Institut für Technische Informatik und Kommunikationsnetze
Computer Engineering and Networks Laboratory

Daniela Brauckhoff

Patterns for Service Deployment in Programmable Networks

Diploma Thesis DA-2003.03
November 2003 to August 2004

Supervisor: Matthias Bossardt
Co-Supervisor: Lukas Ruf
Professor: Bernhard Plattner

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Abstract

New network services such as firewalls, video scaling, and load balancing have been introduced in the last years. These services require packet processing functionality for filtering, forwarding, or transforming packets to be placed within the network. Programmable networks offer this functionality in a very flexible way. In addition to the packet forwarding mechanisms provided in conventional networks, programmable networks enable application-specific packet handling. Deploying new services in programmable networks, however, poses many challenges. Service logic has to be installed on multiple routers distributed all over the network. Moreover, routers that are suitable to host a service must be identified. Manual deployment of distributed services is a time consuming and error-prone process. Thus, there is a clear need for automation.

In this work we present a service deployment framework that automates the identification of suitable nodes, and initiates the installation of service logic on a set of selected nodes. We have implemented a novel pattern-based approach that allows tailoring of resource discovery strategies to service-specific needs. Moreover, network resources are saved by exclusively gathering information relevant to the deployment of the service in question. We define a generic format for specifying the information that is to be gathered from candidate nodes. To evaluate the utility and performance of our implementation, we conducted system tests with an example set of deployment patterns in a local testbed, and in the large-scale PlanetLab test environment. We deployed the ASD framework on all available PlanetLab nodes, and successfully used the implemented example patterns to identify a node that is suitable to deploy our example tunnel service. To optimize the performance of the ASD framework considering resource discovery latency and traffic, service-specific patterns must be developed that gather relevant information in a way that is suited to the network conditions, e.g. network size and connectivity. This implementation provides the fundamentals for developing and testing service-specific patterns.

Contents

Acknowledgments

I would like to thank Prof. Adam Wolisz, head of the Telecommunication Networks Group at the Technical University of Berlin, and Prof. Bernhard Plattner, head of the Computer Engineering and Networks Laboratory at the Swiss Federal Institute of Technology Zürich, for rendering this thesis possible.

I am also particularly grateful to Matthias Bossardt, my supervisor, who always took the time to listen to my concerns and questions. Our weekly discussions proved to be inspiring, constructive, and problem-solving.

Lukas Ruf, my co-supervisor, and the Service Group members helped me through the course of this thesis, providing the infrastructure and additional support.

I would also like to thank my former supervisor at the Technical University of Berlin, Morten Schläger, for teaching me the lesson that the presentation of your work is as important as the work itself.

Chapter 1

Introduction

1.1 Service Deployment Challenges

In the last years numerous distributed network services such as firewalls, video scaling, and load balancing have been proposed by the research community. These services require the installation of service logic on multiple routers at specific locations, for instance on bottleneck links, within the network. In addition to these topological requirements, services require certain resources such as bandwidth, CPU power, or memory space to be available on routers that host the services. Lastly, routers must provide the functionality to install and run the service logic. The complexity of deployment for these services challenges conventional service deployment mechanisms. Manual deployment of distributed services is time consuming and error-prone. The only feasible solution is to automate the service deployment process.

Service deployment, manual or automatic, involves two successive phases, the network-level and the node-level deployment. At the network level, nodes that are suitable to deploy a particular service have to be identified, then a subset of all suitable nodes has to be selected, and finally the node-level deployment has to be initiated on the selected nodes. Node-level deployment includes the installation of service logic on the selected nodes, and service-specific configuration of these nodes.

This thesis concentrates on network-level functionality. To identify a set of suitable nodes, node status data for all candidate nodes has to be evaluated. Most of the existing approaches to network-level service deployment focus on continuous distribution of node status data, which is evaluated at deployment time. However, for networks that potentially support a broad range of services, continuous distribution of all relevant node status data is inefficient or even impossible. Moreover, for many services the number of nodes to be queried is restricted at deployment time. An alternative approach more suited for the given scenario is on-demand service deployment. On-demand refers to the fact that all service deployment steps are executed at deployment time.

1.2 Claims

In general terms, we implement and validate an automatic, on-demand service deployment framework targeting the network-level. This framework provides an interface to node-level deployment protocols. Hence, it can be integrated with existing approaches to service deployment that target the node level.

Our approach proposes a secure, flexible, and modular deployment protocol that is based on exchangeable patterns. This enables tailoring of the resource discovery mechanisms to the topological requirements of a particular service. Moreover, its extensibility allows the protocol to be adapted to upcoming services.

We introduce a generic format for specifying service deployment requests. This generic format includes service requirements related to resource discovery, as well as installation and configuration specific requirements. We claim that it supports all network-level service deployment steps, and thus enables automation of the whole service deployment process.

1.3 Outline of the Thesis

In *Chapter 2*, related research activities are explored. We examine approaches that target network-level service deployment, as well as research activities addressing related areas such as resource discovery, overlay networks, and distributed management. Furthermore, we position our work in the context of related research activities.

Chapter 3 focuses on design decisions made during the development process. Proceeding on a target scenario, we define the design requirements for the Automated Service Deployment (ASD) framework. Furthermore, we analyze target services of the framework, and introduce the concept of patterns for service deployment. This chapter concludes with an overview of the architecture derived from the design requirements.

Our approach to service description is introduced in *Chapter 4*. We present the generic service description template as a general format for the definition of service deployment requirements. Moreover, we favor a layered approach to service description that allows for a separation of static and dynamic service deployment requirements. We illustrate our idea of service description with an example.

In *Chapter 5* the ASD protocol specification is presented. We address protocol-specific design requirements such as security, reliability, and modularity. The defined ASD message types are described in detail. Moreover, functionality provided by the user interface, and interfaces to other protocols such as the Overlay Topology Generation and the Node-level Deployment protocol are specified.

Chapter 6 describes our implementation of the ASD framework in detail. We present the implementation of the ASD management station and the ASD daemon that provide

the core ASD protocol functionality. Moreover, the example patterns, which we have implemented, are described.

In *Chapter 7* we present a comprehensive evaluation of our work. The utility of the ASD framework is evaluated considering its support for the usage and development of different service-specific patterns. Furthermore, functional and performance tests are described, and results are evaluated. The tests include the deployment of an example service, as well as resource discovery in a large-scale research network. Finally, the ASD framework is compared with related approaches.

The contributions of this thesis are summarized in *Chapter 8*. Moreover, our work is analyzed from a critical viewpoint to identify open problems and potential tasks for further improvement. The thesis is concluded with a discussion of future work.

Chapter 2

Related Work

In this chapter we present related research activities. Since our work targets automated service deployment at the network level, competing approaches to service deployment concentrating on the identification of nodes able to run service components will be analyzed. Other approaches concerned with identifying nodes matching certain requirements, not necessarily in a context of service deployment, are summarized in the section Resource Discovery. Further related areas that contribute to service deployment in a broader sense are Overlay Networks and Distributed Network Management. We conclude this chapter by positioning our work in the field of research activities.

2.1 Network-Level Service Deployment

The *Hierarchical Iterative Gather-Compute-Scatter (HIGCS) algorithm* [6] is an approach to automated network-level service deployment addressing programmable networks. An hierarchical overlay is constructed in order to aggregate information (e.g. node capabilities) collected from nodes. Aggregation results are distributed via signaling messages within the network. However, the optimization of these hierarchies for service deployment is not further specified. The main contribution of this approach is the introduction of service categories for deployment strategies. Service deployment for these categories was implemented in a simulation environment. Furthermore, XML is proposed as the specification language for service requirements, but the actual evaluation process is not described.

Our work owes much of its design philosophy to *Pattern-based Service Deployment* [2] which uses a service-specific deployment protocol that is constructed from reusable protocol building blocks, namely navigation patterns and aggregators. Whereas navigation patterns describe the flow of information between nodes, aggregators determine the information to be collected from nodes and the algorithms used to summarize this information. Information is gathered on-demand to support a broad range of services. Navigation patterns and aggregators tailored to certain service categories have been implemented and simulated. This thesis will propose a framework that enables the usage of different navigation patterns and aggregators. Moreover, the simulated patterns will be extended with a failure handling mechanism and serve as test patterns for the framework.

In [3] a service framework featuring *Self-configuring active services* is presented. It provides an active network control software (ANCS) that accepts demands from applications using a network programming interface. Discovery of available processing resources is in fact realized using the OSPF approach described in the next section. A signaling protocol is then used to set up the service by deploying code on the selected nodes. Active pipes are proposed as a network programming interface that allows the specification of communication and processing requirements. However, the interface distinguishs only modules that are to be installed exactly once within the network, and modules that are to be installed on all matching nodes. Moreover, installation conditions must be specified for each module separately, since node groups are not supported.

2.2 Resource Discovery

Resource Discovery is concerned with locating an appropriate subset of a system to host a service, computation, or experiment. Approaches described in this section do not provide service deployment functionality themselves, but could be used along with an independent service deployment tool.

SWORD [7] proposes a scalable resource discovery service especially targeting large-scale infrastructures. It supports an expressive query language that allows users to specify the ranges of resource quantities their application needs. When more nodes than needed are available, candidate nodes are filtered. Reporting nodes periodically send measurement reports for a predefined set of single-node attributes to distributed SWORD server nodes handling user queries. Additional attributes must be distributed to all reporting nodes before they can be used. After all relevant single-node attributes have been retrieved, inter-node measurements for the nodes meeting the single-node constraints are obtained. The SWORD server combines single- and inter-node measurement results. Finally, the optimizer maps the selected nodes to the node groups specified in the query. SWORD implements three different patterns for distributing query requests. Furthermore, those patterns are not aware of any service topologies. Contrary to our requirements, SWORD does not provide any security or privacy features.

In *Dissemination of Application Specific Information* [8] an approach to resource discovery using OSPF is described. The OSPF routing protocol was extended to enable external applications to distribute their own attributes using the OSPF protocol. Each node periodically sends information about its available resources to adjacent neighbors using the OSPF protocol. Received information is stored in the node's network information base. This approach is similar to the one previously described as it also uses distributed databases to store information about individual nodes. Hence, it shows the same drawback: Information about nodes is gathered continuously. Moreover, the proposed approach does not provide any flexibility regarding the distribution of information since it makes only use of the flooding-based dissemination protocol of OSPF. And in comparison with SWORD, the specification and distribution of requests (here between OSPF routers) is not as sophisticated.

2.3 Further Related Concepts

Further related concepts can be found in the area of distributed network management, as well as in connection with overlay networks. Research activities that are not directly concerned with service deployment or resource discovery are presented in this section.

Sophia [11] proposes a distributed system, a so-called shared Information Plane, for managing and controlling complex, large-scale networks. It uses a declarative logic language (Prolog) to express statements about the actual and desired system state. Sensors running on nodes throughout the network collect data about aspects of the system. Statements are evaluated in a distributed programming environment, and a set of actuators performs actions on evaluation results. The core implementation can be extended with loadable user modules adding new functionality. However, Sophia addresses network monitoring and management and does not support a comprehensive resource query language.

The *X-Bone* [9] is a system for rapid, automated deployment and management of overlay networks. Furthermore, a framework for application deployment within virtual networks using the X-Bone was presented in [10]. While traditionally virtual networks were used to support network-level services, such as multicast and QoS, the developed framework enables the deployment of arbitrary applications. However, a node selection or resource discovery process is not supported by the current implementation.

2.4 Positioning of our Work

This thesis proposes a network-level service deployment framework. We specifically address automated service deployment in overlay networks. Our approach can be characterized as distributed, out-of-band, and on-demand. One of the main contributions is the introduction of a service description language which allows the specification of service-specific deployment requests.

