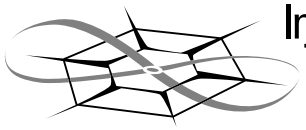
The University of Kansas



Information and Telecommunication Technology Center

Technical Report

Designing a Framework for Dynamic Deployment of Network Services in an Active Network Domain

Vishal Zinjuvadia, Gary Minden, and Joseph Evans

ITTC-FY2003-TR-19740-09

February 2003

Defense Advanced Research Projects Agency and the
United States Air Force Research Laboratory,
contract no. F30602-99-2-0516

Abstract

In the last five years, the networking research community has worked on the Active Networking Technology. The main focus in the effort has been on developing technology that allows the rapid deployment of new functionality, such as data processing or control protocols, by dynamically inserting mobile code segments into the network. Active networks permit applications to inject programs into the nodes or local and, more importantly, wide area networks. This supports faster service innovation by making it easier to deploy new network services. This creates a tremendous opportunity for service providers to offer value-added Managed Network Services (MNS).

But this trend imposes challenges for network management. Current network management techniques offer several limitations for active networks. For the managed services, the main challenge is the complexity: value added services are large distributed programs that have to execute in an unpredictable large runtime environment (the Internet), raising questions related to dynamic deployment, monitoring and end-to-end guarantees. Use of formal methods can be made to analyze the complexity of the management system.

In this thesis, we propose an Active Network Management system with focus on dynamically deploying network services. We provide formal verification of the proposed protocol using SPIN/Promela verification system.

TABLE OF CONTENTS

List Of Figures

List Of Tables

Chapter 1

Introduction

In today's networks, the additions of new network services is restricted by standardization and compatibility concerns. As a result, new services and protocols are deployed at a rate much slower than the emergence of network applications that may benefit from the new services. A number of investigations have been encouraged by this requirement, and over the past few years, a concept called Active Networking has emerged. Active Network proposes the generalization of the processing capability of the network elements by incorporating the programmable packet processing ability into the network elements. Not only the header elements, but also the contents of transiting packets may be processed in transit. Such capability can open up a new set of possibilities for many collaborative and adaptive applications and network algorithms. At present, applications have to depend on the services provided by accepted standards. It has very limited option if it requires anything beyond what is provided by the standards. Active networking technology aims to radically change the situation, where any protocol/service can be loaded into the network elements dynamically even if it is required by a limited and special set of participating entities.

Even though Active Networking technology promises faster service deployment, it brings additional complexity to the network management functionality. The current network management techniques offer several limitations to the Active Networks. The Active Networking technology requires a lightweight architecture that can be rapidly deployed. It must be extensible in nature and provide adequate hooks for new functionality. Such an infrastructure should be highly scalable. The service providers should be able to deploy the infrastructure in an incremental fashion. This ability can prove to be highly useful while upgrading an existing network service to a larger customer-base. Clearly, the management framework offered by current techniques is unable to handle the service deployment needs of active networks. It is essential to redefine the requirements for network management keeping in mind the characteristics of active applications.

In this thesis work, we explore the requirements for an active network management system focusing primarily on the aspect of dynamic deployment of network services. We define the different entities involved with their respective roles and the mutual interactions that take place during the various stages of service deployment. The resulting architecture is formally verified using SPIN/Promela verification system.

The remainder of the introduction is organized as follows. The following section discusses the related research work going on in this area. The next section argues the need for a service management framework for the active network infrastructure in

which new services can be introduced readily. Thereafter, we discuss the overall architecture of the proposed framework.

1.1 Background and Motivation

Networking as a field is characterized by rapid change. New technologies and new applications have emerged quickly for at least a decade, and this trend shows no signs of abating. In turn, new ways of using the network often benefit from new services within the network that enhances functionality or improve performance to better accommodate the new modes of use. The ability to accommodate new services within the infrastructure has gained more importance than ever before. Given this situation, it becomes extremely important to have a sound management infrastructure to back the dynamic nature of the network.

The currently employed management techniques [20] are based on a passive paradigm. The dynamic nature of the management system dictates a rather active approach where the managed entities should provide a snapshot of the current system state to the Service Manager. Moreover, the dynamic changes in the managed services and the network management system will have to be synchronized and coordinated with the dynamic changes of the active network. It should also be possible to adopt newly developed services to reuse existing network management paradigms.

Despite the need for flexibility, the process of changing network services in the Internet is lengthy and difficult. Backward compatibility is necessary to provide

connectivity between new and old service areas since it is not possible to upgrade all portions of the network simultaneously. Similarly, incremental deployment is necessary to allow a new service region to grow until the service is available across the entire network. A number of issues including but not limited to dynamic service deployment mechanism, management of the services deployed and security need to be addressed.

Overlay model for network services is one solution that has come up. The new services are deployed as an overlay, that is, as a layer on top of the old network service instead of in place of it. This strategy is currently used to provide IP multicast service encapsulated within IP to hosts that participate in the MBONE[1]. Overlays used in this manner can be a temporary step towards full deployment. They are useful for experimentation and early provisioning of a new service because, by their very nature, they isolate it from the old network service. They are not a long-term solution because overlays duplicate mechanism and incur performance overheads, often beyond that of packet encapsulation. In case of the MBONE, for example, multiple copies of a message will traverse a link – the very situation multicast seeks to avoid – when the overlay topology does not match the underlying topology. Further, services such as real-time cannot be deployed as an overlay effectively because the underlying IP layer does not provide the required functionality, in this case bandwidth reservation.

[1] Refer to the MBONE Deployment Working Group of the IETF for current details

Additionally, active elements typically need to adapt to and even control the network behaviors. They must thus be able to access data concerning network performance and configuration, as well as effect configuration changes to control network resources. Therefore, in contrast with traditional network applications, which are entirely separated from management software, active applications will need to integrate monitoring and control capabilities. This thesis work does not deal with the service monitoring aspect and is left for future studies.

1.2 Related Work

Research in active networking is gaining popularity. Some of the projects currently contributing to research in this area are: ANTS in MIT [5]; [3] in University of Kansas; SwitchWare in Upenn [33]; [5] in Georgia Tech; NetScript in Columbia [10], and more.

At the University of Pennsylvania and Bellcore, the Switchware project [5] is developing a programmable switch approach that allows digitally signed type-checked modules to be loaded into a network node. Out-of-band program loading is used to support value-added services as specified by the Advanced Intelligent Network (AIN) concept of the telecommunications industry. In Switchware, formal methods are applied to assure the security of the network by identifying the security properties of the underlying infrastructure for which theorems can be proved.

A complementary part of the effort is the development of PLAN [4], a programming language for active networks. PLAN is intended to be compact, so that small

forwarding programs can be carried directly in each packet. These programs can refer to node resident code for privileged operations or common-case processing that is too large to carry directly. PLAN is also designed to facilitate the safe operation of the network.

The Netscript project [10] at Columbia University is focusing on network management. It is designed to support new routing, packet analysis, signaling and management tasks. Netscript consists of a data flow style programming language for scripting agents that process packet stream and a Virtual Network Engine or execution environment within which agents are run. An overall program may consist of many agents that are distributed across nodes. The goal is to enable the programming of remote nodes, including intermediate systems, as easily and quickly as end-systems.

Network management research related to active technologies has only recently been started. Many approaches taken today can be seen as a generalization of the concept of mobile code [10] for building an active management middleware, i.e. a software layer between the management applications and the managed objects. [23] uses mobile agents to provide network management. Most agent based systems [24] [5] confine themselves to application layer thus limiting the number of services that can be provided (for e.g., Packet filtering, Congestion control, etc.). [9] addresses the need to interact with the network layer information but makes use of agent based technology. Another interesting work presented by Hjalmtysson et al [5] suggests a new router design, where installed agents can manipulate data streams in a router.

Many of the modern solutions to the IP network management challenge, use high level distributed file or object environment such as: CORBA, Java ORB, Java RMI, Styx, DCOM, and Directory Enabled Networks (DEN). Often, these solutions hide the cost of communication leading to some deterioration in performance.

Hewlett-Packard, in its effort to give service providers a more complete view of their enterprise networks has rolled out the OpenView VantagePoint [18] software family. It can automatically find a server as it is added to a network, find out what services and applications are running on the server, and set up policies to monitor and manage those services. The tools of this software family also can adjust the amount of information they want to collect from software agents distributed around a network – so if a problem is detected, they can start collecting more data.

The ability to accommodate new services within the infrastructure is of prime importance. This fact is receiving increasing recognition. The Next Generation Internet (NGI) initiative explicitly recognizes the need for infrastructure that is able to accommodate new services as they emerge [27]. In the Internet today, new switching technologies such as Multi-Protocol Label Switching [12] are being explored to simultaneously boost raw forwarding performance and ease the introduction of new routing services. In telecommunications networks, the Intelligent Network architecture is being standardized to speed the development of value-added services. The Open Signaling community[2] [22] is incorporating programmability in the control plane as a means to express new services. This has enabled the relatively rapid

[2] See http://comet.ctr.columbia.edu/opensig/ for more information.

introduction of value-added services, such as 1-800 numbers and call forwarding, by standardizing interfaces to switches and other network equipment. These architectures are specialized to telecommunication tasks, in which there is strong separation between signaling and data transfer tasks, and it is not clear how to best apply them to the concept of an active network based on the Internet architecture.

A part of this thesis work deals with specification and verification of a protocol framework. There has been a substantial amount of research done in the field of protocol specification and verification. The initial approaches could be divided into two categories: implementation-oriented and purely verification-oriented.

The research community has also explored hybrid approach to system verification. Implementation is generally performance-driven and hence imperative or procedural languages such as C or Pascal are usually chosen for implementation. Programs written in these languages, however, are amenable for verification. It is difficult [19] to describe properties of the system using the imperative "assignment-sequence" style of programming. Pure verification languages such as NuPrl [30] and PVS [34], on the other hand, are not used for implementation purposes because they are slow and hinder performance.

A hybrid model that combines specifications and implementation characteristics is the Ensemble model. In this model, the protocol framework is implemented in a high-level language (OCaml). Programs written in OCaml are amenable for verification and yet are able to provide reasonable performance characteristics [36]. The OCaml representation is then converted into the language of a theorem prover (NuPrl) using

an automatic conversion process. NuPrl then reasons about the protocol framework. Specifically, Ensemble optimizes the framework and then verifies the optimizations that were carried out in NuPrl. The optimized framework is then converted back into the implementation language OCaml.

1.3 Service Management Framework at a glance

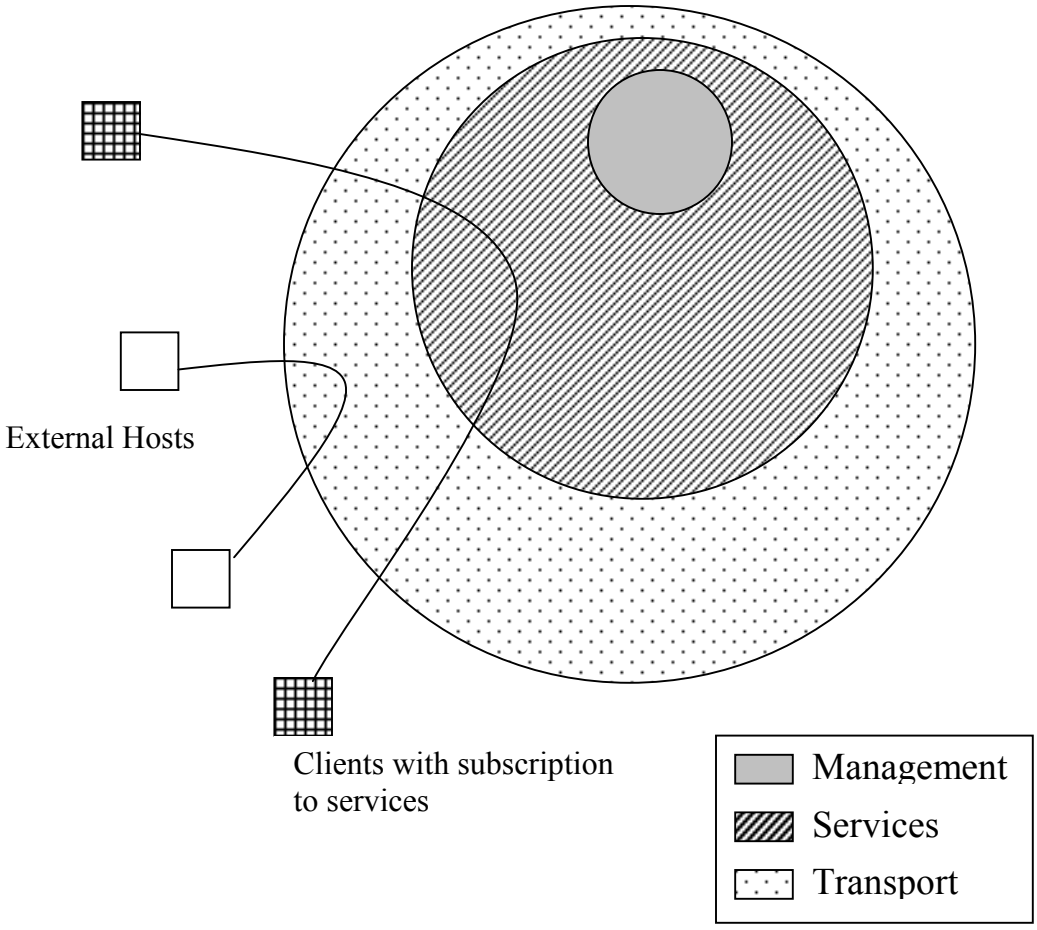


Figure 1.1 Different layers in the Service Management Protocol Framework

The Management subset of the above diagram represents the core management functionality of the Service Management Framework. It comprises of entities that completely trust each other. It is a highly secure environment, and no external entity is allowed to access it. Next is the Services layer. It comprises of entities that provide interfaces to the authorized clients to make requests for service installation/uninstallation as well as service updates. The outermost layer is the Transport layer that carries traffic that just needs forwarding services from the routers.

1.4 Potential New Services

The following examples are intended to show the kinds of new services that can be readily introduced with the proposed service management framework, but are difficult to deploy dynamically in today's Internet.

1.4.1 Multicast

Deploying reliable multicast in a network requires setting up the Multicast server, and managing the multicast group. With a framework for dynamic deployment of services, a multicast server can be dynamically set up at a network node by sending a request to the Service Manager. Appropriate parameters passed on to the Multicast Server enable it to manage the different groups willing to access the multicast service.

1.4.2 QoS Route Establishment

End-to-End QoS provisioning is currently achieved using manually configuring the links from the source to the destination node. Instead, using the programmable packet processing, smart packets can be sent along the required path and each intermediate host configured for the required traffic characteristics. The client provides the required traffic parameters to the Service Manager to dynamically setup the QoS route.

1.4.3 Multipath routing and forwarding

Based on the traffic patterns, multipath routing capabilities can be dynamically installed on a router to handle the traffic bursts and divert it onto disjoint paths to improve available bandwidth or reliability. The dynamic nature of the proposed service management framework complements the unpredictability in the traffic patterns.

Depending on the requirements, several other services are expected to come up giving rise to numerous network applications.

1.5 Thesis Organization

The remainder of the thesis describes the proposed service management framework with focus on dynamically deploying network services. It includes a SPIN prototype of the system that evaluates the validity of the protocol and its effectiveness as a management system for active networks.

Chapter 2 provides an overview of the service management framework. It discusses the protocol state machine for the individual entities and describes the interaction between the entities. Chapter 3 discusses proposed protocol in further detail. It describes the types of messages, message parameters and encoding of messages. It also discusses the various error conditions and recovery mechanism employed by the protocol. Chapter 4 is devoted to the specification and verification of the service management framework using SPIN and Promela. The final chapter summarizes the contribution of this thesis and possible future extension for the framework.

The Appendix includes the Promela source of the system specification.

Chapter 2

Overview of the Service Management framework

This chapter describes the proposed active network management system in greater detail. It begins with the design principles of the proposed architecture. The functional description of the service management framework follows it. Then we talk about the different entities involved and their responsibilities in the functioning of the system.

