IEEE DISTRIBUTED SYSTEMS ONLINE 1541-4922 © 2005 Published by the IEEE Computer Society
Vol. 6, No. 7; July 2005

NeCoMan: Middleware for Safe Distributed-Service Adaptation in Programmable Networks

Nico Janssens, *Katholieke Universiteit Leuven*
Wouter Joosen, *Katholieke Universiteit Leuven*
Pierre Verbaeten, *Katholieke Universiteit Leuven*

NeCoMan middleware customizes dynamic adaptation of point-to-point-based network services in programmable networks, taking into account the properties of the network services involved as well as the reconfiguration semantics.

Programmable networks let third parties dynamically reprogram the network elements. By opening the execution environment of routers, firewalls, base stations, and so on, users and service providers can adapt the behavior of these network elements to meet their own needs. Programmable networks are therefore an interesting technology for building adaptive networks as well as for supporting the evolution of networking software.[1]

This article focuses on safe runtime adaptation of point-to-point-based network services in programmable networks. Many network services such as compression, fragmentation, reliability, and encryption conform to a point-to-point-based distributed-service model. This model represents a pair of tightly coupled components (not necessarily situated on neighboring nodes) that must cooperate to offer a distributed service. To prevent the runtime addition, replacement, or removal of these network services from jeopardizing a programmable network's correct functioning, the network programmer must coordinate the adaptation of the protocol stacks accommodating the service components. For example, safe runtime deployment of an MPEG software encoding service on two programmable nodes requires, among other actions, installing and activating the MPEG decoder before bringing the encoder into use. Otherwise, both the encoding service and the network integrity could be compromised during adaptation, since the encoded packets most likely can't be processed correctly when arriving at their destination.

To conduct safe distributed-service adaptation in programmable networks, we've developed the NeCoMan (*Ne*twork *Co*nsistency *Man*agement) middleware. This middleware coordinates the addition, replacement, and removal of point-to-point services among programmable nodes at runtime. The novelty of this middleware is in its ability to customize adaptation, taking into account both the characteristics of the network services that are involved and the reconfiguration semantics. Earlier work[2] introduced the need for and design of the NeCoMan middleware, its initial adaptation algorithm, and a limited set of customizations. Here, we present an updated version of the adaptation algorithm as well as an extensive set of customizations.

Requirements and motivation

Three main requirements have driven the NeCoMan middleware's development. First, NeCoMan should relieve the developer of point-to-point network services from implementing ad hoc adaptation algorithms. To accomplish this, NeCoMan must coordinate the safe distributed adaptation of an extensive set of point-to-point services. By packaging this adaptation complexity in the middleware, we aim to make dynamic adaptation of programmable networks less complex and error prone.

Second, distributed adaptations must be executed as transparently as possible. The quality requirements imposed on computer networks imply that service disruption caused when adapting programmable nodes must be kept to a minimum. At best, end users shouldn't be able to detect that an adaptation has occurred inside the network—that is, they shouldn't experience unexpected delays or service disruptions. To cope with this requirement, NeCoMan aims to speed up the adaptation process whenever possible by taking into account the characteristics of the services involved as well as the reconfiguration semantics.

Finally, to promote reuse, the NeCoMan middleware should be decoupled from the execution environment of the programmable nodes that will be adapted. NeCoMan therefore can't use the adaptation primitives of one specific node execution environment. Instead, NeCoMan coordinates the recomposition of flow-oriented, component-based protocol stack architectures, such as Click[3] and DiPS (Distrinet Protocol Stack)/CuPS (Customizable Protocol Stack).[4] Consequently, adaptations of a protocol stack composition are expressed in terms of creating and removing components and connectors (mediating invocations among components). This type of node architecture exposes the following characteristics:

- Nodes become reprogrammed out-of-band: the recomposition of a protocol stack is initiated independently from the actual data packets traversing through the network.
- Data packets don't carry information that would let a node decide how to process them, except when required during adaptation.

Basic algorithm for safe distributed-software adaptation

Because NeCoMan must coordinate the adaptation of an extensive set of point-to-point services, the basic adaptation algorithm (see figure 1a) doesn't take into account the services' characteristics. We'll discuss the customizations of this general-purpose adaptation algorithm in the next section.

