Starfish: Fault-Tolerant Dynamic MPI Programs on Clusters of Workstations (Extended Abstract)

Adnan M. Agbaria Roy Friedman
Department of Computer Science
The Technion
Haifa 32000
Israel
{adnan,roy}@cs.technion.ac.il

Abstract

This paper reports on the architecture and design of Starfish, *an environment for executing dynamic (and static) MPI-2 programs on a cluster of workstations. Starfish is unique in being efficient, fault-tolerant, highly available, and dynamic as a system internally, and in supporting fault-tolerance and dynamicity for its application programs as well. Starfish achieves these goals by combining group communication technology with checkpoint/restart, and uses a novel architecture that is both flexible and portable and keeps group communication outside the critical data path, for maximum performance.*

1 Introduction

Employing clusters of workstations as a cost effective alternative to parallel computers has been the goal of much research in the past few years [8, 13], especially in light of the remarkable advances in both computing power of PCs and networks speed. While the idea of building such clusters is very appealing, the realization of this goal is quite complicated, due to the numerous non-trivial issues that have do be dealt with. These include maintaining the promised performance at the application level, manageability of the cluster, and fault-tolerance. Other issues like job/process scheduling must also be considered, although these are well studied problems, and the usage of clusters vs. parallel computers does not significantly alter known solutions from the parallel world.

One major obstacle in the way of building such clusters, which has been successfully dealt with recently, lies in the fact that legacy operating systems present high overhead in accessing the network. Thus, naive usage of fast networks fails to deliver on the promised network speed. U-Net [9], NOW [8], Fast-Messages [28], and other projects have been able to overcome this problem by developing user-level network interfaces, that bypass the operating system kernel, offering applications near optimal bandwidth and latency.

Thus, largely speaking, obtaining low network latency and high bandwidth can be considered a solved problem (at least from the research point of view). However, the problems of achieving cluster manageability, high-availability and checkpoint/restart without hurting the performance of applications under normal conditions are still far from resolved. In particular, the distributed nature of these clusters and the relatively high probability of partial failures that is inherent in distributed environments make these problems hard.

In this work we describe Starfish, a system that tries to tackle these issues using a novel architecture that combines group communication technology and checkpoint/restart. Starfish as a system is manageable, highly available, and adjusts to dynamic changes in the cluster. For its applications, Starfish provides hooks to handle dynamic cluster changes, as well as checkpoint/restart facilities. The initial implementation of Starfish runs on Linux, and supports both Myrinet [3] (for performance) and IP (for convenience). Porting to other fast networks like ServerNet^{TM} [4] is planned, and only requires writing a thin layer of code, as will be described later in this paper. Also, the bulk of the code is written in OCaml [6]; this part is fully portable to most variants of Unix, as well as Windows NT. The only parts that need to be rewritten for these operat-

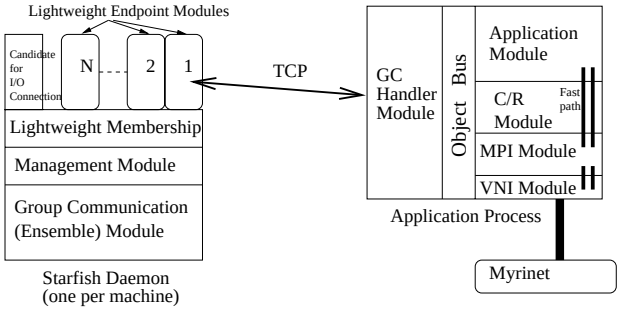

Figure 1. Starfish architecture

ing systems is the actual saving and restoring of the process state, and adapting the VNI code for Myrinet (or other fast networks) to these systems.

Each Starfish node runs a *Starfish daemon* (or simply a *daemon*), as illustrated in Figure 1. All Starfish daemons form a *process group* [10], using the Ensemble group communication toolkit [5, 22]. As described later in this paper, these daemons are used to interact with clients, spawn MPI programs to which we refer to as *application processes*, track and recover from failures, and to maintain the configuration of the system. In particular, daemons utilize a lightweight group mechanism ala [33], in a manner similar to the group daemon proposed in [11], for keeping track of and reporting partial application and system failures.

Each application process is composed of 5 major components, as illustrated in Figure 1, and as reported in Section 2. These include, a *group handler*, which is responsible for communicating with the daemon, an *application part*, which includes the user supplied MPI code, a *checkpoint/restart module*, an *MPI module*, and a *virtual network interface* (or *VNI* for short). These modules communicate internally using an object bus based listener model. Also, the application part has a separate *fast data path* to and from the MPI module, which guarantees low latency and minimal impact on performance.

