical mapping in *MOVE2* is around 150 microseconds/Mbits. From the latency test, we can observe that the CPU overhead due to virtualization in *MOVE1* is about 0.3 microseconds per transaction; and the CPU utilization overhead due to virtual-physical mapping in *MOVE2* is fewer than 2 microseconds per transaction.

Figure 6-74 shows the TCP connection setup overhead for *Vanilla* and *MOVE1*. Note again that since connection setup occurs before migration, there is no virtual-physical mapping overhead associated with connection setup, therefore this measurement is not applicable to *MOVE2*. From the figure we can see that the over-

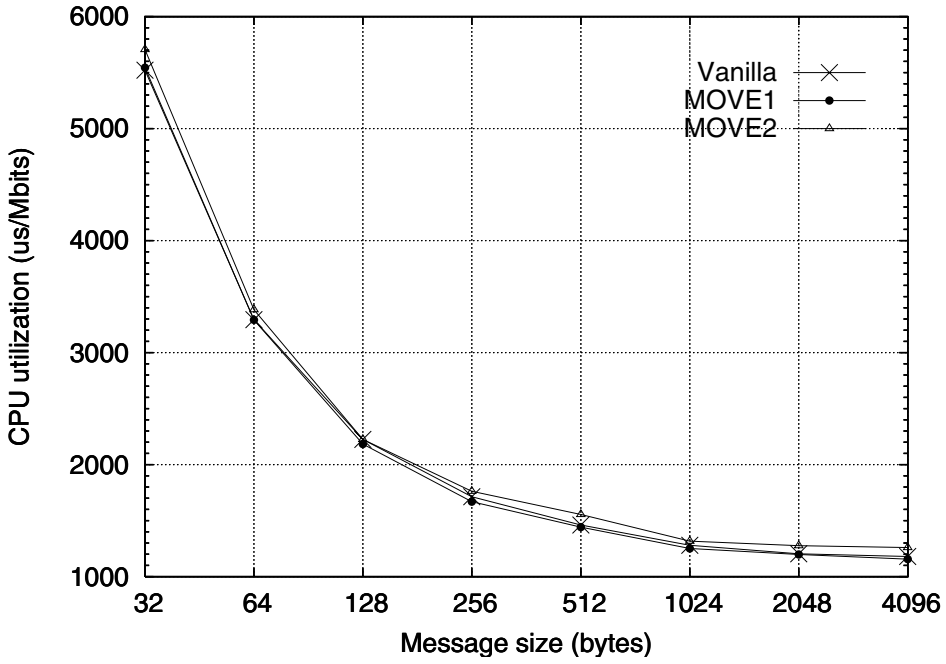


Figure 6-72. CPU utilization overhead, throughput test

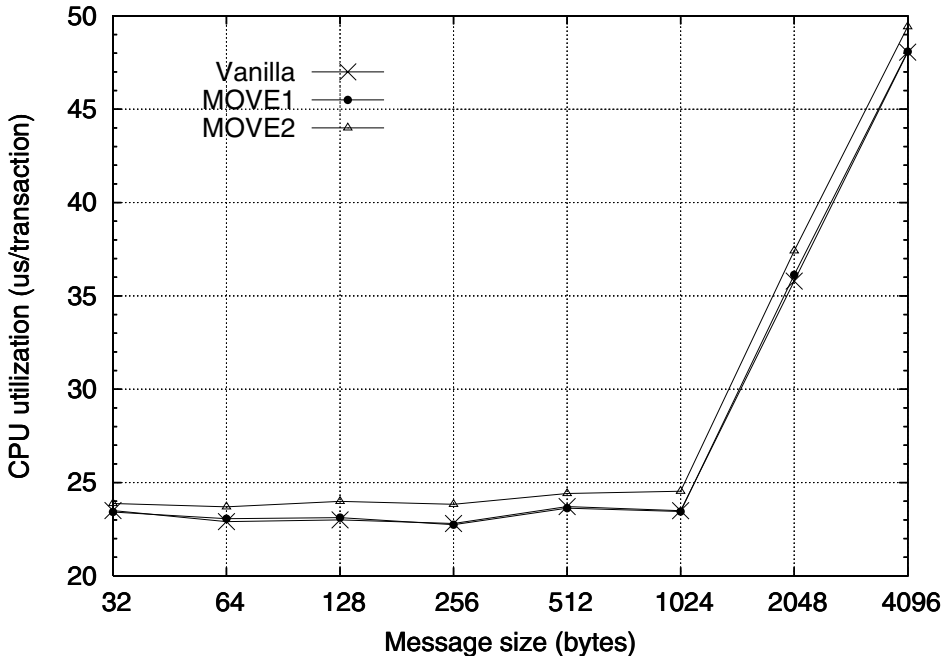


Figure 6-73. CPU utilization overhead, latency test

head is fewer than 10 transactions per second. Note the similar increase of transaction rate above the reply message size of 128 bytes we also observed in the *server* system in Section 6.3.4.

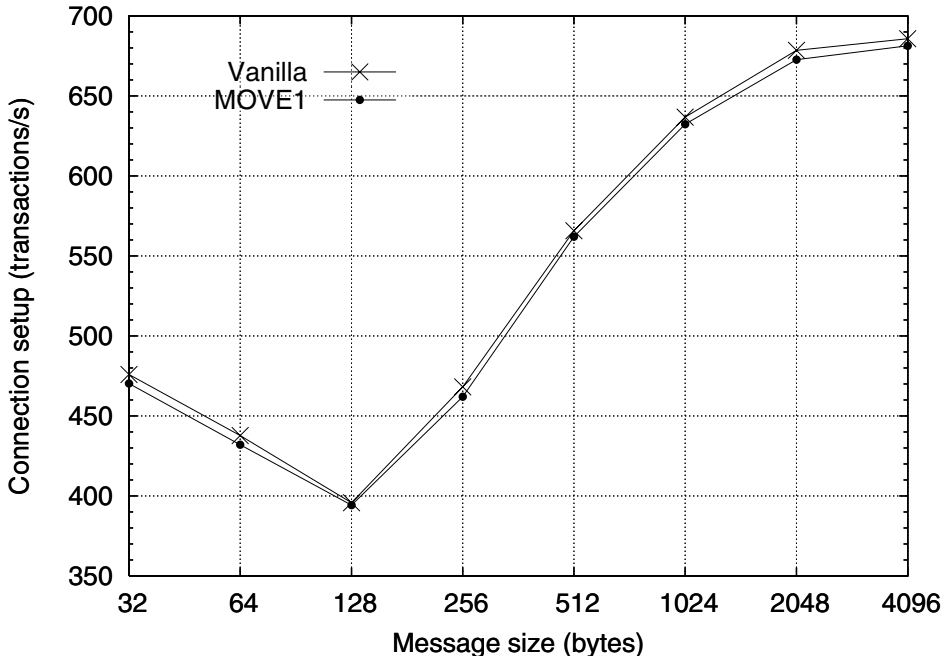


Figure 6-74. TCP connection setup overhead

6.4 Host and Service Location Mechanism Studies

We present our empirical studies of DDNS and SRV RR for their suitability as mobile host and service location mechanisms.

6.4.1 Empirical DDNS studies

The first study we conduct is to find out the TTL of name records provided by some service providers who offer DDNS for free; since the TTL determines how long a host name to IP address mapping can be cached and therefore limits the fre-

quency a host or service can migrate. We sign up with several free DDNS providers and our findings are shown in Table 6-2. We see that it's possible to have TTLs as low as 1 minute, which means that a mobile host or service can migrate as often as once every minute. We deem it adequate for most applications.

DDNS provider	Name record TTL (seconds)
`2mydns.org`	60
`dyndns.org`	60
`no-ip.org`	60
`blrf.net`	300
`myip.org`	300
`afraid.org`	3600

Table 6-2. Name record TTL of some DDNS providers

We then study if TTLs lower than 60 seconds will be honored by DNS servers and applications. To do this, we setup our own *named* [11] version 9.2.1 DNS server and domain *move.cs.columbia.edu* as a child of the Columbia Computer Science Department DNS server. Our *named* DNS server run on a Dell Dimension XPS R400 with a 800MHz Pentium II CPU, 256MB RAM, and an 100Mbits 3Com 3c905B ethernet NIC. We create name records such as `foo.move.cs.columbia.edu` on our DNS server with TTLs set to 5 seconds or 0 seconds (no caching allowed) and use a suite of popular network applications to resolve these names. The applications we use for our studies are listed in Table 6-3. All the applications can resolve the updated names according to the TTL setting. That is, with TTL=5 seconds, an update of the host name to IP address mapping is reflected to the applications after 5 seconds; and with TTL=0 second, an update is instantly reflected. For GUI based web

browsers, we do have to restart the browser for it to correctly resolve the names; simply pressing the "reload" button does not resolve the correct names since GUI web browsers typically cache a page (independent of DNS caching) for a variable amount of time, typically on the order of minutes.

Category	Application
remote login	`telnet, ssh`
file transfer	`ftp`
email	`pine, evolution, thunderbird`
web browser	`lynx, netscape, mozilla, opera, firebird`
media player	`mplayer, realplay, xmms`

Table 6-3. Applications used for DDNS TTL test

6.4.2 Transparent SRV RR lookup measurements

Because no DDNS service providers support SRV RR lookup and update, we used our own DNS server that we use for the DDNS TTL tests for the SRV RR related tests as well. On the client side, we use our transparent SRV RR lookup mechanism described in Section 5.6 in Chapter 5 with those applications in Table 6-3 that we use for the DDNS TTL test. Again, all the applications can reach their services on the correct host after the services are migrated and their corresponding SRV RR are updated on our DNS server. Similar to the DDNS TTL tests in the previous section, GUI web browsers need to be restarted to resolve the SRV RR to the correct host.

We also measure the performance overhead of the socket calls that are intercepted. We compare the execution time, the time between when the function is entered and when the function is exited, of the original socket calls with that of our version.