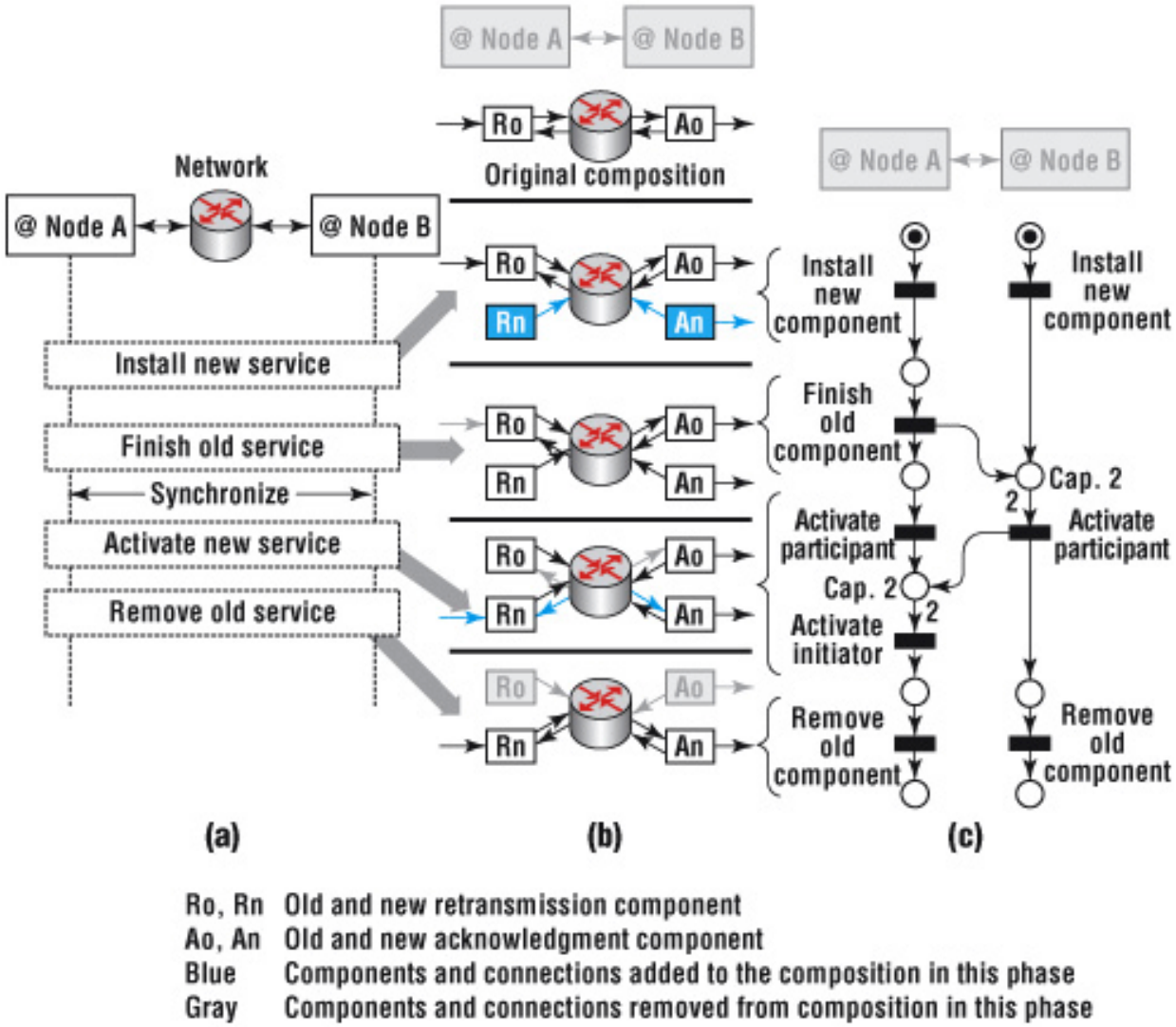

Figure 1. Basic algorithm for safe distributed-software adaptation: (a) the basic algorithm, (b) replacement of a reliability service by a new version, and (c) implementation of the basic adaptation NeCoMan uses to replace a reliability service.

We exemplify all phases of the basic adaptation algorithm with the runtime replacement of a reliability service by a new version that had a bug fixed (see figure 1b). This reliability service

consists of a retransmission component that collaborates with an acknowledgment component. To recover from packet loss, the retransmission component resends data packets (triggered by a time-out) until it receives an acknowledgment packet from the acknowledgment component. Both components depend on each other to offer this service. On the one hand, the retransmission component expects the acknowledgment component to filter out duplicate data-packet resends and to return an acknowledgment packet for each data packet that has arrived correctly. On the other hand, the acknowledgment component requires the retransmission component to react to these acknowledgments by removing the acknowledged data packets from its retransmission queue. To clarify the basic adaptation algorithm, the Petri net in figure 1c illustrates how NeCoMan conducts the reliability service's replacement.

The basic adaptation algorithm comprises four phases:

1. installing the new service,
2. finishing the old service,
3. activating the new service, and
4. removing the old service.

In addition, NeCoMan employs a synchronization protocol to coordinate the local execution of these phases among participating network nodes whenever necessary.

Installing the new service

As a first condition for safe distributed service adaptation, the new service components must be made available on both programmable nodes before the new service can be activated. The adaptation algorithm therefore begins by installing the new service, resulting in the coexistence of the old service (still in use) and the new version (not yet activated). This involves dynamically loading the components that implement the new service into the execution environment of both programmable nodes and then connecting these components into the protocol stack composition. Since the new service is not yet activated in this phase, the latter is limited to constructing the connectors that won't cause service invocation on the new components (more specifically, the connectors that mediate the outgoing invocations).

When replacing the reliability service, the service installation phase includes integrating the new retransmission and acknowledgment component into the protocol stack composition of the sending and the receiving network nodes, respectively. In addition, as illustrated in figure 1b, NeCoMan instructs the underlying node's execution environment to establish all connections that will not mediate packets to the new service components.