This architecture is also very flexible and portable, in the sense that it allows us to implement several different checkpoint/restart protocols, both coordinated and uncoordinated checkpointing [16, 30], and to run them side by side. In particular, we can run the same application with two different checkpoint/restart protocols, and compare them. Also, we can easily port Starfish to various networks, since all that is required is to provide a relatively thin interface layer inside our VNI.

Another interesting feature of Starfish relates to its API. The additional functionality of Starfish is supported through additional downcalls and upcalls. For each of the upcalls, there is some default handling procedure, and hence applications that do not wish to handle these upcalls can simply ignore them. Similarly, applications are not required to issue any downcalls. This allows Starfish to run regular MPI programs, without any modifications. Naturally, such programs will only enjoy part of Starfish capability, e.g., system initiated checkpointing, but not all the potential benefits of the system, e.g., user initiated checkpointing and dynamic reconfiguration. Conversely, programs that use the additional API calls can be automatically, or semi-automatically, transformed back into standard MPI programs by eliminating all Starfish specific downcalls. Since these calls only deal with checkpointing and reconfiguration, such programs will then run correctly on any standard MPI implementation.

Starfish is currently in advanced stages of development; we have an initial prototype of the system, and plans for an initial release later this year. Some initial performance measurements for this prototype are reported in this paper.

The rest of this paper is organized as follows: The general architecture underlying Starfish is presented in Section 2. Section 3 elaborates on the fault-tolerance and high-availability aspects of Starfish. An initial performance evaluation of the current prototype is presented in Section 4. We compare our work to related work in Section 5, and conclude with a discussion in Section 6.

2 Starfish Architecture

As discussed in the introduction, and as illustrated in Figure 1, each Starfish node runs a daemon, where all Starfish daemons are members of the same process group, called the *Starfish group*. This group is managed by the Ensemble group communication system [5, 22]. The collection of these daemons form Starfish' *parallel environment*, and they are responsible for spawning application processes, keeping track of applications health, managing the configuration and settings of the cluster, communicating with clients, and for providing the hooks necessary to provide fault tolerance for applications. In particular, all configuration and control messages, including those related to applications (but not data messages) are sent by the daemons using Ensemble.[1] The internal structure of daemons is described in detail in Subsection 2.1.

Given the parallel nature of MPI programs, each application is expected to be divided into several concur-

[1] Ensemble ensures reliable ordered message delivery, as well as consistent automatic failure and recovery detection. See Section 3.1 for more details.

rent processes, each potentially running on a separate node. As described shortly, each application process consists of a Starfish run-time environment and user supplied code. Starfish run-time library is responsible for interacting with the daemons, for checkpoint and restart, and for implementing MPI, whereas the user code is any given MPI C program. Subsection 2.2 below elaborates on the internal structure of the application process.

Finally, the user process and the daemon communicate with each other through a local TCP connection. The description of this protocol appears in Subsection 2.3 below.

2.1 Daemon Internals

Starfish daemons need to maintain some application specific data for each application processes running on the same machine, as well as some shared state that defines the current cluster configuration and settings. We maintain this data by employing the design illustrate in Figure 1. Daemons compose of four main modules: *Ensemble* (the group communication system we use), the *management module*, the *lightweight membership module*, and several instances of the *lightweight endpoint module* [21]. Daemons may exchange *control messages* among themselves; these messages may be generated by either module, and are sent through Ensemble, but are not passed to application processes. (See Table 1.)

In our design, a single management module is used to maintain the configuration and shared settings of the cluster. Each application that runs on Starfish is associated with a lightweight process group, whose members are the daemons on the nodes that run the corresponding application processes. For each application process we instantiate a lightweight endpoint module. This module is the one that is responsible for the connection with the application process, and for passing messages and events to and from that process. The lightweight membership module is responsible for deducing the lightweight group membership from the entire Starfish process group, and for translating membership and message events between the main Starfish group and each of the lightweight groups. (See also Figure 2.)

Note that typically the lightweight groups are only subsets of the main Starfish process group. This is because on a large cluster, each application spans less than the entire cluster. Thus, not every change in the Starfish group needs to propagate to every lightweight group. Similarly, if an application process on one node terminates, but the machine itself continues to run, then this should result only in a membership change