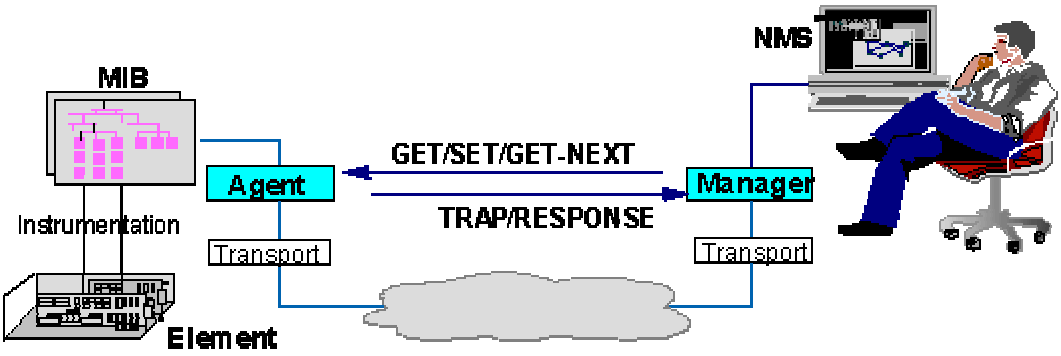


Figure 2.1 The Architecture of a Network Management System

In recent years, new management paradigm proposals tried to overcome some of the key deficiencies of the SNMP model. The Management by Delegation (MbD) [37]

paradigm proposes a distributed hierarchy of managers. At the lower layers of the hierarchy, managers closer to the managed entities will be responsible for monitoring and controlling their operations. Managers in higher levels of the hierarchy oversee several managers to distribute the management duties. MbD is a very scalable proposition when compared to the model in Figure 2.1. The proposed service management framework for dynamic deployment of active network services derives in part from the MbD architecture.

Following are the design principles that would guide the design targeting specifically the network management domain with focus on service deployment.

2.1 Design Principles

2.1.1 Generality and Simplicity

The node should be general enough to support the different levels of active networking – from capsules to programmable switches. The architecture should not be dependent on the type of service to be deployed.

2.1.2 Modularity

The components responsible for different functionality should be separated and should have clearly defined API between them. Adding a service or removing one should not involve any change to the management infrastructure.

2.1.3 Safety and Security

At each step, security mechanisms including Source authentication, Integrity check and Policy verification, or any subset thereof should be employed.

2.1.4 Extensibility

Active Networking is an area that is being explored and is in its nascent stages. As research continues, new possibilities may be discovered leading to more sophisticated applications of active networking. The model developed must be extensible in a manner so as to take advantage of the new discoveries.

The remaining part of the section describes the different entities as well as a typical interaction between them.

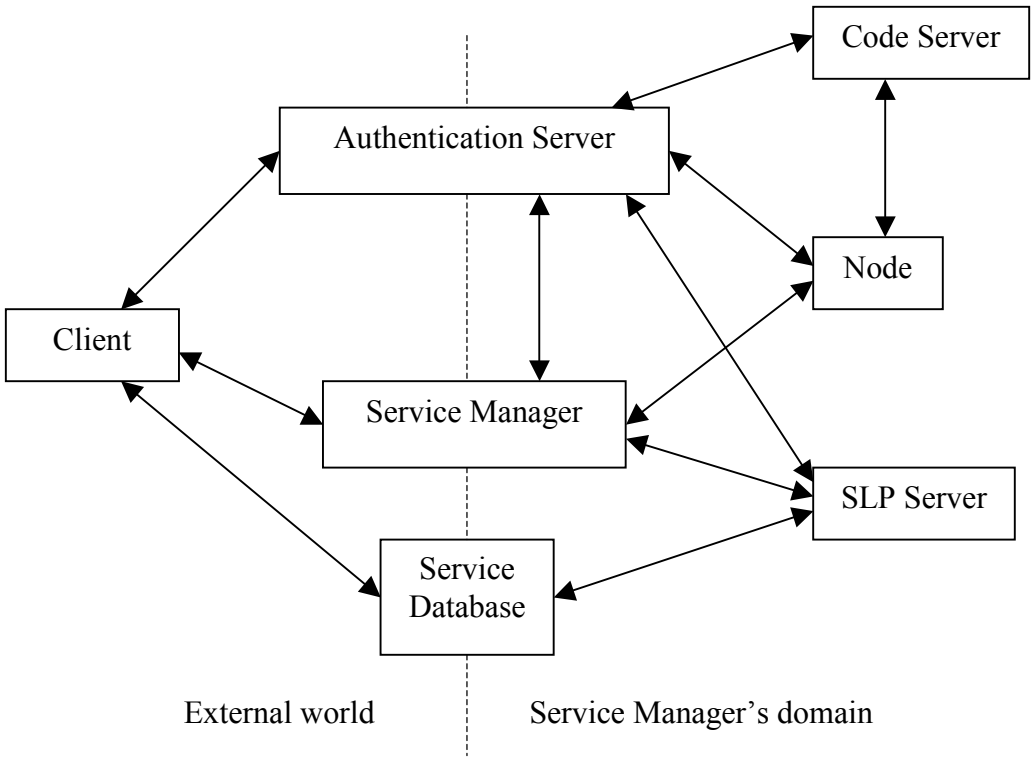


Figure 2.2 Components of the Service Management Framework

Figure 2.2 shows the entities involved in the proposed service management framework for active networks. Applications residing on the client use the Service Manager's network for certain services. They request the installation of the service if it is not already present. Adequate security mechanisms are employed at the Service Manager's end to authenticate the client requests. Additional entities such as the Authentication Server, Code Server and the Service Locator help the Service Manager in processing the client's request. Each of the above entities is connected to its neighbors by link layer channels. Compatibility with existing routers supports

incremental deployment of the architecture itself, a necessary condition to bootstrap an active network management infrastructure in the Internet.

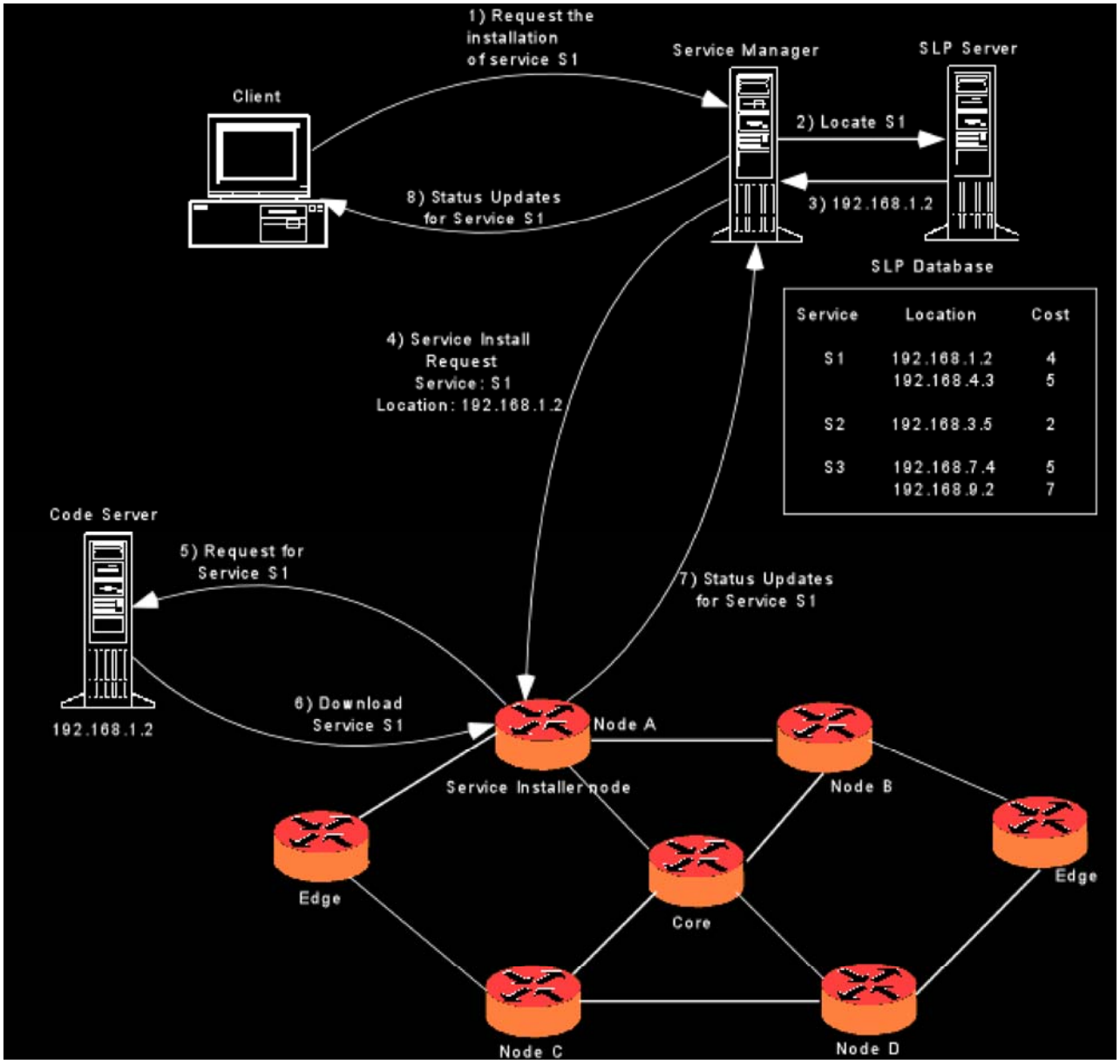


Figure 2.3 Typical interaction between different entities in the Service Manager's Network. The Client has requested a Service S1 to be installed

2.2 Typical Interaction between the different entities

The following is a step-by-step description of the sequence of events that occur when a client needs to install a service in the Service Manager's network.

The client requests the Service Manager to install a particular service at one of the member nodes in the network. The Service Manager receives the request and authenticates it with the help of Authentication Server. The Service Manager checks to see if it knows the Code Server where the requested service can be found. If not, it sends out a Service Location Request to the Service Locator requesting the location of the service in question.

Thereafter, the Service Manager locates the Service Installer node where the service needs to be installed and sends the Service Install Request to it with the Service Identifier as well as the address of the Code server. The Service Installer node authenticates the Service Manager's request and requests the Code Server for the modules related to the service to be installed. The service is downloaded from the Code Server. The Service Installer node carries out appropriate authentication checks to verify the modules downloaded from the Code Server.

After downloading and verifying the service modules, the Service Installer node installs the service. A particular preconfigured amount of resources are allocated to every service. The node constantly gathers statistics about the resources consumed by the service. In case the resources consumed exceeds the allocated maximum amount of resources, the service is uninstalled and a notification message with appropriate status code is sent to the Service Manager.

The Service Installer nodes provide the Service Manager with a snapshot of the network at regular intervals. So at any instant, the Service Manager can have an overall picture of the network. In case the client needs the status updates about the service it had requested, it sends a Service Update Request message to the Service Manager. The Service Manager provides the client with the required information in a Service Update Response message.

Since the Service Manager maintains an up-to-date picture of the network statistics, it makes decisions based on the overall network behavior. Decisions like choice of the Service Installer node, Code Server or Authentication server, connection admission control are based on the current network information.

There are several situations that cause the installed services to be uninstalled. Many of them are covered in the chapter on Error Conditions and Recovery mechanisms. In case the client does not want the service any more, it sends a Service Uninstall message to the Service Manager. After the necessary authentication procedures, the Service Manager directs the Service Installer node to uninstall the service and release any resources allocated to it.

The Service Manager then sends the Service Uninstall Response message back to the client and terminates the session.

A basic set of services (for e.g., “ping” to verify link connectivity with neighboring member-nodes) is always installed on each Service Installer node as it joins the Service Manger’s network. These services provide the Service Manager with the statistics at each node and notify it in case any failure is detected.

2.3 Protocol State Machine

The state transition based models [13] represent a network protocol in terms of a finite state machine. The finite state machine is the simplest and the most general tool for specifying the behavior of a protocol. All the possible input parameters are paired with the current state to give an output state for the system.

Going by the design principles, the overall functionality of the service management framework with respect to service deployment has been divided into several independent sub-tasks and different entities have been specified for each of those sub-tasks.

In this section we shall describe the state machine for the individual entities involved in the proposed protocol framework. Two assumptions are made while specifying the protocol. The nodes running the different entities are assumed to be connected at all times and the transport protocol running is assumed to be providing reliable communication between the nodes.

The states are shown in closed oval shapes while the numbered arrows connecting these oval shapes denote the action that causes the transition from the initial state to the final state. For each of the entities, the various states and the actions causing the state transition are described in the following sub-sections.

2.3.1 Service Manager

The Service Manager is the central controlling entity that manages the process of deploying the services at the client's request. It coordinates the actions of the other

entities such as the Authentication Server, Service Installer node, Service Locator and the Code Server.

The Service Manager listens for client's requests to install the services at particular nodes in its network. The Service Manager is responsible for authenticating the message, processing it and directing the other entities during the process of executing the client's request. The authentication procedure must include source authentication, message integrity check as well as a policy verification with the Service Level Agreement database to make sure that the client is allowed to install the requested service at that instant of time.

Additionally, the Service Manager also collects network statistics from the member nodes and maintains a snapshot of the network at any instant of time. It also obtains service-related information from the nodes where the services have been installed and provides the clients with updates about the service status when requested by the client. Following is the finite state machine for the Service Manager.

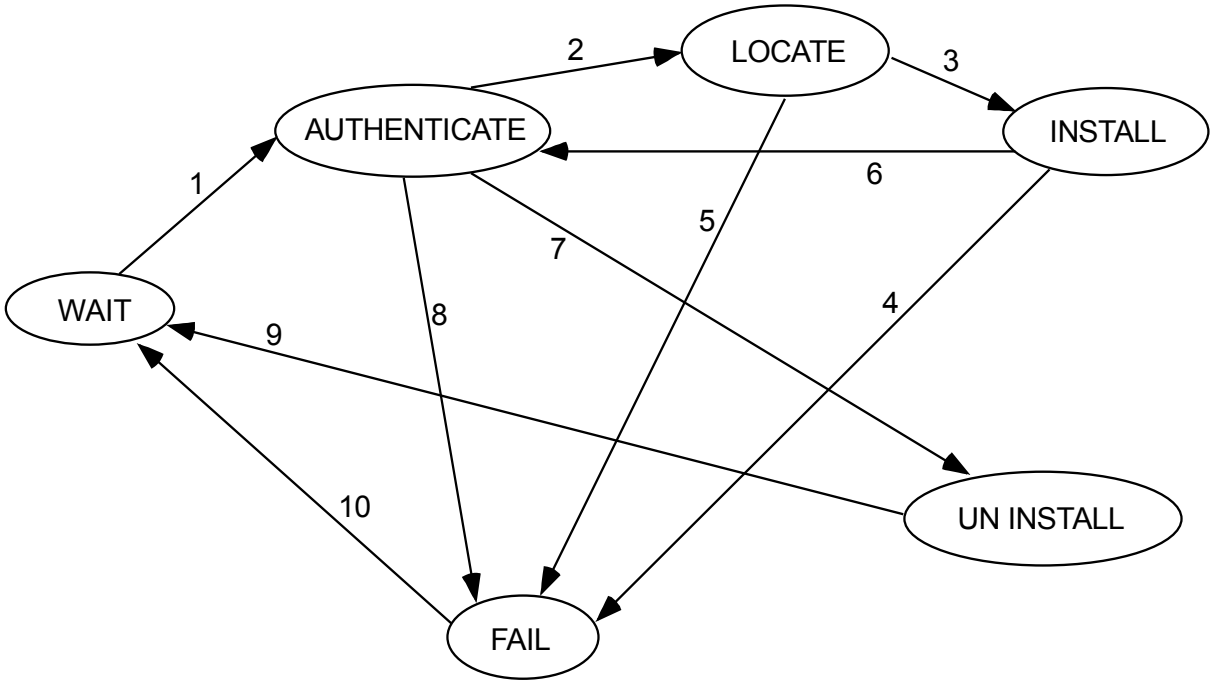


Figure 2.4 Finite State Machine for the Service Manager

States

WAIT The Service Manager is waiting for a request from the client.

AUTHENTICATE The Service Manager sends the Service Install/Uninstall request message from the client to Authentication Server for authentication.