Finishing the old service

A second condition for safe service adaptation concerns the execution state of the old service that's still in use. Before activating the new service, all components implementing the old service must be *quiescent*, or consistent and frozen.[5] When software components are consistent, they don't include results of a partially completed service execution—the execution of the communication protocol used by both collaborating components has completed. By forcing software components to freeze, they have no pending invocations to be processed and are not processing any invocation.

Finishing the old reliability service involves only driving the old retransmission component to a quiescent state. If the component's retransmission queue is empty (indicating all transmitted data packets arrived correctly), then both the old retransmission and acknowledgment components are in a consistent state. To remain in this state during replacement, the old retransmission component must be frozen as well. This is accomplished by intercepting all data packets mediated to the old retransmission component for reliable transmission and deactivating the timer of the old retransmission component that triggers packet resends.

When the old service is finished and the components implementing the new service are installed on both network nodes, NeCoMan can safely initiate the new service's activation. A synchronization protocol guarantees that both participating nodes comply with this requirement before activating the new service (see figure 1a).

Activating the new service

Activating the new service implies redirecting invocations from the old service components toward the new service components as well as starting the execution of the active objects. When activating components that will collaborate with each other, we must coordinate the order in which they are brought into use. A component *C* can only be activated safely if all the other components on which *C* depends are already active to service *C*'s invocations. For point-to-point network services, these dependencies are formalized by the communication protocol that both service components use. Safe activation of a service therefore requires first activating the component or components that participate in the execution of that protocol before bringing into use the component that initiates the protocol execution.

Figure 1c illustrates how this third condition for safe distributed-service adaptation is fulfilled when activating the new reliability service. According to the communication protocol the new reliability components use, both the new acknowledgment component and the retransmission component participate in offering reliable communication by respectively processing data and acknowledgment packets. In addition, the new retransmission component initiates this service's execution by sending data packets to the acknowledgment component. To activate the new service participants, connectors are removed and created to redirect the new data and acknowledgment packets that will arrive from the network toward the new acknowledgment component and retransmission component, respectively. Next, after receiving a synchronization message indicating that the new

acknowledgment component is in use, NeCoMan instructs the node's execution environment to start the new retransmission component's timer and to redirect the data packets that are intercepted for finishing the old reliability service to this new retransmission component. At this point in the adaptation process, the service initiator is activated, and the new reliability service processes all packets in transit.

Removing the old service

As a final condition for safe service adaptation, the old service can be removed safely only when all its components have reached a quiescent state. Because this was a precondition as well for activating the new service, the old reliability components can be safely removed after activating the new version.

Customizations

Since the basic algorithm coordinates the distributed adaptation of a broad variety of services, it lacks two important requirements we impose on service adaptation in programmable networks. First, to guarantee service continuity, service disruption during the adaptation process must be minimized. This requires customizing the basic adaptation algorithm to service specific characteristics to optimize adaptation when possible. Second, NeCoMan should restrict the contribution needed from the service developer to conduct a safe adaptation. NeCoMan therefore must tailor the implementation of each phase in the adaptation algorithm to the service's characteristics.

To meet both requirements, the basic adaptation algorithm must be customized to reflect

- the characteristics of the distributed services that are involved and
- the reconfiguration semantics.

Characteristics of distributed services

Several characteristics of distributed services enable the NeCoMan middleware to customize the basic algorithm.

Stateful vs. stateless service. Depending on whether the services are *stateful* or *stateless*, we can switch the order of finishing the old service before activating the new one as adopted by the basic adaptation algorithm.

When a service is stateful, it shares its execution state with its clients. While replacing a stateful

service, the execution state of the old and new versions of that service must be kept consistent with the state that the client using the service expects. Consider the replacement of a TCP-like service by a revised version after the former has successfully completed a session setup—that is, having reached the `ESTAB` state. Activating the new TCP-like service before the old version reaches the `CLOSED` (finished) state would compromise the correct functioning of the client (which still expects the service to be in the `ESTAB` state). The old TCP-like service therefore must be finished before the new one is activated, like the basic adaptation algorithm does.

When a service is stateless, such as the reliability service, its execution state is irrelevant to its clients. When replacing such a stateless service, the new version of the service can be safely activated before the old one is finished. So, the NeCoMan middleware allows switching the order of executing the finishing and activating phase. Because finishing a service can be very time-consuming, NeCoMan can this way reduce the service disruption time by redirecting packets to the new service components before finishing the old version. To illustrate this, monitoring the retransmission queue of the old retransmission component *after* the new one is activated limits the impact of finishing the old reliability service on the service continuity.

Activating the new service before finishing the old service imposes several adjustments to the basic adaptation algorithm (see figure 2).