The measurements are performed on an IBM ThinkPad T770 with a 233MHz Pentium CPU, 160MB RAM, and an 100MBits LINKSYS PCM200 PCCard. The results are shown in Table 6-4. We can see that our `gethostbyname`/`getaddrinfo`, `getpeername` calls incur a very small amount of overhead. For `connect`, the overhead in a LAN is fairly large, about 6ms; but this overhead stays constant in the WAN case. The overhead is due to two trips to the local DNS server, one for SRV RR lookup, one for current host name lookup returned by the SRV RR.

socket call		original (ms)	intercepted (ms)
`gethostbyname`		2.28	2.31
`getaddrinfo`		74.55	77.18
`connect`	LAN	0.47	6.49
	WAN	95.41	101.59
`getpeername`		0.0086	0.0096

Table 6-4. Execution overhead of intercepted socket calls

6.5 Summary

We have presented in this chapter various performance measurements, using both application benchmarks and micro benchmarks, for our MOVE prototype implementation to demonstrate its feasibility. We showed that, with a variety of applications, endpoint migration mechanisms, network connectivity configurations, and transport protocols, MOVE's handoff protocol H2O imposes minimal impact on the end-to-end connectivity perceived by the transport protocols and applications. Our scalability tests showed that MOVE virtualization and mapping overhead stay constant with increased number of simultaneous connections or rate of

new connections. We showed that MOVE's virtualization overhead for stationary connections is essentially negligible; and for migrated connections MOVE's virtual-physical mapping overhead is also very low. We finally presented our empirical studies on the suitability of DDNS, along with the SRV RR, as the mechanism for mobile host and service location.

7 Related Work

This thesis has touched upon a few aspects of the broad mobile computing area where many prior arts exist. These aspects include: mobile communication, hand-off, service availability, and process migration. We survey these work in this chapter. We first present, in Table 7-1, a summary of the difference between MOVE and the prior arts, whose main deficiencies are indicated by the shaded cells in the table. We then present more detailed account of each aspect in the rest of the chapter.

7.1 Mobile Communication Architectures

A variety of approaches have been taken in previous work in providing communication mobility in current IP data networks. These approaches often work at a particular layer in the protocol stack and can be loosely classified as network layer solutions, transport layer solutions, application layer solutions, and split connection solutions. Despite the seemingly large variation of the mechanisms employed by these approaches, their fundamental differences can be traced back to the very mechanism that they use to address the key technical problems we described in Chapter 2: state inconsistency, conflict, and synchronization. In the comparison below, we first describe the general characteristics of each class of the solutions, followed by a more detailed look at each individual solutions within the class.

		Transparent				Non-transparent		
			Network layer					
		MOVE	MobileIP	MobileIP w/ handoff extensions	Others	Transport layer	Application layer	Split connection
Fine-grain and unlimited mobility	Fine-grain migration	Yes	**No**	**No**	**No**	Yes	Yes	Yes
	Process migration integration	Yes	**No**	**No**	**No**	**No**	**Limited[a] or No**	**No**
	Unlimited mobility	Yes	Yes	Yes	Yes	Yes	Yes	No
	Work across NAT	Yes	**Limited[b]**	**Limited**	**No**	**Yes or No**	**No**	**No**
Secure and flexible migration	Migration security	pre-computed DH	IPsec	IPsec	IPsec or No	on-the-fly DH or No	on-the-fly DH or No	SOCKS or No
	Suspension/ resumption	Yes	**No**	**No**	**No**	**No**	**Limited[c] or No**	**No**
Easy deployment	New infrastructure	No	**Yes**	**Yes**	**Yes**	No	No	Yes
	Backward compatibility	Yes	Yes	Yes	Yes	Yes	Yes	Yes
	Transport change	No	No	No	No	**Yes**	No	No[d]
	Transport dependency	No	No	No	No	**Yes**	**Yes**	**Yes**
High performance	Connection setup overhead	Low	Low	Low	Low to High	Low to High	**High**	Low to High
	Pre-move I/O overhead	Low	Low	Low	Low to Low+	Low	**High**	Low[e]
	Post-move I/O overhead	Low+	Low+ to High	Low+ to High	Low+	Low	**High**	Low
	Handoff latency	0.5 *RTT*[f]	**O(*RTT*)**	**intra-domain: O(*rtt*[g]) inter-domain: O(*RTT*)**	**O(*RTT*)**	**1.5 or 2 *RTT***	**> 1.5 *RTT***	**2.5 *rtt* or 0.3-1.4 seconds**

Table 7-1. Comparison of MOVE and other mobility solutions

a. ROCKS/RACKS [139] supports checkpointing/restarting MPI based applications

b. With RFC3519 extension [85], only works from no NAT to NAT

c. ROCKS/RACKS can prevent TCP from timing out but not applications

d. While I-TCP [34] doesn't modify TCP, it violates TCP end-to-end semantics

e. Consider connection between mobile endpoint and MSR (mobility support router, for I-TCP)/proxy (for MSOCKS [89]) only; same for post-move I/O overhead

f. *RTT*: round trip time between mobile endpoint and stationary endpoint or between mobile endpoint and home agent (for MobileIP [71][104])

g. *rtt*: round trip time between mobile endpoint and MAP (mobility anchor point, for handoff extension architectures) or between client and proxy (for MSOCKS)

7.1.1 Network layer solutions

Network layer solutions preserve connection states at both transport-and-above layers and network-and-below layers across migration. Therefore they address the inconsistency problem in a way similar to MOVE. However, their "virtual namespace" is only provided to the entire host rather than individual connections. Therefore they can only provide migration at the granularity of an entire host rather than individual connections. Network layer solutions must also deal with state conflict in the transport layer due to address and port reuse; and individual solutions differ in the way they address this problem. None of the existing network layer solutions addresses the cross address space synchronization problem, with the exception of MobileIP, to which an extension has been proposed to partially address the problem. Existing network layer solutions either do not address security problem or rely on IPsec. And with the exception of MobileIP, none has presented studies on its handoff behavior.

MobileIP is the most well-known network layer mobility mechanism; recent versions [71][104] have consolidated various improvements to the original proposals [38][70][103]. MobileIP supports its "virtual namespace" by assigning each host a "home" IP address that is carried along with the host wherever it moves. It resolves the conflict problem by requiring that the assignment of the home IP address is permanent and non-reusable, which needs a global infrastructure. The main problem with MobileIP, however, is the use of the home IP address for both locating and tracking the host, which are two different issues. As a result, it mandates the requirement of additional network infrastructures (i.e., home/foreign

agents), which history has shown to be extremely difficult to deploy, even when there is no need for them. For example, the most common mobile hosts today are client machines that are never used as server machines, which means there was never a problem of locating them; the mobility problem in this case is entirely just the tracking problem. And we've shown that you don't need any new network infrastructure for tracking. The coupling of locating and tracking also results the "triangle routing" (aka dog-leg routing) problem, which can be addressed [105] but it adds even more complexity to the architecture. [85] proposes an extension to partially address the synchronization problem in the case a mobile host moves from a public network without a NAT to a private network behind a NAT. But the solution requires more infrastructure support and has unresolved security issues. MobileIP's handoff performance is generally considered inadequate due to the need for the mobile host to interact with the home agent on every move; its security mechanism in the absence of IPsec, called return routability procedure, also adds to infrastructure complexity and handoff delay.Therefore a large body of work has been proposed to improve its handoff performance, which we discuss in the next section.

A few other network layer solutions that use network address virtualization have been previously considered [66][131][132][136]. [131][132] supports its "virtual namespace" with a constant virtual IP (VIP) address that is set to be the initial physical IP address of a host in its native network. However, they did not address the conflict problem. [131][132] also employs a more complex virtual-physical mapping mechanism that requires network infrastructure support and host OS

kernel changes. [66] describes a way to implement a MobileIP equivalent service using its redirection mechanism. It shows how new connections are redirected but does not prescribe a way to migrate existing connections. The same per host permanent home IP address "virtual namespace" is assumed to avoid the conflict. [66] requires extra round trip delays to determine virtual-physical mapping for all connection setup, regardless of wether a machine moves or not. [66] rewrites TCP stack values which requires TCP/IP stack changes and makes it dependent on not only TCP but also the TCP implementation as well. [136] employs reserved E-class IP addresses as the VIP for its "virtual namespace". This avoids conflict since the E-class addresses are assumed to be never used as hosts' physical IP addresses. However, the limited number of usable E-class addresses causes a few problems: (1) extra round trip delays to negotiate VIPs for all connection setup, regardless of whether a machine moves or not; (2) possibility of conflict between VIPs since there are more machines then the number of VIPs. [136] by default doesn't address this issue but rather rely on the low possibility of the conflict happening. It provides an option to reserve a static physical IP as the VIP to avoid the conflict, which is equivalent to MobileIP's permanent home IP address approach; and (3) complexity of managing VIPs such as selection and garbage collection policies. None of these approaches addresses the synchronization problem and presents studies of their handoff behavior. Among them, only [136] considers security issues that arise due to host mobility; it assumes IPsec which is not yet widely deployed.

ROAM [142] is a host mobility approach using a peer-to-peer overlay network called *i3*. Communications on *i3* are commenced with a rendezvous-based abstrac-

tion where all packets are of the form (*id*, *data*) and the *id* is location independent. While the overlay network possesses interesting properties pertinent to mobility, ROAM suffers from some of the same problems that MobileIP does. First ROAM requires the deployment of *i3* servers to form the overlay. Second, similar to MobileIP's home address, the *id* in *i3* is used not only just for locating a mobile host, but also for routing packets to the mobile host. The *i3* overlay therefore has the same problem of coupling the issues of host locating and connection tracking. The efficiency of ROAM to support *i3*-unaware legacy applications is likely to suffer because to transmit a packet, it must be converted from IP address namespace to the *i3 id* namespace, routed on the overlay, and then converted back from *i3 id* namespace to IP address space. The authors have suggested that one can potentially use *i3* only for control traffic (e.g., exchanging new IP addresses when hosts move) but leave data traffic for the underlying physical network in order to reduce the overhead. However, doing so also reduces the functionality of the entire *i3* overlay down to the equivalent of a DDNS.

[96] proposes an interesting approach that exploits the similarity between mobility and multicasting. Unfortunately, a scalable multicast infrastructure does not yet exist today.

7.1.2 Transport layer solutions

Transport layer solutions do not preserve all connection states at transport layer, therefore the migration is no longer transparent to the transport protocols. As we pointed out in Chapter 2, state inconsistency, conflict, and synchronization prob-

lems do not apply to non-transparent migration solutions. Instead, the mobility functions are provided by modifying the transport protocol, TCP in this case, itself.

TCP-R [58], Migrate [121][122] and M-TCP [128] are transport layer solutions that "re-synchronize" some of the TCP states (e.g., tuple, sequence number, etc.) at the new location by modifying TCP on both endpoints. As a result, TCP-R, Migrate, and M-TCP cannot be used with existing unmodified transport protocol stack implementations. M-TCP also requires server application (the endpoint that moves) change.