LOCATE The Service Manager attempts to locate the requested service by querying the Service Locator.

INSTALL The Service Manager directs the Service Installer node to install the requested service by providing it with the required information.

FAIL Failure occurred at one of the three stages – Authenticate, Locate or Install.

UNINSTALL The Service Manager directs the Service Installer node to uninstall the service.

Events

{1} The Service Manager receives a Service Install Request message.

{2} The Authentication Server successfully authenticates the Service Install Request message.

{3} Code Server(s) for the requested service has (have) been successfully located by the Service Locator.

{4} The service is preempted because of a fatal error or competition for resources with a service with higher priority.

{5} The Service Locator could not locate a Code Server for the requested service.

{6} The Service Manager receives a Service Uninstall Request message.

{7} The Authentication Server successfully authenticates the Service Uninstall Request message.

{8} The authentication of the Service Install/Uninstall Request message failed.

{9} The session with the client is terminated and the associated resources are released.

{10} The session with the client is terminated and the associated resources are

released.

2.3.2 Client

The client is an entity external to the Service Manager's network. The user or the client who wants to install a service in the Service Manager's network should have a Service Level Agreement registered with the Service Manager. Based on several parameters including the current status of the agreement and the amount of resources needed, the Service Manager makes a decision about whether to install the service.

The client forms the service-install request packet and sends it to the service manager. If the response from Authentication Server is positive, the client assumes that the service manager has installed the service. The client requests for status updates from the service manager while the service is installed. The client sends a service-uninstall request packet after it is done using the service.

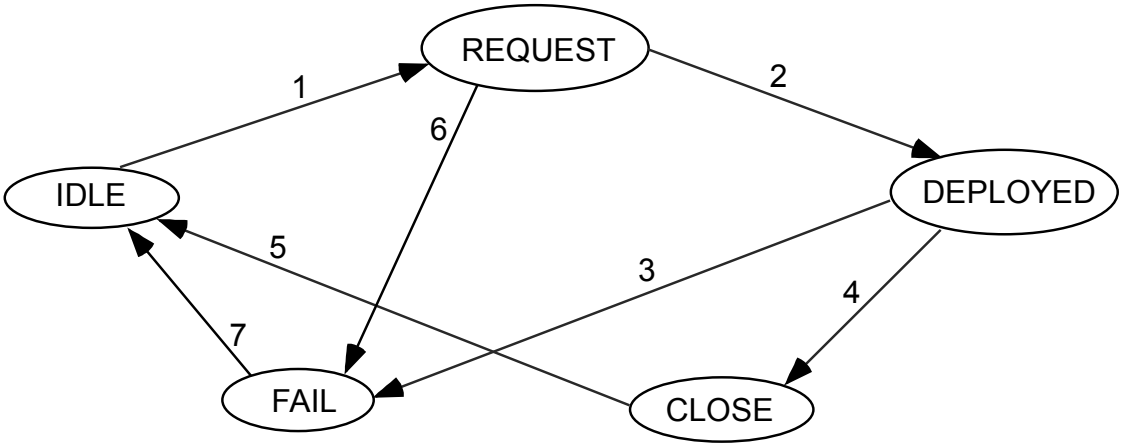


Figure 2.5 Finite State Machine for the Client

States

IDLE Client does not need any services installed in the Service Manager's domain.

REQUEST Client needs a service to be installed in the Service Manager's domain and sends a Service Install Request. It is waiting for the Service Manager to acknowledge the request.

DEPLOYED The Service Manager sends a Service Install Response with status set to SUCCESS. Service is installed and the client can use it. In this state, the client queries the Service Manager for status updates on the service at regular intervals.

CLOSE The client does not require the installed service any more. So it request the Service Manager to uninstall it.

FAIL The Service Manager sends a Service Install Response with status set to FAILURE. Due to some failure, the service is not installed.

Events

{1} Clients need a service to be installed in Service Manager's network.

{2} Client receives a Service Install Response with status set to SUCCESS.

{3} The installed service is preempted because of a fatal error or competition for resources with services having higher priority.

{4} Client sends a Service Uninstall Request to the Service Manager.

{5} Session with the Service Manager is terminated & resources released.

{6} Client receives a Service Install Response with status set to FAILURE.

{7} The session with the Service Manager is terminated & the associated resources are released.

2.3.3 Authentication Server

The Authentication Server is responsible for verification of messages when requested by a particular entity. It is assumed that the Authentication Server is a trusted entity[3]. The Authentication Server maintains a complete database of the security information for all the member nodes. Issues related to security of the database are not dealt with as a part of this thesis. The Authentication Server listens for requests from the other entities for authentication of messages that they have received. When a particular entity receives a message from its peer, in order to verify its authenticity, it generates an authentication message and sends it to the Authentication Server. The Authentication Server authenticates the message (exact semantics depend on the security mechanism used) and sends back either a SUCCESS or FAILURE message depending on the outcome of the authentication test.

Since the Authentication Server is expected to process a large number of requests (one per each message), its stability is of tremendous importance. In a way the speed at which the Authentication Server handles the requests decides the overall

performance of the system. So, we propose to have multiple Authentication Servers. A single point of failure is also avoided by providing multiple Authentication Servers.

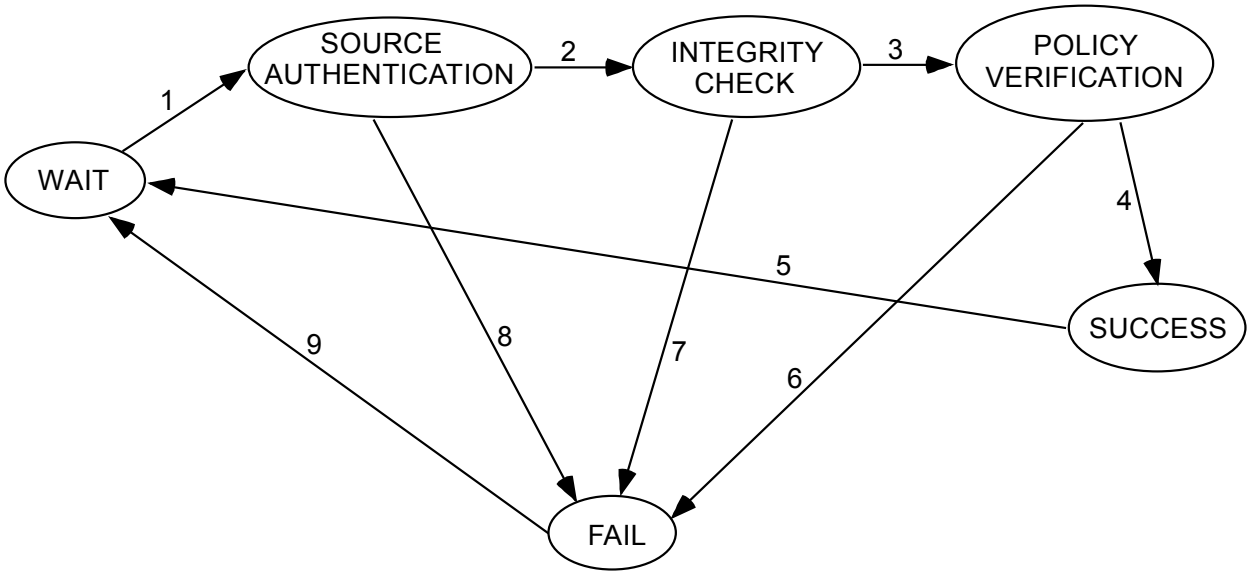


Figure 2.6 Finite State Machine for the Authentication Server

States

WAIT The Authentication Server listens for requests from the other entities for authentication.

SOURCE AUTHENTICATION Authentication is performed in 3 stages. Source Authentication is carried out in this state. This is to verify that the actual source of the received message is the same as that mentioned in the message.

INTEGRITY The integrity of the message is checked in this state. This is

[3] A more general model would be that of a hierarchy of Certificate Authorities. This extension is left for future studies.

CHECK	to make sure that the message was not tampered with or inadvertently modified while in transit.
POLICY VERIFICATION	Depending on the previous records, the Service Manager installs filters in order to block certain requests from misbehaving clients. In this state, the Authentication Server accesses the policy database to check whether there are any filters installed for the source entity in the message.
SUCCESS	The Authentication Server responds to the Authentication Request message with the status set to SUCCESS.
FAIL	One of the three stages in the authentication process failed. The Authentication Server responds to the Authentication Request message with the status set to FAILURE.

Events

{1} The Authentication Server receives a request for authenticating a message.

{2} The Authentication Server could successfully authenticate the source of the message.

{3} The message was not tampered with or inadvertently modified while in transit.

{4} The message obeys the policy regulations set by the Service Manager.

{5} A response is sent to the requesting peer with the status set to SUCCESS.

{6} The message violates the policy regulations set by the Service Manager.

{7} The message has been tampered while in transit.

{8} The Authentication Server could not verify the source of the message.

{9} A response is sent to the requesting peer with the status set to FAILURE.

2.3.4 Service Locator

When the Service Manager processes a service install request from the client, it is essential for it to know whether the requested service is available in its domain. Moreover there is a need for a mechanism by which new services can be added and existing services removed or modified in the Service Manager's domains. [11] suggests a simple protocol that takes care of all these details.

The Service Locator needs to maintain a database consisting of all the services and the nodes that advertised those services. It also maintains a cost factor with each service, which is referred in case the same service is offered by more than one node in the Service Manager's network. The Service Manager queries the Service Locator about a particular service. If the Service Locator locates the entry for the service in its database, it returns a SUCCESS message with the information about the node offering the service and the associated cost. Otherwise a FAILURE message is returned.

The Service Locator must provide the member nodes with necessary APIs to add/delete/modify a particular service entry.

Following is the finite state machine for the Service Locator followed by a brief explanation about the various states and the actions that causes state transition.

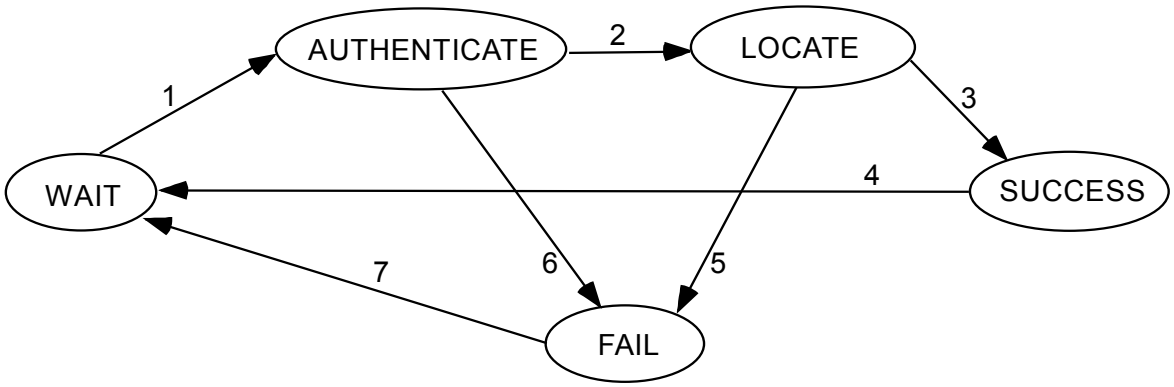


Figure 2.7 Finite State Machine for the Service Locator

States

WAIT The Service Locator is waiting for Service Location requests from Service Manager.

AUTHENTICATE The Service Location request is sent to the Authentication Server for authentication.

LOCATE The Service Locator accesses the Service Location database and attempts to locate the Code Server(s) where the requested service can be found.

SUCCESS The Service Locator successfully located one or more entries for the requested service in the Service Location database.

FAIL Either the Authenticate or the Locate state failed.

Events

{1} The Service Locator receives a Service Location request from the Service Manager.

{2} The Authentication Server successfully authenticates the Service Location request.

{3} The Service Locator successfully locates Code Server(s) for the requested service.

{4} A Service Location response with status set to SUCCESS is sent to the Service Manager.

{5} The Service Locator could not locate a Code Server for the requested service.

{6} The Authentication Server failed to authenticate the Service Location request message.

{7} A Service Location response with status set to FAILURE is sent to the Service Manager.

2.3.5 Code Server

A service consists of one or more modules that function together. For example, Reliable Multicast service requires the protocol elements that provide the reliable communication and the protocol elements that provide multicast capability. The node where the service is to be installed might not have the required modules. In this case,

it needs to obtain assistance from the Service Manager's network. In the service install request, along with the Service ID, the Service Manager's also provides the Service Installer node with the information about the Code Server where the service modules can be found. A Code Server is an entity residing on a network node that maintains a repository of the service modules. Along with the service modules, the repository also contains policy information for all the member nodes. This is primarily to filter requests coming from nodes not a member of the Service Manager's domain. It is also useful to provide the Service Manager with a fine control over the access to the service modules. Before the service can be installed at a node in the Service Manager's network, the node contacts the Code Server to obtain the required modules for the service. The Code Server accesses the repository of service modules, and provides the node with the requested modules.

The Code Server needs to support the remote code-loading process on the active network platform. As with any other entity, the Code Server needs to authenticate the node's request before providing any of the requested modules. The requesting node should be provided with the modules as well as some authentication value to ensure the integrity of the modules. For e.g., the hash algorithm could be run over the service module and the residual value be encrypted and sent along with the modules.

The Code Server may prove to be a single point of failure for the Service Manager's network. So, as in the case of Authentication Server, the network must have multiple Code Servers. [3] suggests failover mechanisms to provide continual service in case of failure of the primary Code Server.

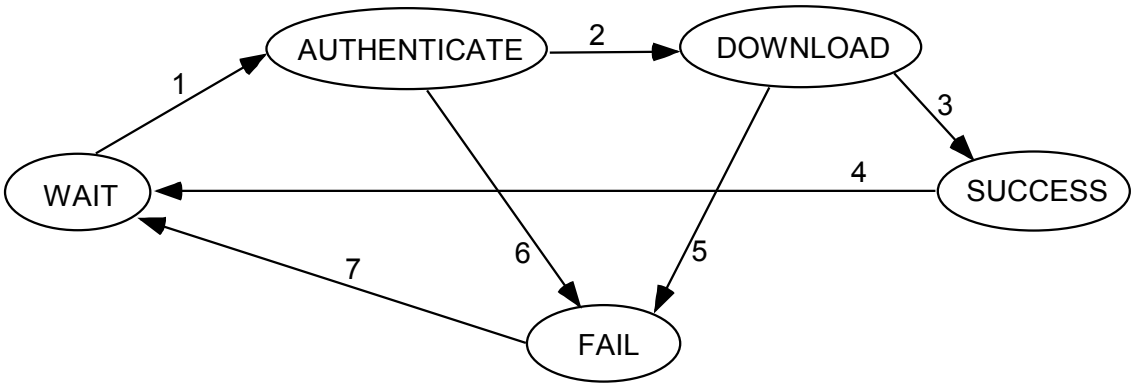


Figure 2.8 Finite State Machine for the Code Server

States

WAIT The Code Server is waiting for requests for service modules from the Service Installer nodes.

AUTHENTICATE The Code request from the Service Installer request is sent to Authentication server for authentication.

DOWNLOAD The Service Installer node attempts to download the required service modules from the Code Server.

SUCCESS The Code Server responds to the Service Installer node with status set to SUCCESS.

FAIL Either the Authenticate or the Download state failed.

Events

{1} The Code Server receives a request for service modules from the Service

Installer node.

{2} The Authentication Server successfully authenticates the request for service modules.

{3} The Service Installer node successfully downloads the required service modules from the Code Server.

{4} A response message with status set to SUCCESS is sent to the Service Installer node.

{5} A fatal failure occurred during the download of the requested service.

{6} The Authentication Server failed to authenticate the Code request message.

{7} A response message with status set to FAILURE is sent to the Service Installer node.

2.3.6 Service Installer node

The Service Installer node install the services requested by the Service Manager. They do not communicate directly with the client. Service Manager acts as a mediator between the client and the Service Installer node. The Service Manager also requests the Service Installer node to provide a snapshot about the network activity at regular intervals.

Following is the finite state machine for the Service Installer node followed by a brief description about the states and the actions that cause state-transition.