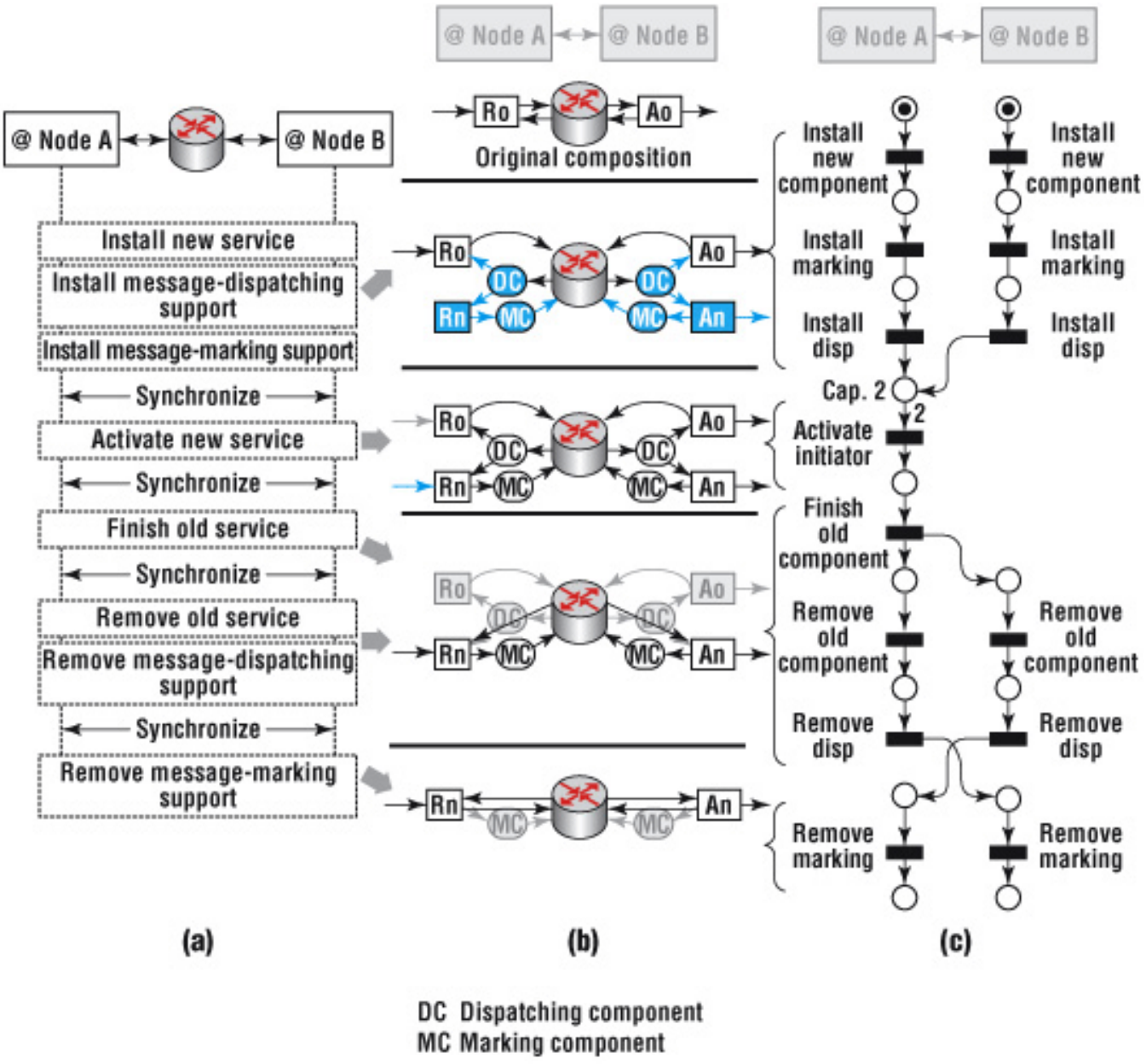

Figure 2. (a) A customized adaptation algorithm, (b) replacing a reliability service with a new version, and (c) implementing NeCoMan's customized adaptation to replace a reliability service.

Installing the new service. First, since both the old and the new (reliability) services will execute in parallel during the adaptation process, support is needed to distinguish between packets that each service processes. This includes support to

- mark transmitted packets to identify the service components they were processed by,
- demultiplex these packets when they arrive at the peer node, and
- delegate them to the appropriate service component.

Consequently, the first condition for safe service adaptation (as described earlier) must be made more stringent: the new service components as well as the associated support for distinguishing packets must be made available on both programmable nodes before the new service can be safely activated.

Regarding the packet-distinguishing support, during adaptation NeCoMan only marks packets (using *marking components*) that the new (reliability) service components have processed. Packets are demultiplexed by *dispatching components* ("disp" in figure 2), which are integrated in the protocol stack composition after installing the new service components (see figure 2b). These components delegate the unmarked packets to the old service component, while the others are delivered to the corresponding new service component.

Activating the new service. A synchronization message from the participating nodes guarantees that the components implementing the new (reliability) service and the packet-distinguishing support are installed on both network nodes. Once that message is received, the new (reliability) service is activated (see figure 2). Because the dispatching components deliver packets arriving from the network to the appropriate component—that is, the old or new retransmission and acknowledgment components—both protocol stacks are prepared to deal with packets the new service components process. Therefore, installing the packet-distinguishing support has already fulfilled the third condition for safe service adaptation (defining the order of activating the new service components). So, activating the new service boils down to redirecting packets that must be transmitted to the new service initiator (more specifically, to the new retransmission component for the reliability service).