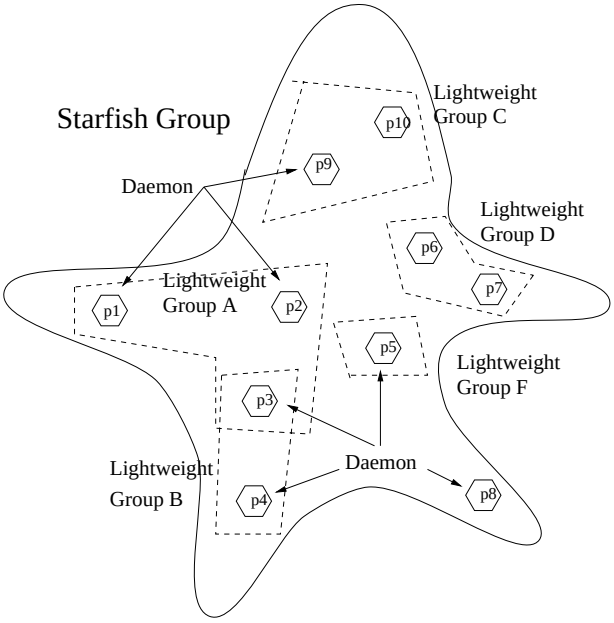

Figure 2. Lightweight groups within a Starfish group. Here, all daemons are members of the same Starfish group. Additionally, daemons p_1, p_2, and p_3 share the same lightweight group, indicating that there is an application that spans all three machines. Similarly, p_3 and p_4 share a lightweight group, etc. Note that p_8 does not appear in any lightweight group, indicating that no application process is currently running on the corresponding machine.

of the corresponding lightweight group, and need not be reported to all members of the main Starfish group, or in other lightweight groups. Also, messages that are sent inside a lightweight group should only be delivered to members of the same lightweight group.

Of course, it would have been possible to allocate a separate full blown process group for each application. But as indicated in [21], the lightweight group approach is more efficient. Also, the structure we described allows us to maintain both consistent cluster wide information, which is independent from a specific application, and manage multiple applications on top of this cluster, mimicking a parallel computer.

2.2 Application Process Internals

Each application process consists of the following components, as illustrated in Figure 1: a group handler, an application module, a checkpoint/restart mod-

ule, a MPI module, and a virtual network interface, hereafter called VNI. All modules communicate by posting events on an object bus, that invokes the corresponding event handlers at each of the listening module. Using an object bus allows us to completely decouple the modules, and also to potentially post the same events to more than one module. Finally, in order to orchestrate these modules, we have implemented our own scheduler.

Application processes are involved in five types of messages: *data messages*, *coordination messages*, *checkpoint/restart messages*, *lightweight membership messages*, and *configuration messages*.[2] (See Table 1.) There is a significant difference between data messages and other types of messages: Data messages result from the user supplied MPI code, and have strict performance requirements, while other messages are generated by Starfish itself and do not require the same responsiveness as data messages. Thus, we employ a fast data path between the MPI implementation and the application module, that does not go through the object bus. This ensures the required low latency for data messages, while still being able to provide manageability and strong guarantees for the other types of messages.

To send other types of messages, the generating module post an event for the group handler module and the group handler translates this event to a message on the TCP connection with the daemon. In the case of coordination messages and checkpoint/restart messages the daemon broadcast them in the relevant lightweight group using Ensemble. In the other direction, messages received by the group handler module are translated to events on the object bus, to be invoked at the corresponding modules in the application process. Using the daemon, and therefore Ensemble, for disseminating coordination messages greatly simplifies our code, and enjoys the strong delivery guarantees of Ensemble, which also simplify our protocols.

Lightweight membership messages and configuration messages are part of the protocol executed between the application processes and the daemons, and are discussed in the next subsection. Coordination messages are sent among application processes (potentially) located on different nodes for general coordination tasks, while checkpoint/restart messages are used by the various checkpoint/restart protocols. The set of checkpoint/restart messages seems to be rich enough to express all checkpoint/restart protocols we have encountered. Since lightweight membership messages and checkpoint/restart messages are exchanged by application processes, and daemons only serve as a reliable middle communication layer, these messages are opaque to daemons.

[2] As discussed earlier, there are also *control messages*, but in our terminology these messages are exchanges solely by daemons, and are therefore not discussed in this section.

2.2.1 Optimizing Receives

As in the MPI standard, Starfish supports both *blocking* and *non-blocking* send and receive operations. In blocking mode, a send or receive operation does not return until the message has been fully sent or received, while in non-blocking, operations return immediately. The eager way of implementing sends is to immediately send a message to its destination [20]. This, however, requires that the destination be prepared to read a message from the network, even if the application process there has not yet performed a matching receive operation.

In starfish we overcome this problem by introducing a low priority thread, called the *polling thread*. This thread continuously polls the network, so whenever a message arrives, the polling thread receives the message and puts it in a queue of received messages, for further handling by the application at a later time.

A nice feature of the polling thread is that it eliminates much of the runtime overhead of issuing a receive operation at the application level. A receive operation, particularly in blocking mode, pauses the process execution in order to receive an in-transit message. Moreover, when using the regular TCP/IP stack, receiving a message from the network involves a system call and user-level/kernel interaction, which is costly. When using the polling thread, the time required for kernel interaction is interleaved with other operations, yielding fast receive operations.