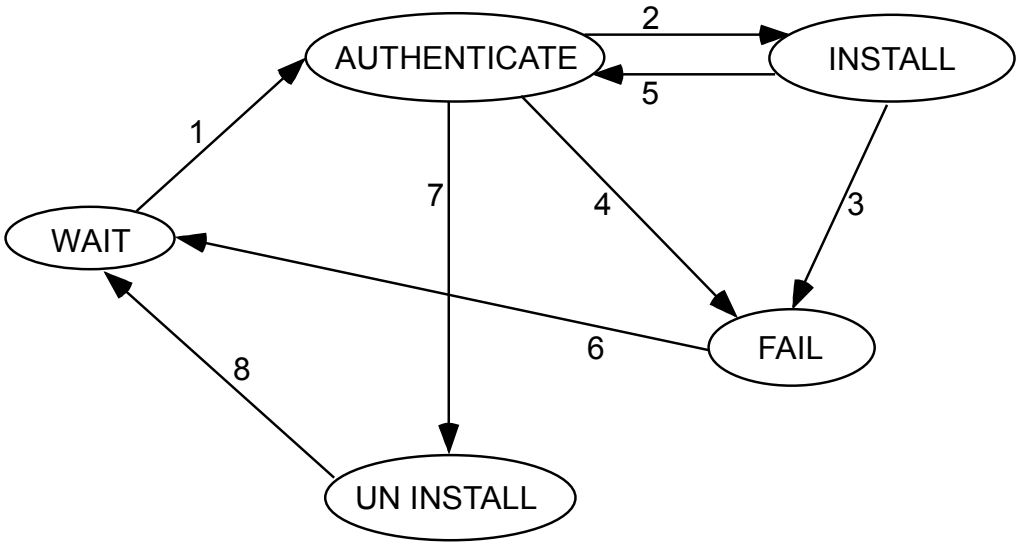


Figure 2.9 Finite State Machine for Service Installer Node

States

WAIT The Service Installer node is waiting for a request from the Service Manager.

AUTHENTICATE The Service Installer node sends the Service Install/Uninstall request message from the Service Manager to the Authentication Server for authentication.

INSTALL The Service Installer node installs the requested service

PREEMPT The service to be preempted because of other competing service with higher priority or a fatal error.

UNINSTALL The service is uninstalled on receipt of a Service Uninstall message from the Service Manager

FAIL Either the Authenticate or the Install state failed.

Events

{1} The Service Installer node receives a Service Install request from the Service Manager.

{2} The Authentication Server successfully authenticated the Service Install request message.

{3} The installed service is preempted because of a fatal error or competition for resources with a service with higher priority.

{4} The Authentication Server failed to authenticate the Service Install/Uninstall request message.

{5} The Service Installer node receives a Service Uninstall request from the Service Manager.

{6} The Service Installer node notifies the Service Manager of the preemption of the installed service and terminates the session

{7} The Authentication Server successfully authenticated the Service Uninstall request message.

{8} The Service Installer node terminates the session with the Service Manager after the service is uninstalled.

Chapter 3

Internals of the Protocol Framework

The initial chapters dealt with the requirements for a management protocol to develop a framework for dynamically deployable network services and the associated entities that take part in the overall process. We discussed the individual responsibilities of the various entities and how they interact with each other in order to provide the required services to the clients. It becomes necessary to go into further details about the interactions between the different entities. This chapter deals with the details of the protocol framework. The initial sub-sections focus on the message parameters generally used for message transfer between two peer entities. We then discuss the error conditions that could occur and the recovery actions that the framework should take to maintain a stable state.

3.1 Message Parameters

We shall consider a client requesting the installation of a service at a particular node in the Service Manager's network.

Different types of messages are exchanged between the entities at different stages of service deployment. Though we shall not provide the exact semantics of the messages involved, the discussion shall certainly include the most prominent aspects that need to be taken care of. Before discussing the message types and their contents, let us discuss the essential parameters that would accompany a message and the functions they serve.

Following are the fields that carry useful information as a part of the messages exchanged among the different entities in the proposed service management framework.

3.1.1 Entity ID

Each entity in the Service Provider's network identifies itself from the others using the Entity ID. Several policy decisions are taken based on the Entity ID.

3.1.2 Message type

Message type field carries information about the type of the message. Depending on the type of message, appropriate packet processing function is called to handle the packet.

3.1.3 Service ID

Service ID field is used to indicate the service to which the current message refers. A client can have multiple services installed. Service ID is used to differentiate between the services when client and the server exchange control messages.

3.1.4 Authentication field

Currently undefined, this field should carry information that could be used to authenticate the sender. The receiver extracts the authentication field and sends it to the Authentication Server along with the sender's identity. The Authentication Server decides based on certain checks whether the message is valid and returns its response to the client.

3.1.5 Result (Success/Failure)

This field contains the result of the request that was received from the peer. It indicates whether the request could be satisfied or not. Additional TLVs could be used to give a more detailed description about the result. For e.g., failure occurred because of the lack of resources, etc.

3.1.6 Service Update Parameter

After the service is installed, client can request the Service Manager to provide it with the information about service status or about certain service parameters. For e.g., some parameter that reflects the Quality of Service that is provided by the Service

Manager's network. This parameter field is included in the Service Update Response from the Service Manager to the client.

3.1.7 Session ID

There could be multiple service-request sessions active at the same time. Session ID field is used by the entities involved to differentiate between them.

3.1.8 Sequence numbers

To provide protection against Replay-Attacks, Sequence numbers need to be present in the packets. The preferred form of Sequence numbers would be linear (no wrapping takes place). If the number of messages is likely to exceed the maximum number that could be carried by the field, a lollipop mechanism of wrap-around should be used (start with 'p', suppose wrap-around takes place at 'r', then start from 'q' where 'r' > 'q' > 'p').

3.1.9 Service Setup Priority

Each Service is associated with a Setup Priority. When the service installation is requested, if the resources available are insufficient for the service installation, the Setup Priority of the new service and the Holding Priority of the currently active service are compared. If the Setup Priority is higher, the existing service is pre-empted to free the required resources.

3.1.10 Service Holding Priority

Each Service is associated with a Holding Priority. When the service installation is requested, if the resources available are insufficient for the service installation, the Setup Priority of the new service and the Holding Priority of the currently active service are compared. If the Setup Priority is higher, the existing service is pre-empted to free the required resources.

It is advisable to have the Setup Priority of a service less than the Holding Priority to avoid the situation where a newly installed service is pre-empted by a new service having the same Setup Priority as the installed service.

3.1.11 Service Location

This field conveys the address of the node (e.g. code-server) from where the required service can be downloaded. The address could be an IPv4 or an IPv6 address. So it should be encoded as a TLV, where the type and the length field will indicate the type of the address.

3.1.12 Traffic Parameters

For services like QoS provisioning, along with the Service Install Request packet the client also sends traffic parameters that reflect the type of service level expected by the client. The traffic parameters include the attributes such as the Committed Data Rate, Committed Burst Size, Conditioning Action and Service Frequency. They are described in brief below.

3.1.12.1 Committed Data Rate

Committed Data Rate indicates the rate in bytes per second at which the client requests its data to be allowed to traverse the Service Monitor's network according to the Service Level Agreement.

3.1.12.2 Committed Burst Size

Depending on the traffic type, occasional bursts in client's traffic are allowed by the Service Manager's network. Committed Burst Size is the value of the burst traffic in bytes per second that is specified by the Service Level Agreement.

3.1.12.3 Service Frequency

The value of service frequency field is proportional to the number of times the client would request the service updates & service parameters.

3.1.13 Failure Information

While processing peer's message, if an error is encountered, a notification message is sent to the peer with an appropriate status code. In order to assist the peer in taking appropriate corrective action, it is necessary that some additional information be provided along with the notification message. For e.g., information about the excess amount of resources (CPU time, memory, etc.) consumed by Service or the stage where authentication process fails (Source Authentication, Message Integrity or Policy Verification stage) can prove to be useful to the peer.

3.2 Special Attributes

Apart from the fields mentioned above, there is a need to convey additional information that is useful primarily for administrative purposes. Following are the situations that require special handling.

- A member node of the network boots for the first time or reboots.
- When a particular entity in the Service Manager's network is heavily loaded with requests and cannot process any more requests, it needs to inform the peer to hold on to the further requests for a fixed duration of time. This is the case of Application level congestion.
- If a peer wants that a particular request be given immediate attention, it needs to inform the peer in order to avoid queuing delays at the receiver end.
- When the value of certain parameters received in a message from the peer are not acceptable, the receiving node needs a mechanism to negotiate the value of those parameters. For e.g., the Traffic parameters in the Service Install Request message.
- When the client needs to modify certain parameters of a previously installed service, it needs to convey the information to the Service Manager.

3.3 Common Header

Among the fields mentioned above, the Message type, Message Length, Authentication field, Session ID and Sequence Number fields should accompany all

the messages exchanged. They help the receiver in authenticating the sender and identifying the session to which the message belongs. So these fields should be a part of the Common Header that accompanies all the messages.

3.4 Message Encoding

The above fields should be encoded as TLVs instead of static encoding (as in TCP or IP headers). There are several reasons why TLV encoding is preferred compared to static encoding of the packet fields. TLV encoding is suitable for deployment of Composible Services. The order of TLVs is not important and new TLVs can be added or existing ones removed without much effort. But TLV encoding increases processing delay. On the other hand, though static encoding is efficient in terms of processing time required, it is not flexible enough and is recommended for protocols that are not expected to change drastically over time. Addition or removal of fields requires change in the processing functions since the packet fields are expected in a fixed sequence by the processing functions. Also byte alignment is a concern in static encoding. Extra bits need to be padded to achieve the required alignment.

Following is a detailed description about the error conditions that the protocol framework could encounter and the recovery actions that must be taken depending on the severity of the error.

3.5 Error Conditions & Recovery Mechanisms

The introduction chapter described the basic failure conditions that can occur during the various stages of the service installation procedure. We also touched upon some basic recovery mechanisms that can be used in case of failures.

While processing the message from a peer, if a failure occurs, a notification message is sent to the peer. It may prove to be useful if the peer is informed of the exact cause of the failure. One way to do it is include a parameter called “Status Code” in the notification message. This field carries information that describes the cause of failure to the peer.

The remaining portion of the section describes the different status codes in detail along with the corrective actions that are needed to restore the system state.

3.5.1 Service Unavailable

If the service requested by the client is not present in the database of the Service Locator, the Service Location request fails. This message causes a notification message being sent from the Service Manager to the client with the status code set to “Service Unavailable”. Such a message could also be sent if the Service Locator timeouts. One possible reason why the Service Locator timeouts while responding to the Service Location request is excess load on Service Locator that causes new Service Location requests to be dropped. In this case, the notification message sent by

the Service Manager should advise the client to try requesting the installation of the service at a later stage when the Service Locator is ready to accept more requests.

3.5.2 Resources Unavailable

The service that is to be installed needs a certain amount of resources. At the time of request, if the node is not in a position to allocate the requested resources to the service the Service Install Request sent by the Service Manager fails. So a notification message is sent to the client with the status code field set to "Resources Unavailable". Resources could be the active node, CPU time, memory, etc., anything that is needed to handle the client's requests and install the service. There could be several reasons that lead to the denial of the request. The node could be down or rebooting. So it cannot handle any requests. The node might be experiencing excessive load causing a lack of memory or CPU time for any new services. In either case, the client must be advised to make the request at a later stage in the notification message that is sent.

Let us consider the case where the Service Installer node supports the concept of per-service setup and holding priorities. When the currently available resources are not sufficient to handle the client's request, the node where the service needs to be installed chooses a service from those currently active such that the requested service has a higher Setup priority than the Holding priority of the chosen service. If such a service exists, its resources are released and the corresponding session is terminated to make way for the new request. A Service Install Response message is sent to the client corresponding to the new request. A notification message with appropriate

status code is sent to the client whose service was pre-empted to free resources for the new service.

There is a separate class of errors that occur due to corruption of the bits while the message is in transit.

3.5.3 Errors due to corrupted fields

Due to corruption in the fields carried by the packet, the node cannot process the packet correctly and registers an error. For example, error in the message type TLV causes the packet to be discarded because the receiving node cannot identify the message. A variation of this error condition would be when the receiving node can identify the message type, but is not expecting to receive it. Such an error can occur if the underlying transport protocol does not provide reliable communication and one of the messages gets delayed or lost.

Similarly, corruption of the session identifier field of packet causes an error because the message cannot be demultiplexed to a session handler process.

Corruption of the message length TLV is discovered when the node finds that the length of the received message is not in agreement with the length that is mentioned in the TLV.

The failure caused by an invalid Sequence number TLV is of particular interest because of the serious possibilities. Let us discuss it in more detail.

As in the case of TCP, the two entities must agree upon an initial sequence number when a new session is initiated between them. The subsequent messages exchanged

between the two entities will include the Sequence number TLV incrementing it by one every time. If a node detects that the sequence number carried in the message by the peer is not in agreement with the expected value, a failure is declared. There can be several reasons causing it. Some of them are as follows.

- Packets are received out of order (probably because of an unreliable transport protocol) causing some packets to be delayed or dropped due to congestion.
- An intruder is attempting a replay attack trying to send some previously captured packets.

It is difficult to figure out the exact reason for such a failure condition. For all the failures caused by corrupted fields, the receiving node should send notification message to the peer with appropriate status code. The session with the peer that sent the erroneous message should be terminated releasing any resources allocated to it. If the peer is a client, any services installed on its behalf should be uninstalled and appropriate notifications should be sent to the client.

3.5.4 Timeout errors

Another class of errors occurs when an entity is waiting for a message from a peer and the hold timer expires before the message is received.

Timeout can occur whenever any node is waiting for a particular message from another node and a configured amount of time passes before the message arrives from the other node. The amount of time for which the node waits for the message from the peer is known as the timeout interval.

Timeout can be caused due to many reasons. One of them is packet level failure because of which even though the packet reaches the destination node, it does not make its way to the application that is listening for it and hence it appears as a timeout to the application. Timeout may also occur because of heavy traffic that causes the packets to be dropped on their way to the destination node. But to the application both these mechanisms are totally indistinguishable. So though packet level failures do not constitute a new category of failure conditions, they are mentioned here for the sake of completion.

Entity that timeouts	**Possible reasons for timeout**
Active node	• While responding to service install requests • While providing service updates
Service manager	• While responding to client's service install/uninstall requests. • While sending updates to clients
Authentication Server	• While responding to authentication request from any other entity who is entitled to use the Authentication Server
Service locator	• While responding to service location requests from a node or service manager

Table 3.1: Possible reasons for timeout errors

A notification message that has the status code set to "Hold timer expired" is sent to the peer and the session is closed.

3.5.5 Errors causing Service Preemption

There could be several reasons for a service to be pre-empted while it is active in the Service Manager's domain.

Consider a new service request arrives with the Setup priority higher than the Holding priority of the currently active service. If the available resources are insufficient for installation of a new service, the current service is pre-empted to free the resources required by the new service.

Consider the case where the Service Installer node detects a violation of the access privileges granted to the service at the time of installation. The violation could be of any nature. For e.g., the service uses up more than its share of resources thus starving other processes. The resources include CPU time, memory, etc. One more example could be that the service attempts access to a resource that currently does not belong to that service. The service that violates the access criteria should be pre-empted immediately to prevent any damage to the other active services. All the resources held should be released.

If the node running the service reboots or is brought down administratively, the service gets removed.

In all the cases, the Service Manager must be informed about the preemption of the service. The Service Manager sends a preemption notification to the client and the session with the client is terminated.

3.5.6 Authentication Failure

The Authentication Server receives authentication requests from various entities and responds depending on whether the security information is found authentic or not. There can be three ways as specified below when the authentication may fail.

- Source authentication: The Authentication Server fails to verify that the source of the packet is what the security information implies.
- Message integrity: The Authentication Server notices some anomalies leading to a conclusion that the packet may have been tampered on its way to the receiving entity.
- Policy check: The current Service Level Agreement (SLA) does not allow the requested action to be performed. For example, client requests a particular service to be installed while according to the current SLA entries, it is not authorized to do so.

In case of an Authentication failure condition the session with the peer must be closed releasing all the associated resources. A notification message must be sent to the peer with an appropriate status code.