Finishing and removing the old service. When synchronization messages guarantee that all new service components are activated, the old service is finished and removed. Additional synchronization messages ensure that the old service components are all quiescent before removing them. After the old components are finished, the new service processes all the packets that are between both nodes. Then, the packet-distinguishing support becomes redundant and is removed. This imposes an additional condition for safe service adaptation: coordinating the order of removing the marking and dispatching components. For both nodes, the marking component can be removed only if the dispatching component is removed from the peer node (see figures 2b and 2c).

To illustrate this, suppose this rule is ignored and the old retransmission component, the related packet-dispatching component, and the marking component of the new retransmission component are removed from node A before the old acknowledgment component is removed from node B. As a result, unmarked data packets that the new retransmission component processes will be delivered incorrectly by node B's dispatching component to the quiescent, old acknowledgment component. Hence, the correct functioning of the reliability service is broken.

Guaranteed packet ordering. When finishing the old service after the new version has been activated, both services execute concurrently during the adaptation. Consequently, packets processed by both versions most likely will get mixed. This customization therefore can't be applied when replacing a stateless service that ensures packets are delivered in the same order they're sent.

Consequently, when a stateless service offers guaranteed packet ordering, the NeCoMan middleware executes the basic adaptation algorithm to ensure that no message-ordering guarantees are violated during adaptation.

Unidirectional vs. bidirectional communication. The distributed dependencies between both cooperating components of a point-to-point service can be either unidirectional or bidirectional. The communication protocol that both collaborating components use formalizes these dependencies. Depending on whether the service uses a unidirectional or bidirectional communication protocol, a different implementation for installing and removing packet-distinguishing support is employed. This service characteristic therefore only customizes the implementation of the adaptation algorithm when the new service is activated before the old one finishes.

For unidirectional communication (see figure 3a), the service that will be added, replaced, or removed only covers the downstream or the upstream path between both nodes (such as an MPEG encoding service). Consequently, packet-marking and -dispatching components will be installed on and removed from only the sending node and the receiving node, respectively.

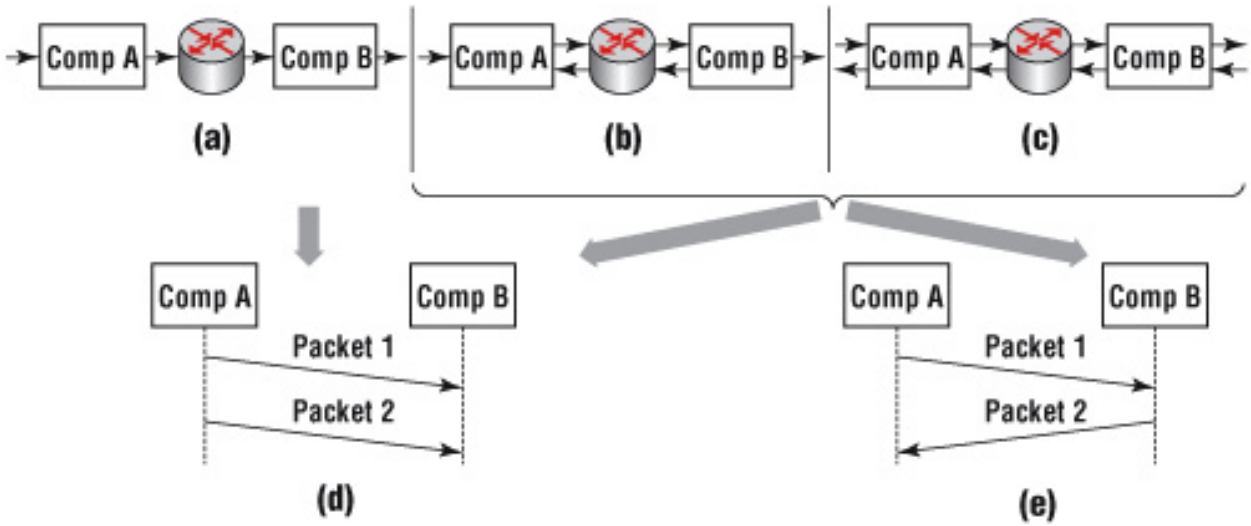

Figure 3. Communication characteristics. A (a) unidirectional communication protocol with one service initiator (component A), (b) bidirectional communication protocol with one service initiator (component A), (c) bidirectional communication protocol with both components service initiator, (d) asynchronous communication protocol, and (e) synchronous communication protocol.

When a service such as a reliability service adopts a bidirectional communication model (see figures 3b and 3c), both downstream (data packets) and upstream communication (acknowledgment packets) are part of the same service. In that case, the NeCoMan middleware installs and removes marking and dispatching components at both nodes to deal with the outgoing (transmitted) and the incoming

packets (arriving from the network).