2.3 Interaction Between Daemon and Application Process

Brevity precludes us from specifying the exact protocol between the daemons and the application processes. Instead, here we only outline the main features of this protocol. The full version of this paper provides a more detailed description of the protocol.

As mentioned in the previous subsection, two types of messages are designated for the interaction between daemons and application processes. These are configuration messages and lightweight membership messages. Lightweight membership messages inform application processes about new views, and are used by an application process to terminate its membership in a lightweight group. Configuration messages, on the other hand, are used to inform the application process of various configuration parameters, and used to

Message type	Sent between
Control	Starfish daemons
Coordination	Application processes through daemons
Data	Application processes through MPI and VNI modules using fast path
Lightweight membership	Lightweight endpoint module and application processes
Configuration	Local daemon and application processes
Checkpoint/Restart	Checkpoint/restart modules through daemons

Table 1. Summary of message types in Starfish

synchronize between the application process and the daemon upon initialization and termination of the application process.

3 Manageability, Dynamicity, Fault Tolerance, and High Availability

In this section we elaborate on how we achieve cluster manageability and high-availability, how we support dynamic changes in the environment at both the clusters and the application, and provide fault-tolerance for the application. We split the discussion into general cluster aspects, which are reported in Subsection 3.1, and to application aspects, which are reported in Subsection 3.2.

3.1 Starfish Manageability, Dynamicity, and High Availability

3.1.1 Manageability

Starfish can be managed from any computer connected to the LAN on which the cluster runs, either directly, or through the Internet. Managing the cluster is done by opening a TCP connection to one of the daemons, on which an ASCII based protocol is used.[3] Through this connection, the cluster administrator can add or remove nodes from the cluster, disable and (re)enable nodes, and control the parameters of the cluster. The management protocol starts with a login session, in which the client side has to authenticate itself as an administrator to the cluster, and identify the connection as a *management connection*.

The management module of Starfish handles management connections, and takes care of forwarding configuration commands to all other daemons in the system. The use of ensemble's reliable and totally ordered delivered mechanism is instrumental here, in maintaining coherent state between all cluster daemons [10].

A similar protocol that also employs a TCP connection is used between clients and any of the cluster nodes in order to submit applications for either interactive or batch execution. This protocol also begins with a login session, but is identified as a *user session*, and is thus limited to submitting, suspending, resuming, and deleting applications. (A user can only suspend, resume, and delete its own applications.)

Note that user commands regarding an application have to be propagated to all daemons that manage the corresponding application processes, and in some cases should be forwarded from these daemons to the application processes themselves. This is done by reliably multicasting the corresponding messages in the appropriate lightweight group; the lightweight membership module at a daemon that receives such a message directs it to the corresponding lightweight endpoint module. The lightweight endpoint module than takes appropriate actions, and if necessary, passes the message to the application process as well.

3.1.2 Dynamicity

Starfish supports dynamic changes in the clusters. These changes can be the result of a node being added or deleted from the cluster, or might happen due to failures and recoveries of nodes. Each such change causes the group communication module to generate a new *view* event, which reports this change to all members of the cluster. Also, changes that affect only lightweight groups are reported in the lightweight group only, by the lightweight membership module. One of the main benefits of using an underlying group communication system is that our code does not need to explicitly keep track of such changes, since we can rely on Ensemble to report these changes for us.

3.1.3 High-Availability

Starfish is a highly-available system, in the sense that a failure of a few nodes does not cause the entire system to crash or hang. Instead, the system continues to run applications and to be available to clients. In particular, if none of the application processes of a given

[3] In the future, we plan to build a management GUI console, but currently managing the cluster requires textual interaction. At any event, having a textual protocol is useful for debugging and for automatic testing.

application was located on a failed node, then this application continues to run transparently. Similarly, as discussed in the next subsection, even if the application had one of its processes on a failed node, Starfish provides enough hooks to allow the application to overcome this failure, either by restarting the failed process on a different node, or by restructuring the computation.

Note that although the system as a whole does not stop due to a failure of a single node, client connections to this node might be lost. However, if the client reconnects to the system, he/she can continue the disrupted session from the point where the connection was cut off. At the moment clients have to explicitly choose the server they wish to connect to. However, in the future we plan to make this more transparent, using a *one-IP* type of solution [14].

3.2 Applications Dynamicity and Fault Tolerance

3.2.1 Dynamicity

Many applications can benefit from having more nodes added to them on-the-fly, while some applications might be able to accommodate loss of a few nodes on-the-fly. This is typical in applications that have trivial parallelism, since in this case usually each node works independently on a given subset of the computation space. Thus, changing the number or nodes dynamically simply requires restructuring the computation subspace on which each node computes so that the entire compute space is covered with no duplicates.