3.5.7 Service Level Agreement (SLA) expired

The Service Level Agreement (SLA) is an agreement between the client and the service manager that allows the client to use certain services during certain period of time in the service manager's network. There are various ways in which a service request can fail due to reasons related to SLA expiration.

- Before service installation: When the Authentication Server receives the service install request packet, it inquires the SLA database whether at that point of time, the client is authorized to install the service. In this case, the SLA would indicate that the client's SLA does not permit it to install the service. So before the service is installed, the failure is notified to the client
- After service installation: If the Authentication Server detects that at the time of service install request, the client has the required authorization to install the service, it sends a positive acknowledge (assuming other checks to be successful) to the service manager and the service gets installed. But if during the course of service execution in the service manager's network, the SLA database notifies the service manager of the expiry of the client's SLA, the service manager revokes the service and sends a failure notice to the client.

Note that SLA expiry before the service installation is different from the failure of policy check performed during authentication. Failure of policy check implies that the client is not allowed to install a particular service at any time in the service manager's network. SLA expiry before a service gets installed means the client is not allowed to install a particular service at this point of time. The client could register for the service (probably by subscribing to it) and continue to use it later.

The Service Manager sends a notification message with the status code set to "SLA Expired" to the client and then closes the session. The client needs to setup a new SLA with the Service Manager in order to continue to use the service.

3.5.8 Unknown/Internal Error

An unknown or internal error occurred while processing the message sent by the peer. The exact cause of the error is unknown possibly because it is not exposed to the protocol layer. In any case, the session with the peer should be closed.

3.6 Fatal Error Conditions

Certain error conditions are considered fatal and it is important to prevent such errors from occurring in future. Errors such as Access Violation, Authentication failure and Bad Sequence number are considered to be fatal.

Error conditions such as Resources Unavailable and Timeout indicate that the node has allocated its resources to different services and is not able to accept any more service install request. This may be an indication of a Denial-of-Service attack going on against the node. It is important to identify such attacks and eliminate the source. [27] suggests mechanisms to identify and deal with Denial-of-Service attacks.

When such fatal errors occur, the relevant information (for e.g., type of error, the entity or client responsible for the error, etc.) must be logged. Depending on the previous logs and the type of error, future requests from a peer or a client should be blocked. The Service Manager must revoke the client's secret key. If the client is a member of a group identified by the Service Manager, the Service Manager should also change the group key by redistributing a new group key to the other members of the group.

Chapter 4

Specification and Verification of the Proposed Service Management Framework

Active networks enable users to customize network processing through the deployment of application-specific protocol frameworks, such as the one described in this thesis, into the nodes of the network. The performance and security of the network is compromised if the injected code is inserted by a malicious entity or contains unintentional mistakes or if the protocol framework does not work as expected. A major requirement in such systems is to enable developers to construct protocol frameworks that operate reliably.

In order to evaluate the user-defined protocol framework, formal methods of specification and verification have proved to be useful. Specification is the process of describing a system and its properties. Formal specification uses a language with mathematically defined syntax and semantics. Properties described by the specification can include functional behavior, timing behavior, performance

characteristics or internal structure. Verification is the process of mathematically proving the exactness of the specification.

In this chapter, we describe the different approaches adopted by related research work on verification of protocols and protocol framework. The rest of the chapter is organized as follows. We begin with a brief description about the research work that is going on in verification of protocols and protocol framework. We then compare and contrast two different approaches for verification of protocols and chose one of them for the proposed protocol framework. We then describe SPIN verification system and Promela language in brief. Thereafter, we describe the specification details for the proposed service management framework followed by the verification using SPIN.

There have been a number of studies on protocol verification [24][31][29] that deal with the issue of correctness of a given protocol specification by testing it for safety and liveness properties. The safety and the liveness properties of a protocol specification provide a measure of its correctness. Verification of safety properties guarantee that the protocol does not violate any constraints imposed or the system always ends in one of the valid end-states determined by the designer. Verification of liveness properties tests that the protocol does not deadlock and that it always makes progress.

4.1 Verification Approaches

Historically, there are two major approaches to verification of systems: model checking and theorem proving. A theorem proving system takes an abstract description of a system in terms of algebraic or logic formulae and attempts to prove properties of the system. Model checking relies on building a finite state model of the system and checking that a desired property holds in that model. Following is a comparison between the two approaches in the context of their applicability to the specification and verification of communications protocols and protocol frameworks.

4.1.1 Theorem Proving

Theorem proving is a technique where both the system and its desired properties are expressed as formulae in some mathematical logic. This logic is given by a formal system, which defines a set of axioms and a set of inference rules. Theorem proving is the process of finding the proof of a property from the axioms of the system. Theorem provers rely on techniques like structural induction, rewrite-rules and proofs by contradiction to prove properties of systems. But finding proofs in theorem proving systems is a difficult process. General search procedures have had noteworthy success in solving various combinatorial problems, but in general proving properties of arbitrary systems can be hard. A theorem proving system cannot easily prove temporal behavior. Making statements about temporal properties requires the notion of ‘state’ to be embedded in the system for which theorem proving systems are not well equipped.

Theorem proving systems also require that the description of the system be abstracted so that the properties can be clearly specified. While useful for verification of the properties, a consequence of this strategy is that the implementation differs substantially from the specification. This makes it difficult to ascertain if the implementation preserves the properties expressed by the specification.

4.1.2 Model Checking

As explained earlier, model checking is a technique that relies on building a finite model of a system and checking that the desired property holds in that model. Generally, the check is performed as an exhaustive state space search that is guaranteed to terminate since the model is finite-space. In contrast to theorem proving, model checking is automatic and fast [10]. Model checking can be used to check partial specifications, so it provides useful information about the model's correctness even before the system is completely specified. Another advantage is that it produces counter-examples, which provide a useful aid for debugging. Model checking is also useful for checking temporal properties that communication protocols exhibit. Since model checkers use state-space search, it is relatively easy to determine temporal ordering of events.

The principal problem with model checking is the state explosion problem. This issue is particularly relevant for protocol composition. Each protocol is represented by a finite state-space and as protocols are composed, the state space of the composed framework grows exponentially. However, many techniques such as bit-state hashing

[16], partial order reduction [8] and BDDs [28] have been identified that reduce the size of the state space. It is thus apparent that the advantages of using a model checking system for verifying communication protocol frameworks far outweigh its limitations.

4.2 The SPIN Model Checker

SPIN [17] is a generic verification system that supports the design and verification of asynchronous process systems. SPIN verification models are focussed on proving the correctness of process interactions, and they attempt to abstract as much as possible from internal sequential computations. Process interactions can be specified in SPIN with rendezvous primitives, with asynchronous message passing through buffered channels, through access to shared variables, or with any combination of these.

As a formal methods tool, SPIN aims to provide:

1) An intuitive, program-like notation for specifying design choices unambiguously, without implementation detail,
2) A powerful, concise notation for expressing general correctness requirements, and
3) A methodology for establishing the logical consistency of the design choices from 1) and the matching correctness requirements from 2).

In SPIN the notations are chosen in such a way that the logical consistency of a design can be demonstrated mechanically by the tool. SPIN accepts design specifications written in the verification language PROMELA (a Process Meta Language) [32], and it accepts correctness claims specified in the syntax of standard

Linear Temporal Logic (LTL). There are no general decision procedures for unbounded systems, and one could well question the soundness of a design that would assume unbounded growth. Models that can be specified in PROMELA are, therefore, always required to be bounded, and have only countable distinct behaviors. This means that all correctness properties automatically become formally decidable, within the constraints that are set by problem size and the computational resources that are available to the model checker to render the proofs.

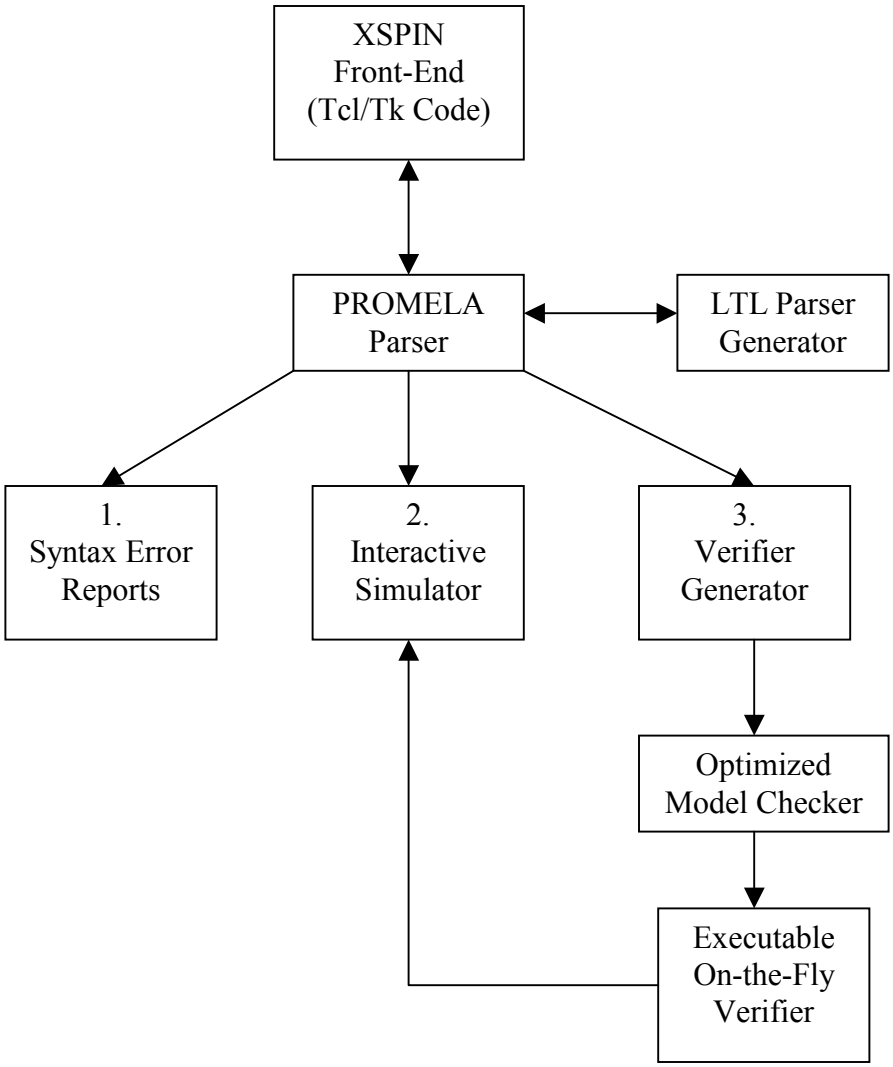


Figure 4.1 Architecture of the SPIN Model Checker

The basic structure of the SPIN model checker is illustrated in Figure 4.1. The typical mode of working is to start with the specification of a high level model of a concurrent system, or distributed algorithm, typically using SPIN's graphical front-end XSPIN. After fixing syntax errors, interactive simulation is performed until basic confidence is gained that the design behaves as intended. Then, in a third step, SPIN

is used to generate an optimized on-the-fly verification program from the high level specification. This verifier is compiled, with possible compile-time choices for the types of reduction algorithms to be used, and executed. If any counter examples to the correctness claims are detected, these can be fed back into the interactive simulator and inspected in detail to establish and remove their cause.

4.3 Promela

SPIN is described in a modeling language called Promela (Process or Protocol Meta Language). The language allows for the dynamic creation of concurrent processes. Communication via message channels can be defined to be synchronous (i.e., rendezvous), or asynchronous (i.e., buffered). XSPIN is a graphical front-end to drive SPIN (written in Tcl/Tk).

Given a model system specified in Promela, SPIN can perform random or interactive simulations of the system's execution or it can generate a C program that performs a fast exhaustive verification of the system state space. During simulations and verifications SPIN checks for the absence of deadlocks, unspecified receptions, and unexecutable code. The verifier can also be used to prove the correctness of system invariants and it can find non-progress execution cycles. Finally, it supports the verification of linear time temporal constraints; either with Promela never-claims or by directly formulating the constraints in temporal logic.

The verifier is setup to be fast and to use a minimal amount of memory. The exhaustive verifications performed by SPIN are conclusive. They establish with

certainty whether or not a system's behavior is error-free. Very large verification runs, which can ordinarily not be performed with automated techniques, can be done in SPIN with a ``bit state space'' technique. With this method the state space is collapsed to a few bits per system state stored. Although this technique doesn't guarantee certainty, the coverage is better, and often much better, than that obtained with traditional random simulation.

Promela programs consist of processes, message channels, and variables. Processes are global objects. Message channels and variables can be declared either globally or locally within a process. Processes specify behavior while channels and global variables define the environment in which the processes run. The syntax of Promela is C-like.

4.4 Specification of the Service Management Framework

4.4.1 Terminology

The terms that are used to describe the service management framework are described below.

Network Entity

A Network Entity is defined to be a process that implements a part of the proposed protocol framework. It interacts with other Network Entities in the process of

providing the required functionality. Examples of Network Entities are Service Manager, Code Server, Service Locator, etc.

Service

A Service is a functionality that is provided by using the resources in the Service Manager's network.

Client/User

A Client/User is an entity external to the Service Manager's network. Clients subscribe for the services provided by the Service Manager.

4.4.2 Formal Model

The principal idea behind the specification of this protocol framework is to model the framework as succinctly as possible in order to be able to study its structure, and establish and verify its behavior. A correctly defined service management framework consists of the various entities functioning in a synchronous fashion such that the overall system behavior meets the specification.

The specification and verification of the system is carried out by modeling the execution of the service management framework for active networks. The interaction between the different entities is achieved by passing the relevant information along with the control messages through the communication channels. Global environment variables constantly provide a snapshot of the network conditions and monitor the

progress of the requests from client. Using primitives available in SPIN, the access to shared variables is restricted to avoid undefined states.

4.4.3 System Specification

Verification in SPIN involves defining the model in its input language, Promela. The Promela program is fed to the SPIN model checker that tests the correctness, completeness and consistency of the composition. The service management framework is defined by modeling the properties of different entities involved. The final model is fed to the verification system.

In this section, we outline the salient features of a specification. We discuss how the system is modeled, how the entities are defined and how their properties are specified for later verification. We describe verification techniques utilized by SPIN to test models and describe the verification of various properties of the model.

4.4.4 System Model

The system model consists of specifications for the network entities as well as for the service management framework. External sources (client) of inputs have been added to test the functionality of the system. Global state information provides an up-to-date snapshot of the system allowing us to determine the correctness of the system. Events such as unidentified message, source authentication failure, message integrity check failure, policy verification failure, packet corruption, access violation, packet loss and

so on have been modeled in the specification developed. Every node has a node ID to identify itself. A node can be running one or more network entities. For e.g., same node could run a Service Manager and the Service Locator. Each entity has its own ID to help the receiving entity identify the sender. The connectivity between the different entities is provided using channels primitive in Promela.

Different packet types have been defined for communication between different entities. The structures describing the different packet types include the basic parameters that particular packet type must include. The basic idea while using SPIN verification system is to abstract the system specification so as to reduce system state. Only the elements that influence the outcome of the execution of the system are modeled leaving all the other details out. For implementation purposes, additional elements could be added.

4.4.5 Service Management Entities

Every entity is a separate process in Promela. Using the language constructs in Promela, the properties and interfaces of each entity are specified.

The communication channels between the different entities were synchronous. In other words, a message could be sent to an entity only if the other entity is already listening for messages. All the entities have two channels – one for incoming and one for outgoing messages. Each entity listens on its incoming channel for messages from the peer. After it receives the message, it processes it and updates the global state information in case the system state has changed. After processing the peer's

message, the entity sets the “status code” field in the response message depending on the outcome of processing. The message instantiations are local to the entities. A response message is generated and sent to the peer. Whenever a failure occurs, the global state is updated with relevant information and the system halts.

4.4.6 Message types

`mtype` is used to declare the different message types in Promela. Following shows the different message types used in the specification.

```
mtype = {
          svc_install_request, svc_install_response,
          svc_uninstall_request, svc_uninstall_response,
          svc_update_request, svc_update_response,
          authenticate_request, authenticate_response,
          sla_request, sla_response,
          slp_request, slp_response,
          code_request, code_response,
          unknown
};
```

Only one mtype-definition is allowed which must be global and at most 256 symbolic constants can be declared; an `mtype` variable is 8 bits wide.