TCP-R describes two security modes, called optimistic and pessimistic approach. The optimistic approach relies on TCP sequence number for rudimentary protection therefore is insecure. The pessimistic approach uses public key encryption and the keys are exchanged at connection setup time, similar to the H2O Diffie-Hellman key exchange. However, the keys are exchanged using TCP options therefore it supports TCP only; there are also no details given for the approach. The security mechanism of Migrate is based on Elliptic Curve Diffie-Hellman key exchange but the computation of secret key is performed per connection at connection setup time, which results in high connection setup overhead. A connection certificate is mentioned in M-TCP but no discussion on its use for security.

TCP-R and Migrate require 1.5 RTT for connection handoff while M-TCP requires 2 RTT. Although TCP-R is capable of individual connection handoff, it is designed in the context of host migration and compared against MobileIP. Therefore, no process migration is mentioned in TCP-R. Neither Migrate nor M-TCP is inte-

grated with process migration either. They rather assume that an "identical" process already exists on the target host. This necessarily restricts them to "transactional" type of applications where the application states for processing individual transactions are easily duplicated. For example, an HTTP connection through which a file is being requested can be migrated by simply recreating the transport connection states at the "identical" process on the target host without migrating any application states, as long as the same file is also available on the target host.

Among the three solutions, only Migrate can work across NAT boundaries.

Emerging transport protocols such as SCTP (Stream Control Transmission Protocol) [127] have mobility-savvy features such as support for multihoming, which allow an endpoint to use multiple IP addresses for a connection. An SCTP extension described in [126] further allows an endpoint to dynamically add and delete IP addresses associated with a connection, therefore making SCTP a mobility enabled transport protocol [114]. This is indeed a welcome sign that new transport protocols have started to take mobility into consideration. Current SCTP mobility support, however, has a few limitations:

- New IP addresses can be added only through an existing connection, which means that the mobile endpoint must have simultaneous connections to the stationary endpoint from both the old and the new IP addresses. In a network such as WiFi where one interface can only be associated with one access point at a time, this requires the mobile endpoint

have at least two interfaces for mobility support.

- Since only IP addresses, not port numbers, can be dynamically changed for a connection, SCTP mobility support cannot work across NAPT.
- SCTP mobility support does not address the security problem itself, but rather leave it up to IPsec.

Note that MOVE can also readily take advantage of multihoming by transparently migrating connection(s) from one interface to another as we've shown in Section 6.1.1 in Chapter 6 when we migrated a connection between a WAN and a LAN.

7.1.3 Application layer solutions

Application layer solutions do not preserve any connection states at transport layer and therefore are also non-transparent migration solutions for which state inconsistency, conflict, and synchronization problems do not apply, as we pointed out in Chapter 2. Unlike transport layer solutions, however, the mobility functions are provided not by modifying the transport protocols themselves but rather by emulating the migration through closing the old connection and opening a new one.

All application layer solutions, such as [98][109][139][141], are based on introducing a "shim" layer, generally a socket library wrapper, between the application and the transport protocol (again TCP) to emulate the migration. Because in-flight data, those that have been acknowledged by TCP but not yet delivered to the application, can be lost due to closing the old connection, these solutions must

employ double-buffering and go-back-N (or similar) mechanisms to recover lost in-flight data. These mechanisms essentially duplicate many of TCP's functions and create substantial network I/O performance overhead [139], not only for migrated connections, but also for stationary connections as well.

Of these solutions, [141] and [109] do not address migration security issues. [98] mentions a "seed" for migration authentication but no details are given. Only [139] provides security protection for migrating connections, which is based on the Diffie-Hellman key exchange. Like Migrate, it computes the shared secret key per connection at connection setup time and has high connection setup overhead.

While none of these solutions presented studies of their handoff behavior, one can infer that they will require at least 1.5 RTT in order to open a new connection. The go-back-N for recovering lost in-flight data and authentication (if provided) will add additional handoff delay.

[139] is integrated with MPICH [64] and supports checkpointing/restarting connections for MPI [13] based applications. [141], [109], [98] are not integrated with process migration. In addition, [98] is a pure Java [62] based socket library and supports connection migration for Java applications only.

Lastly, none of these solutions, or application layer solutions in general, can work across NAT boundaries. This is because application layer solutions rely on the ability of the mobile endpoint, after it moves, to establish a new connection to the stationary endpoint. However, if the stationary endpoint is behind a NAT device,

which masquerades the IP address and port number of the stationary endpoint, establishing a new connection is not always possible. For example, the stationary endpoint may be a client behind a NAT device which initiates a connection to a mobile server; after the server moves, it cannot connect back to the client due to the NAT device.

7.1.4 Split connection solutions

I-TCP [34] splits a TCP connection between a mobile host (MH), which is assumed to be on a wireless network, and a fixed host (FH), which is assumed to be on a wired network, into two connections using a mobile support router (MSR), which is a special base station with I-TCP support. The split is such that it's transparent to the FH but not transparent to the MH; much like the way NAT works. The split of a connection offers two advantages:

- It separates TCP performance characteristics such as flow control and congestion control on the wireless link from those on the wired link. The separation is desirable because it allows separate control on the two vastly different type of links. For example, a different version of TCP that is specially tuned for the wireless link such as [35][41] or even a non-TCP transport protocol can be used between the MH and the MSR to improve the performance of the overall connection.
- It allows the MSR to hide the movement of the MH from the FH by transferring TCP states and socket buffers of the MSR-FH connection to a new MSR and reestablishing a new MH-MSR connection after the MH moves.

I-TCP, however, has a few serious drawbacks. First, it needs network infrastructure support for the MSR. Second, the MSR breaks the end-to-end TCP acknowledgement semantics because it separates acknowledgements for the wireless MH-MSR part and the wired MSR-FH part of the connection, i.e., the MSR acknowledges packets sent from the FH to the MH before delivering them to the MH. As a result, the MSR must buffer packets for the MH and can easily run out of buffer space since the FH has no idea that it's talking to a slow wireless MH. In other words, the split connection is a double-edged sword. Third, every time the MH changes the MSR, TCP states and socket buffers must be transparently transferred to the new MSR. This is a rather complex task and requires careful kernel design and networking code modification on the MSR. As shown in [34], transferring large socket buffers (32KB) can take as long as 1.5 seconds, which severely impacts its handoff performance. Note that large sockets buffers are needed to prevent the MSR from running out of buffer space. Finally, I-TCP does not address migration security and process migration issues, and is limited to client mobility only.

MSOCKS [89] is another split connection solution which is based on a proxy utilizing the TCP Splice [88] technique. A proxy transparently splices a single TCP connection between a client and a server. It handles the disconnecting and reconnecting of the client-proxy half while maintaining the proxy-server half intact in the face of client migration. MSOCKS can be classified as transport layer solution since TCP Splice is TCP specific. MSOCKS can also be classified as an application layer solution. Because the client-proxy half connection is "migrated" by closing the old one and opening a new one, the same way application layer

solutions "migrate" a connection. MSOCKS library must emulate the migration to the client and the proxy must emulate the migration to the server. Therefore, MSOCKS suffers from the same problems as those with application layer solutions. Indeed, one can think of MSOCKS as another form of application layer solution, where the proxy plays the role of the socket library wrapper for the server. Comparing to the socket library wrappers, the advantage of MSOCKS is that it doesn't have to touch the server. However, it requires kernel change on the proxy and supports only client mobility. The scope of client mobility is also restricted to the same proxy. MSOCKS relies on SOCKS [84] for its security and requires 2.5 RTT between the client and the proxy for handoff delay.

7.1.5 Summary

To summarize and as a comparison, MOVE is a transparent migration architecture, similar to network layer solutions. Unlike network layer solutions, however, MOVE's virtual namespace extends to individual connections therefore MOVE provides migration at the granularity of individual connections yet still remains transport-independent. MOVE addresses state inconsistency, conflict, and synchronization problems through a light-weight virtualization, privatization, and labeling mechanism that requires no explicit management of virtual address space, and incurs no delay to connection setup and no mapping when connections do not move. MOVE handoff is secure and its delay is 0.5 RTT, i.e., a single one-way trip between the two communication endpoints. MOVE is integrated with a general purpose process migration mechanism that supports migration of legacy applications running on commodity OS without requiring changes to either the applica-

tions or the OS. To some extent, one may classify MOVE as a multi-layer solution since the mechanisms for supporting the CELL namespace abstraction and H2O handoff protocol happen at multiple layers, as illustrated in Figure 5-1 in Chapter 5.

7.2 Handoff Mechanisms

Numerous approaches have been proposed to provide better handoff support for the original MobileIP. These work follow the same general idea: introduce a MAP (mobility anchor point) that is close to the mobile host (MH) so that traffic can be redirected more quickly without involving the home agent (HA) when the MH moves within the same micro-mobility domain under the MAP. [51][69][82][123][137] are direct extension to the MobileIP, while [42][46][110] define their own micro-mobility domain with proprietary routing protocols. We give a brief description of these approaches. We note that all the approaches fall back to MobileIP when the MH moves across MAPs.

7.2.1 Extensions to MobileIP

Low Latency Handoffs in MobileIPv4 [51] assumes the availability of an "advance notice" from layer 2 before the current link is dropped; it also assumes the new foreign agent (FA) is known before the handoff. It suggests three ways for low-latency layer 3 handoff. The first is called pre-registration, which registers the MH with the new FA (nFA) through the old FA (oFA) before the layer 2 handoff. The second is called post-registration, which sets up a tunnel between the oFA and the nFA to

forward traffic to the MH after it has moved the nFA but not yet registered. The third combines both methods one and two. It performs the advance registration and tunnel setup in parallel. If pre-registration can be completed before the layer 2 handoff, no forwarding is necessary; otherwise, traffic for the MH is forwarded as specified in the post-registration method.

Fast Handovers for MobileIPv6 (FHMIPv6) [82] is essentially the low latency handoffs for MobileIPv4, except that it always use the third method above even when pre-registration is successful so the nFA can buffer in-flight packets tunneled to it by the oFA during MH's layer 2 handoff. It also provides a simple way to handle the case when the "advance notice" from layer 2 and the nFA are not available before the layer 2 handoff.

Hierarchical MobileIPv6 mobility management (HMIPv6) [123] organizes FAs into a tree hierarchy and assigns each MH with two Care-of Addresses (CoA): a regional CoA (RCoA) and an on-link CoA (LCoA). This allows an MH to perform a "regional registration" [67] with the "crossover" FA, the lowest common ancestor of the oFA and the nFA. Essentially, the crossover FA, functioning as the MAP, becomes the "local HA" for the MH which uses the RCoA as the "local home address" for the MH and performs binding between the RCoA and the LCoA. The original HA performs binding between the home address (HoA) and the RCoA. When the MH moves within a domain under the same MAP, no binding update by the HA is necessary since the RCoA does not change. This way, the registration traffic is confined within a local domain, eliminating the delay of having to register

with a distant HA every time the MH moves.