Another feature of Starfish that allows applications to cope with dynamic changes is the checkpoint/restart capability. Specifically, checkpoint/restart allows Starfish to migrate application processes from one node to another, e.g., if a better nodes becomes available, or a new node is added to the cluster.

3.2.2 Fault Tolerance

Starfish offers two forms of fault-tolerance for applications: The main fault-tolerant mechanism employed by Starfish is checkpoint/restart. The checkpoint/restart module of Starfish is capable of performing both coordinated and uncoordinated checkpoint, which is either system driven or application driven. Thus, when a node failure occurs, Starfish can automatically restart the application from the last checkpoint. Also, in some versions of uncoordinated checkpointing, it is enough to only start the failed process from its last stored checkpoint. As we mentioned earlier in this paper, this allows Starfish to run multiple checkpoint/restart protocols side by side. In particular, we can compare various checkpoint/restart protocols on the same platform.

The other form of fault-tolerance offered by Starfish is more application dependent, and is suitable mostly to applications that can be trivially parallelized. For such applications, whenever a node that runs one of the application processes crashes, a view event is delivered to all surviving application processes. This is done by having the application process registering a listener handler with the object bus for membership events. (Note that applications that cannot utilize view changes simply do not register listeners for membership events, and their programming model remains the conventional MPI model.) Once the surviving members learn about the failure of a node, they can repartition the data sets on which each process computes, and continue to run without interruption.

When an application is submitted to Starfish, the client can also determine the fault-tolerant policy that should be applied to this application, i.e., should automatic restart or view notifications be used, and some rules regarding how to choose the node on which a process will be started after a partial failure. For compatibility, there is also an option to kill an application whenever one of the nodes dies in the middle of its execution, which mimics non fault tolerant systems.

4 Performance

In this section we report on some initial performance measurements conducted on the prototype of Starfish. The performance measurements were obtained using 300 MHz Pentium II computers, connected by both Ethernet and Myrinet. In the case of Ethernet, we used the regular IP stack, while with Myrinet we used the BIP user-level interface [1].

Checkpoint time: Figure 3 shows both the serial checkpoint time and the parallel checkpoint time using the stop-and-sync protocol [16]. Note that as expected, the checkpoint time grows linearly with the size of the checkpointed data. The checkpointing time is on the order of seconds. Hence, if a checkpoint is taken once every hour, it would only slow down the entire execution time by less than 1%. Also, the hardware used for these measurements is not the most advanced, and employs regular IDE bus and controller. Newer and faster hardware is likely to result is faster saving times.

The smallest data point in Figure 3 is for a checkpoint file of size 632KB. This corresponds to checkpointing an empty program, or in other words, this indicates the checkpoint overhead imposed by our system. We attribute this low number to our architecture,

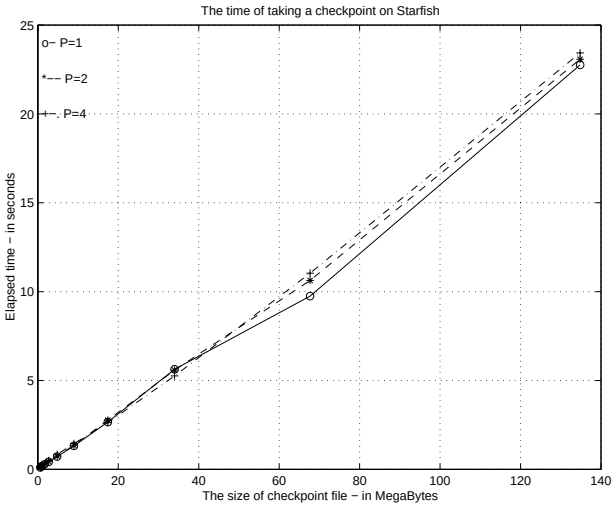

Figure 3. Checkpoint time vs. data size. The smallest data point is 632KB, which takes 0.104061 seconds for one node, 0.131898 seconds for two nodes, and 0.149219 for four nodes