The advantage of `mtypes` over `#defines` is that the former type of symbolic constants is recognized by Spin and during simulations the symbolic names are used instead of the values they represent.

4.4.7 Atomic Statements

The "`atomic`" construct in Promela is used to execute a group of statements in one indivisible step; i.e., without interleaved execution of other processes. For example following group of statements are executed sequentially without any interleaving.

```
atomic {
     failure = 0;
     if
     :: svc_req_pkt.msgtype = svc_uninstall_req;
     :: svc_req_pkt.msgtype = unknown;
     fi;
     svc_req_pkt.svc_id = svc_id;
     svc_req_pkt.auth_id = host_id;
     svc_req_pkt.result = 0;
};
```

An `atomic` statement is enabled if its first statement is. During its execution, control can only be transferred outside the scope of an atomic statement by an explicit `goto` or at a point where a statement within its scope becomes blocked. If this statement subsequently becomes enabled again, execution may continue at that point.

There is no constraint on what may occur inside the scope, other than that no nested atomic or `d_step` is allowed. In particular, it is possible to jump to any (labeled) location within the scope of an atomic statement.

4.4.8 Modeling Unidentified Message Error

Errors such as "Unidentified message" can occur due to several reasons. Such events can be modeled using the non-determinism that Promela offers in the "`if`" construct.

```
if
:: statements
...
:: statements
fi;
```

The "`if`" statement selects one among its options (each of them starts with `::`) and executes it. An option can be selected if its first statement is enabled. A selection blocks until there is at least one selectable branch. If more than one option is selectable, one will be selected at random.

```
if
:: svc_req_pkt.msgtype = svc_install_req;
:: svc_req_pkt.msgtype = unknown;
fi;
```

As both the statements in the "`if`" block are assignment statements, both are selectable. So one of them would be selected randomly. This causes the `msgtype`

field to be set to the `svc_install_req` or `unknown` with equal probability. When the receiving entity discovers that the `msgtype` is `unknown`, an unidentified message error is flagged.

In the same manner the authentication failure and the service download failure errors are modeled.

4.4.9 Modeling Authentication Failures

```
if
:: auth_pkt.msgtype = authenticate_msg_res;
     auth_pkt.result = 1; /* Successful */
     auth_output!auth_pkt;
:: auth_pkt.msgtype = authenticate_msg_res;
     auth_pkt.result = 0; /* Failure */
     auth_output!auth_pkt;
fi;
```

Either one of the two possible statements is executed with equal probability. Depending on which one gets selected, the received message is either successfully authenticated or fails.

4.4.10 Modeling Service Download Failures

```
if
:: codesr_pkt.result = 1;
```

```
     codesr_pkt.hash_encrypt = 1; /* Successful */
     to_installer!codesr_pkt;
:: codesr_pkt.result = 0;
     codesr_pkt.hash_encrypt = 0; /* Failure */
     to_installer!codesr_pkt;
fi;
```

`hash_encrypt` is the field that carries the encrypted hash of the service module. Depending on which statement gets executed, the service download would be declared successful (`hash_encrypt = 1`) or unsuccessful (`hash_encrypt = 0`).

4.4.11 Modeling Access Violation by Installed Service

We make use of the constructs for repetition provided by Promela to model the change in the status of the service while it is installed. The repetition construct "`do`" is similar to a selection, except that the statement is executed repeatedly, until control is explicitly transferred to outside the statement by a `goto` or `break`.

```
do
:: statements
...
:: statements
od;
```

Following statements show the repetition construct "do" being used to model the change in the status of the service. If the service is installed (shown by svc_stats[svc_id].installed == 1), the status of the service is constantly changed.

```
do
::   if
     ::svc_stats[svc_id].installed == 1;
          if
          ::svc_stats[svc_id].status_code = 0;
          ::svc_stats[svc_id].status_code = 1;
          ::svc_stats[svc_id].status_code = 2;
          ::svc_stats[svc_id].status_code = 3;
          ::svc_stats[svc_id].status_code = 4 ->
               svc_stats[svc_id].installed = 0;
          fi;
     fi;
od;
```

The status code of 4 has been assigned for service violation. So, if the status code of a service is changed to 4, it is uninstalled and so the installed flag is turned off.

4.4.12 Modeling Wait Intervals

Though there is no concept of time in Promela, approximate time delays can be created using the repetition construct as shown below,

```
do
::    if
      :: count > 5 -> break;
      fi;
:: count = count + 1;
od;
```

This control would break out of the repetition statement once the value of `count` reaches 5. Though this does not guarantee a constant time delay every time the repetition statement gets executed, but it is sufficient if the entity wants to block itself for a short time interval before proceeding.

4.4.13 Temporal claims

Temporal claims are defined by Promela never claims and are used to detect behaviors that are considered undesirable or illegal.

When checking for state properties, the verifier will complain if there is an execution that ends in a state in which the never claim has terminated; i.e., has reached the closing } of its body.

A never claim is intended to monitor every execution step in the rest of the system for illegal behavior. Such illegal behavior is detected if the never claim matches along a computation.

The never claim used in the specification is as follows. It shows that along every computation, each system state in which `failure` is true should not be followed by a state where `svc_installed` is true.

```
never
{
      do
      :: failure -> break
      :: skip
      od;
      do
      :: svc_installed;
      od;
}
```

Let us analyze it using a similar never claim as an example.

Let p and q be two boolean expressions and consider the property that

``along every computation, each system state in which p is true (a p-state) is eventually followed by a q-state"

The following never claim verifies whether the property holds; i.e., it will detect any violation of the property:

```
never {
        do
        :: p -> break
        :: skip
        od;
accept:
        do
        :: !q
        od
}
```

The first repetition terminates only in p-states. Such a state should eventually be followed by a q-state. The second repetition (hence the never claim) cannot terminate, so the never claim either eventually blocks because the computation sequence reaches a q-state or matches because the (infinite) computation cycles through an acceptance state. The latter occurs precisely if there are no subsequent q-states. Because the analyzer guarantees an exhaustive search for computations along which the never claim is matched, a computation violating the property is guaranteed to be detected (if there is one).

4.5 Verification of the Service Management Framework

SPIN is used to perform on-the-fly verification of the Promela specification generated for the system. SPIN enables the verification of liveness and safety properties as well as temporal properties of the model. In SPIN, the verification of these two classes of

properties is performed separately. Verification of safety properties involves checking for correctness and completeness of the composition. This implies checking for any assertion violations and testing for any unreachable code. Verification of liveness properties involves ensuring that the system does not enter into any deadlock or livelock. Temporal properties can be defined and verified to ascertain specific behavioral properties of the model.

4.5.1 Correctness and Completeness Verification

Correctness of a system requires the individual entities be structurally sound. It ensures that all the entities are invoked correctly, there is no violation of read/write sequence and accessing packet variables and all constraints set by the entities are satisfied. Checking the syntax of the specification enables us to catch any incorrect calls to component interfaces. Testing the model for safety properties automatically flags any violation of the write/read sequence for packet variables. Assertions are used to specify the constraints of the entities. Therefore, any violations of the constraints placed by the module are also flagged while checking for safety properties.

Checks on the safety properties of the system describe what is allowed to happen. However, just because safety properties hold does not guarantee that the system is functioning correctly. Liveness restricts the long-term behavior of the system by specifying what must eventually happen. Progress must be guaranteed, i.e., there are not deadlocks or livelocks. Every entity should either progress towards completion or

be explicitly marked acceptance cycles. Checking the model for non-progress cycles using the SPIN verifier catches these conditions. The SPIN verifier also checks to see if there is any unreachable code, i.e., states that the system can never reach.

4.5.2 Verification Results

It is essential that the verification process be carried out on a specification model that closely resembles a real-life scenario. Apart from specifying and verifying a simple model as described in Figure 2.3, several other extensions were tested. In order to determine the complexity of the different models verified, the amount of memory used in the process of verification as well as the number of state transitions that took place were measured. Following are the results and the corresponding plots.

4.5.2.1 Varying the number of Client processes

Varying the number of active client processes, the amount of memory needed to verify the model was observed. Following table shows the observed values. As we can see there is a large increase in the amount of memory consumed with the increase in the number of active client processes.

Number of client processes	Amount of memory used in megabytes
1	2.542
2	21.480
3	241.143

Table 4.1 Amount of memory used with the increase in the number of active client processes

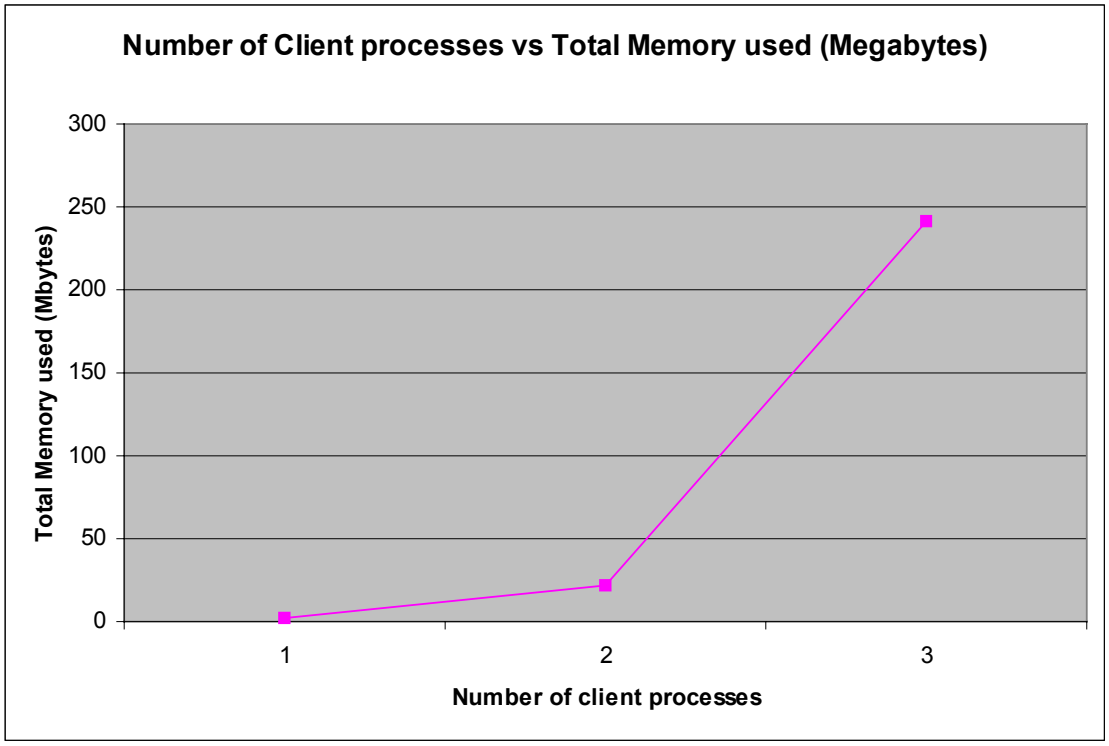


Figure 4.2 Increase in the total memory used with the increase in the number of active client processes

The numbers of state transitions were measured by varying the number of active client processes. Following table shows the observed valued. Again, we can see the large increase in the number of state transitions with the increase in the number of client processes.

Number of client processes	Number of state transitions
1	734
2	214587
3	3134220

Table 4.2 Number of state-transitions with the increase in the number of active client processes

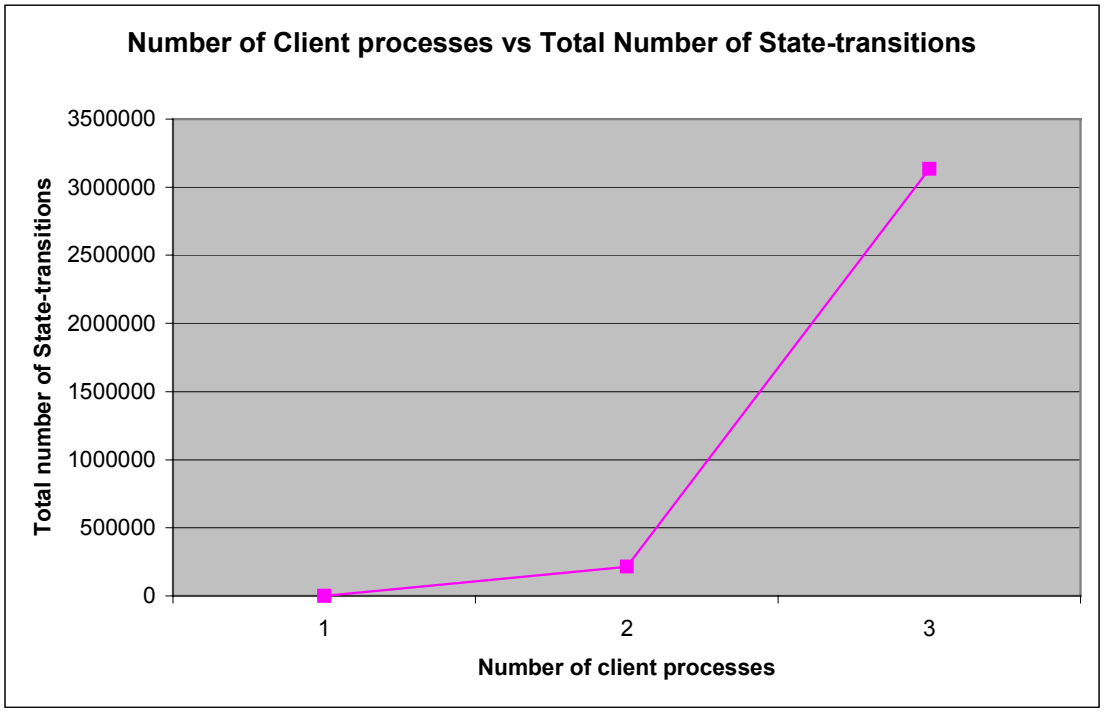


Figure 4.3 Increase in the number of state-transitions with the increase in the number of active client processes

4.5.2.2 Varying the number of Service Install Requests

The number of Service Install Requests sent by the clients indicates the amount of load on the Service Manager. The client may send requests in sequential order, i.e. send the next request after the current request is completely processed, or in burst mode where the client bursts all the requests at a time and then waits for the Service Manager to process them. Note that in the burst mode, the Service Manager needs to queue incoming requests while one of them is being processed. Following tables show the observed values of the memory used while the number of request was varied in both sequential and burst modes. Observe that in the burst mode, there is a large

increase in the amount of memory used with the increase in the number of Service Install Requests.

Number of service install requests (sequential)	Amount of memory used in megabytes
1	2.542
2	2.747
3	4.385
4	8.891
5	20.053
6	44.425
7	97.685
8	188.722

Table 4.3(a) Amount of memory used with the increase in the number of Service Install Requests (Sequential)

Number of service install requests (burst)	Amount of memory used in megabytes
1	2.542
2	15.250
3	136.701
4	230.605

Table 4.3(b) Amount of memory used with the increase in the number of Service Install Requests (Burst)

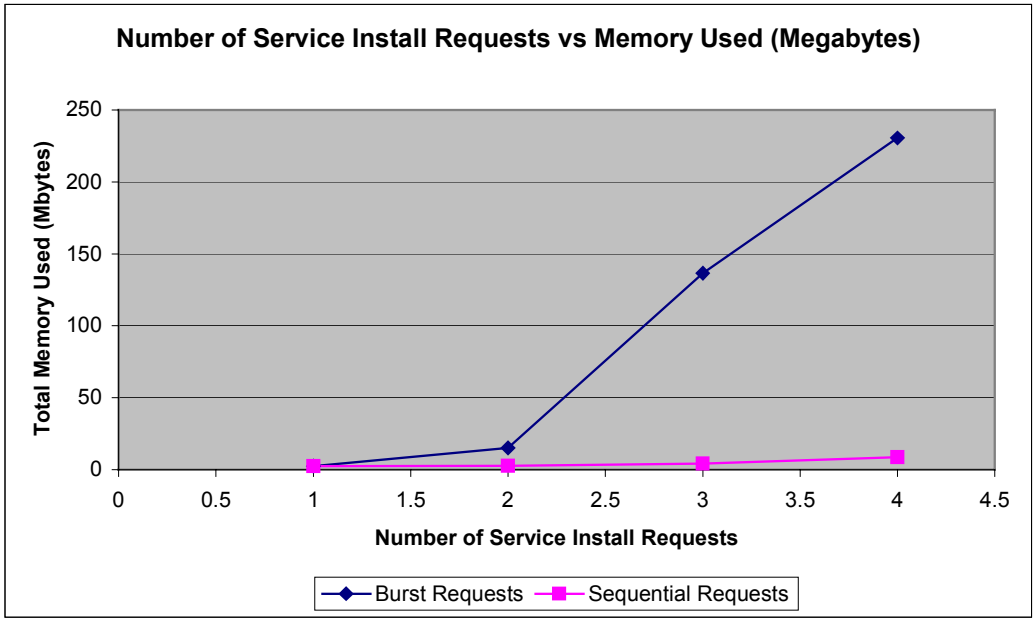


Figure 4.4 Increase in the total memory used with the increase in the number of Service Install Requests

Following table shows the increase in the number of state transitions measured with the increase in the number of service install requests.