Synchronous vs. asynchronous communication. The adaptation algorithm's finishing phase is implemented in a different manner depending on whether the old service that will be removed uses a synchronous or asynchronous communication protocol. For a synchronous communication protocol, the protocol's end message arrives at the node initiating the service protocol (see figure 3e). Since in this case the initiator is aware of the service being finished, finishing involves only driving the service initiator to a quiescent state. This is also the case when finishing the reliability service, which involves only monitoring the retransmission component until its retransmission queue is empty.

When the service employs an asynchronous communication protocol (more specifically, when the protocol doesn't end in the service-initiating component as illustrated in figure 3d), both nodes must be quiescent to finish the old service. This imposes an additional condition for safe service adaptation that must be fulfilled. The component initiating the old service protocol must be quiescent before driving the component where the protocol ends to a quiescent state. Finishing, for instance, an MPEG-software-encoding service requires first freezing the encoder, then monitoring the decoder until all encoded packets have arrived.

Service initiators. Since a new service is brought into use by activating the new-service-initiating component, the service activation phase gets customized depending on the number of service-initiating components. When the service contains one service-initiating component, service activation involves waiting for the synchronization message and starting the execution of the new service only at the node accommodating that component (as illustrated in figures 1c and 2c). Activating a service when both components are initiators (for example, when reliable communication is provided for the upstream as well as the downstream communication link) requires distributed activation of both service initiators.

Fail-safeness. When a service is *fail-safe*, the network's correct functioning in the absence of one of the service components won't be compromised. For example, protocol boosters conform to this definition.[6] Consequently, coordinated adaptation becomes redundant, because incorrectly adapting a service doesn't compromise the network's integrity. In that case, the synchronization messages are omitted, speeding up the adaptation process.

Reconfiguration semantics

In addition to the distributed service's specifications, the reconfiguration semantics also define a customized adaptation algorithm. These reconfiguration semantics specify the adaptation properties, as imposed by the service developer or the network environment.

Instant vs. delayed activation. Depending on the type of network service to be deployed, the service developer can adopt a different activation policy. Some service adaptations, such as security patches

or bug fixes, require instant activation. In this case, the new service becomes activated immediately after installation. For other services, delayed activation is more appropriate. Take, for example, an MPEG encoding service being employed to accommodate a wireless link in a home network when the bit error rate exceeds a certain threshold. To reduce the impact of the bandwidth consumption that the complete adaptation process causes, the encoding service can be installed in advance. Afterward, the protocol stack monitoring the link quality activates the encoding service when the bit error rate exceeds the threshold. This involves customizing the adaptation algorithm to allow for an external entity (in this case the monitoring support) to initiate the new service's activation.

Adaptation operation. Depending on the type of service adaptation—service addition, removal, or replacement—we can define two optimizations. First, adding a new service is similar to replacing a dummy service, which contains no functionality, with this new version. Since a dummy service is inherently consistent at any moment in time, there's no need to finish the dummy service before activating the newly added service. Second, coordinating the distributed adaptation process isn't required when replacing a point-to-point service with a new version that is semantically equivalent with the old one and that employs the same communication protocol. When these conditions are fulfilled, the new service components' behavior is backward compatible with that of the old components, omitting the need for synchronizing the local protocol-stack adaptations. An example of such an adaptation is replacing an MPEG encoding service with a new version extended with nonfunctional support, such as logging.

Completeness

Since we aim to limit the contribution needed from the service developer, NeCoMan must provide a complete set of customizations, as well as an extensive set of service characteristics to identify which customizations apply to each service. We defined the customizations we listed by generating all combinations and implementations of the basic adaptation algorithm's phases. From this set, we selected the customizations that both fulfill the conditions for safe service adaptation and cause minimal service disruption. We therefore claim that, although we haven't yet provided a formal proof, the presented set of customizations most likely is complete for service adaptation in component-based, flow-oriented, protocol stack architectures.

However, because each service can comprise a virtually unlimited set of functionalities, we can't claim the completeness of the listed service characteristics that classify the customizations. These service characteristics are important, though, since they all reflect the dependencies between the service components as well as the dependencies between the service and its clients. Consequently, these characteristics affect the coordination of the adaptation process.

Current status and evaluation

As a proof of concept, we implemented the NeCoMan middleware in Java on top of DiPS/CuPS, a

component framework for building runtime-adaptable protocol stacks. However, we developed the NeCoMan middleware independently from the underlying node architecture. NeCoMan therefore invokes adaptations of the protocol stack composition in terms of creating, removing, linking, and unlinking components (as Jeff Kramer and Jeff Magee proposed[5]). In addition, extra primitives are required from the node execution environment to let NeCoMan activate, deactivate, and finish the service components. Finally, the current implementation uses TCP socket communication for reliable exchange of the synchronization messages.