Based on previous work that explored service categories, and navigation patterns and aggregators for service deployment, we implement an extensible ASD framework architecture supporting service-specific deployment patterns. Extensibility is addressed by providing (i) clearly defined interfaces between the patterns and the ASD framework, and (ii) a mechanism for loading locally unavailable patterns from a pattern server at run-time. Furthermore, example patterns will be implemented for testing purposes.

The ASD protocol provides end-to-end reliability and security using TCP on the transport layer combined with TLS authentication and encryption. Moreover, our approach will provide a node-level interface that enables integration with existing node-level deployment tools such as Chameleon [4].

Resulting from the analysis of related research efforts, we reason that no other approach features all of the described characteristics. In particular, this thesis appears to be the

first attempt to introduce service description on the network-level.

Chapter 3

Design Considerations

This chapter focuses on the design decisions made during the development process. We define a target scenario that helps us gathering design requirements for the service deployment framework. Furthermore, we analyze target services for the framework and their service requirements. This will influence the design of the service descriptor. Finally, we investigate into navigation and aggregation patterns, and capability functions, since the ASD framework must support all kinds of patterns.

3.1 Scenario and Design Requirements

In our target scenario a network manager wants to deploy a virtual private network (VPN) in three geographical separated networks. Deployment of a VPN involves the following steps: (i) specification of node requirements for hosting the VPN, (ii) identification of nodes matching these requirements in each network, and (iii) configuration of the selected nodes to set up the VPN. The goal of this thesis is to provide a framework that automates the process of service deployment as much as possible.

The specification of node requirements must be done by the network manager. But in order to automate the remaining deployment steps, the ASD framework must provide a *service description language* that defines the form in which node requirements are to be specified. Also, assuming one of the three networks hosts much more nodes than the others, the network manager might want to select a node with higher CPU power to deploy the VPN in this network. Hence, the service description must support *node groups* allowing the specification of multiple requirement sets.

In order to identify a set of suitable nodes, we need to obtain information about the requested characteristics from (eventually) all nodes within the three networks. Our approach favors *on-demand* collection of node status data. We argue that distributing node status data continously is less efficient for the given scenario for three main reasons: First, services such as a VPN are not to be deployed that frequently. Thus, the frequency of network state changes is likely to be higher than the rate of deployment requests. Second, the ASD framework is tailored to support a broad range of services resulting in a huge amount of node status data

to be distributed. Third, the set of nodes from which node status data is to be collected can be restricted at deployment time. For instance, to deploy the VPN service only nodes within one of the three target networks are queried. Moreover, we prefer *distributed evaluation* to a central approach in order to avoid the concentration of status data messages and computations on a single node.

Resource discovery tasks to be performed by the ASD protocol can be inferred from the design requirements. For the given scenario these are (i) distribution of service requests in the three target networks, (ii) evaluation of the specified VPN node capabilites, and (iii) aggregation of the gathered information. We reason that *modularity* achieved by separating these tasks from each other guarantees flexibility and extensibility of the framework. Using different navigation patterns the distribution of service requests can be optimized for the service to be deployed, and navigation patterns can be tailored to user demands. For instance one pattern is used to find a suitable node to deploy the VPN service, and a different pattern is used to find the best node out of all suitable nodes. Also, in each case a different aggregation pattern is used. Capability functions specify how node requirements are mapped to the actual capabilities. Hence, the use of different capability functions enables an infinite set of node requirements.

Since forwarding mechanisms are used to distribute service requests within the target networks we reason that the propagation of ASD messages must be *reliable*. Moreover, we want to address how much of its control over the network the network manager is willing to pass to the ASD framework. We argue that the network manager wants to keep control over the final selection of nodes. Thus, the framework must present resource discovery results to the network manager before node-level deployment is initiated. We believe *security* is a key factor for the acceptance of automated service deployment.

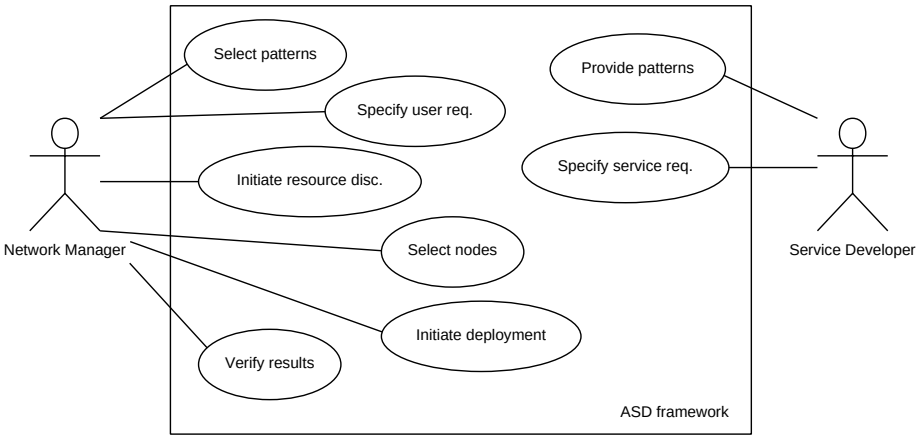

Figure 3.1: Use case diagram modeling the context of the ASD framework

The use case diagram in Figure 3.1 illustrates the interaction of the network manager with

the ASD framework. It also introduces the service developer entity which provides *description templates*, deployment patterns, and the service components for one or more services. Description templates specify the requirement attributes necessary to deploy a service. The VPN service descriptor for instance would specify a target network attribute. Static service requirements can be included in the template as well. This further facilitates the interaction of the network manager with the ASD framework.

Requirement	Specification
Distributed evaluation	general
On-demand distribution	general
Extensibility	general
Service description language	description-specific
Descriptor templates	description-specific
Node groups	description-specific
Modularity	protocol-specific
Security	protocol-specific
Reliability	protocol-specific

Table 3.1: Design requirements for the ASD framework

Finally, requirements to be met by the design of the service deployment framework are summarized in Table 3.1. They define the characteristics of the final system, and serve as a guideline for the conceptual work.

3.2 Target Service Overview

The ASD framework is supposed to support a broad range of services. In this section we analyze target services in order to identify their service requirements. One group of target services are overlay networks, including previously discussed virtual private networks. Other target services provide more packet-based services, such as filtering, forwarding, and transformation.

3.2.1 Overlay Networks

Tunnels are the most basic form of overlay networks. They provide a virtual point-to-point connection between two nodes in different networks. To set up a tunnel automatically, the network manager must assign a name to identify the tunnel, specify a tunnel mode (e.g. IPv6-in-IPv4), and its endpoints. Additionally, QoS parameters, such as maximum number of hops between tunnel endpoints, and a minimum bandwidth for the tunnel may be defined. According to the given parameters, tunnel endpoints will be selected to deploy the tunnel service upon them.

Virtual Private Networks provide secure connections between multiple networks by

combining tunneling mechanisms and IP security. The network manager must specify the security mode, a name for the VPN, and the members participating in the VPN. Furthermore, the service user may determine QoS constraints, such as CPU power and memory for security functions on VPN nodes, or bandwidth requirements. This service has to be deployed on each endpoint of the VPN.

In *peer-to-peer networks* a different approach is used. Autonomous peers located at the edge of the Internet form application-specific virtual networks. Typical applications of p2p networks are file sharing (i.e. Napster, Gnutella), and distributed storage or computation. In [12] the authors argue that the lack of centralized control in p2p networks results in a tremendous amount of signaling traffic between peers. Furthermore, they describe a service providing effective management for p2p networks through the use of active networking technology at the application layer. Hence, management of p2p networks is also a potential target service for automated service deployment.

3.2.2 Packet Filtering

Active Reliable Multicast (ARM) is a service that combines packet filtering with forwarding mechanisms. In the upstream direction, negative acknowledgments (NACKs) from multiple users are filtered to suppress NACK implosion. In the downstream direction, routers are able to perform retransmissions of previously cached multicast data [13]. Topology-wise this service is best to be deployed on fork nodes of the multicast tree.

Concast is a programmable network layer service, providing information fusion in the upstream direction, e.g. to collect feedback from multicast applications [14]. Concast extends the filtering functionality, as provided in ARM to control NACK implosion, by allowing the receiver to specify the mapping from sent datagrams to delivered datagrams. Hence, topological requirements are identical for both services.

Active Monitoring is another similar service. By filtering the collected data before sending it to the central management station, transmission overhead is reduced. Nodes running this service must be able to collect data and filter out irrelevant information, before sending a compact summary to the monitoring station. Again, the same topological requirements as for ARM apply.

Firewalls provide security for networks, filtering data packets and streams that leave and enter the network. For large networks, several firewalls must be set up and configured. Firewalls are best to be deployed at the network border. Required memory for caching data streams is proportional to the traffic handled by the firewall. Hence, the network manager might want to select nodes providing a maximum of free memory to deploy the firewall.

Application Security Gateways provide another type of security service. They handle all incoming and outgoing traffic for a single application (e.g. SSH), providing security for its users. This service has to be deployed on exactly one secure node within the network.

Moreover, this node must provide sufficient bandwidth to handle all user traffic.

Video Scaling services improve the quality of video distribution by scaling the data rate of video streams as congestion occurs in the network. Video Scaling is realized by either decreasing the image resolution or modifying the frame rate. In [15] an approach to video scaling based on WaveVideo encoding is presented. Each indivdual packet of the video stream is labeled with a tag. This allows for selectively dropping packets with high-frequency coefficients. Thus, the image quality can be reduced by eliminating high-frequency parts of the video stream. Smart locations to deploy the WaveVideo Scaling service are nodes where congestion is likely to occur, for instance bottleneck links in a path or multicast tree. In [16] a service similar to WaveVideo Scaling is proposed. Under congestion an Intelligent Discard mechanism is used to reduce the bandwidth in a way tailored to application requirements. More specifically, an active congestion mechanism for handling MPEG is presented.

3.2.3 Packet Forwarding

Web caching services reduce network traffic through caching of web contents on intermediate network nodes. A typical application where web caching can be advantageous is the provision of stock quotes. Client requests are intercepted at intermediate nodes to check if the desired quotes are available in the local cache. Otherwise, the request is forwarded to the server. As this service involves caching, available memory becomes a service requirement.

Load balancing services distribute accesses from clients to popular servers among geographically dispersed replication servers. In [18] an approach to load balancing using Active Anycast is presented. Clients send their requests to a group of servers, identified by anycast addresses. A load balancing node receiving this request, selects a suitable server from the viewpoint of load balancing and changes the anycast address to the unicast of the selected server. Load information can be piggybacked with the data exchanged between clients and servers and collected when passing load balancing nodes.

3.2.4 Payload Data Transformation

Real-time Multimedia Transcoding [19] is an application layer service allowing different systems to access the same data, which is converted into the appropriate format within the network. The best location to deploy this service depends on network status information, such as bandwidth, delay, and free resources.

3.3 Service Deployment Patterns

Modularity of the deployment protocol is one of our design requirements. Therefore, the network-level service deployment protocol is constructed from reusable components, namely navigation patterns, aggregation patterns, and capability functions. As our objective is to support the exchangeability of these patterns, we have to explore similarities and differences between the various patterns of each group.

3.3.1 General Remarks

The concept of navigation patterns and aggregators was first introduced in [20]. Patterns are distributed programs running in parallel on nodes within the network. Navigation patterns describe the flow of control among these nodes. Whereas operations, to be performed on visited network nodes, are specified by aggregators. The use of navigation patterns and aggregators for service deployment was studied in previous work [2]. In this thesis a further separation of the aggregator functionality is introduced. In our approach the aggregation pattern is responsible for aggregating information gathered in one node. Mapping of node requirements against actual node capabilities, however, is taken over by the capability function.