S-MIP [69] combines HMIPv6 and FHMIPv6, adding a movement tracking component that is not covered by FHMIPv6 and a hybrid handoff mechanism that is lossless. Movement tracking is done by adding a Decision Engine (DE) in the network that maintains, through periodical feedback from access routers, a global view of the connection states of any mobile devices in the domain, as well as their movement pattern. The hybrid handoff mechanism is termed as "mobile node initiated but network determined" since supposedly the MH has the best idea of when to move and the network has the best idea of where to move. It relies on simultaneous binding and knowledge of the nFA to allow packets to be bi-casted to both the oFA and the nFA to prevent packet loss. However, as with all bi-cast schemes, it must handle the packet duplicate and reordering problem created by the two separate packet streaming to the MH: one is forwarded via the oFA, the other is directly routed to the nFA. The proposed Synchronized-Packet-Simulcast (SPS) scheme assumes some type of sequencing capability of access routers. This capability is achieved by marking packets with a special bit and having the access routers maintain two separate buffers for the marked and unmarked packets.

[137] is the only approach that actually tries to make as little change to standard MobileIP as possible. Unfortunately, their approach still requires adding a special layer 2 bridge to connect the two WiFi networks between which an MH moves. By default, the bridge does not forward traffic between the two networks since they are two different layer 3 networks. Once an MH moves from one network to the

other, the bridge will "snoop" packets destined for the MH on the old network and forward them to the new network. The bridge essentially functions as a hardware tunnel between the oFA and nFA during the period of standard Mobile IP handoff. It of course will only work in a broadcast network such as WiFi. And all networks the MH will potentially visit must be connected by the special bridge.

7.2.2 Domain-based solutions

HAWAII [110], Cellular IP [42], and EMA [46] all share the same basic architecture design, know as the domain-based architecture that separates the notion of micro-mobility and macro-mobility. Each approach defines its own domain and provides specialized routing and handoff signaling support for micro-mobility within the domain. They all assume standard MobileIP for the macro-mobility across domains. Their differences only lie in the details of how a path to an MH is setup and the handoff procedure.

Within a HAWAII domain, traditional IP routers are extended with mobility support functions that handle specialized control messages to setup host-based routes to MHs. A hierarchical structure of routers similar to that of HMIPv6 is employed so path setup messages only travel as far up in the hierarchy as to the closest cross-over router. Two path setup schemes are proposed. In the forwarding path setup scheme, old base-station (oBS) forwards traffic to the new base-station (nBS) until the crossover router starts to divert traffic directly to the nBS. In the non-forwarding path setup scheme, the crossover router simply diverts traffic (once it receives the path setup control message) either by unicast to the nBS (if the MH is able to

communicate with multiple base-stations during handoff) or by bi-cast to both the oBS and the nBS (if otherwise).

Within a Cellular IP domain, IP routers are replaced by special base-stations using a hop-by-hop routing scheme that resembles to the layer 2 spanning-tree packet forwarding scheme. The domain gateway periodically flood the network with a beacon so the base-stations can build a spanning tree rooted at the gateway. Base-stations maintain soft state routing cache to MHs by observing traffic from the MHs to reduce explicit control messages. A special route-update packet is used by MHs to immediately refresh the routing cache of upstream base-stations after it moves. Two handoff procedures are described. In the Hard Handoff, the MN simply sends a route-update packet to refresh the routing cache in the routers between the nBS and the crossover router. In the Semisoft Handoff, it's assumed the MH is able to communicate with the nBS before losing connection to the oBS. The MH will send a route-update message to the nBS before handoff so the nBS can start the process of refresh routing cache towards the crossover router before the MH arrives.

Within an EMA domain, a new routing algorithm called TORA (Temporally-Ordered Routing Algorithm) is proposed that supports both prefix-based routing for stationary base-stations and host-based routing for MH. The details of TORA and how it is used to support handoff is presented in [46]. Two handoff procedures, break-before-make (BBM) and make-before-break (MBB), are considered. In both cases, a temporary tunnel between the oBS and nBS is attempted before the

MH breaks its connection to the oBS and the tunnel is torn down (if it was created) after the MH makes its connection to the nBS. The tunnel is not really needed in the case of MBB but is attempted anyway as a backup.

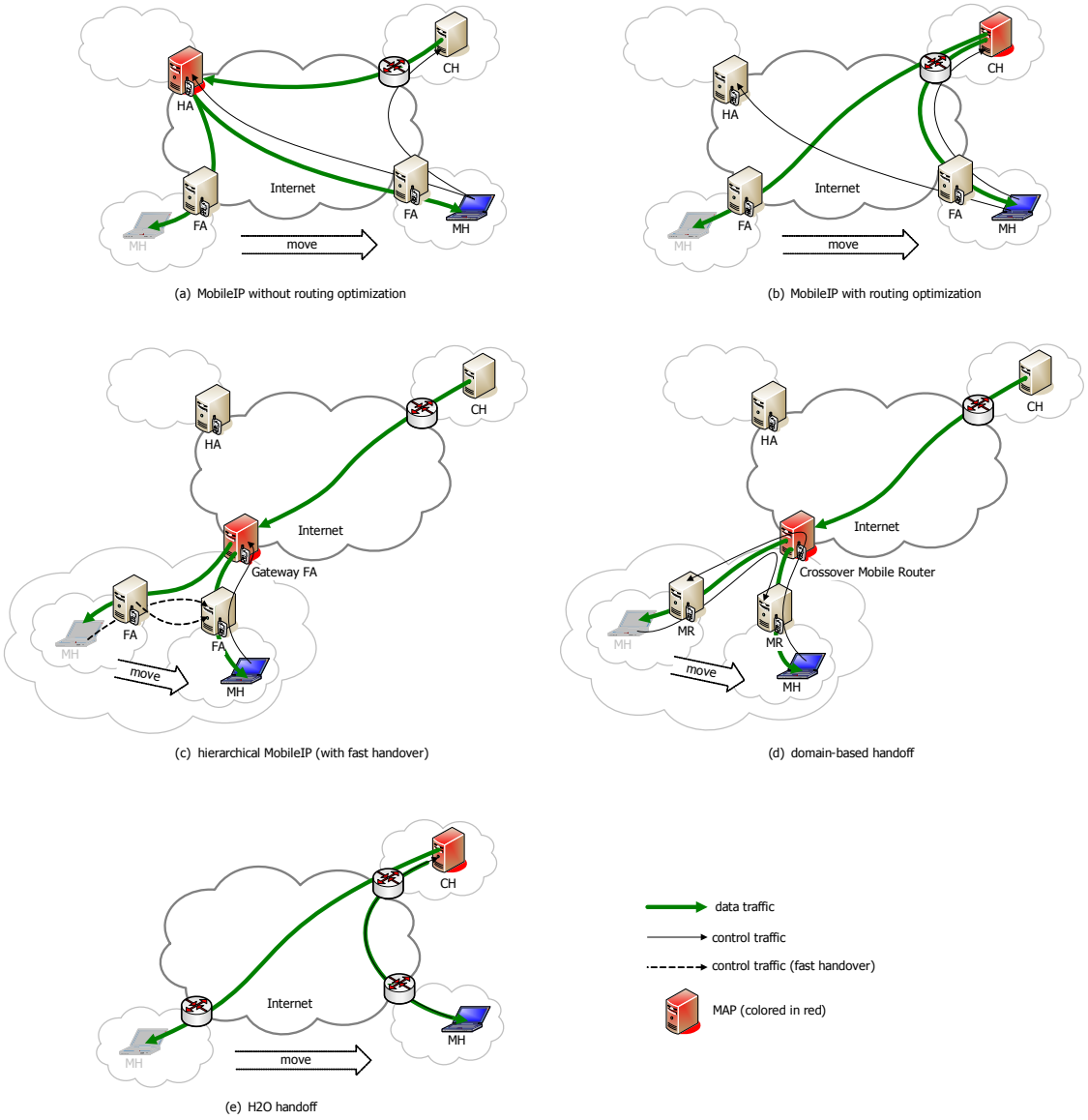


Figure 7-1. Visual handoff comparison

To summarize, existing solutions designed to improve MobileIP handoff performance all require introducing complex functions in the network infrastructure.

H2O, on the other hand, functions end-to-end completely within the communication endpoints. We present a visual comparison of these handoff solutions and H2O in Figure 7-1. Note that the overlapping of data connection and control connection in H2O illustrates the in-band signaling of H2O.

7.2.3 Others

[129] describes a domain-based handoff scheme in which the MAP multicasts traffic to the base-station where the MH is currently connected, as well as all the neighboring base-stations; although only the base-station where the MH is currently connected actively forwards packets to the MH using unicast. The idea is to preload neighboring base-stations with packets destined for the MH so that if it moves to any one of them, there will be a few packets buffered at the base-station for the MH therefore reducing packet loss and handoff delay.

FASTMIP [53] describes a domain-based handoff scheme in which the MH and all base-stations are equiped with a GPS (global positioning system) device. By making use of the position information, the base-station where the MH is currently connected can estimate the direction in which the MH is traveling and send duplicate packets destined for the MH to several perspective base-stations that the MH is likely to connect next.

7.3 High Service Availability Mechanisms

Although relatively little attention has been paid to minimizing service disruption due to a scheduled server maintenance, such as applying patches or upgrading

system software/hardware, much research effort [26][27][29][80][90][99][138][140] has been put into providing TCP failover in server clusters in order to minimize the downtime due to unexpected failure. Special TCP handoff mechanisms [31][101][102][120][130] have also been proposed for web server clusters in order to improve their performance and scalability. We briefly survey these work in this section.

7.3.1 Fault tolerance with TCP failover

Most TCP failover mechanisms to date focus on failover of TCP connection states and require TCP stack and/or application change on the server. Another important aspect of fault tolerance, failover of server application states for arbitrary non-deterministic applications, remains an open issue.