Conclusion

The current version of the NeCoMan middleware still has room for improvement. First, NeCoMan coordinates the adaptation of one-to-one services. However, many services, such as the MPEG-software-encoding service, employ a one-to-many or even many-to-many communication model. Second, although we designed NeCoMan to be reused on top of various component-based, flow-oriented, protocol stack frameworks, extra validation is needed to confirm its portability. Finally, the current version of NeCoMan cannot select the most optimal adaptation algorithm for each service. This ability is required for additional specification of the service. For instance, when replacing a stateless service that takes a long time to finish, activating the new service before the old one finishes will most likely reduce service disruption. But, when the service reaches the finished state in a negligible period of time, the basic adaptation algorithm might be a better choice. This solution needs less synchronization and lacks the potential performance overhead that the packet-distinguishing support causes.

We will continue to work on these issues in future research. (See the Related Work sidebar for a discussion of previous work in the field.) In addition, we're currently working on a formal proof of the customizable adaptation algorithm's correctness as well as of the customizations' completeness.

Acknowledgments

We performed the research for this article with financial support from the Fund for Scientific Research Flanders, Belgium—F.W.O. RACING #G.0323.01.

References

1. A.T. Campbell et al., "A Survey of Programmable Networks," *ACM SIGCOMM Computer Communications Rev.* , vol. 29, no. 2, 1999, pp. 7-23.
2. N. Janssens et al., "NeCoMan: Middleware for Safe Distributed Service Deployment in Programmable Networks," *Proc. 3rd Workshop Adaptive and Reflective Middleware* (RM 2004), ACM Press, 2004, pp. 256-261.

3. E. Kohler et al., "The Click Modular Router," *ACM Trans. Computer Systems* , vol. 18, no. 3, 2000, pp. 263-297.
4. N. Janssens et al., "Towards Hot-Swappable System Software: The DiPS/CuPS Component Framework," *Proc. 7th Int'l ECOOP Workshop Component-Oriented Programming* (WCOP 2002), 2002, http://research.microsoft.com/~cszypers/Events/WCOP2002/06_Janssens.ps.
5. J. Kramer and J. Magee "The Evolving Philosophers Problem: Dynamic Change Management," *IEEE Trans. Software Eng.*, vol. 16, no. 11, 1990, pp. 1293-1306.
6. W.S. Marcus et al., "Protocol Boosters: Applying Programmability to Network Infrastructures," *IEEE Communications Magazine* , vol. 36, no. 10, 1998, pp. 79-83.

Nico Janssens is a PhD research student at Katholieke Universiteit Leuven's Department of Computer Science. His research interests include dynamic software reconfigurations, programmable networks, and adaptive middleware. He received his master's degree in computer science from K.U. Leuven. Contact him at DistriNet, Dept. of Computer Science, K.U. Leuven, Celestijnenlaan 200A, B-3001 Leuven, Belgium; nico.janssens@cs.kuleuven.be.

Wouter Joosen is a professor at Katholieke Universiteit Leuven's Department of Computer Science. His research interests include software architecture for distributed systems, aspect-oriented and component-based software development, adaptive middleware, security solutions, and secure software. He received his PhD in computer science from K.U. Leuven. Contact him at DistriNet, Dept. of Computer Science, K.U. Leuven, Celestijnenlaan 200A, B-3001 Leuven, Belgium; wouter.joosen@cs.kuleuven.be.

Pierre Verbaeten is a full professor at Katholieke Universiteit Leuven's Department of Computer Science. His research interests include open system software, dynamic configuration and integration, and ad hoc networks. He received his PhD in computer science from K.U. Leuven. He's a member of the ACM and IEEE. Contact him at DistriNet, Dept. of Computer Science, K.U. Leuven, Celestijnenlaan 200A, B-3001 Leuven, Belgium; pierre.verbaeten@cs.kuleuven.be.

Cite this article: Nico Janssens, Wouter Joosen, and Pierre Verbaeten, "NeCoMan: Middleware for Safe Distributed-Service Adaptation in Programmable Networks," *IEEE Distributed Systems Online*, vol. 6, no. 7, 2005.

Related Work

A lot of research has been done in the field of dynamic distributed software adaptations. We focus on two research tracks closely related to our work.

Programmable networks

Wen-Ke Chen and his colleagues present a system to coordinate the dynamic adaptation of distributed services and communication protocols.[1] Although this architecture and the NeCoMan middleware both coordinate distributed service adaptations, we point out three differences. First, Chen's adaptation algorithm always activates the new service before finishing the old one. When guaranteed packet ordering is required, the switch-over of incoming packets (arriving from the network) from the old to the new service components is deferred until the old components have processed all incoming packets. In addition, their adaptation algorithm only replaces services. Second, unlike NeCoMan, Chen's adaptation system is an open framework that requires the service components' developer to implement part of the adaptation algorithm. Examples include the activation of a new service component as well as some optimizations to the adaptation algorithm. Finally, Chen's adaptation system wasn't designed to be reused on top of various node architectures. The system is an extension of Cactus, a framework for constructing highly configurable middleware and protocol stacks.[2] The adaptation system is therefore tightly coupled to the Cactus syntax as well as to its architecture.