3.3.2 Navigation Patterns

As we stated earlier, one reason for making patterns exchangeable is the usage of service-specific navigation patterns. To classify services according to their topology requirements has the advantage that services of the same group can share navigation patterns. We extend the basic topology categories defined in [6] with a new category, named tree-based topology.

Path-based topologies require deployment of service functionality on nodes on a path between source and destination. One subgroup are continuous path-based topologies. They require the same functionality to be deployed on each node on the path. This topology is for instance used to deploy the unicast WaveVideo service. In contrast, sparse path-based topologies require certain (possibly different) functionality components to be present on a set of nodes on the path. An example for this category is Transcoding between end users, where service functionality is deployed on one node on the path.
Fence-based topologies require service deployment on nodes along a path orthogonal to the data path. This category applies to services acting on traffic crossing network borders such as firewalls.
Node-based topologies require functionality to be deployed on single nodes. Services that fall into this category are Application Security Gateways, overlay networks, and Web Caching.
Tree-based topologies, finally, require service components to be deployed on nodes forming a tree structure. Services belonging to this category are Active Reliable Multicast, Concast and Monitoring.

In general, most of the previously identified topology categories are best served with a navigation pattern that first explores the network to find suitable nodes, then summarizes the collected information at a central point, and finally informs the selected nodes to carry out the node-level deployment. Only continuous path-based topologies are better served with a pattern applying a sequential summarization and selection of nodes as it follows the path from the source to the destination. In order to identify the input parameters for navigation patterns, we examine two example patterns as described in [2].

The *scout* navigation pattern visits a destination node, and returns back to its origin after some action has been taken on the destination node. An *mScoutBegin* message containing

the address of the destination node is sent to trigger the scout pattern. The node that receives this message selects a node on the path to the destination based on its IP-forwarding table, and sends an *mScout* message to the selected node. This procedure is repeated until an *mScout* reaches the destination node. After the corresponding aggregation function has been executed on the destination node, an *mScoutBack* message is sent back towards the originating node.

The *constrained-remoteEcho* navigation pattern comprises three phases. In the expansion phase the network is flooded with *mExplorer* messages. Each node that receives such a message forwards it to all neighbor nodes, except the one the *mExplorer* message was received from. The contraction phase starts as a subsequent *mExplorer* message is received by a node. An *mEcho* message is sent to the upstream node. After the upstream node has received *mEcho* messages from all nodes it has sent an *mExplorer* message to, it generates itself an *mEcho* message. This continues until the originating node is reached. In the last phase the *scout* pattern is used to contact individual nodes. Furthermore, the sending of *mExplorer* messages can be constrained to specific network domains or multicast groups. These domains can be explored remotely by using part of the *scout* navigation pattern.

Common to both example navigation patterns is that they need a specific target. This can be an individual node as for the *scout* pattern, or a group of nodes forming a domain or multicast tree as for the *constrained-remoteEcho* pattern. Another parameter for the latter pattern might be a timeout value, if we extend the pattern to handle failures during navigation pattern operation.

3.3.3 Aggregation Patterns

Aggregation patterns are executed on visited nodes. They define functions for aggregating information obtained on the local node with information gathered at other visited nodes. Which aggregation function is executed in a particular case depends on the state of the navigation pattern. Hence, the aggregation pattern must always match the navigation pattern that it is used with. As an example we describe a modified version of the *findBest-Inform* aggregation pattern proposed in [2].

The *findBest-Inform* aggregation pattern matches the previously described *constrained-remoteEcho* navigation pattern. In general, it returns exactly one node per node group that matches best the node requirements. On arrival of an *mExplorer* or *mEcho* message the aggregation function corresponding to the actual state of the navigation pattern is executed. The *onEcho* aggregation function triggers the associated capability function to determine whether the visited node fulfills the node requirements. During the contraction phase of the *constrained-remoteEcho* pattern, *mEcho* messages contain information about the node that matches best the node requirements. The *onEcho* aggregation function compares this node‘s score with the one of the local node, and remembers the one with the higher score for future evaluation. Furthermore, the *onTimeout* aggregation function is executed when a timeout has occurred.

The described aggregation pattern only returns one node per node group. But some services might need to be deployed on multiple nodes meeting the same requirements. In this case another aggregation pattern must be used. Hence, we reason that for each node group the number of required nodes must be given as parameter to the aggregation pattern.

3.3.4 Capability Functions

We introduce capability functions to separate the aggregation of collected information from the evaluation of node capabilities. Capability functions are, as aggregation patterns, executed on visited nodes. In fact, the result of the capability function is used as input for the aggregation pattern. Hence, it is important to define a clear interface between both. In contrast to aggregation patterns, capability functions are unaware of the navigation pattern's actual state, and must not match the navigation they are used with.

Since capability functions define rules for the mapping node requirements to actual node capabilities, parameters given to the capability function are identical with the node requirements. For instance, a capability function evaluating the CPU load of nodes could take a minimum value, or even a range of CPU load values as input parameter.

3.4 Architecture Overview

Finally, we present an overview of the architecture derived from the gathered requirements. Figure 3.2 illustrates the identified system components.

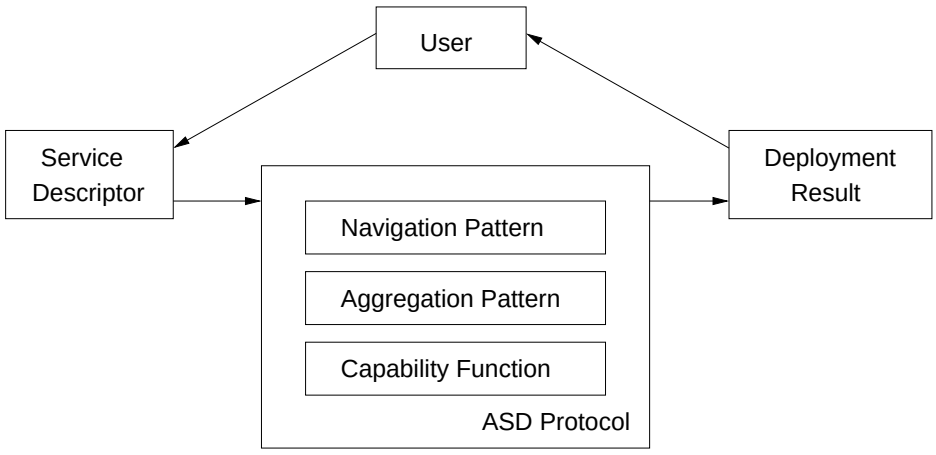

Figure 3.2: Overview of the ASD framework architecture

The service descriptor, specifying the service requirements, is to be defined by the user e.g. a network manager. The ASD protocol, constructed from navigation patterns, aggregation patterns, and capability functions, distributes the service descriptor within the network, and

gathers evaluation and deployment results. The obtained results are then presented to the user for verification.

Chapter 4

Service Description

In this chapter our approach to service description is introduced. We describe the layered structure of the service descriptor in detail, and illustrate its use with an example. Furthermore, XML is introduced as the service description language.

4.1 Conceptual Overview

The definition of a general service description is one of the main contributions of this thesis. Our approach to service description favors a layered design employing templates. The three description layers are depicted in Figure 4.1.

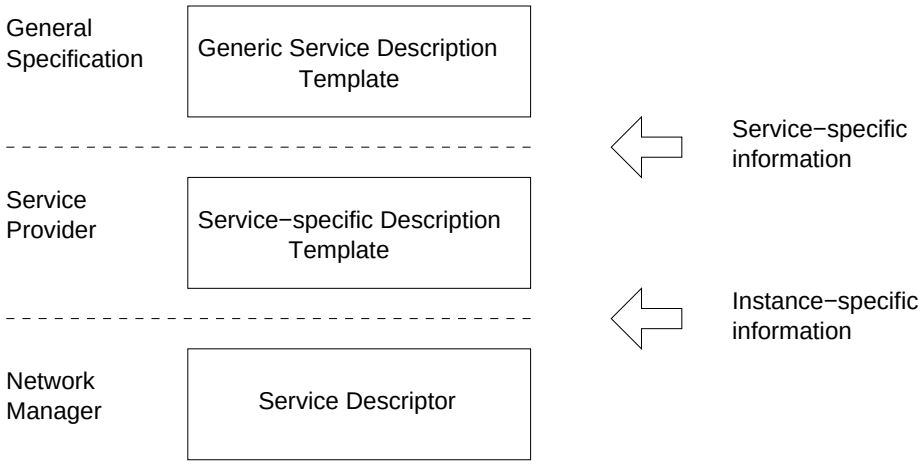

Figure 4.1: Layered design of service description

The generic service description template specifies a general form for describing service requirements. Service providers add service-specific information such as the required patterns to the generic service description template. The resulting service-specific description templates are provided to network managers. At deployment time the network manager adds the

instance-specific information such as target network addresses to the service-specific description template. The resulting service descriptor is then processed by the ASD framework.

4.2 Generic Service Description Template

The definition of a general service description is very challenging, since it must be guaranteed that it serves all existing and future services. We address these needs with a high degree of extensibility. The structure of the proposed generic service description template is illustrated in Figure 4.2.

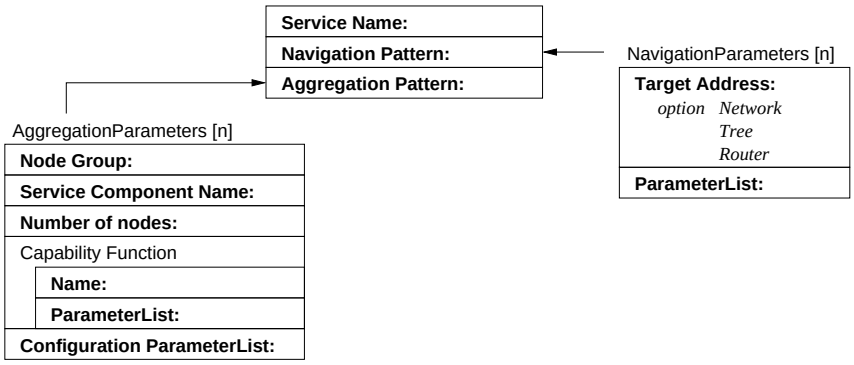

Figure 4.2: Structure of the generic service description template

According to our analysis of target services and patterns, we identify four main service requirement categories: (i) general requirements such as pattern and service names, (ii) topological requirements such as target addresses, (iii) aggregation requirements such as the required number of nodes, and (iv) resource requirements such as the maximum CPU load. Moreover, to meet the design requirements the service description must support node groups.

Topological requirements are addressed by navigation patterns, since they control which nodes are visited during discovery. Thus, topological requirements represent the input parameters for navigation patterns. One navigation parameter identified in the target scenario is the *target address*. The analysis of target services showed that the node search can be restricted to parts of an overlay network. This can be either a subnetwork, a tree structure, or a single router. Furthermore, we identified a need for specifying multiple target addresses as for the VPN service. To address other possible navigation parameters such as the timeout value for the constrained-remoteEcho navigation pattern, we included a *parameter list* which can hold a restricted number key and value pairs.

We decided to arrange aggregation and resource requirements in node groups, since requirements such as the number of nodes or a CPU load maximum are indeed always specific to a certain node group. The service description supports multiple node groups. Thus, a *node*

group parameter is needed to distinguish between multiple groups. This parameter is actually redundant information. An alternative would be to compute a hash over the parameters of each node type. Since this method would require the computation of a hash every time a node is selected, we prefer sending the node group parameter in the service descriptor. From a service deployment viewpoint, node groups can be used to specify node requirements tailored to the service functionality that is to be installed on nodes of that group. Hence, the *service component name* and the *number of nodes* can be specified in the service description for each node group. Moreover, a *configuration parameter list* can hold configuration information for the selected nodes of each node group.