FT-TCP (Fault-Tolerant TCP) [29][138] instruments a wrapper both above (called north-side wrap, NSW) and below (called south-side wrap, SSW) the server TCP stack. The wrapper intercepts, modifies, and logs packets on their way in and out of the TCP stack. The wrapper manipulates TCP sequence numbers to deal with the output commit problem [52] and to allow rollback recovery of server TCP states from the log. Early FT-TCP work [29] assumes an external mechanism to recovery application states. Later FT-TCP work [138] adapts it to "hot backup" systems using primary-back [39] and "cold backup" systems with message logging [52]. While FT-TCP is client transparent and avoids modifying server TCP stack, handling server application nondeterminism and keeping the replicas identical to the primary, however, remains an unresolved issue. [138] modifies two popular

server applications, *Darwin* streaming server and *Samba* file server, to show the feasibility of FT-TCP with *some* applications.

ST-TCP (Server fault-Tolerant TCP) [90] is based on the primary-backup protocol. It maintains an active backup server at all times to allow fast failover. An ST-TCP backup server uses ethernet tapping, originally presented in [99], to learn the complete packet exchange between the client and the primary server. While ST-TCP is client transparent and ethernet tapping incurs low performance overhead during fault-free operation, ST-TCP requires modifying server TCP stack in order to synchronize the TCP states on the primary and backup server. For synchronizing server application states, ST-TCP assumes that either the application is completely deterministic therefore identical application states on the backup server can be derived from the tapped packet stream, or a leader/follower consistency protocol such as [37] is available for nondeterministic applications.

[80] describes a mechanism very similar to ST-TCP but claims to have faster failover time (no comparison is given). Backup servers in [80] use promiscuous mode of their NIC to receives all packets exchanged between the client and the primary server. [80] is also client transparent but requires modifying server TCP stack. [80] assumes that server applications are deterministic.

[26][27] describes a fault-tolerant web service that can provide fault-tolerance for HTTP requests being processed at the time of server failure. It assumes a standard standby back system with message logging. Unlike FT-TCP, however, [26][27] logs at the granularity of HTTP requests rather than TCP packets, which allows it to

recovery in progress requests. But this also limits [26][27] to HTTP only. The logging mechanism of [26][27] deals with the output commit problem by placing the backup server *before* the primary server, i.e., a client HTTP request passes through the backup server first, which ensures that a copy of the request is saved before passing it onto the primary server. [26][27] also implicitly assumes that the server application, a web server in this case, is deterministic and its states are completely determined by the sequence of HTTP requests.

[87] proposes another HTTP specific fault tolerant system for web server clusters. It places a lot of functions on the frontend switch which makes it susceptible to a single point of failure. [87] pre-establishes several persistent HTTP [55] connections between the frontend switch and the backend servers. A client's connection to the switch is then spliced with one of the pre-established switch-server connections to create one seamless client-server connection. Requests for static contents can be redirected to a different server in case of failure by simply splicing the client-switch connection to another idle persistent switch-server connection. To support requests for dynamic contents, [87] uses the switch to cache the reply and only forward it to the client when the entire content has been stored. [87] also supports session based requests, which requires the ability to recover intermediate session states from the failed server. This is achieved by keeping a backup server that snoops packets for the primary server using the primary server's IP address as an alias. In addition, the server application (*apache*) is modified to support a proprietary protocol between the primary and the backup server to ensure that requests and replies are logged before they are posted to the client and the data-

base server. [87] implicitly assumes that the server application is deterministic.

[140] describes a connection failover mechanism for web server clusters in which a frontend dispatches client requests to backend servers, which are organized into a ring. Each backend server is also a backup server for a fixed number n of its predecessor servers; the paper considers the case of n=1. [140] implements its own protocol called BTCP (Backup TCP) to replace TCP for backup server functions. It's main purpose is for the backup server to derive TCP states on the primary server by passively observing the packets sent from the client to the primary server, which are also forwarded to the backup server by the frontend. While this approach alleviates the need for explicit synchronization between the primary server and the backup server, it also has a few drawbacks. Since the backup server only sees the incoming packets, certain important information on the primary server is not available, such as when the primary server sends FIN or RST to terminate or abort a connection. The solution adopted by [140] is to use a timeout. Connection failover is achieved by converting the passive TCP states maintained by BTCP on the backup server to regular TCP states, and reissuing the last request before primary server failure on the backup server. This approach, however, has a few drawbacks. It requires the server application to be stateless, i.e., no application states are required on the backup server to serve the request. This necessarily limits the type of applications to simple request/reply transactions such as serving static web page. It also requires the server application to be idempotent; and it's not entirely transparent to the client since the failed (and restarted) request may have already sent some data back to the client.

7.3.2 Performance and scalability with TCP handoff

TCP handoff is a mechanism designed specifically for the widely used web server clusters [43], typically consist of a frond-end dispatcher and a group of backend servers, to support client transparent content-aware request distribution, with the benefits of smart load balancing, cache affinity, etc. TCP handoff in this context deals with transferring TCP connection states only. Server applications are explicitly assumed to be stateless, i.e., no application states beyond those trivially replicated ones such as static web page are needed in order to serve a request. Depending on the particular approach, the handoff can happen between the frontend dispatcher and the backend server, or among backend server themselves. Most approaches require modifying server TCP stack; some also require modifying server applications.

LARD [31][101] is one of the early system proposed for content-aware request distribution, with the focus on improving performance by exploiting content locality. In order to support content-aware dispatching at the frontend, the frontend must establish a TCP connection prior to dispatching a request. To reduce the load on the frontend, once the target backend server is chosen, the TCP connection states on the frontend are handed off to the chosen server, which can then reply directly to the client. LARD modifies the TCP stack on both the frontend dispatcher and the backend server to support the handoff, e.g., creating a new connection on the target server without going through a regular 3-way handshake. Early work [101] supports single handoff from the frontend to the backend at connection setup time. Later work [31] extends handoff support to persistent HTTP/1.1 and allows a con-

nection to be handed off from one backend server to another at any time using backend forwarding. [130] describes an implementation of the TCP handoff with backend forwarding based on STREAMS TCP/IP in HP-UX11.0.

Socket cloning [120] describes a handoff mechanism similar to LARD. It modifies both the server application and TCP stack to clone both the socket states and transport states of a connection from one server to another. After cloning, the original server forwards incoming packets to the cloned server, while outgoing packets go from the cloned server to the client directly. To avoid explicit state synchronization between the original and the cloned server, additional mechanism is introduced on the original server to derive implicit synchronization by observing the acknowledgements from the client.

KNITS [102] proposes a TCP handoff mechanism in which content-aware request dispatching is performed at the backend server. The frontend dispatcher in KNITS is a layer-4 switch that sprays a request to a designated backend server using simple layer-4 information. The designated server parses the request content and may handoff the request to another optimal server if necessary. Unlike LARD, handoff in KNITS is not achieve by backend forwarding from the designated server to the optimal server. Instead, the designated server informs the frontend switch about the handoff, which then redirects further requests directly to the optimal server without going through the designated server. The redirection is transparent to the client and the optimal server but requires all traffic between the backend servers go through the frontend switch. KNITS uses an application proxy

on the backend servers to avoid directly modifying the server applications.

Half-pipe anchoring [81] advocates a backend handoff scheme in which a dedicated backend server performs layer-7 switching and hands off requests to an appropriate optimal server. The terms "control pipe" and "data pipe" are used to refer to the connection from the client to the dedicated server and the connection from the optimal server to the client, respectively. The main idea of splitting the two "half-pipes" between the dedicated and the optimal server is to relax the requirement of performing layer-7 switching on optimal servers therefore allowing heterogeneous cluster of special optimal servers. Handoff between the dedicated and the optimal server is coordinated by modifying their TCP stack to support a proprietary "split-stack" protocol.

7.4 Process Migration Systems

Process migration is also a well traveled area with a large body of prior work and a variety of approaches proposed, such as special purpose OSes, user-level migration, language and middleware support, etc. We briefly survey these approaches. Interested readers are referred to the more in-depth presentation of [93].

7.4.1 Special purpose OSes

Several research OSes have been developed specifically with process migration support in mind, such as Accent [111], Amoeba [95], Charlotte [32], Chorus [116], MOSIX [36], Sprite [49], and V [45]. These are distributed OSes with a single system image across a cluster of machines. Process migration is supported by careful

kernel design to provide a global namespace and location-independent execution. While suitable for a small cluster of machines, these solutions require new OSes or substantial changes to existing ones, therefore limiting their general purpose usage and deployment. The single system image design also hinders their usage in cluster environments where each machine is independent, which has become increasingly common. Finally, these solutions typically handle certain process states such as IPC, open files, and system calls by forwarding requests to a home machine on which the migrated process originated. This leaves undesirable dependency on the home machine since if the home machine fails the migrated process on another machine will fail as well.

7.4.2 User-level migration

User-level migration mechanisms that do not require special purpose OSes and can run on unmodified commercial OSes, such as CoCheck [108], Condor [86], libckpt [106], and MPVM [44], have been proposed. However, providing transparent process migration without kernel support such as those mentioned in the previous section is much more challenging. Therefore, these solutions are primarily targeted for long-running "well-behaved" applications that do not pose significant OS requirements and use only a limited set of system calls. For example, these applications cannot use common OS services such as IPC. These restrictions severely limit the kinds of applications that can be migrated.

7.4.3 Language and middleware support

Mobile objects and mobile agents are another form of migration. These systems

provide programming languages and middleware toolkits so that programmers can explicitly incorporate migration capability into their applications. Examples of these systems include Abacus [30], Emerald [74], Globus [57], Legion [63], Rover [73], and Telescript [21]. While language constructs provide a high-level abstraction for defining and encapsulating application states so that they can be easily migrated, these solutions require applications to be (re)written using the new programming languages or toolkits. Therefore, they can not migrate legacy applications.

7.4.4 OS virtualization

Virtualization at OS level has recently been proposed in [100][118] as a mechanism for supporting process migration. [118] introduces a capsule abstraction that provides a virtual private namespace to a group of processes that can be migrated as a unit. However, implementation of capsule requires extensive OS changes. [100] is our choice of process migration mechanism to enable the fine-grain connection migration capability of MOVE. We have combined the Pod abstraction of [100] with our CELL abstraction to create a unified migration abstraction, called zPod, for both process and connection states.

7.4.5 Virtual machine monitors

Virtualization at machine hardware level, a technique commonly known as virtual machine monitors (VMM) [22][40][60][135], has long been recognized as an important technology for supporting resource partitioning and multiplexing, and software isolation and portability. Since the virtual machine encapsulates an entire OS

environment, it can be used to support process migration by migrating the entire OS environment from one machine to another assuming sufficient similarity in the underlying machine systems. For example, VMotion [23] from VMware [22] can migrate a live VM between two co-located machine through dedicated fast (giga-bit) network. However, since VMMs operate below the OS, they cannot take advantage of OS specific mechanisms to reduce migration cost. All applications to be migrated must be running in the VMM and be migrated all together, which has high migration cost in terms of suspension/resumption time and image size as shown in [100].