Robbert van Renesse and his colleagues present Ensemble, a hierarchical framework for constructing adaptive protocol stacks, implementing an adaptation by replacing the entire protocol stack.[3] An adaptation in Ensemble is coordinated by the Protocol Switch Protocol (PSP), a fault-tolerant protocol that synchronizes the participating stacks, assists them in finalizing their state, and performs the necessary agreement before resuming communication. Similar to the NeCoMan middleware, the approach aims to facilitate the programming complexity of dynamic distributed adaptations.[3]

The PSP's adaptation algorithm is comparable to the basic adaptation algorithm we present, but it doesn't provide support for dynamically loading services. In addition, the algorithm coordinates the replacement of entire protocol stacks rather than coordinating the recomposition of operating protocol stacks. Consequently, coordinating the sequence of activating the new protocol stacks is unnecessary, since a stack can be invoked only when it's fully operational.

Adaptive and reflective middleware

Lasagne, an architecture for building runtime-adaptable middleware layers, uses a different approach

from ours to guarantee service consistency during dynamic adaptation.[4] Rather than replacing components with new versions (so as to change the system's composition), the Lasagne model employs a wrapper-based component-extension mechanism. Customizations are based on additive refinements of stable core components, to be expressed on a per-request basis. Once a client selects an extension, this client-specific customization propagates with the message flow of the entire collaboration, providing system-wide execution consistency. Consequently, conducting safe distributed-software adaptation boils down to coordinating the proper installation of the wrappers on each host before being selected by a client request.

Fabio Kon and his colleagues describe two different implementations of reflective middleware systems: dynamicTAO, which is an extension of the TAO ORB, and OpenORB, developed at the University of Illinois and Lancaster University, respectively.[5] In dynamicTAO, *component configurators* reify the distributed dependencies between components. These first-class entities can be extended by inheritance to provide customized implementations dealing with different kinds of components. This way, middleware developers can write code that coordinates safe, dynamic reconfiguration of the middleware components. Francisco José da Silva e Silva and his colleagues used these component configurators for the same purpose.[6]

In OpenORB, dynamic reconfigurability is achieved with the extensive use of reflection, with a clear separation between base- and meta-level. Two metaspace models support *structural reflection* (the ability to reify the program currently executing): *interface* and *architecture*. The interfaces metamodel defines a component's external representation in terms of a set of provided and required interfaces. The architectural model, in turn, defines a component's internal implementation in terms of the software architecture. The associated metaobject protocol provides the ability to dynamically inspect and adapt the software architecture. Geoff Coulson and his colleagues discuss OpenORB's application to the area of programmable networks.[7]

Finally, Nanbor Wang and his colleagues illustrate how the *CIAO/QuO* middleware expands the capabilities of conventional component middleware to facilitate static and dynamic quality-of-service provisioning.[8] *Qedo* works toward the same goal, targeting applications in the telecommunications domain.[9] Dynamic adaptations in these systems don't typically need coordination, since an adaptation only involves disconnecting and reconnecting remotely collaborating components.

References

1. W.-K. Chen et al., "Constructing Adaptive Software in Distributed Systems,"*Proc. 21st Int'l Conf. Distributed Computing Systems* (ICDCS 2001), IEEE CS Press, 2001, pp. 635–643.
2. G.T. Wong, M.A. Hiltunen and R.D. Schlichting, "A Configurable and Extensible Transport Protocol,"*Proc. 20th Ann. Joint Conf. IEEE Computer and Communications Societies* (INFOCOM 2001), IEEE CS Press, 2001, pp. 319–328.
3. R. van Renesse et al., "Building Adaptive Systems Using Ensemble,"*Software—Practice and Experience* , vol. 28, no. 9, 1998, pp. 963–979.
4. E. Truyen et al., "Consistency Management in the Presence of Simultaneous Client-Specific Views,"*Proc. 18th Int'l Conf. Software Maintenance* (ICSM 2002), IEEE CS Press, 2002, pp. 501–510.
5. F. Kon et al., "The Case for Reflective Middleware,"*Comm. ACM,* vol. 45, no. 6, 2002, pp. 33 –38.
6. F.J.S. Silva, M. Endler, and F. Kon, "A Framework for Building Adaptive Distributed Applications,"*Proc. 2nd Workshop Adaptive and Reflective Middleware,* ACM Press, 2003, pp. 161–168; http://doi.ieeecomputersociety.org/10.1109/ICDCS.2000.840918.
7. G. Coulson et al., "Reflective Middleware-Based Programmable Networking,"*Proc. 2nd Workshop Adaptive and Reflective Middleware* (RM 2003), ACM Press, 2003, www.cs.wustl.edu/~corsaro/papers/RM2003/p8-coulson.pdf.
8. N. Wang et al., "QoS-Enabled Middleware," in *Middleware for Communications,* John Wiley & Sons, 2003, pp. 131–162.
9. T. Ritter et al., "A QoS Metamodel and Its Realization in a CORBA Component Infrastructure,"*Proc. 36th Ann. Hawaii Int'l Conf. System Sciences* (HICSS 2003), IEEE CS Press, 2003, p. 318a; http://doi.ieeecomputersociety.org/10.1109/HICSS.2003.1174879.