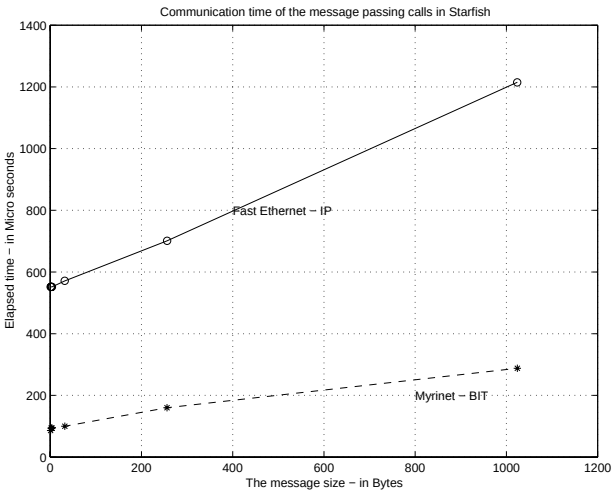

Figure 4. Round-trip delay vs. data size. The smallest data point is 1 Byte, at 86 microseconds for BIP/Myrinet and 552 microseconds for TCP/IP

in which the run-time system on each node is divided between the application process and the daemon. The daemon, which accounts for most of the code, is shared between all processes on the same node, and is written in a way that we never have to save or recover its state. The only part of the run-time system that needs to be saved is the one that is included in the application process, and that part is relatively small.

Round-Trip Delay: In order to measure the application level round-trip latency, we have implemented a simple ping-style application. That is, one node sends a short message to another node, who immediately replies. We then measure the elapsed time between sending the message and receiving the reply at the application level. This is done repeatedly a hundred times to get the average round-trip latency. We have measured the round-trip delay with both TCP/IP and BIP/Myrinet. The results of these measurements are reported in Figure 4. It can be seen that the round-trip delay grows linearly with the size of the data.

The round-trip time for an empty message is 86 microseconds using BIP/Myrinet and 552 microseconds using TCP/IP, or in other words, roughly 46 microseconds one way with BIP/Myrinet and 226 microseconds one-way using TCP/IP. This time is the net overhead imposed by our system in handling a message, plus the network latency. It includes getting the message from the application and putting it on the network, and retrieving the message from the network, and then all the way back to the application. Also, these measurements were obtained on a non optimized prototype, running as byte-code. From our experience, the actual time for the optimized native code are expected to be much smaller.

System Overheads: Figure 5 reports on the time a message spends in each layer of our code. Here again, we refer to the non-optimized prototype, running as byte-code. Also, note that the time spent in each layer is independent of the message size, since messages are never copied in our code.

5 Related Work

As discussed earlier in this paper, Starfish supports several approaches to checkpoint/restart and employs group communication for providing fault-tolerance and high availability. In addition, Starfish allows dynamic changes in the number of running processes. Most of these features and other properties of Starfish have been studied. Thus, there are several distributed systems, in academia as well as in industry, that have some of these features.

There are several systems that offer checkpoint/restart capabilities, e.g., Condor [25], Manetho [15], and LoadLeveler [2], and quite a few protocols and techniques for checkpoint/restart have been

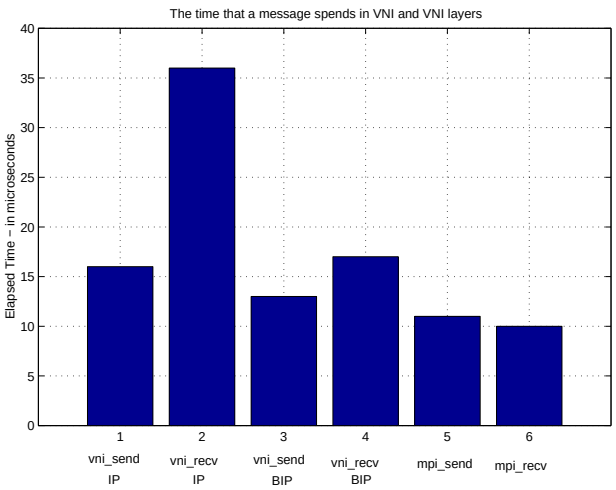

Figure 5. Layers overhead for sending and receiving messages

proposed. Generally, checkpoint/restart protocols can be categorized as either coordinated, in which case all processes coordinate their checkpointing to form a global consistent state [12, 16, 30], or as uncoordinated, in which case every process can perform checkpointing independently [30, 32, 34, 27]. One of the important aspects of Starfish architecture is that it enables us to implement and study both coordinated and uncoordinated checkpointing within a single framework.

Given the large number of high-performance distributed computing systems built, it is impossible to mention all of them. Here we discuss the ones we feel are more related to our work, and examine their architecture and functionality compared to Starfish.

Condor is a distributed system that runs on a cluster of workstations [25]. Condor provides an environment for executing serial and parallel applications on clusters. Moreover, it supports checkpoint/restart in order to provide fault tolerance and process migration [26]. Condor employs the sync-and-wait protocol [29], which is coordinated, and imposes several restrictions on checkpoint/restart in programs. As we mentioned before, our architecture allows us to implement, side-by-side, both coordinated and uncoordinated protocols. Also, we believe that using Starfish architecture we can remove most of the restrictions imposed in Condor.