Number of service install requests (sequential)	Number of state-transitions
1	734
2	26083
3	190348
4	746122
5	2272250
6	6113040
7	15331100
8	35165100

Table 4.4 (a) Number of state-transitions with the increase in the number of Service Install Requests (Sequential)

Number of service install requests (burst)	Number of state-transitions
1	734
2	101890
3	1407480
4	2853760

Table 4.4 (b) Number of state-transitions with the increase in the number of Service Install Requests (Burst)

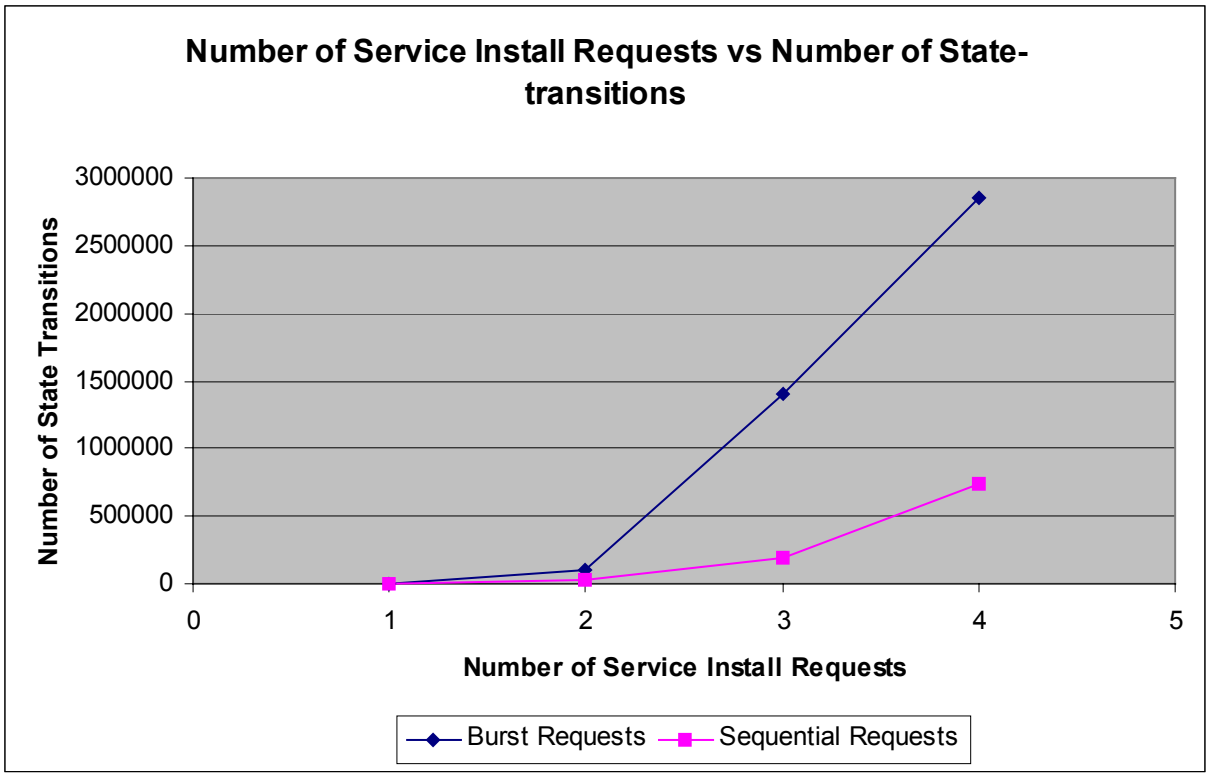


Figure 4.5 Increase in the number of state-transitions with the increase in the number of Service Install Requests

4.5.3 Observations

Following are the observations based on the statistics plotted in the previous sub-sections.

- Increase in the number of entities adds to the overall complexity of the system evident from the increase in the number of state-transitions and the amount of memory used for verification
- Increase in the number of requests handled by the Service Manager causes the following behavior.
    - If the requests are made in order, there is a moderate increase in complexity as shown by the Figures 4.4 and 4.5.
    - In case of burst mode of requests, there is a huge increase in the complexity of the system evident from the large increase in the number of state-transitions and the amount of memory used for verification

Chapter 5

Summary And Future Work

Active Networking provides a new paradigm of networking in which users are able to create and inject custom services and protocols in the network. This thesis proposes a new model for a framework that manages such services and protocols. The properties of the new protocol framework model are identified by studying the limitations of current models and analyzing requirements of protocol frameworks for active networking.

The thesis begins by identifying the requirements of a Service Management framework. Overall functionality is distributed among separate modules such that each module acts independently of the other. A message verification process that involves a trusted Authentication Server follows every message exchange. Redundancy is achieved by having multiple instances of the entities that are crucial for the functioning of the system. We describe the finite state machines for the various entities involved in the framework. A typical interaction between the different entities shows the events that take place when a client makes a Service Install Request.

The different types of message involved and several packet fields that are sent in the packets are discussed. Several failure conditions that may occur during the processing of the client's request are explained. It is followed by the recovery conditions needed to restore the state of the system.

This thesis work also describes the specification and verification of the proposed protocol framework using SPIN/Promela verification system. The functionality of the individual components has been modeled as concurrent processes in Promela. Several different scenarios are considered and each of them is verified. Statistics about the complexity of the system in each of the scenario is plotted and the overall trend is discussed. The statistics indicate that the system complexity increases with the increase in the number of entities or number of requests handled by the Service Manager.

Though the Service Management Framework proposed by the thesis work is functionally complete, following are the extensions that could add new features to the framework. The proposed Service Management framework does not discuss sophisticated mechanisms to monitor the services installed. Additional modules can be added to provide the service-monitoring feature.

There can be an exchange of services between the member nodes by certain extensions to the Service Manager module. Each participating node registers the service that it can provide with the Service Manager. The Service Manager advertises the list of all available services to the member nodes. A node that needs a particular

service initiates a session with the member node that provides it. Issues like the level of trust associated with each member node need to be tackled for such an extension.

In some circumstances, the client would like to install some of its services on the Service Manager's network. So, along with the Service Install Request message, the client also provides the service bytecodes that will be executed on the member nodes of the Service Manager's network. This scenario demands higher level of trust verification between the Service Manager and the client.

Bibliography

[1] AN Working Group. “Architectural framework for active networks”. August 31 1998.

[2] A. Kulkarni, G. Minden, R. Hill, Y. Wijata, S. Sheth, H. Pindi, F. Wahhab, A. Gopinath, and A. Nagarajan. “Implementation of a prototype active network”. In OPENARCH’98, pages 130-143, April 1998

[3] A. Kulkarni, G. Minden, V. Frost, J. Evans. “Survivability of Active Network Services”. University of Kansas, 1999

[4] CRA. Research challenges for the next generation internet. “Report from the Workshop on Research Directions for the Next Generation Internet”, May 1997

[5] D. Alexander, W Arbaugh, M. Hicks, P. Kakkar, A. Keromytis, J. Moore, C. Gunter, S. Nettles, and J. Smith. “The SwitchWare active network architecture”. IEEE Network, 12(3): 29:36, May/June 1998.

[6] D. Wetherall. “Service Introduction in an Active Network”. Ph.D. Dissertation. Massachusetts Institute of Technology. November, 1998

[7] D. Wetherall, J. Guttag, D. Tennenhouse, “ANTS: A Toolkit for Building and Dynamically Deploying Network Protocols”. The First IEEE Conference on Open Architectures and Network Programming (OPENARCH ’98), San Francisco, April 1998.

[8] D. Peled. “Combining Partial Order Reductions with On-the-fly Model Checking”. International Conference on Computer Aided Verification. pp. 377-390. Stanford, CA. LNCS 818, 1994.

[9] E. Amir, S. McCanne and R. Katz. “An active service framework and its application to real-time multimedia transconding”. In SIGCOMM’98, September 1998.

[10] E. Clarke and J. Wing. “Formal Methods: State of the Art and Future Directions”. ACM Computing Surveys. Vol. 28, No. 4, pp. 626-643, December 1996.

[11] E. Guttman, C. Perkins, J. Veizades, M. Day. “Service Location Protocol, Version 2 RFC 2608”. June 1999.

[12] E. Rosen, A. Viswanathan, R. Callon. “Multiprotocol Label Switching Architecture”, RFC 3031. January 2001

[13] G. Bochmann. “Finite State Description of Communication Protocols”. Computer Networks, Vol. 2, Oct 1978.

[14] G. Goldszmidt and Y. Yemini. “Distributed Management by Delegation”. International Conference on Distributed Computing Systems, IEEE Computer Society, Vancouver, British Columbia, Canada, June 1995.

[15] G. Hjalmtysson and A. Jain. “Agent-based approach to service management – towards service independent network architecture”. IFIP/IEEE International symposium on Integrated Network Management – IM’97, pages 715-729, San Diego, May 1997

[16] G. Holzmann. “An Analysis of Bit-State Hashing”. IFIP/WG6.1 Symposium on Protocol Specification, Testing and Verification. pp. 301-314, Warsaw, Poland, June 1995.

[17] G. Holzmann. “The Model Checker SPIN”. IEEE Transaction on Software Engineering, Vol. 23, No. 5, May 1997

[18] Hewlett-Packard OpenView VantagePoint family of products (http://www.openview.hp.com/products/vpwindows/index.asp)

[19] J. Backus. “Can Programming be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs”. Comm. Of the ACM, 21(8), August 1978

[20] J. Case, M. Fedor, M. Schoffstall and J. Davin, “A Simple Network Management Protocol (SNMP), STD 15, RFC 1157”. May 1990.

[21] J. Kiniry and D. Zimmerman. “A hands-on look at java mobile agents”. IEEE Internet Computing. 1(4): 21-30, July/August 1997

[22] J. Merve, S. Rooney, L. Leslie and S. Crosby. “The Tempest – A Practical Framework for Network Programmability”. IEEE Network Magazine, 12(3), May/June 1998

[23] J. Postel. “Service Mappings”, RFC 795. September 1981

[24] M. Diaz. “Modeling and Analysis of Communication and Cooperation Protocols using Petri net based models”. Computer Networks, Vol. 6, 1982.

[25] M. Hicks, P. Kakkar, J. Moore, C. Gunter and S. Nettles. “PLAN: A programming language for active networks”. International Conference on Functional Programming (ICFP) ’98, pages 86-93. ACM, September 1998

[26] M. Zapf, K. Herrmann, K. Geibs, and J. Wolfgang. “Decentralized snmp management with mobile agents”. IFIP/IEEE International Symposium on Integrated Network Management – IM’99, Boston, May 1999

[27] P. Ferguson, D. Senie. “Network Ingress Filtering: Defeating Denial of Service Attacks which employ IP Source Address Spoofing, RFC 2267”. January 1998

[28] P. Godefroid. “Symbolic Protocol Verification with Queue BDDs”. Logic in Computer Science, pp. 198-206, New Brunswick, NJ, July 1996

[29] P. Hermman and H. Krumm. “Compositional Specification and Verification of High Speed Protocols”. Research Report No. 540/1994, University of Dortmund, Dortmund, Germany, 1994

[30] P. Jackson. “The NuPrl Proof Development System, Version 4.1 Reference and User’s Guide”. Cornell University, Ithaca, N.Y., February 1994.

[31] P. Merlin. “Specification and Validation of Protocols’. IEEE Transaction on Communications, Vol. 27-11, November 1979.

[32] R. Gerth. “Concise Promela Reference”. Eindhoven University, August 1999

[33] S. Bhattacharjee, K. Calvert and E. Zegura. “An architecture for active networks”. HPN’97, April 1997

[34] S. Owre, N. Shankar, J. Rushby, D. Stringer-Calvert. “PVS System Guide, Version 2.3”. SRI International, September 1999

[35] Unified Modeling Language (UML) Resource Page, (http://www.omg.org/uml)

[36] X. Leroy, D. Doligez, J. Garrigue, D. Remy, J. Vouillon. “The Objective Caml System, Release 3.01”. Institue National de Recherche en Informatique et en Automatique, March 2001.

[37] Y. Yemini, G. Goldszmidt, S. Yemini, “Network Management by Delegation”. International Symposium on Integrated Network Management (ISINM ’91), pp. 95-107, Washington DC, April 1991