Resource requirements represent the input parameters for the capability function. To support a high degree of flexibility, the service description provides the possibility to specify not only different node requirements but also different evaluation rules. Therefore, the *capability function name* and another extensible *parameter list* containing the resource requirements can be specified for each node group.

We argue that the presented service description covers all network-level service deployment steps as depicted in Figure 4.3. Part of this figure was presented in [6], and the distinction between network and node level was added in [2].

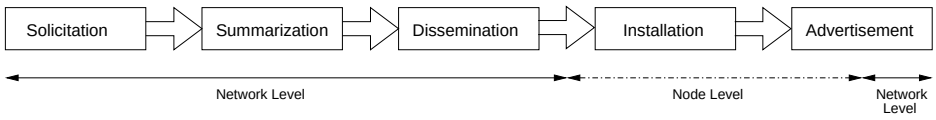

Figure 4.3: Service Deployment Steps

In the solicitation step of service deployment a set of suitable nodes is identified. This means that all nodes that are potential candidates for deploying the service must be visited and queried. The following parameters are applied in the solicitation step: The *navigation pattern name* specifies the navigation pattern that is used for distributing the service request to all candidate nodes. The *target address* can constrain the number of candidate nodes visited. The *capability function name* specifies the evaluation rules for a certain node group, and the extensible *parameter list* specifies desired node capabilities. Furthermore, the *node group* parameter allows the definition of different node requirement sets. For evaluation the suitability of a node, a test deployment for the specified *service component* might be necessary.

Gathered information is aggregated in the summarization step of service deployment. The result is a set of nodes suitable to deploy a certain service. Parameters applying to the summarization step are the *aggregation pattern name*, and the *number of nodes* required for each node type. The aggregation pattern summarizes the gathered information according to the given node groups and numbers.

In the dissemination step of service deployment, selected nodes are requested to install the service components. Hence, the *service component name* is needed in the dissemination step. Moreover, some services might require additional configuration of nodes. Configuration information is provided in the form of an extensible *parameter list.*

Finally, the advertisement step informs whether the service deployment was successful. The service descriptor, however, does not participate in this step.

4.3 Service-specifi c Descriptor Template

In order to define service-specific description templates, the service provider adds service specific-information to the generic template. This service-specific information usually includes the service name, and the patterns to be used. Moreover, navigation and aggregation parameter attributes are tailored to the requirements of the pattern in use. We want to illustrate this refinement considering as example a unidirectional tunnel overlay service.

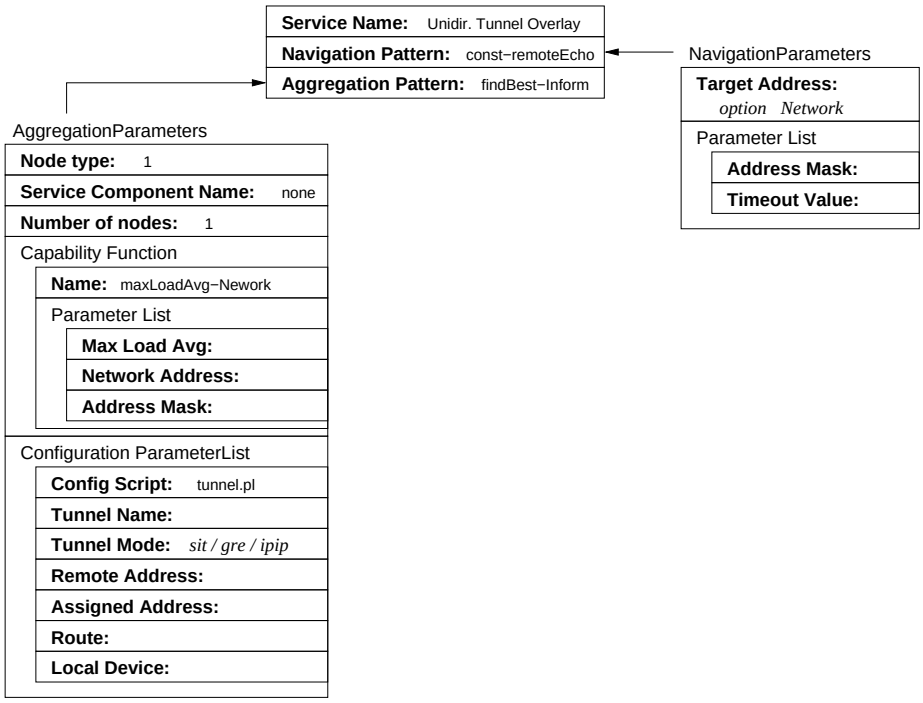

Figure 4.4: Descriptor template for a unidirectional tunnel overlay service

In Figure 4.4 the description template for a unidirectional tunnel service is depicted. To describe this particular service, the service and the pattern names are included in the template. We decided to use the example patterns described in 3.3, since they target overlay services such as tunnels. The *maxLoadAvg-Network* capability function is used to find a node with a

certain maximum load average in a given network.

Furthermore, navigation parameters are limited to one column, since the service is only to be deployed within one network. The target address is further limited to network addresses, as we are not interested to search a multicast tree or a single router. Moreover, the parameter list is tailored to the *constrained-remoteEcho* pattern which needs a timeout value and an address mask as input.

Aggregation parameters are also limited to one column, or node group. This is because the unidirectional tunnel service is only to be deployed on one node. Hence, for a bidirectional tunnel service either two nodes of the same node group (with the same requirements) or two different node groups and one node per node group could be specified. Obviously, for the unidirectional tunnel service the required number of nodes is one. Furthermore, no service component name is specified in the description template, since no node-level functionality needs to be deployed on the tunnel endpoint. Instead, the once selected node has to be configured to set up the tunnel overlay. Hence, configuration information necessary to set up the tunnel such as tunnel name and mode is specified in the corresponding parameter list.

The capability function parameter list is tailored to the *maxLoadAvg-Network* capability function. The network manager is requested to specify the network where the tunnel endpoint is to be deployed and the maximum CPU load value for the selected node. One could claim that this information is already given with the target address. But notice that in contrast to the target address the capability network parameter is node group specific.

4.4 Service Description Language

We decided to use XML as the service description language, since it provides a structured, easily extensible, and platform-independent way for specifying information. Moreover, the XML Schema Definition Language can be used to constrain the contents of XML documents. Thus, the layered structure of service description as presented in Figure 4.1 can be realized using XML documents and schemes.

In general, templates will be implemented as XML schemes and descriptors as XML documents. Since the generic service description template only specifies attributes but no actual values for these attributes, it can be represented by an XML schema. The service-specific description template, however, specifies constraints on the attributes but also attribute values such as the service name. Thus, the service-specific description template is represented by an XML schema and a corresponding XML document. This service-specific XML document is completed with instance-specific information such as a particular target address by the network manager at deployment time.

Chapter 5

Automated Service Deployment Protocol

In this chapter the ASD protocol specification is described. We present the ASD protocol stack that is designed to meet the previously defined protocol-specific requirements. Furthermore, the ASD protocol messages are described in detail. We define a general message format and introduce the different message types. Finally, interfaces provided by the ASD protocol are presented.

5.1 Conceptual Overview

From the given target scenario we extracted three protocol-specific requirements: (i) reliability, (ii) security, and (iii) modularity. The easiest way to address reliability is to rely on TCP which offers end-to-end reliability to the application layer. Security is also addressed by existing protocols such as IPSec [22] and TLS [21]. We favor the TLS protocol, since it provides security tailored to the transport layer including end-to-end privacy and data integrity between applications. The resulting protocol stack is depicted in Figure 5.1.

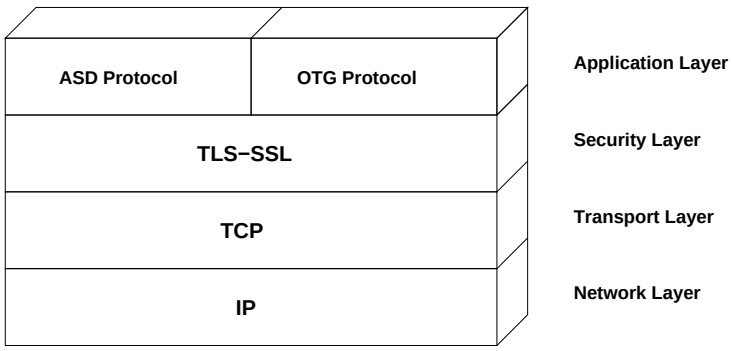

Figure 5.1: Overview of the protocol stack

Moreover, we specified that the ASD framework targets overlay networks. Thus, the generation of overlay network topologies must be supported by the framework. Overlay topologies

can be generated based on different criteria such as bandwidth or delay measurements. In this case, two nodes would become direct neighbors in the overlay if the delay on the link connecting the nodes is smaller than the specified maximum. We decided to separate this functionality from the actual ASD protocol, since it is not an integral part of network-level service deployment. Overlay related functionality such as the generation of overlay topologies, and determination of direct neighbors in the generated topology, is provided by the Overlay Topology Generation (OTG) protocol.

Modularity is addressed by the previously described distribution of deployment functionality to navigation patterns, aggregation patterns, and capability functions. Since patterns are exchangeable, the ASD protocol must provide a unified interface to all patterns. However, patterns are only used in the solicitation and summarization step of service deployment, or in other words only for resource discovery. The ASD protocol does not use patterns in the dissemination step, since service installation is a lot simpler than resource discovery. In order to install a service the selected nodes are requested to install the service components and configured as specified in the service descriptor. Furthermore, our target analysis showed that usually only a few nodes are required to deploy a service.

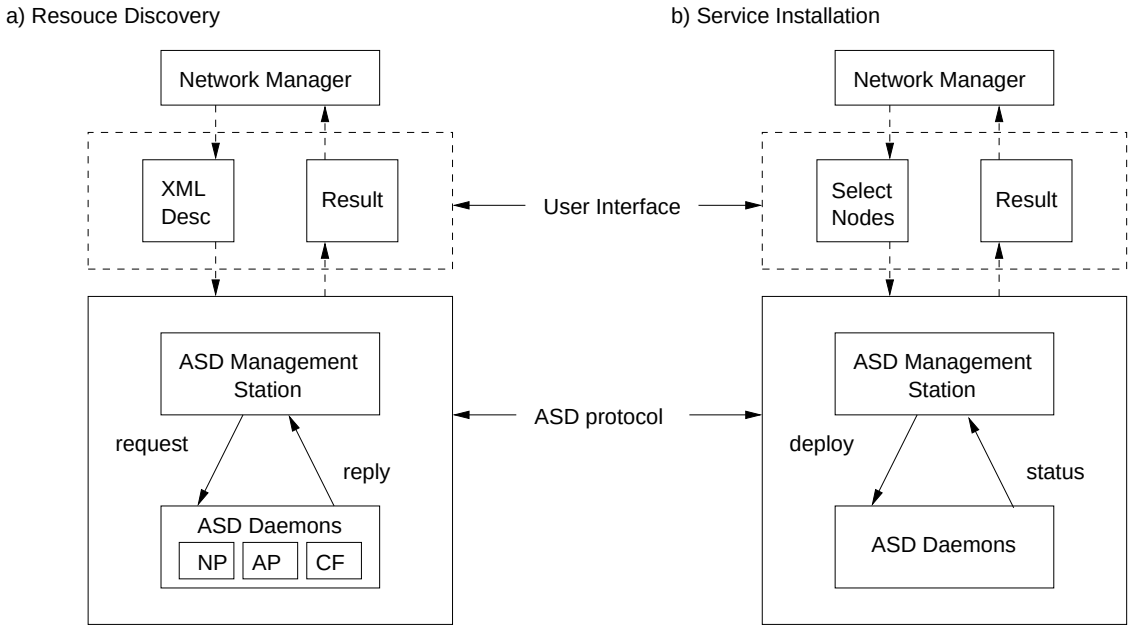

Figure 5.2: ASD protocol design overview

In Figure 5.2 the ASD protocol design is illustrated. We defined two protocol entities: the ASD management station that provides the user interface, and distributed ASD daemons. Protocol operation for the resource discovery step is depicted in Fig. 5.2a. The network manager initiates a deployment operation via the user interface on the management station specifying a start node for resource discovery. The management station sends a discovery

request to the ASD daemon running on the given start node. Resource discovery is governed by distributed patterns (NP, AP, CF) running on the ASD daemons. Finally, a reply is send back to the management station and the discovery result is displayed via the user interface. Hence, most of the protocol logic for the resource discovery step is provided by the patterns. Protocol operation for the service installation step is depicted in Fig. 5.2b. The network manager selects the nodes on which the service is to be deployed. The management station sends installation requests directly to all selected nodes. On reception of an installation request the ASD daemon running on the node carries out the installation and configuration and sends a reply back. Finally, the deployment result is displayed via the user interface.