Manetho is a distributed system that runs on a cluster of workstations [15]. This system uses a novel combination of rollback-recovery and process replications to provide fault tolerance and high availability; Manetho uses coordinated checkpointing protocol as described in [12], and uses process replication to provide high availability to servers in the system [10]. Aside from supporting both coordinated and uncoordinated protocols, Starfish high availability mechanisms are somewhat different than Manetho, as we use a group communication system to manage our cluster.

Libckpt is a transparent checkpointing library on uniprocessors running UNIX [31]. It provides a mechanism for enabling fault-tolerance for long-running programming. Libckpt implements most optimizations that have been proposed to improve the performance of checkpointing, including, e.g., incremental checkpointing, forked checkpointing, copy-on-write checkpointing [30]. However, libckpt is merely a library, whereas Starfish is a complete system. Also, libckpt does not address high availability and dynamicity as we do.

LoadLeveler is a distributed system that runs on a cluster of workstations [2]. It provides an environment for executing serial and parallel applications with dynamic scheduling. In addition, it supports checkpoint/restart only with serial jobs in order to balance workload and provide process migration, and is thus incomparable to Starfish.

Legion is an object-based meta-system [19]. It has been built on a collection of connected hosts to provide a virtual computer that can access all types of data and physical resources. Legion is designed to be a world-wide virtual computer, while Starfish is designed to be a reliable and highly available distributed system for executing message-passing applications on clusters.

HPVM is a distributed system that runs on a cluster of PCs with Windows NT [13]. This system achieves high performance communication by using modern processors (300 MHz Pentium II) and FM protocol for communication on Myrinet [7, 3]. In addition, the system includes efficient implementation of standard scientific computing APIs such as MPI [17]. However, HPVM does not support fault tolerance or high availability.

Millipede is a Distributed Shared Memory (DSM) system that runs on a cluster of workstations. It supports various consistency models of DSM [18], as well as thread migration inside the cluster for load-sharing and to improve the locality of memory references [24]. Millipede however, does not support fault tolerance, parallel I/O, security, and more [23]. On the other hand, Starfish system supports process migration to provide load-balancing, and in other cases to provide fault tolerance [30].

6 Discussion

Clusters of workstations offer a potential for cost effective high-performance computing. However, building usable clusters is in inherently difficult task. Successful implementations of such clusters must retain high-performance, while addressing issues like manageability, fault-tolerance, high-availability, and coping with dynamic changes in the environment.

In this paper we report on the design and architecture of Starfish, a system that tries to tackle these issues. We believe that Starfish architecture is novel in the specific way it addresses all of the above concerns. Starfish serves as a good testbed for new checkpoint/restart protocols and is also a fairly portable system in its design. The performance of the initial Starfish prototype is promising, although some additional fine tuning is still needed.

Looking into the future, developing newer and faster checkpoint/restart protocols, in particular ones that utilize fast networks, is a natural research direction. Also, trying to eliminate many of the restrictions that typical checkpoint/restart systems impose would also be desirable.

Acknolwdgements: We would like to thank the anonymous referees for their helpful comments.

References

[1] Basic Interface for Parallelism. http://lhpca.univ-lyon1.fr/bip.html.

[2] LoadLeveler home page. http://www.austin.ibm.com/software.

[3] Myricom Home Page. http://www.myri.com.

[4] Tandem Home Page. http://www.tandem.com.

[5] The Ensemble Home Page. http://www.cs.cornell.edu/Info/Projects/Ensemble.

[6] The OCaml Home Page. http://pauillac.inria.fr/ocaml.

[7] Y. Amir, L. E. Moser, P. M. Melliar-Smith, D. Agarwal, and P. Ciarfella. Fast Message Ordering and Membership Using a Logical Token-Passing Ring. In *Proc. of the 13th International Conference on Distributed Computing Systems*, pages 551–560, May 1993.

[8] T. Anderson, D. Culler, and D. Patterson. A Case for NOW (Network of Workstations). *IEEE Micro*, February 1995.

[9] A. Basu, V. Buch, W. Vogels, and T. von Eiken. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. In *Proc. of the 15th ACM Symposium on Operating Systems Principles*, pages 40–53, December 1996.

[10] K. Birman. The Process Group Approach to Reliable Distributed Computing. *Communications of the ACM*, 36(12):37–53, December 1993.

[11] K. Birman, R. Friedman, and M. Hayden. The Maestro Group Manager: A Structuring Tool For Applications With Multiple Quality of Service Requirements. Technical Report TR96–1619, Department of Computer Science, Cornell University, March 1996.

[12] K. M. Chandy and L. Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Transanctions on Computer Systems*, 3(1):63–75, February 1985.