Appendix A

Promela Source

```
mtype = {
            svc_install_req, svc_install_res,
            svc_uninstall_req, svc_uninstall_res,
            svc_update_req, svc_update_res,
            authenticate_msg_req, authenticate_msg_res,
            mon_req, mon_res,
            sla_req, sla_res,
            slp_req, slp_res,
            code_req, code_res,
            unknown
      };

typedef svc_install_pkt {
      mtype msgtype;
      byte svc_id;
      byte auth_id;
      byte result;
      byte codesr
};

typedef svc_update_pkt {
      mtype msgtype;
      byte svc_id;
      byte auth_id;
      byte attr
};

typedef authenticate_pkt {
      mtype msgtype;
      byte auth_id;
      byte rcd_auth;
      bool result
};

typedef sla_pkt {
      mtype msgtype;
      byte svc_id;
      byte auth_id;
      bool result
};

typedef slp_pkt {
      mtype msgtype;
      byte svc_id;
      byte auth_id;
      byte codesr;
```

```
      bool result
};

typedef mon_pkt {
      mtype msgtype;
      byte svc_id;
      byte auth_id;
      bool result
};

typedef code_pkt {
    mtype msgtype;
      byte svc_id;
      byte auth_id;
      byte hash_encrypt;
      bool result;
};

typedef service_statistics {
      bit installed;
      byte status_code
};

chan client_svcmgr[2] = [0] of { svc_install_pkt };
chan authmgr_chan[2] = [0] of { authenticate_pkt };
chan svcmgr_svcinstaller = [0] of { svc_install_pkt };
chan slp_svcmgr[2] = [0] of { slp_pkt };
chan mon_svcmgr[2] = [0] of { mon_pkt };
chan sla_svcmgr[2] = [0] of { sla_pkt };
chan codesr_installer[2] = [0] of { code_pkt };

bit failure;
bit svc_installed;
bit unidentified_msg;
bit authentication_failure;
bit sla_expired;
bit service_unavailable;
bit code_error;
bit access_violation;

service_statistics svc_stats[5];

proctype client (byte host_id; byte svc_id)
{
      svc_install_pkt svc_req_pkt, svc_res_pkt;
      authenticate_pkt auth_pkt;
      chan from_svcmgr = client_svcmgr[0];
      chan to_svcmgr = client_svcmgr[1];
      byte count1 = 0;



      do
      ::    if
```

```
            :: count1 > 1 -> break;
            fi;
      ::

      atomic {
            count1 = count1 + 1;
            svc_req_pkt.svc_id = svc_id;
            svc_req_pkt.auth_id = host_id;
            svc_req_pkt.result = 0;
            byte count = 0;
            failure = 0;

            if
            :: svc_req_pkt.msgtype = svc_install_req;
            :: svc_req_pkt.msgtype = unknown;
            fi;
      };


      to_svcmgr!svc_req_pkt ->
      atomic {
            from_svcmgr?svc_res_pkt;
            if
            :: auth_pkt.msgtype = authenticate_msg_req;
            :: auth_pkt.msgtype = unknown;
           fi;
           auth_pkt.auth_id = host_id;
            auth_pkt.rcd_auth = svc_res_pkt.auth_id;
            auth_pkt.result = 0;
            authmgr_chan[0]!auth_pkt -> authmgr_chan[1]?auth_pkt;
      };

      if
      :: auth_pkt.result == 1 ->
            if
            :: svc_res_pkt.result == 1;
                  failure = 0;
            :: else ->
fail1:                failure = 1;
            fi;
      :: else ->
fail2:            failure = 1;
      fi;

      do
      ::    if
            :: count > 5 -> break;
            fi;
      :: count = count + 1;
      od;

      atomic {
            failure = 0;
            if
```

```
            :: svc_req_pkt.msgtype = svc_uninstall_req;
            :: svc_req_pkt.msgtype = unknown;
            fi;
            svc_req_pkt.svc_id = svc_id;
            svc_req_pkt.auth_id = host_id;
            svc_req_pkt.result = 0;
      };

      to_svcmgr!svc_req_pkt ->
      atomic {
            from_svcmgr?svc_res_pkt;
            if
            :: auth_pkt.msgtype = authenticate_msg_req;
            :: auth_pkt.msgtype = unknown;
            fi;
            auth_pkt.auth_id = host_id;
            auth_pkt.rcd_auth = svc_res_pkt.auth_id;
            auth_pkt.result = 0;
            authmgr_chan[0]!auth_pkt;
            authmgr_chan[1]?auth_pkt;
      };



      if
      :: auth_pkt.result == 1;
            if
            :: svc_res_pkt.result == 1;
                  failure = 0;
            :: else ->
fail3:                  failure = 1;
                  fi;
      :: else ->
fail4:      failure = 1;
      fi;


      od;


}

proctype monitor (byte host_id; byte svc_id)
{
      mon_pkt mon_req_pkt, mon_res_pkt;
      authenticate_pkt auth_pkt;
      chan from_svcmgr = mon_svcmgr[1];
      chan to_svcmgr = mon_svcmgr[0];

      do
      ::    if
            ::svc_stats[svc_id].installed == 1;
                  if
                  ::svc_stats[svc_id].status_code = 0;
```

```
                    ::svc_stats[svc_id].status_code = 1;
                    ::svc_stats[svc_id].status_code = 2;
                    ::svc_stats[svc_id].status_code = 3;
                    ::svc_stats[svc_id].status_code = 4;
                    fi;
              fi;
      od;
}

proctype slp_manager (byte host_id)
{
      slp_pkt slp_req_pkt, slp_res_pkt;
      authenticate_pkt auth_pkt;
      chan from_svcmgr = slp_svcmgr[1];
      chan to_svcmgr = slp_svcmgr[0];
      do
      :: from_svcmgr?slp_req_pkt;
            if
            :: slp_req_pkt.msgtype == slp_req;

      atomic {
            if
            :: auth_pkt.msgtype = authenticate_msg_req;
            :: auth_pkt.msgtype = unknown;
            fi;
            auth_pkt.auth_id = host_id;
            auth_pkt.rcd_auth = slp_req_pkt.auth_id;
            auth_pkt.result = 0;
            authmgr_chan[0]!auth_pkt;
            authmgr_chan[1]?auth_pkt;
      };
                  if
                  :: auth_pkt.result == 1;
                  atomic {
                        if
                        :: slp_res_pkt.msgtype = slp_res;
                        :: slp_res_pkt.msgtype = unknown;
                        fi;
                        slp_res_pkt.auth_id = host_id;
                        if
                        :: slp_res_pkt.result = 1;
                              slp_res_pkt.codesr = 5;
                              to_svcmgr!slp_res_pkt;
                        :: slp_res_pkt.result = 1;
                              slp_res_pkt.codesr = 5;
                              to_svcmgr!slp_res_pkt;
                        :: slp_res_pkt.result = 1;
                              slp_res_pkt.codesr = 5;
                              to_svcmgr!slp_res_pkt;
                        :: slp_res_pkt.result = 0;
                              slp_res_pkt.codesr = 0;
                              to_svcmgr!slp_res_pkt;
                        fi;
                  };
```

```
                  :: else ->
                  atomic {
                        if
                        :: slp_res_pkt.msgtype = slp_res;
                        :: slp_res_pkt.msgtype = unknown;
                        fi;
                     slp_res_pkt.auth_id = host_id;
                     slp_res_pkt.result = 0;
                     slp_res_pkt.codesr = 0;
                     to_svcmgr!slp_res_pkt;
                  };
                  fi;
            :: else ->
                  unidentified_msg = 1;
            fi;
      od;
}

proctype sla_manager (byte host_id)
{
      sla_pkt sla_req_pkt, sla_res_pkt;
      authenticate_pkt auth_pkt;
      chan from_svcmgr = sla_svcmgr[1];
      chan to_svcmgr = sla_svcmgr[0];
      do
      :: from_svcmgr?sla_req_pkt;
            if
            :: sla_req_pkt.msgtype == sla_req ->

      atomic {
            if
            :: auth_pkt.msgtype = authenticate_msg_req;
            :: auth_pkt.msgtype = unknown;
            fi;
            auth_pkt.auth_id = host_id;
            auth_pkt.rcd_auth = sla_req_pkt.auth_id;
            auth_pkt.result = 0;
            authmgr_chan[0]!auth_pkt;
            authmgr_chan[1]?auth_pkt;
      };
                  if
                  :: auth_pkt.result == 1 ->
                  atomic {
                        if
                        :: sla_res_pkt.msgtype = sla_res;
                       :: sla_res_pkt.msgtype = unknown;
                       fi;
                        sla_res_pkt.auth_id = host_id;
                        if
                        :: sla_res_pkt.result = 1;
                        to_svcmgr!sla_res_pkt;
                        :: sla_res_pkt.result = 1;
                        to_svcmgr!sla_res_pkt;
```

```
                        :: sla_res_pkt.result = 1;
                        to_svcmgr!sla_res_pkt;
                        :: sla_res_pkt.result = 0;
                        to_svcmgr!sla_res_pkt;
                        fi;
                  };
                  :: else ->
                  atomic {
                        if
                      :: sla_res_pkt.msgtype = sla_res;
                      :: sla_res_pkt.msgtype = unknown;
                      fi;
                      sla_res_pkt.auth_id = host_id;
                      sla_res_pkt.result = 0;
                      to_svcmgr!sla_res_pkt;
                  };
                  fi;
            :: else ->
                  unidentified_msg = 1;
            fi;
      od;
}

proctype svc_manager (byte host_id)
{
      svc_install_pkt svc_pkt;
      authenticate_pkt auth_pkt;
      sla_pkt sla_req_pkt, sla_res_pkt;
      slp_pkt slp_req_pkt, slp_res_pkt;
      mon_pkt mon_req_pkt, mon_res_pkt;
      chan from_client = client_svcmgr[1];
      chan to_client = client_svcmgr[0];
      chan to_svcinstaller = svcmgr_svcinstaller;
      chan from_slamgr = sla_svcmgr[0];
      chan from_slpmgr = slp_svcmgr[0];
      chan from_mon = mon_svcmgr[0];
      chan to_slamgr = sla_svcmgr[1];
      chan to_slpmgr = slp_svcmgr[1];
      chan to_mon = mon_svcmgr[1];

      do
      :: from_client?svc_pkt;
            atomic {
                  service_unavailable = 0;
                  sla_expired = 0;
                  authentication_failure = 0
            };
            if
            :: svc_pkt.msgtype == svc_install_req ->
            atomic {
                  if
                  :: auth_pkt.msgtype = authenticate_msg_req;
                  :: auth_pkt.msgtype = unknown;
                  fi;
```

```
        auth_pkt.auth_id = host_id;
        auth_pkt.rcd_auth = svc_pkt.auth_id;
        auth_pkt.result = 0;
        authmgr_chan[0]!auth_pkt;
        authmgr_chan[1]?auth_pkt;
};
        if
        :: auth_pkt.result == 1 ->
        atomic {
              if
              :: sla_req_pkt.msgtype = sla_req;
              :: sla_req_pkt.msgtype = unknown;
              fi;
              sla_req_pkt.svc_id = svc_pkt.svc_id;
              sla_req_pkt.auth_id = host_id;
              sla_req_pkt.result = 0;
              to_slamgr!sla_req_pkt;
              from_slamgr?sla_res_pkt;
        };
        if
        :: sla_res_pkt.result == 1 ->
        atomic {
              if
              :: slp_req_pkt.msgtype = slp_req;
              :: slp_req_pkt.msgtype = unknown;
              fi;
              slp_req_pkt.svc_id = svc_pkt.svc_id;
              slp_req_pkt.auth_id = host_id;
              slp_req_pkt.result = 0;
              slp_req_pkt.codesr = 0;
              to_slpmgr!slp_req_pkt;
              from_slpmgr?slp_res_pkt;
        };
        if
        :: slp_res_pkt.result == 1 ->
              atomic {
              if
              :: svc_pkt.msgtype = svc_install_res;
              :: svc_pkt.msgtype = unknown;
              fi;
              svc_pkt.auth_id = host_id;
              svc_pkt.result = 1;
              svc_pkt.codesr = slp_res_pkt.codesr;
              to_client!svc_pkt;
              if
              :: svc_pkt.msgtype = svc_install_req;
              :: svc_pkt.msgtype = unknown;
              fi;
              to_svcinstaller!svc_pkt;
              };
              :: else ->

              atomic {
              service_unavailable = 1;
```

```
                        goto install_failed;
                        };

                        fi;

                  :: else ->
                        atomic {
                              sla_expired = 1;
                              goto install_failed
                        };
                        fi;
                  :: else ->
                  atomic {
                        authentication_failure = 1;
                        goto install_failed
                  };
                  fi;

            :: else ->
            if
            :: svc_pkt.msgtype == svc_uninstall_req ->
            atomic {
            if
            :: auth_pkt.msgtype = authenticate_msg_req;
            :: auth_pkt.msgtype = unknown;
            fi;
            auth_pkt.auth_id = host_id;
            auth_pkt.rcd_auth = svc_pkt.auth_id;
            auth_pkt.result = 0;
            authmgr_chan[0]!auth_pkt -> authmgr_chan[1]?auth_pkt;
      };
            if
            :: auth_pkt.result == 1 ->

            atomic {
            svc_stats[svc_pkt.svc_id].installed = 0;
            if
            :: svc_pkt.msgtype = svc_install_res;
            :: svc_pkt.msgtype = unknown;
            fi;
            svc_pkt.auth_id = host_id;
            svc_pkt.result = 1;
            to_client!svc_pkt;
            };
            :: else ->
                  atomic {
                              authentication_failure = 1;
                              goto install_failed;
                        };
            fi;
            fi;
      fi;
od;
```

```
install_failed:
            atomic {
                  if
                  :: svc_pkt.msgtype = svc_install_res;
                  :: svc_pkt.msgtype = unknown;
                  fi;
                  svc_pkt.auth_id = host_id;
                  svc_pkt.result = 0;
                  svc_pkt.codesr = slp_res_pkt.codesr;
                  to_client!svc_pkt
            };
}

proctype code_server (byte host_id)
{
    code_pkt codesr_pkt;
    authenticate_pkt auth_pkt;
    chan from_installer = codesr_installer[1];
    chan to_installer = codesr_installer[0];
    do
    :: from_installer?codesr_pkt;
        if
        :: codesr_pkt.msgtype == code_req;
        atomic {
            auth_pkt.msgtype = authenticate_msg_req;
            auth_pkt.auth_id = host_id;
            auth_pkt.rcd_auth = codesr_pkt.auth_id;
            auth_pkt.result = 0;
            authmgr_chan[0]!auth_pkt;
            authmgr_chan[1]?auth_pkt;
        };
            if
            :: auth_pkt.result == 1;
                  atomic {
                        if
                        :: codesr_pkt.msgtype = code_res;
                       :: codesr_pkt.msgtype = unknown;
                       fi;

                        codesr_pkt.auth_id = host_id;

                        if
                        :: codesr_pkt.result = 1;
                        codesr_pkt.hash_encrypt = 1;
                        to_installer!codesr_pkt;

                        :: codesr_pkt.result = 1;
                        codesr_pkt.hash_encrypt = 1;
                        to_installer!codesr_pkt;

                        :: codesr_pkt.result = 1;
                        codesr_pkt.hash_encrypt = 1;
                        to_installer!codesr_pkt;
```

```
                        :: codesr_pkt.result = 0;
                        codesr_pkt.hash_encrypt = 0;
                        to_installer!codesr_pkt;
                        fi;
                    };
                :: else ->
                atomic {
                    codesr_pkt.msgtype = code_res;
                    codesr_pkt.auth_id = host_id;
                    codesr_pkt.result = 0;
                    codesr_pkt.hash_encrypt = 0;
                    to_installer!codesr_pkt;
                };
                fi;
            :: else ->
                unidentified_msg = 1;
            fi;
        od;
}


proctype svc_installer (byte host_id)
{
      svc_install_pkt svc_pkt;
      authenticate_pkt auth_pkt;
      code_pkt codesr_pkt;
      chan from_svcmgr = svcmgr_svcinstaller;
      chan to_codesr = codesr_installer[1];
      chan from_codesr = codesr_installer[0];

      do
      :: svc_installed = 0;
        :: from_svcmgr?svc_pkt;
            if
            :: svc_pkt.msgtype == svc_install_req ->
            atomic {
            if
            :: auth_pkt.msgtype = authenticate_msg_req;
            :: auth_pkt.msgtype = unknown;
            fi;
            auth_pkt.auth_id = host_id;
            auth_pkt.rcd_auth = svc_pkt.auth_id;
            auth_pkt.result = 0;
            authmgr_chan[0]!auth_pkt;
            authmgr_chan[1]?auth_pkt;
            };
            if
            :: auth_pkt.result == 1;
            atomic {
                  if
                  :: codesr_pkt.msgtype = code_req;
                  :: codesr_pkt.msgtype = unknown;
                  fi;
                codesr_pkt.svc_id = svc_pkt.svc_id;
```

```
                codesr_pkt.result = 0;
                codesr_pkt.hash_encrypt = 0;
                  to_codesr!codesr_pkt;
                 from_codesr?codesr_pkt;
            };
                  if
                  :: (codesr_pkt.result == 1) &&
                  (codesr_pkt.hash_encrypt == 1);
install:
      atomic {

            svc_installed = 1;
            svc_stats[svc_pkt.svc_id].installed = 1;
            svc_stats[svc_pkt.svc_id].status_code = 1;
            run monitor (3, svc_pkt.svc_id);
      };
                  :: else ->
                  atomic {
                              code_error = 1;
                              svc_installed = 0;
                        };
                        fi;
                  :: else ->
                        authentication_failure = 1;
                  fi;
            :: else ->
                  unidentified_msg = 1;
            fi;
      od;
}

proctype auth_manager ()
{
      authenticate_pkt auth_pkt;
      chan auth_input = authmgr_chan[0];
      chan auth_output = authmgr_chan[1];

      do
      :: auth_input?auth_pkt;
            if
            :: auth_pkt.msgtype == authenticate_msg_req;
            if
            :: auth_pkt.msgtype = authenticate_msg_res;
                        auth_pkt.result = 1;
                        auth_output!auth_pkt;
            :: auth_pkt.msgtype = authenticate_msg_res;
                        auth_pkt.result = 1;
                        auth_output!auth_pkt;
            :: auth_pkt.msgtype = authenticate_msg_res;
                        auth_pkt.result = 1;
                        auth_output!auth_pkt;
            :: auth_pkt.msgtype = authenticat_msg_res;
                        auth_pkt.result = 0;
                        auth_output!auth_pkt;
```

```
            fi;
      :: else ->
            unidentified_msg = 1;
      fi;
      od;
}


never
{
      do
      :: failure -> break
      :: skip
      od;

      do
      :: svc_installed
      od;
}


init
{
      atomic {
            run svc_manager (2);
            run svc_installer (4);
            run auth_manager ();
            run sla_manager (5);
            run slp_manager (7);
            run code_server (6);
            run client (5, 2);
      };
}
```

Appendix B

Simulation Trace