Finally, we point out that no previous process migration mechanisms support migration of open network connections.

8 Conclusion

We have presented in this thesis a novel mobile communication architecture, MOVE, that solves key technical problems necessary for supporting the mobile communication needs of existing and emerging network applications. We have focused our attention on specific aspects of mobility problems that lack adequate support in current network and system infrastructures. In particular, we focused on the mobility of the end-to-end communication between two endpoints rather than the mobility of endpoints themselves. We further focused on the problem of tracking the mobile end-to-end communication rather than the problem of locating the mobile endpoints. To that end, we have made the following contributions in this thesis:

- We have identified the functional requirements of the network and system infrastructures needed for supporting general purpose mobile communications, which include: easy deployment, fine-grain and unlimited mobility, secure and flexible migration, and low performance overhead. We have also identified fundamental problems that must be resolved in order to meet these requirements, which are: state inconsistency, state conflict, and cross address space state synchronization problems.
- We have developed new concepts and mechanisms to solve these fundamental problems. In particular, we have developed the CELL namespace

abstraction, which provides a virtual, private, and labeled namespace for individual connections, to cleanly and uniformly address the three fundamental problems. We have also developed light-weight and efficient mechanisms, such as per-connection virtual network interface, lazy assignment, and connection label, etc. to support our CELL namespace abstraction.

- We have developed a new handoff protocol and security mechanism to enable fast and secure migration of end-to-end connections. In particular, our H2O handoff signaling protocol resonates the fundamental end-to-end tenet [117] with the key observation that the cost of introducing additional complexity in the network layer to reduce packet loss does not necessarily translate into end-to-end benefit. Our security mechanism is based the well-known Diffie-Hellman protocol with key observations to mitigate the overhead of expensive key computation. Combining the CELL virtual-physical mapping, H2O handoff signaling, and our security mechanism, MOVE can migrate connections securely in just one packet in a single one-way trip from the mobile endpoint to the stationary endpoint. We have also developed a migration helper mechanism to support connection migration through suspension/resumption with potentially extended period of disconnection time.

- We have seamlessly integrated our MOVE connection migration mechanisms with the Zap process migration mechanisms through a unifying zPod abstraction, which provides a virtual and private namespace for both connection states and process states. We have demonstrated the power of

combining the two in a proxy-based server cluster environment to enable zero service disruption for arbitrary stateful applications during server maintenance, without introducing additional cluster configuration and management complexity.

- We have designed and implemented a prototype of our MOVE architecture on a commodity OS platform, i.e., LINUX x86. We have shown that all MOVE functions can be implemented completely within the endpoints and without requiring any change to existing network infrastructure, OS, and applications. We have also shown that all MOVE functions are backward compatible and can interoperate with existing network infrastructure, OS, and applications.
- We have evaluated our MOVE prototype and shown the performance of our MOVE prototype. We demonstrated that MOVE's handoff has very little impact on the network connectivity perceived by the transport protocols and applications. We showed that MOVE has no negative impact on the system's scalability. We also showed that MOVE's virtualization and virtual-physical mapping functions introduce very low network I/O performance overhead before and after connections are migrated.

With the rapid increase of ubiquitous mobile computing devices and universal network connectivity, there is a pressing need for developing new networking functionality to support the mobile communication needs of the applications. While we have no doubt that future networking infrastructure, protocols, and applications will be increasingly mobility-savvy, existing ones will continue to

function for many years to come. Therefore, developing and deploying new networking functions is often a long and enduring process. Nevertheless, we believe our work in this thesis is a step forward towards the new global pervasive mobile and network computing era. We also hope that our work in this thesis can give insight on how such new networking functionality can be developed and deployed while allowing existing legacy applications to take advantage of the tremendous benefits offered by the coming reality of ubiquitous mobile computing and communication.

Despite our best intentions, this thesis alone cannot address all the issues of an area as broad as general purpose mobile communication. We discuss several issues that we would have liked to but did not have the manpower to address, and that we intend to pursue further in the future. These issues are listed in the order of their importance in our option, with the most important one first:

- Generalization of MOVE mechanisms. At a more abstract level, MOVE can be considered as a transport layer tunneling mechanism, in similar spirit to tunneling mechanisms at other layers such as network layer tunneling (VPNs) and link layer tunneling (VLANs). Besides supporting mobility by essentially tunneling one connection inside another, this fine-grain transport level tunneling can potentially have many other ramifications. As one brain-storming example, one can associate semantics with the connection label therefore allowing migrated and stationary connections to be discriminated with different security and QoS metric, etc.
- Further process migration development. While the focus of this thesis is on

connection migration rather than process migration, we fully recognize that process migration is one of the essential ingredients for truly fine-grain connection migration. This thesis has touched upon a few issues of process migration from the same virtual private namespace point of view as that we used for connection migration. However, process migration is also a very broad area with many open issues and warrants significant research itself.

- Simultaneous move of both endpoints. While mobile endpoints today are mostly on the client side, we've also seen the need for server side mobility to support high service availability. In addition, peer-to-peer networks are becoming increasingly popular. Therefore, the chance of both endpoints are mobile is rather high in the future. While MOVE does not restrict which endpoint of a connection can move, it does assume that only one endpoint moves at a time. This restriction is necessary so that the mobile endpoint can trivially locate the stationary endpoint. If the stationary endpoint also moves, there needs to be a mechanism for the two endpoints to locate each other after they move simultaneously. One simple solution could be for both endpoints to use a directory service such as DDNS to locate each other whenever they move; this approach however requires infrastructure support. An alternative could be to use a proxy at each location where the mobile endpoints have visited to keep track of the mobile endpoints; this approach does not require infrastructure support but have the downside of leaving states behind.

- More in-depth study of location services. While we have chosen DDNS and conducted studies for its suitability as MOVE's mobile host and service location mechanism, we feel that a single directory service may not be the answer for all possible application types and scenarios. For example, DDNS is a semantically simple name-to-value mapping directory service. For some applications, a more semantically rich type of directory service such as X.500 [6] that supports attribute-based retrieval may be more beneficial. Also depending on the scale of the application, a local scale directory service such as LDAP (Light-weight Directory Access Protocol) [134] or SLP (Service Location Protocol) [68] may be more appropriate.

Bibliography

[1] *Apache HTTP Server Project*. http://www.apache.org

[2] *Cisco's PIX Firewall Series and Stateful Firewall Security*, White Paper, Cisco Systems, Inc., 1997.

[3] *Darwin Streaming Server*. http://developer.apple.com/darwin

[4] *DeleGate*. http://www.delegate.org

[5] *DES modes of operation*, NBS FIPS PUB 81, National Bureau of Standards, U.S. Department of Commerce, 1980.

[6] *The Directory, Part 1: Overview of Concepts, Models and Services*, CCITT Draft Recommendation X.500/ISO DIS 9594-1, CCITT/ISO, December 1988.

[7] *Foundry ServerIron Switch Installation and Configuration Guide*, Foundry Networks Inc., June 2003.

[8] *GNU wget*. http://www.gnu.org/software/wget/

[9] *IEEE Standards for Local and Metropolitan Area Networks: Virtual Bridged Local Area Networks*, 802.1Q, 2003 EDITION, IEEE, 2003.

[10] *IMT-2000 DS-CDMA System*, ARIB STD-T63, Association of Radio Industries and Businesses, September 2002.

[11] *Internet Systems Consortium Inc.* http://www.isc.org/

[12] *lftp*. http://lftp.yar.ru/

[13] *Message Passing Interface Forum*. http://www.mpi-forum.org/

[14] *MPlayer*. http://www.mplayerhq.hu/

[15] *openRTSP*. http://www.live.com/openRTSP/

[16] *Resonate Central Dispatch: In-Depth and Technical*, White Paper, Resonate Inc., October 2001.

[17] *Specification of the Bluetooth System, version 1.2*, Bluetooth Special Interest Group, November 2003.

[18] *Stateful Inspection Technology*, White Paper, Check Point Software Technologies Ltd., 2004.

[19] *The tcpdump project*. http://sourceforge.net/projects/tcpdump/

[20] *tcptrace*. http://jarok.cs.ohiou.edu/software/tcptrace/tcptrace.html

[21] *Telescript Technology: Mobile Agents*, General Magic, 1996.

[22] *VMware Inc.* http://www.vmware.com

[23] *VMware VirtualCenter User's Manual, version 1.0*, VMware Inc.

[24] *vsftpd*. http://vsftpd.beasts.org/

[25] *Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) specifications*, IEEE Standard 802.11, The Institute of Electrical and Electronics Engineers, Inc., 1999.

[26] N. Aghdaie and Y. Tamir, *Client-Transparent Fault-Tolerant Web Service*, Proceedings of the 20th IEEE International Performance, Computing, and Communications Conference, Phoenix, AZ, April 2001.

[27] N. Aghdaie and Y. Tamir, *Implementation and Evaluation of Transparent Fault-Tolerant Web Serivce with Kernel-Level Support*, Proceedings of the IEEE International Conference on Computer Communications and Networks, Miami, FL, October 2002.

[28] P. Almquist, *Type of Service in the Internet Protocol Suite*, RFC1349, IETF, July 1992.

[29] L. Alvisi, T. C. Bressoud, A. El-Khashab, K. Marzullo, and Z. Zagorodnov, *Wrapping Server-Side TCP to Mask Connection Failures*, Proceedings of IEEE InfoCom, Anchorage, Alaska, April 2001.

[30] K. Amiri, D. Petrou, G. Ganger, and G. Gibson, *Dynamic Function Placement in Active Storage Clusters*, Technical Report CMU-CS-99-140, School of Computer Science, Carnegie Mellon University, June 1999.

[31] M. Aron, P. Druschel, and W. Zwaenepoel, *Efficient Support for P-HTTP in Cluster-Based Web Servers*, Proceedings of the 1999 Annual Usenix Technical Conference, Monterey, CA, June 1999.

[32] Y. Artsy, Y. Chang, and R. Finkel, *Interprocess Communication in Charlotte*, IEEE Software:22-28, January 1987.

[33] F. Baker, *Requirements for IP Version 4 Routers*, RFC1812, Cisco Systems, June 1995.

[34] A. Bakre and B. R. Badrinath, *Handoff and System Support for Indirect TCP/IP*, Proceedings of Second Usenix Symp. on Mobile and Location-Independent Computing, April 1995.