5.2 ASD Protocol Messages

ASD protocol messages carry (i) information exchanged by the ASD management station and ASD daemons running on nodes within the managed network, and (ii) information exchanged between ASD daemons. Since different patterns can be used for resource discovery, ASD messages also serve as an abstraction for pattern-specific messages.

5.2.1 Message Format

Each ASD message is preceeded by a header with the byte structure given in Figure 5.3. The total length of the ASD message header is 6 bytes.

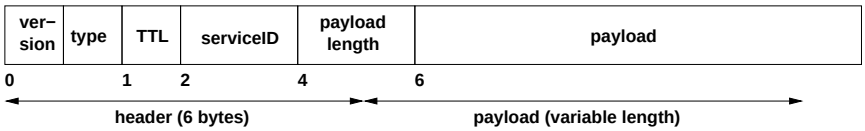

Figure 5.3: ASD protocol message format

The first byte specifies the *protocol version* and the *message type*. Both message fields are defined as 4-bit integers. In this thesis version 1 of the ASD protocol is described. Moreover, four different message types are defined and presented in the remainder of this section.

Field	Type	Description
version	4-bit unsigned integer	Current protocol version
type	4-bit unsigned integer	Message type
ttl	8-bit unsigned integer	Time to live
service ID	16-bit unsigned integer	Unique service ID
payload length	16-bit unsigned integer	Length of payload in bytes

Table 5.1: ASD message header fields

The second byte specifies a *ttl value*. This field is defined as an unsigned 8-bit integer. We included a maximum life time for ASD messages in the header as a security measure to avoid endless forwarding of messages in case a pattern fails. The third and fourth byte specify the *service ID*. This field is defined as an unsigned 16-bit integer. Since the specified service ID must be unique within the managed network, ASD messages are unambiguously assigned to service deployment operations. The fifth and sixth byte specify the *payload length*. This field is also defined as an unsigned 16-bit integer. Due to the variable length of the different message types, we explictly need to specify the number of bytes contained in the payload of ASD messages. In Table 5.1 the header field properties are summarized.

5.2.2 Message Types

We defined four different message types. Each message type applies to a certain service deployment step. *Request* messages (0x00) are used for solicitation of nodes. For summarization of the gathered information *reply* messages (0x01) are exchanged. *Deploy* (0x02) and *status* (0x03) messages are used for the dissemination of installation requests.

Request messages are exchanged (i) between the ASD management station and the the start node, and (ii) between the ASD daemons. The purpose of sending *request* messages is to query nodes whether they fulfill the service requirements. The encoding of payload fields in *request* messages is presented in Table 5.2. Two sections can be identified: the mobile states list, and the service descriptor.

Field	Type	Description
mobile states length	8-bit unsigned integer	Length of mobile states list in bytes
key	16-bit unsigned integer	Key for mobile state
value length	8-bit unsigned integer	Length of value in bytes
value	array of unsigned 8-bit int	Value of mobile state
sd length	16-bit unsigned integer	Length of service descriptor in bytes
service desc	array of unsigned 8-bit int	Service descriptor XML file

Table 5.2: Encoding of payload fields for request messages

The *mobile states list* contributes to the fact that different patterns can be used during resource discovery. It provides a unification of pattern-specific information which is specified as a list of key and value pairs. The *mobile states length* field of the 8-bit unsigned integer type determines the list length in bytes. Hence, the number of key and value pairs is limited by the list length (128 bytes). Each mobile state is identified by a key that must be unique within the space of a distinct pattern. One key (0x01), however, is reserved for communication between the ASD management station and the start node. The variable-length *value* field specifies the actual value of the corresponding mobile state key. The length of the value field must be explicitly specified in the *value length* field.

In the second part of a *request* message the service descriptor is sent. The *service*

desc field contains the descriptor as an array of 8-bit integers. Due to the variable length of the service descriptor, the *sd length* field is required. Since this field is of the 16-bit unsigned integer type, the service descriptor is restricted to a length of 65.536 bytes.

Reply messages are exchanged (i) between ASD daemons, and (ii) between ASD daemons and the ASD management station. In general, *reply* messages are sent in reply to *request* messages to report the evaluation result. The encoding of payload fields in *reply* messages is presented in Table 5.3. *Reply* messages can be divided into three sections: the mobile state list (explained above), the suitable nodes list, and the occurred errors list.

Field	Type	Description
mobile states length	8-bit unsigned integer	Length of mobile states list in bytes
key	16-bit unsigned integer	Key for mobile state
value length	8-bit unsigned integer	Length of value in bytes
value	array of unsigned 8-bit int	Value of mobile state
node list length	16-bit unsigned integer	Length of node list in bytes
node address	32-bit unsigned integer	IP address of node
node group	8-bit unsigned integer	Node group this node is suitable for
result	16-bit unsigned integer	Evaluation result for this node
confi guration length	8-bit unsigned integer	Length of confi guration list in bytes
key	16-bit unsigned integer	Key for node-specifi c confi guration parameter
value length	8-bit unsigned integer	Length of value in bytes
value	array of 8-bit unsigned int	Value of node-specifi c confi guration parameter
error list length	16-bit unsigned integer	Length of error list in bytes
error generator	32-bit unsigned integer	IP address of error generator
pattern type	2-bit unsigned integer	type of error generator pattern
severity	2-bit unsigned integer	Severity of error
reserved bits	4-bit unsigned integer	reserved for future use
error ID	8-bit unsigned integer	Identifi cation of error
parameter length	8-bit unsigned integer	Length of error parameter list in bytes
key	16-bit unsigned integer	Key for error parameter
value length	8-bit unsigned integer	Length of value in bytes
value	array of 8-bit unsigned int	Value of error parameter

Table 5.3: Encoding of payload fields for reply messages

The *suitable nodes list* specifies information about nodes that are suitable to deploy the service in question. The *nodes list length* field of the 16-bit unsigned integer type indicates the byte length of the list. Information that must be specified for each node in the list includes the *node address*, the *node group* a node is suited for, and the evaluation *result*. Furthermore, for certain services it might be necessary to gather configuration information for suitable nodes. For instance, when deploying the unidirectional tunnel overlay service, information about interfaces on suitable nodes can be gathered during resource discovery. This information is stored in the *configuration list* which has the same format as the *mobile*

states list. The *configuration length* field defines the length of the configuration list in bytes. The *key* field holds the key for a particular configuration parameter and the *value* fields holds the actual parameter value for the corresponding key.

The *occurred errors list* contains information about occurred errors. The *error list length* field of the 16-bit unsigned integer type holds the length of the *occurred errors list* in bytes. For each occurred error, the IP address of the *error generator*, and the *pattern id* of the pattern that generated the error are specified. An error can be caused by an ASD daemon (0x00), a navigation pattern (0x01), a capability function (0x02), or an aggregation pattern (0x03). Furthermore, the *severity* field indicates whether the error message is an info (0x00), a warning (0x01) or a severe error (0x02). The *error type* must unambiguously identify a particular error. *Error types* for the ASD daemon will be presented in the implementation chapter. For each error an *error parameter list* can be specified. The *error parameter list* also has the same format as the *mobile states list*.

Deploy messages are exchanged between the ASD management station and ASD daemons. The purpose of sending a *deploy* message is to request a node to install the required service components, and to configure it according to the service requirements. In Table 5.4 the message format for *deploy* messages is depicted.

Field	Type	Description
confi glist length	8-bit unsigned integer	Length of confi guration list in bytes
key	16-bit unsigned integer	Key for confi guration parameter
value length	8-bit unsigned integer	Length of value in bytes
value	array of 8-bit unsigned int	Value of confi guration parameter
nodegroup sd length	8-bit unsigned integer	Length of nodegroup descriptor in bytes
nodegroup service desc	array of unsigned 8-bit int	Node-group-specifi c descriptor

Table 5.4: Encoding of payload fields for deploy messages

Deploy messages contain two sections: the configuration parameter list and the node-group-specific service descriptor. We already described the format of the *configuration parameter list* in connection with the *suitable nodes list*. Since *deploy* messages are sent directly to the selected nodes, additional node information such as the node address must not be specified. The *node-group-specific service descriptor* contains the subset of the attributes in the service descriptor that is necessary for the installation of service components and for configuration of an individual node. These are the service component name and the configuration parameters for the node group of the contacted node.

Status messages are exchanged between the ASD daemons and the ASD management station. According to the protocol specification, *status* messages are sent in reply to *deploy* messages. The message format of *status* messages is presented in Table 5.5. *Status* messages contain two sections: the deployment status and the occurred errors list. The *deployment status* field, defined as an 1-bit unsigned integer, indicates whether service deployment and

configuration were successful. The *occurred errors list* specifies errors that occurred during installation or configuration. We already explained the format of the *occurred errors list* in connection with *reply* messages

Field	Type	Description
deployment status	1-bit unsigned integer	Status of service deployment
error list length	16-bit unsigned integer	Length of error list in bytes
error generator	16-bit unsigned integer	IP address of error generator
pattern type	2-bit unsigned integer	Type of error generator pattern
severity	2-bit unsigned integer	Severity of error
reserved bits	4-bit unsigned integer	reserved for future use
error ID	8-bit unsigned integer	Identifi cation of error
parameter length	8-bit unsigned integer	Length of parameter list in bytes
key	16-bit unsigned integer	Key for error parameter
value length	8-bit unsigned integer	Length of value in bytes
value	array of 8-bit unsigned int	Value of error parameter

Table 5.5: Encoding of payload fields for status messages

5.3 Interfaces

The Automated Service Deployment protocol provides a user interface that facilitates the initiation and control of a service deployment operation. Furthermore, it interfaces the Overlay Topology Generation protocol on the application layer. Since one task of network-level service deployment is the initiation of node-level service deployment on selected nodes, the ASD protocol provides an interface to node-level deployment protocols.

5.3.1 User Interface

The user interface utilizes the ASD protocol functionality to automatically deploy services on behalf of a user. The ASD protocol provides the *findNodes()* function to initiate the resource discovery. The calling thread (a console or graphical user interface) is blocked until the function returns. Thus, only one service can be deployed at a time. For concurrent service deployment means for reserving resources on nodes would be required. Later versions of the protocol could support the deployment of multiple services at a time providing the necessary resource allocation mechanisms. The service descriptor, a unique service ID, and the IP address of the start node must be given as parameters when calling the function. The function returns a list of nodes which are suitable to deploy certain service components, and a list of errors that occurred during deployment.