[13] A. Chien, M. Lauria, R. Pennington, M. Showerman, G. Ianello, M. Buchanan, K. Hane, L. Giannini, G. Koenig, S. Krishnamurthy, Q. Liu, S. Pakin, and G. Sampemane. The Design and Evaluation of an HPVM-based Windows-NT Supercomputer. Unpublished manuscript, 1999.

[14] O. P. Damani, P. Y. Chung, Y. Huang, C. Kintala, and Y. M. Wang. One-IP: Techniques for Hosting a Service on a Cluster of Machines. In *Proc. of the 6th World Wide Web Conference*, April 1997.

[15] E. N. Elnozahy. *Manetho: Fault Tolerance in Distributed Systems Using Rollback-Recovery and Process Replication.* PhD thesis, Houston Unversity, October 1993.

[16] E. N. Elnozahy, D. B. Johnson, and Y. M. Wang. A Survey of Rollback-Recovery Protocols in Message-Passing Systems. Technical Report CMU-CS-96-181, Department of Computer Science, Carnegie Mellon University, October 1996.

[17] M. P. I. Forum. MPI-2: Extensions to the Message-Passing Interface. http://www.mcs.anl.gov/mpi, July 1997.

[18] R. Friedman, M. Goldin, A. Itzkovitz, and A. Schuster. Millipede: Easy Parallel Programming in Available Distributed Environments. *Software: Practice and Experience*, 27(8):929–965, August 1997.

[19] A. Grimshaw and W. Wulf. The Legion Vision of a Worldwide Virtual Computer. *Communications of the ACM*, 40(1), Jan. 1997.

[20] W. Gropp and E. Lusk. Mpich working note: Creating a new mpich device using the channel interface. Technical Report ANL/MCS-TM-000, Argonne National Laboratory.

[21] K. Guo and L. Rodrigues. Dynamic Light-Weight Groups. In *Proc. of the 17th International Conference on Distributed Computing and Systems*, pages 33–42, May 1997.

[22] M. Hayden. The Ensemble System. Technical Report TR98-1662, Department of Computer Science, Cornell University, January 1998.

[23] A. Itzkovitz, A. Schuster, and L. Shalev. The Millipede Virtual Parallel Machine for NT/PC Clusters. http://www.cs.technion.ac.il/Labs/Millipede/millipede.html.

[24] A. Itzkovitz, A. Schuster, and L. Wolfovich. Thread Migration and its Applications in Distributed Shared

Memory Systems. *The Journal of Systems and Software*, 1998. To appear. Also available as Technion CS Technical Report LPCR #9603.

[25] M. Litzkow, M. Livny, and M. Mutka. Condor: A hunter of idle workstations. In *Proc. of the 8th Int'l Conference on Distributed Computing Systems (ICDCS'88)*, 1988.

[26] M. Litzkow, T. Tannenbaum, J. Basney, and M. Livny. Matchmaking: Distributed Resource Management for High Throughput Computing. Technical Report 1346, University of Wisconsin-Madison Computer Sciences, April 1997.

[27] R. H. B. Netzer and J. Xu. Adaptive Independent Checkpointing for Reducing Rollback Propagation. Technical Report CS-93-25, Department of Computer Science, Brown University, September 1993.

[28] S. Pakin, V. Karamcheti, and A. A. Chien. Fast Messages (FM): Efficient, Portable Communication for Workstations Clusters and Massively Parallel Processors. *IEEE Concurrency*, 5(2):60–73, 1997.

[29] J. S. Plank. *Efficient Checkpointing on MIMD Architectures.* PhD thesis, Princeton Unversity, January 1993.

[30] J. S. Plank. An Overview of Checkpointing in Uniprocessor and Distributed Systems, Focusing on Implementation and Performance. Technical Report UT-CS-97-372, Department of Computer Science, Tennessee University, July 1997.

[31] J. S. Plank, M. Bech, G. Kingsley, and K. Li. Libckpt: Transparent Checkpointing Under UNIX. In *Usenix Winter 1995 Technical Conference*, pages 220–232, New Orleans, January 1995.

[32] B. Randell. System Structure for Software Fault Tolerance. *IEEE Trans. on Software Engineering*, SE-1(1):220–232, June 1975.

[33] L. Rodrigques, K. Guo, A. Sargento, R. van Renesse, B. Glade, P. Verissimo, and K. Birman. Reducing Interprocessor Dependence in Recoverable Distributed Shared Memory. In *Proc. of the 13th Int. Symp. on Reliable Distributed Systems*, pages 34–41, 1994.

[34] Y. M. Wang and W. K. Fuchs. Scheduling Message Processing for Reducing Rollback Propagation. In *Proc. IEEE Fault-Tolerance Computing Symposium*, pages 204–211, July 1992.