[35] H. Balakrishnan, S. Seshan, E. Amir, and R. H. Katz, *Improving TCP/IP Performance over Wireless Networks*, Proceedings of 1st ACM International Conference on Mobile Computing and Networking (MobiCom), Berkeley, CA, November 1995.

[36] A. Barak and R. Wheeler, *MOSIX: An Integrated Multiprocessor UNIX*, Proceedings of the USENIX Winter 1989 Technical Conference, pp. 101-112, San Diego, CA, February 1989.

[37] P. Barret, A. Hilborne, P. Bond, P. V. D. Seaton, L. Rodrigues, and N. Speirs, *The Delta-4 Extra Performance Architecture (XPA)*, Proceedings of 20th IEEE Symposium on Fault-Tolerant Systems, 1990.

[38] P. Bhagwat and C. Perkins, *A Mobile Networking System based on Internet Protocol (IP)*, Proceedings of USENIX Symposium on Mobile and Location Independent Computing, pp. 69-82, Cambridge, MA, August 1993.

[39] N. Budhiraja, K. Marzullo, F. Schneider, and S. Toueg, *Primary-backup Protocols: Lower Bounds and Optimal Implementations*, Proceedings of 3rd IFIP Conference on Dependable Computing for Critical Applications, Sicily, Italy, September 1992.

[40] E. Bugnion, S. Devine, and M. Rosenblum, *Disco: running commodity operating systems on scalable multiprocessors*, Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles, Saint-Malo, France, October 1997.

[41] R. Caceres and L. Iftode, *Improving the Performance of Reliable Transport Protocols in Mobile Computing Environments*, IEEE Selected Areas in Communications, **13**(5):850-857, June 1994.

[42] A. T. Campbell, J. Gomez, S. Kim, Z. Turanyi, C. Y. Wan, and A. G. Valko, *Design, Implementation and Evaluation of Cellular IP*, IEEE Personal Communications, Special Issue on IP-based Mobile Telecommunications Networks, June/July 2000.

[43] V. Cardellini, E. Casalicchio, M. Colajanni, and P. S. Yu, *The State of the Art in Locally Distributed Web-Server Systems*, ACM Computing Surveys, **34**(2):263-311, June 2002.

[44] J. Casas, D. L. Clark, R. Conuru, S. W. Otto, R. M. Prouty, and J. Walpole, *MPVM: A Migration Transparent Version of PVM*, Computing Systems, **8**(2):171-216, 1995.

[45] D. Cheriton, *The V Distributed System*, Communications of the ACM, **31**(3):314-333, March 1988.

[46] M. S. Corson and A. O'Neill, *An Approach to Fixed/Mobile Converged Routing*, Technical Report, TR-2000-5, Institute for Systems Research, University of Maryland, 2000.

[47] S. Deering, *Host Extensions for IP Multicasting*, RFC1112, IETF, August 1989.

[48] W. Diffie and M. Hellman, *New Directions in Cryptography*, IEEE Transactions on Information Theory, **22**(6):644-654, November 1976.

[49] F. Douglis and J. Ousterhout, *Transparent Process Migration: Design Alternatives and the Sprite Implementation*, Software - Practice and Experience, **21**(8):757-785, August 1991.

[50] D. Eastlake and P. Jones, *US Secure Hash Algorithm 1 (SHA1)*, RFC3174, IETF, September 2001.

[51] K. El-Malki, P. R. Calhoun, T. Hiller, J. Kempf, P. J. McCann, A. Singh, H. Soliman, and S. Thalanany, *Low Latency Handoffs in Mobile IPv4*, draft-ietf-mobileip-lowlatency-handoffs-v4-05.txt, IETF, June 2003.

[52] E. Elnozahy, L. Alvisi, Y. Wang, and D. Johnson, *A survey of rollback-recovery protocols in message passing systems*, ACM Computing Surveys, **34**(3):375-408, 2002.

[53] M. Ergen, S. Coleri, B. Dundar, R. Jain, A. Puri, and P. Varaiya, *Application of GPS to Mobile IP and Routing in Wireless Networks*, Proceedings of IEEE Semiannual Vehicular Technology Conference, Vancouver, Canada, September 2002.

[54] A. Festag, *Optimization of Handover Performance by Link Layer Triggers in IP-Based Networks: Parameters, Protocol Extensions and APIs for implementation*, Technical Report TKN-02-014, Telecommunication Networks Group, Technical University Berlin, August 2002.

[55] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee, *Hypertext Transfer Protocol -- HTTP/1.1*, RFC2068, IETF, January 1997.

[56] N. A. Fikouras, A. J. Könsgen, and C. Görg, *Accelerating Mobile IP Hand-offs through Link-layer Information*, Proceedings of the International Multiconference on Measurement, Modelling, and Evaluation of Computer-Communication Systems, Aachen, Germany, September 2001.

[57] I. Foster and C. Kesselman, *Globus: A Metacomputing Infrastructure Toolkit*, Proceedings of the Workshop on Environments and Tools for Parallel Scientific Computing, Lyon, France, August 1996.

[58] D. Funato, K. Yasuda, and H. Tokuda, *TCP-R: TCP mobility support for continuous operation*, IEEE International Conference on Network Protocols, pp. 229-236, Atlanta, GA, October 1997.

[59] B. Gleeson, A. Lin, J. Heinanen, G. Armitage, and A. Malis, *A Framework for IP Based Virtual Private Networks*, RFC2764, IETF, February 2000.

[60] R. P. Goldberg, *Survey of virtual machine research*, IEEE Computer Magazine, **7**(6):34-45, 1974.

[61] D. M. Gordon, *A survey of fast exponentiation methods*, Journal of Algorithms, **27**(1):129-146, April 1998.

[62] J. Gosling, B. Joy, and G. Steele, *The Java Language Specification*, Addison-Wesley Publishing Company, Inc., 1996.

[63] A. Grimshaw and W. Wulf, *The Legion Vision of a Worldwide Virtual Computer*, Communications of the ACM, **40**(1):39-45, January 1997.

[64] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, *A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard*, Parallel Computing, **22**(6), September 1996.

[65] A. Gulbrandsen, P. Vixie, and L. Esibov, *A DNS RR for specifying the location of services (DNS SRV)*, RFC2782, IETF, February 2000.

[66] S. Gupta and A. L. N. Reddy, *A Client Oriented, IP Level Redirection Mechanism*, Proceedings of IEEE INFOCOM'99, pp. 1461-1469, March 1999.

[67] E. Gustafsson, A. Jonsson, and C. Perkins, *Mobile IP Regional Registration*, Internet draft, draft-ietfmobileip-reg-tunnel-06.txt, IETF, March 2002.

[68] E. Guttman, C. Perkins, J. Veizades, and M. Day, *Service Location Protocol, Version 2*, RFC2608, IETF, June 1999.

[69] R. Hsieh, Z. G. Zhou, and A. Seneviratne, *S-MIP: A Seamless Handoff Architecture for Mobile IP*, Infocom'03, San Francisco, March 2003.

[70] J. Ioannidis, D. Duchamp, and G. Q. Maguire, *IP-based Protocols for Mobile Internetworking*, Proceedings of ACM SIGCOMM, pp. 235-245, 1991.

[71] D. B. Johnson and C. Perkins, *Mobility Support in IPv6*, draft-ietf-mobileip-ipv6-16.txt, IETF, March 2002.

[72] R. Jones, *Netperf: a Network Performance Benchmark*, Information Networks Division, Hewlett-Packard Company, February 1996. http://www.netperf.org/netperf/NetperfPage.html

[73] A. D. Joseph, J. A. Tauber, and M. F. Kaashoek, *Mobile Computing with the Rover Toolkit*, IEEE Transactions on Computers, **46**(3):337-352, March 1997.

[74] E. Jul, *Migration of Light-weight Processes in Emerald*, IEEE Technical Committee on Operating Systems Newsletter, **3**(1):20-23, 1989.

[75] J. Jung, E. Sit, H. Balakrishnan, and R. Morris, *DNS Performance and the Effectiveness of Caching*, IEEE/ACM Transaction on Networking, **10**(5), October 2002.

[76] S. Kent, *IP Authentication Header*, Internet Draft, draft-ietf-ipsec-rfc2402bis-07.txt, IETF, March 2004.

[77] S. Kent, *IP Encapsulating Security Payload (ESP)*, Internet Draft, draft-ietf-ipsec-esp-v3-08.txt, IETF, March 2004.

[78] S. Kent and K. Seo, *Security Architecture for the Internet Protocol*, draft-ietf-ipsec-rfc2401bis-03.txt, IETF, September 2004.

[79] T. J. Killian, *Processes as Files*, Proceedings of USENIX Summer Conference, pp. 203-207, Salt Lake City, UT, June 1984.

[80] R. Koch, S. Hortikar, L. Moser, and P. Melliar-Smith, *Transparent TCP Connection Failover*, Proceedings of DSN, San Francisco, CA, June 2003.

[81] R. Kokku, R. Rajamoni, L. Alvisi, and H. Vin, *Half-Pipe Anchoring: An Efficient Technique for Multiple Connection Handoff*, Proceedings of the 10th International Conference on Network Protocols, Paris, France, November 2002.

[82] R. Koodli, *Fast Handovers for Mobile IPv6*, draft-ietf-mobileip-fast-mipv6-06.txt, IETF, March 2003.

[83] H. Krawczyk, M. Bellare, and R. Canetti, *HMAC: Keyed-Hashing for Message Authentication*, RFC 2104, IETF, February 1997.

[84] M. Leech, M. Ganis, Y. Lee, R. Kuris, D. Koblas, and L. Jones, *SOCKS Protocol Version 5*, RFC1928, IETF, March 1996.

[85] H. Levkowetz and S. Vaarala, *Mobile IP Traversal of Network Address Translation (NAT) Devices*, RFC3519, IETF, April 2003.

[86] M. Litzkow, T. Tannenbaum, J. Basney, and M. Livny, *Checkpoint and Migration of UNIX Processes in the Condor Distributed Processing System*, Technical Report #1346, University of Wisconsin Madison Computer Sciences, April 1997.

[87] M. Y. Luo and C. S. Yang, *Constructing Zero-loss Web Services*, Proceedings of IEEE InfoCom, Anchorage, AK, April 2001.

[88] D. Maltz and P. Bhagwat, *TCP splicing for application layer proxy performance*, IBM Research Report 21139 (Computer Science/Mathematics), IBM Research Division, March 1998.

[89] D. A. Maltz and P. Bhagwat, *MSOCKS: An Architecture for Transport Layer Mobility*, Proceedings of the IEEE INFOCOM'98, pp. 1037-1045, San Francisco, CA, 1998.