This information could be displayed via a graphical user interface or simply a console. Important is that the interface provides the means for a user to either select nodes to deploy

service components, or to stop the deployment process. If nodes are selected, the *deployOnNodes()* function can be used to contact the nodes and initiate node-level deployment and configuration. The list of nodes that have been selected must be given to the function as parameter. Additionally, functions to unload, deactivate, or reconfigure a deployed service can be implemented, but this is out of the scope of this thesis.

5.3.2 OTG Protocol Interface

The Overlay Topology Generation (OTG) protocol provides the ASD protocol with overlay topology information. The *getNeighbors()* function is used by the ASD protocol to request a list of its direct neighbors within a certain range. An IP address, and an address mask that restrict the range to a particular domain or multicast tree are given to the function as parameter. The function returns a list of all identified neighbors and their associated cost within the given range. This cost information can apply to delay, jitter, or bandwidth on the link to the neighbor. Experiments with the ASD protocol will show which metrics are of interest for the patterns. An empty list is returned, if no valid neighbor can be identified.

Furthermore, the ASD protocol uses the *getNodeInDomain()* function to request a random node within a certain range. This function is used for instance by the constrained-remoteEcho pattern to flood a certain domain from a node that resides outside of this domain. An IP address and an address mask identifying the range are given as parameter to the function. The function returns the address of one node within the range.

5.3.3 Node-level Deployment Interface

Node-level service deployment protocols are interfaced by the ASD protocol to request the installation of service components on individual nodes. The *testDeployment()* function is used by the ASD protocol during resource discovery to determine whether the node provides the functionality that is needed to install the requested service component. The *deployServiceComponent()* function is used by the ASD protocol to install a requested service component on a selected node. Both functions take as parameter the name of the service component that is to be installed, and return a boolean value that specifies whether the (test) deployment was successful.

Chapter 6

Implementation

In this chapter we present the actual Java-implementation of the ASD protocol. Firstly, the implementation of the core ASD protocol, including the ASD network management station and the ASD daemon, is presented. Secondly, we introduce the implementation of an example navigation pattern, aggregation pattern and capability function.

6.1 Overview

In Fig. 6.1 an overview of the implemented packages is presented. We subdivided the ASD framework functionality into six packages. Dashed arrows represent import dependencies. Hence, two of main packages in the first row import classes from the packages in the second row. For reasons explained in the remainder of the chapter, we implemented the ASD protocol using the latest Java version which was only available as a beta-version (J2SE 5.0 Beta 2). Hence, when the final version of J2SE 5.0 is available, eventually adjustments have to be made.

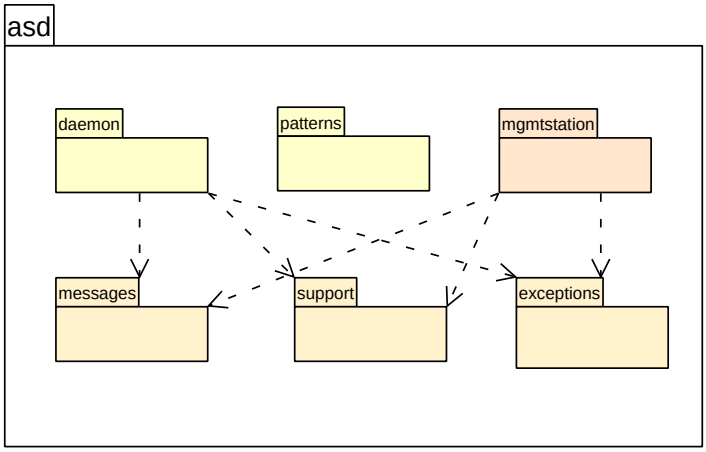

Figure 6.1: Overview of the implemented packages

The *asd.daemon* package defines the core ASD daemon functionality including interfaces to navigation patterns, aggregation patterns, and capability functions. The core ASD management station functionality is provided by the *asd.mgmtstation* package. The different ASD message types are defined in the *asd.messages* package, and ASD exceptions for failure handling can be found in the *asd.exceptions* package. Support classes used by the ASD management station and the ASD daemon are provided in the *asd.support* package. Finally, the example patterns are defined in the *asd.patterns* package.

6.2 ASD Network Management Station

Functionality provided by the ASD management station was specified in the last chapter. We implemented this functionality in four different packages: asd.mgmstation, asd.messages, asd.exceptions, and asd.support. Classes provided by these packages and their import dependencies are illustrated in Fig. 6.2. However, this section concentrates on the asd.mgmtstation package, since it provides the core ASD management station functionality. For description of the other classes refer to the corresponding sections.

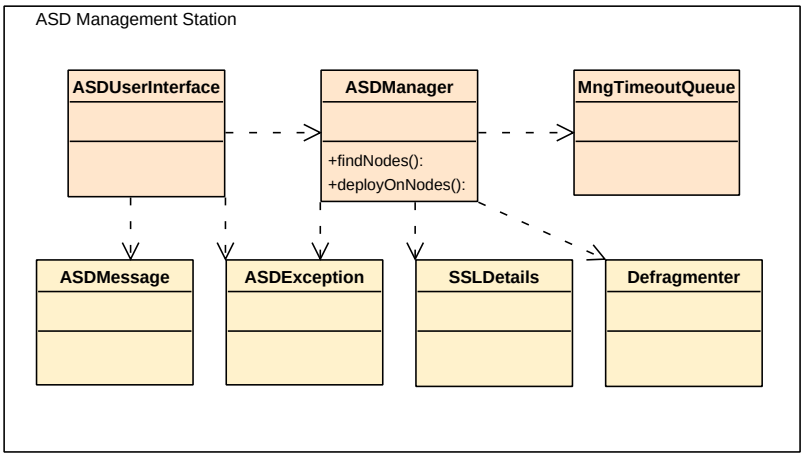

Figure 6.2: Class diagram of the ASD management station

The asd.mgmtstation package provides three classes: `ASDUserInterface`, `ASDManager`, and `MngTimeoutQueue`. The `ASDUserInterface` class defines the user interface. It instantiates the `ASDManager` class to utilize its *findNodes()* and *deployOnNodes()* methods. The `ASDManger` on the other hand instantiates the `MngTimeoutQueue` class, and classes provided by other packages(`ASDMessage`, `ASDException`, `SSLDetails`, and `Defragmenter`).

6.2.1 ASDUserInterface

The `ASDUserInterface` provides a basic user interface, and thus the means for a network manager to interact with the ASD framework. We previously identified functionality that the

user interface must provide: (i) specification of service requirements, (ii) initiation of resource discovery, (iii) selection of suitable nodes or interruption of the deployment operation, and (iv) verification of deployment results. The interface is realized as a simple console that displays information to the user, and waits until the user enters requested information. In a later version this simple interface can be replaced with a graphical user interface offering more comfort.

Remind that service requirements are to be specified in an XML document called the service descriptor. However, the basic `ASDUserInterface` does not include means for generating the service descriptor. The service descriptor must be generated beforehand, and made available to the `ASDUserInterface` by specifying the filename in a configuration file. Other required start-up arguments to be specified in the configuration file are the start node that administrates the automatic service deployment, and a unique service ID for this deployment operation. Optionally, the port ASD daemons are supposed to be running on can be changed by specifying a different port. The default port is 5000. An example configuration file is presented in the Appendix . Resource discovery can be initiated by starting the `ASDUserInterface` with a valid configuration file as argument.

Before the `ASDUserInterface` instantiates the `ASDManager` it creates the pattern error ID mappings. For each pattern a properties file must be specified that contains one entry (*errorID = error message*) for each exception it defines. To start the resource discovery the `ASDUserInterface` invokes the *findNodes()* method provided by the `ASDManager`. This method returns a list of suitable nodes and a list of error messages occurred during resource discovery. The `ASDUserInterface` displays the list of error messages using the previously created pattern error ID mappings. To generate the daemon error ID mappings, it uses the *constructErrorIDMappings()* method provided by the `ASDException` class. The returned list of all suitable nodes is displayed on the console, and the user is asked whether the deployment should be continued. If the answer is *'yes'*, the user is requested to make his selection. Otherwise the `ASDUserInterface` is terminated.

To start the installation and configuration of the selected nodes the `ASDUserInterface` invokes the *deployOnNodes()* method provided by the `ASDManager`. This function returns a list of deployment results for all selected nodes. The `ASDUserInterface` displays this list on the console and exits with a zero-status. In contrast, if an exception occurs the `ASDUserInterface` immediately exits with a non-zero status indicating abnormal termination.

6.2.2 ASDManager

The `ASDManager` implements the core functionality of the ASD management station protocol entity. Thus, its main task is to communicate with the distributed ASD daemons. According to the protocol specification the `ASDManager` uses TCP on the transport layer combined with the TLS protocol to guarantee data integrity and privacy for its connections. Furthermore, we decided to implement the `ASDManager` in a single thread, since concurrent deployment

operations are not supported by the ASD framework. Refer to Figure 5.2 for an overview of the communication between the ASD management station and the ASD daemons during resource discovery and service installation and configuration.

Resource discovery

Resource discovery functionality of the ASD management station is provided by the *findNodes()* method. This functionality includes sending of a discovery request to the ASD daemon running on the start node, and waiting for a reply from the contacted start node. The setup of an TLS/SSL connection to a particular node involves multiple steps. First an `SSLContext` is generated using the *getSSLContext()* method. A `KeyStore` for the previously created client keys is instantiated. Subsequently, an `SSLSocketFactory` is created, and an `SSLSocket` connected to the start node on the specified port is obtained. Finally, the socket must be configured to use the client mode.

The discovery request is generated by the *constructInitialRequest()* method. The payload encoding of request messages is given in Table 5.2. To construct the mobile states list, methods provided by the `ASDMessage` class are utilized. The service descriptor is serialized by the *serializeServiceDesc()* method and placed in the request message. Finally, the header fields are specified and the message itself is serialized. To send the request an `OutputStream` on the previously created socket is obtained, and the message is written to it. Furthermore, a timeout is set to limit the time the `ASDManager` is waiting for a reply from the start node.

To read the reply message from the socket an `InputStream` is obtained. Data is read from the input stream until a whole message is received. Methods provided by the `ASDMessage` class are used to de-serialize the received message. The `ASDManager` checks whether the received message is of the expected message type. If this is not the case, an exception is thrown by the `ASDManager`. Otherwise a `FindNodesResult` is constructed from the received list of suitable nodes and the received errors list. Finally, the socket is closed. The constructor for the `FindNodesResult` class in given in Table 6.1, since it is part of the ASD frameworks user interface.

```
public FindNodesResult(ASDMessage.SelectedNodesList shList,
  ASDMessage.OccurredErrorsList oeList){
  this.shList = shList;
  this.oeList = oeList;
}
```

Table 6.1: Constructor for the FindNodesResult class

Installation and Configuration

Installation and configuration functionality of the ASD management station is provided by the *deployOnNodes()* method. This functionality includes sending of installation requests

to all selected nodes, and waiting for the replies from all contacted nodes. We anticipate that for distributed services usually more than one node is selected to deploy a service. Hence, we decided to send installation requests simultaneously applying nonblocking I/O for performance reasons. To combine nonblocking I/O and TLS security was one of the most challenging parts of the implementation. Since this combination is not supported by the actual Java version (J2SE 1.4.2), we had to use the J2SE 5.0 Beta 2 version which provides the desired functionality. But naturally, there was no example code available how to combine nonblocking I/O and TLS security using the beta-version.

Nonblocking I/O (without TLS support) in Java is provided by the *java.nio.channels* package. For each socket a `SocketChannel` is created and put into nonblocking mode. A `Selector` can be used to monitor the registered channels. The *javax.net.ssl* package provides TLS support for Java. In the J2SE 5.0 version new classes have been added to this package that enable transport independent usage of TLS. The `SSLEngine` class operates on input and output streams, independent of the transport mechanism. Hence, to attain our objective we use normal sockets supporting nonblocking I/O as the underlying transport mechanism.

Usage of the `SSLEngine` requires four buffers. One for (i) outbound application data, (ii) outbound network data, (iii) inbound network data, and one for (iv) inbound application data. Application data, or plaintext, is data which is consumed or produced by an application. Its counterpart is network data, which consists of either handshaking and/or ciphertext data, and is destined to be transported via an I/O mechanism [23]. In our implementation the `SSLEngine` and its associated buffers are held by the `SSLDetails` class. The `ASDManager` uses a `HashMap` to map `SSLDetails` instances to particular channels.

To send the installation request one channel per selected node is created and registered with the selector. Furthermore, the deploy message is generated and a copy is put in the outbound application data buffer of the `SSLEngine` associated with each channel. The rest of the code is enclosed in a while loop that is only interrupted when replies from all selected nodes have been received, or the channels have timed out. In each iteration all registered channels are polled and a list of all channels that are ready for I/O is obtained. For each channel in the list, we determine the kind of I/O operation the channel is ready for. Supported operations are connect, read, and write I/O operations. Accept operations are not handled by the `ASDManager`, since the client mode was used when connecting to the selected nodes. At the end of each iteration timed out channels are closed.

Connect I/O operations are handled by the *handleSocketConnect()* method. For non-blocking channels *finishConnect()* is called to complete the TCP connection before using it. Remind that we use normal sockets. Thus, the SSL handshake must be handled separately by the `SSLEngine`.

Write I/O operations are processed by the *handleSocketOutput()* method. Before the deploy message can be send on a channel, however, the SSL handshake must be finished. Handshake data is generated by the `SSLEngine` and encrypted, like application data, by the

engine's *wrap()* method. Encrypted data is put in the outbound network data buffer, and subsequently written to the channel. Each call to *wrap()* returns an `SSLEngineResult` which indicates the status of the `SSLEngine` and (optionally) how to interact with the engine to make progress in the SSL handshake. Application data has been sent if the `SSLEngineResult` status indicates that the SSL handshake is finished. In this case a timeout must be set for the deploy message that has been sent. Therefore, the `MngTimeoutQueue` class is used. Moreover, we turn off the `OP_WRITE` bit for this channel, since we are not interested to write more data.

Read I/O operations are processed by the *handleSocketInput()* method. Again, only after the SSL handshake is finished application data can be received. All input data on a channel is first copied to the associated engine's inbound network data buffer. To decrypt the received data the engine's *unwrap()* method is called. If the received data is SSL handshake information the handshake advance is handled by the `SSLEngine`. Otherwise, the decrypted data is put in the corresponding inbound application data buffer. Moreover, the `SSLEngineResult` returned by the call to *unwrap()* must be checked. This is necessary, since the `SSLEngine` requires the creation of a delegated task for any operation that potentially may block. It also indicates when the peer has sent an SSL closure handshake message. If application data has been received the `Defragmenter` class is used to obtain a complete message which is then processed by the *handleStatusMessage()* method. Since no more data will be sent on this channel, the engine's *closeOutbound()* method is called to generate an SSL closure message. Finally, the write interest option is turned on again to send the closure message on the channel.

```
public Entry(String address, boolean status,
  ASDMessage.OccurredErrorsList oeList){
  this.hostAddress = address;
  this.status = status;
  this.oeList = oeList;
}
```

Table 6.2: Constructor for the DeployOnNodesResult.Entry class

Handling of a received message requires verification of the message type, de-serializing the message, generation of a `DeployOnNodesResult.Entry` instance, and canceling the timeout. The `DeployOnNOdesResult.Entry` contains information extracted from the received status message (see Table 6.2). If the timer for a channel expires before a message is received, an error message is generated and added to the occurred errors list of the entry for this channel. Moreover, the deploy status for the contacted node is set to false. An `ArrayList` of these entries (one per channel) is returned by the *deployOnNodes()* method.

6.2.3 MngTimeoutQueue

The timeout mechanism provided by the `Socket` class is sufficient for the *findNodes()* method. For the *deplyOnNodes()* method, however, a more sophisticated mechanism is

required, since multiple channel timeouts have to be managed. Therefore, we created the `MngTimeoutQueue` class which provides a sorted list of `TimedSelectionKey` objects. A `TimedSelectionKey` instance holds a `SelectionKey` identifying a particular channel, and the associated timeout value.

The `MngTimeoutQueue` class provides the *isEmpty()* method that can be used to check whether any timeouts are set, and the *containsKey()* method that allows to check whether a timeout is set for a particular key. Moreover, a *setTimeout()* method and a *cancelTimeout()* method are implemented. Both methods utilize the *binarySearch()* method provided by the `Collections` class. A `Comparator` that defines the order of the sorted list must be passed to the *binarySearch()* method, since the list entries do not implement `Comparable`. To obtain a list of all timed out keys, the *getTimedOutKeys()* method can be used.

6.3 ASD Daemon

As for the ASD management station, functionality provided by the ASD daemon was specified in the last chapter. We implemented the pattern-independent part of this functionality in four packages: asd.daemon, asd.message, asd.exceptions, and asd.support. Classes provided by these packages and their import dependencies are illustrated in Fig. 6.3.

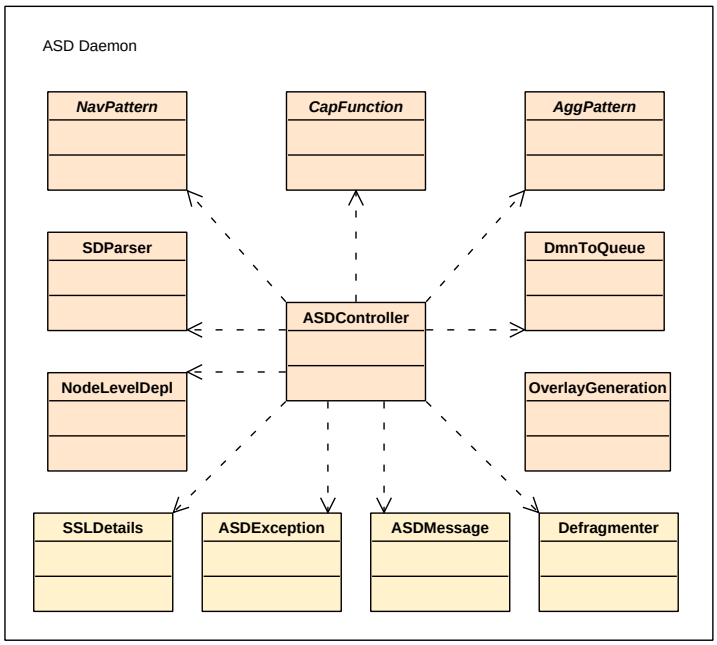

Figure 6.3: Class diagram of the ASD daemon

However, this section concentrates on the asd.daemon package, since it provides the core ASD daemon functionality. For description of the other classes refer to the corresponding

sections. The asd.daemon package comprises six classes: `ASDController`, `SDParser`, `OverlayGeneration`, `NodeLevelDeployment`, `DmnTimeoutQueue`, and the abstract classes `NavigationPattern`, `AggregationPattern`, and `CapabilityFunction`.

The `ASDController` class provides the main server loop. Therefore, it instantiates `SDParser`, `NodeLevelDeployment`, `DmnTimoutQueue`, and subclasses of the abstract `AggregationPattern`, `NavigationPattern`, and `CapabilityFunction` classes. Moreover, it instantiates different classes from other packages (`SSLDetails`, `Defragmenter`, `ASDException`, and `ASDMessage`).

6.3.1 ASDController

This class implements the core ASD daemon functionality. Hence, it provides means for the communication between ASD daemons and for communicating with the ASD management station. Moreover, pattern loading and management mechanisms are provided by the `ASDController`. Like for the `ASDManager`, TCP is used together with TSL to provide data integrity and privacy for the daemon's connections. Since we did not implement a TLS key distribution mechanism, all daemons use the same client and server keys. We decided to implement the `ASDController` single-threaded, since concurrent deployment operations are not supported. Furthermore, simultaneous message processing for one deployment operation is not required, since navigation patterns rely on finite state machines.

Before a service deployment operation can be initiated on the ASD management station, ASD daemons must be started, ideally, on all nodes within the managed network. Therefore, the main class of the ASD daemon, the `ASDController`, is started with a valid configuration file as argument. This configuration file can optionally specify the port ASD daemons are running on, a local pattern directory, a remote pattern directory, and the pattern server's IP address. If no values are specified the default configuration presented in Table 6.3 is used. Where `homeDirectory` identifies the home directory of the ASD daemon implementation.

```
private class DefaultConfiguration {
  public String port = “5000”;
  public String localPatternDir = homeDirectory + “/classes”;
  public String remotePatternDir = “/∼brauckhoff/”;
  public String patternServer = “pc-4207.etzh.ch”;
}
```

Table 6.3: Default configuration values

The `ASDController` implements a continuously-running server that accepts connections from other daemons and from the ASD management station. Moreover, it opens connections to other ASD daemons (acting as a client) to forward resource discovery requests. In order to reduce the overhead we decided to use the same connection for request and reply messages, instead of closing connections and opening a new connection for the reply. Hence,

the `ASDController` must manage multiple connections to different nodes. To achieve this we use the same concept as introduced in 6.2.2. The only difference is that also a `ServerSocketChannel` is registered with the selector to accept connections. Hence, in addition to connect, read, and write operations, accept I/O operations must be handled by the `ASDController`.

Incoming connection requests are processed by the *handleSocketAccept()* method. The connection is accepted, and the resulting socket channel is configured to use the nonblocking mode. Furthermore, a `Defragmenter` is attached to the channel to hold incomplete ASD messages before the channel is registered with the selector. `SSLDetails` holding the corresponding `SSLEngine` and buffers are created and stored in a hash using the channel's `SelectionKey`.

The *handleSocketConnect()* method is identical to the one used on the ASD management station (see 6.2.2 for the details). Also the structure of the *handleSocketInput()* method remains unchanged, apart from the fact that application data is handled by a method which is much more complex than the one used on the ASD management station. Only the *handleSocketOutput()* method had to be extended with functionality to send SSL closure messages on behalf of the patterns.

Processing of application data is handled by the *handleServiceDeployment()* method. This method processes all received request, reply, and deploy messages. Firstly, received messages are de-serialized and the service ID contained in the message header is added to a hash using the channel's `SelectionKey`. This is necessary to identify the patterns to be used for processing a message that belongs to a particular deployment operation (identifiyed by a unique service ID). The method proceeds with (i) obtaining the set of pattern instances for the received service ID, (ii) processing messages according to the message type, and (iii) handling the result of the message processing.

Request Message Processing

Request message processing is implemented in the *handleRequestMsg()* method. First, the `SDParser` is used to parse the service descriptor that is received in each request. Refer to 6.2.2 for a detailed description of the `SDParser` class. Moreover, if this request is the first message of a particular deployment operation (identified by the service ID) the set of pattern instances required to process the message is loaded. As specified in the requirements, a pattern loading mechanism must provide the means to load locally unavailable patterns from a pattern server.