[90] M. Marwah, S. Mishra, and C. Fetzer, *TCP Server Fault Tolerance Using Connection Migration to a Backup Server*, Proceedings of DSN, San Francisco, CA, June 2003.

[91] U. Maurer, *Towards the equivalence of breaking the Diffie-Hellman protocol and computing discrete logarithms*, Advances in Cryptology, **839**(271-281, 1994.

[92] J. Midgley, *Autobench*. http://www.xenoclast.org/autobench/

[93] D. Milojicic, F. Douglis, Y. Paindaveine, R. Wheeler, and Z. Songnian, *Process Migration*, HPL-1999-21, HP Laboratories Palo Alto, February 1999.

[94] D. Mosberger and T. Jin, *httperf: A tool for measuring web server performance*, ACM First Workshop on Internet Server Performance, pp. 59-67, Madison, WI, June 1998.

[95] S. J. Mullender, G. v. Rossum, A. S. Tanenbaum, R. v. Renesse, and H. v. Staveren, *Amoeba – A Distributed Operating System for the 1990s*, IEEE Computer, **23**(5):44-53, May 1990.

[96] J. Mysore and V. Bharghavan, *A New Multicasting-Based Architecture for Internet Host Mobility*, Proceedings of ACM Mobicom, September 1997.

[97] K. Nichols, S. Blake, F. Baker, and D. Black, *Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers*, RFC2474, IETF, December 1998.

[98] T. Okoshi, M. Mochizuki, Y. Tobe, and H. Tokuda, *MobileSocket: Toward Continuous Operation for Java Applications*, IEEE IC3N'99, Boston, MA, October 1999.

[99] M. Orgiyan and C. Fetzer, *Tapping TCP Streams*, Proceedings of the IEEE International Symposium on Network Computing and Applications (NCA), Boston, MA, February 2002.

[100] S. Osman, D. Subhraveti, G. Su, and J. Nieh, *The Design and Implementation of Zap: A System for Migrating Computing Environments*, Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI 2002), pp. 361-376, Boston, MA, December 2002.

[101] V. Pai, M. Aron, G. BAnga, M. Svendsen, P. Drushel, W. Zwaenepoel, and E. Nahum, *Locality-Aware Request Distribution in Cluster-Based Network Servers*, Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, CA, October 1998.

[102] A. Papathanasiou and E. V. Hensbergen, *KNITS: Switch-based Connection Hand-off*, IEEE InfoCom, New York, NY, June 2002.

[103] C. Perkins, *IP Mobility Support*, RFC2002, IETF, October 1996.

[104] C. Perkins, *IP Mobility Support for IPv4, revised*, draft-ietf-mobileip-rfc2002-bis-08.txt, Internet Draft, September 2001.

[105] C. Perkins and D. B. Johnson, *Route Optimization in Mobile IP*, draft-ietf-mobileip-optim-11.txt, Internet Draft, September 2001.

[106] J. S. Plank, M. Beck, G. Kingsley, and K. Li, *Libckpt: Transparent Checkpointing under Unix*, Proceedings of Usenix Winter 1995 Technical Conference, pp. 213-223, New Orleans, LA, January 1995.

[107] J. Postel, *INTERNET PROTOCOL*, RFC791, Information Sciences Institute, University of Southern California, September 1981.

[108] J. Pruyne and M. Livny, *Managing Checkpoints for Parallel Programs*, 2nd Workshop on Job Scheduling Strategies for Parallel Processing (In Conjunction with IPPS '96), Honolulu, Hawaii, April 1996.

[109] X. Qu, J. X. Yu, and R. P. Brent, *A Mobile TCP Socket*, International Conference on Software Engineering (SE97), San Francisco, CA, November 1997.

[110] R. Ramjee, T. L. Porta, S. Thuel, and K. Varadhan, *HAWAII: A Domain-Based Approach for Supporting Mobility in Wide-Area Wireless Networks*, IEEE International Conference on Network Protocols, Toronto, Canada, October 1999.

[111] R. Rashid and G. Robertson, *Accent: a Communication Oriented Network Operating System Kernel*, Proceedings of the 8th Symposium on Operating System Principles, pp. 64–75, December 1984.

[112] P. Reinbold and O. Bonaventure, *IP micro-mobility protocols*, IEEE Communications Surveys and Tutorials, 2003.

[113] Y. Rekhter, B. Moskowitz, D. Karrenberg, G. J. deGroot, and E. Lear, *Address Allocation for Private Internets*, RFC1918, IETF, February 1996.

[114] M. Riegel and M. Tuexen, *Mobile SCTP*, draft-riegel-tuexen-mobile-sctp-03.txt, IETF, August 2003.

[115] R. Rivest, *The MD5 Message-Digest Algorithm*, RFC1321, IETF, April 1992.

[116] M. Rozier, V. Abrossimov, F. Armand, M. Gien, M. Guillemont, F. Hermann, and C. Kaiser, *Chorus (Overview of the Chorus Distributed Operating System)*, Proceedings of the USENIX Workshop on Micro-Kernels and other Kernel Architectures, Seattle, WA, April 1992.

[117] J. H. Saltzer, D. P. Reed, and D. D. Clark, *End-To-End Arguments in System Design*, ACM Transactions on Computer Systems, **2**(4):277-288, 1984.

[118] B. K. Schmidt, *Supporting Ubiquitous Computing with Stateless Consoles and Computation Caches*, Ph.D Thesis, Computer Science Department, Stanford University, August 2000.

[119] J. Silva, J. Carreira, H. Madeira, D. Costa, and F. Moreira, *Experimental Assessment of Parallel Systems*, Proceedings of the 26th International Symposium on Fault-Tolerant Computing, pp. 415-424, June 1996.

[120] Y. F. Sit, C. L. Wang, and F. Lau, *Socket Cloning for Cluster-Based Web Server*, Proceedings of IEEE 4th International Conference on Cluster Computing, Chicago, IL, September 2002.

[121] A. C. Snoeren, D. G. Andersen, and H. Balakrishnan, *Fine-Grained Failover Using Connection Migration*, Proceeding of the Third Annual USENIX Symposium on Internet Technologies and Systems (USITS), Cambridge, MA, March 2001.

[122] A. C. Snoeren and H. Balakrishnan, *An End-to-End Approach to Host Mobility*, Proceedings of 6th International Conference on Mobile Computing and Networking (MobiCom'00), Boston, MA, August 2000.

[123] H. Soliman, C. Castelluccia, K. El-Malki, and L. Bellier, *Hierarchical Mobile IPv6 mobility management (HMIPv6)*, draft-ietf-mobileip-hmipv6-08.txt, IETF, June 2003.

[124] L. Spitzner, *Understanding the FW-1 State Table*, November, 2000. http://www.spitzner.net/fwtable.html

[125] P. Srisuresh and M. Holdrege, *IP Network Address Translator (NAT) Terminology and Considerations*, RFC2663, IETF, August 1999.

[126] R. Stewart, M. Ramalho, Q. Xie, M. Tuexen, I. Rytina, M. Belinchon, and P. Conrad, *Stream Control Transmission Protocol (SCTP) Dynamic Address Reconfiguration*, draft-ietf-tsvwg-addip-sctp-08.txt, IETF, September 2003.

[127] R. Stewart, et al., *Stream Control Transmission Protocol*, RFC2960, IETF, October 2000.

[128] F. Sultan, K. Srinivasan, D. Iyer, and L. Iftode, *Migratory TCP: Highly available internet services using connection migration*, Proceedings of ICDCS, pp. 17-26, 2002.

[129] C. L. Tan, S. Pink, and K. M. Lye, *A Fast Handoff Scheme for Wireless Networks*, Proceedings of the 2nd ACM Interntational Workshop on Wireless Mobile Multimedia, pp. 83-90, Seattle, WA, 1999.

[130] W. Tang, L. Cherkasova, L. Russell, and M. W. Mutka, *Modular TCP Handoff Design in STREAMS-Based TCP/IP Implementation*, Proceedings of the 1st International Conference on Networking (ICN), Colmar, France, July 2001.

[131] F. Teraoka, K. Uehara, H. Sunahara, and J. Murai, *VIP: A Protocol Providing Host Mobility*, Communications of the ACM, **37**(8), August 1994.

[132] F. Teraoka, Y. Yokote, and M. Tokoro, *A Network Architecture Providing Host Migration Transparency*, Proceedings of ACM SIGCOMM, September 1991.

[133] P. Vixie, S. Thomson, Y. Rekhter, and J. Bound, *Dynamic Updates in the Domain Name System (DNS UPDATE)*, RFC2136, IETF, April 1997.

[134] M. Wahl, T. Howes, and S. Kille, *Lightweight Directory Access Protocol (v3)*, RFC2251, IETF, December 1997.

[135] A. Whitaker, M. Shaw, and S. Gribble, *Denali: Lightweight virtual machines for distributed and networked applications*, Proceedings of the USENIX Technical Conference, Monterey, CA, June 2002.

[136] P. Yalagandula, A. Garg, M. Dahlin, L. Alvisi, and H. Vin, *Transparent Mobility with Minimal Infrastructure*, Technical Report 01-30, University of Texas at Austin, June 2001.

[137] H. Yokota, A. Idoue, T. Hasegawa, and T. Kato, *Link Layer Assisted Mobile IP Fast Handoff Method over Wireless LAN Networks*, MobiCom'02, Atlanta, GA, September 2002.

[138] D. Zagorodnov, K. Marzullo, L. Alvisi, and T. C. Bressoud, *Engineering Fault-Tolerant TCP/IP Servers Using FT-TCP*, Proceedings of DSN, San Francisco, CA, June 2003.

[139] V. C. Zandy and B. P. Miller, *Reliable Network Connections*, Proceedings of 8th ACM International Conference on Mobile Computing and Networking (Mobicom '02), pp. 95-106, Atlanta, GA, September 2002.

[140] R. Zhang, T. F. Abdelzaher, and J. A. Stankovic, *Efficient TCP Connection Failover in Web Server Clusters*, Proceedings of IEEE InfoCom, Hong Kong, March 2004.

[141] Y. Zhang and S. Dao, *A "Persistent Connection" Model for Mobile and Distributed Systems*, 4th International Conference on Computer Communications and Networks (ICCCN), Las Vegas, NV, September 1995.

[142] S. Zhuang, K. Lai, I. Stoica, R. Katz, and S. Shenker, *Host Mobility using an Internet Indirection Infrastructure*, First International Conference on Mobile Systems, Applications, and Services (ACM/USENIX Mobisys), San Francisco, CA, May 2003.