In order to load patterns from a remote server, each pattern must be identified by a unique name. We decided to use a scheme similar to Java namespaces. The example patterns we have implemented are preceeded with the unique prefix 'ch.ethz.ee.tik.asd.patterns'. The pattern loading mechanism is implemented in the *createPatternDetails()* method. A `PatternDetails` object holds the loaded pattern instances. The object itself is stored in hash using the service ID to associate the pattern instances with a particular service

deployment operation.

To load the requested classes specified in the service descriptor, the *getInstance()* method provided by the `PatternLoader` class is used. This method, in fact, utilizes the `URLClassLoader`, which attempts to load the requested class from a local URL (local pattern directory), or a remote URL (remote pattern server). If the requested class can be loaded it is instantiated, and the obtained object is returned. Finally, the returned object is casted to the corresponding abstract pattern class. Remind that capability functions are node-group-specific. Hence, eventually multiple capability functions must be loaded. To address this, capability function instances are stored in a hash together with additional information such as the corresponding node group.

Second, the TTL value of the received request message must be examined. This value is decremented each time a request message is forwarded by an ASD daemon to limit the range of the node search. Thus, if a request with a zero TTL value is received the maximum range is reached, and an immediate reply must be sent in response. This is achieved by throwing a `TTLExeededException`. Exception handling mechanisms of the ASD daemon are explained in the remainder of this section.

Third, the patterns are used to process the received request message. Each of the loaded capability functions is started in oder to determine whether this node fulfills the service requirements for the corresponding node group. Pattern interfaces are defined by the abstract classes `NavigationPattern`, `AggregationPattern`, and `CapabilityFunction`. Refer to the corresponding sections for details. The navigation pattern is used to determine the aggregation function that is suitable for the actual pattern state and the received mobile state. Remind that the aggregation pattern must always match the navigation pattern. After the identified aggregation function has been executed the navigation pattern is started to determine the further protocol processing (e.g. sending of a message or closing a channel). Finally, a pattern result is generated based on the information returned by the patterns.

Reply Message Processing

The processing of reply messages is implemented in the *handleReplyMsg()* method. Each reply message (see Table 5.3) contains an occurred errors list. On reception of a reply message the received errors list is merged with the local errors list, which is held by the previously created `PatternDetails` objects. Since node capabilities are only evaluated for request messages, the execution of capability functions is not necessary when a reply message is received. As for request messages, the navigation pattern is used to determine the aggregation function to execute, and the selected function is started. Moreover, the navigation pattern is started, and a pattern result is generated with the information returned by the aggregation and navigation pattern.

Deploy Message Processing

Deploy message processing is implemented in the *handleDeployMsg()* method. As we explained in the protocol specification, patterns are not involved in the installation and configuration step. The processing of deploy messages is handled completely by the ASD daemon. Hence, on reception of a deploy message the `PatternDetails` object for this deployment operation is released. In each deploy message a node-group-specific service descriptor, which contains the configuration and installation information, is sent. As for the service descriptor, the `SDParser` class is used to extract this information from the received byte stream.

The *deployServiceComponent()* method provided by the `NodeLevelDeployment` interface is used to install the service component specified in the node-group-specific service descriptor. The interface must handle the case that no valid service component is specified. Node configuration is handled by the *configureNode()* method. Actually, this method implements node-level functionality. Since the ASD framework will not be tested together with a node-level service deployment protocol, we implemented the node configuration ourself to verify the deployment of the example tunnel service. The `ASDController` executes a configuration script, which must be specified in the node-group-specific service descriptor. Configuration information extracted from the service descriptor, and from the configuration parameter list in the received deploy message is temporarely saved in a configuration file. This file is used as input for the configuration script. In order to be executed by the `ASDController` the script must use Pearl as the scripting language, and have the tainted mode enabled. An overview of the ASD directory structure is given in the Appendix.

According to the protocol specification, a status message specifying the deploy status and the occurred errors is generated, and sent back to the ASD management station. The deploy status includes only the node-level deployment result, but not the configuration result. Since configuration is a node-level task, it might be handled by the node-level deployment protocol anyway. In the actual implementation correct operation of the configuration script has to be verified on the particular node. After the status message has been sent, the corresponding channel is closed by the `ASDController`.

Pattern Result Processing

The pattern result generated from information returned by the patterns on reception of a request or reply message is handled by the *handlePatternResult()* method. Patterns can specify a list of tasks that the ASD daemon executes on their behalf. The five defined tasks are (i) sending a request or reply message, (ii) setting a pattern timeout, (iii) canceling a pattern timeout, (iv) closing a channel and (v) setting a release timeout.

For sending a message the pattern must specify a list of recipients, and the message type. The `ASDController` constructs a message of the requested type. If a request message is to be sent a new connection is established. In contrast, for sending reply messages existing connections are used. We follow the paradigm that on each channel that is opened by an ASD

daemon exactly one request message is sent and one reply message is received. Thus, the *getSelectionKey()* method is used to search all keys registered with the selector for a channel that is connected to the node in question to send the reply message to. Only if no key is found, for instance in case the connection was already closed, a new connection is set up to send the reply. Further investigations into patterns will show whether it is really necessary to send reply messages although no connection is available. For setting up the socket channel a method very similar to the *setupSocketChannel()* method of the `ASDManger` is used. To send the message, it is simply put in the outbound application data buffer of the corresponding engine.

The ASD daemon provides a timeout mechanism that can be utilized by the patterns. We call it pattern timeout, since only one timer is used per pattern on a node. We decided not to use individual channel timeouts (as for the `ASDManager`) because for the ASD daemon timeouts are pattern-specific. For setting the timeout the pattern must specify a timeout value. For canceling a timer no information is needed, since the `ASDController` maintains an association between each pattern and its started timers.

If the pattern wants to close a channel it must specify the node for which the channel is to be closed. This is due to the fact, that patterns are not aware of communication details such as channels. They only deal with IP addresses. The `ASDController` determines the channel that belongs to the specified IP address and sets the `closeOutbound` flag in the `ChannelDetails`. This flag signals the `SSLEngine` that the application has finished sending data. It is evaluated each time the *handleSocketOutput()* method is called. Thus, a channel can only be closed by a pattern after some data has been sent on that channel.

Since patterns are only needed for the resource discovery, their state can be released after the last discovery message has been sent. The patterns must signal the `ASDController` when they are finished. We decided to use a timeout mechanism for releasing the pattern state. This allows delayed request messages to be handled by the pattern. The release timer is implemented utilizing the`Timer` class provided by the *java.util package*. This requires the definition of a `TimerTask` class which specifies the task that is scheduled for execution in the future. `MyTimerTask` implements the abstract *run()* method defined by `TimerTask`.

Pattern Timeouts

Pattern timeouts are managed by the `DmnTimeoutQueue` (see 6.3.2). At the end of each *select()* call a list of all expired pattern timers is obtained using the *getTimedOutPatterns()* method provided by the `DmnTimeoutQueue`. Since timers are started on behalf of a pattern, expired timers must be handled by the pattern as well. Each navigation pattern must implement the *handleTimeout()* method defined by the abstract `NavigationPattern` class. For patterns that do not utilize timers, this method does not need to provide any functionality. The *handleTimeout()* method returns a pattern result, which is processed by the *handlePatternResult()* method as for request and reply messages.

Exception Handling

For exception handling we implemented the `ASDException` class that subclasses `Exception`. Exceptions thrown by the ASD daemon are presented in 6.4. Of course, also patterns can throw `ASDExceptions`. All exceptions are forwarded to the *handleASDExceptions()* method provided by the `ASDController`. Since an exception on one of the channels can effect the pattern state, all exceptions thrown either by the daemon or by the patterns must be signaled to the navigation pattern. Each navigation pattern is required to implement the *handleASDExceptions()* method defined by the abstract `NavigationPattern` class, which is called by the `ASDController`. In case the pattern has already been released the channel on which the exception occurred is simply closed. Otherwise, an error message is added to the occured errors list, and the navigation pattern result is handled by the *handlePatternResult()* method. Independent of the pattern result, the `ASDController` closes the socket channel on which the exception occurred.

6.3.2 SDParser

The `SDParser` provides functionality to extract information from the serialized service descriptors. It uses the `SAXBuilder` provided by the *org.jdom.input package* to convert the byte stream into a JDOM document. Elements and attributes are read from this document using the functionality provided by the *org.jdom package*. Hence, this package must be available on each ASD daemon.

The `SDParser` class provides two public methods: *parseServiceDesc()* and *parseNodeTypeDesc*. Whereas the first method can be utilzed to extract information from the service descriptor, the second method implements functionality to extract information from the node-group-specific service descriptor. Extracted information is stored in the instance fields of the `SDParser` class. Both methods utilize the *getRootElement()* method in order to extract the root element from the descriptor. Since the `SAXBuilder` requires a file structure as input, the serialized descriptor is temporarely stored in a `File` object.

Information extracted from the service descriptor includes the service name, the navigation pattern name, and the aggregation pattern name. Moreover, navigation parameters, aggregation parameters, and the capability function information are extracted. We present the structure of navigation parameters, aggregation parameters, and capabliity function information in Figure 6.4, since they are passed to the corresponding pattern. From the node-group-specific service descriptor the service component name and the configuration parameter list, which is stored in a `Properties` object, are extracted.

6.3.3 DmnTimeoutQueue

The `DmnTimeoutQueue` is very similar to the `MngTimeoutQueue`. It provides the same methods, and uses as well binary search to insert and delete elements. The only difference is

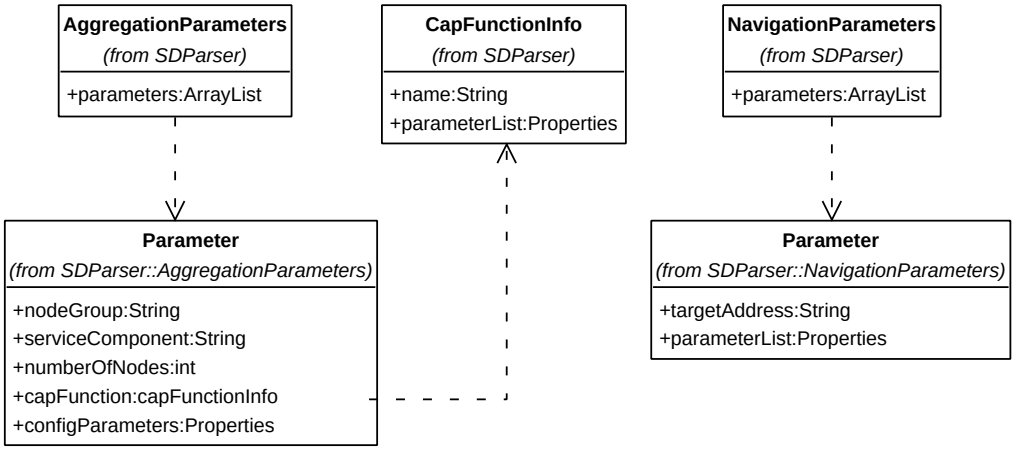

Figure 6.4: Class diagrams of the pattern parameter classes

that the `DmnTimeoutQueue` manages multiple pattern timeouts instead of multiple channel timeouts. Hence, the `TimedSelectionKey` object is replaced with a `TimedPattern` object. For a detailed description of the methods provided by the `DmnTimeoutQueue` refer to 6.2.2.

6.3.4 OverlayGeneration

The `OverlayGeneration` class implements the OTG protocol described in the protocol specification. This is the only class which is not instantiated by the `ASDController` (see Figure 6.3). Instead, the patterns instantiate the `OverlayGeneration` class directly in case they need to utilize the provided functionality.

We implemented a very basic OTG protocol that can be replaced with a more sophisticated approach in a later version. The overlay topology must be generated manually and stored in a file (named topology.txt in the /tmp directory) with a predefined structure. The `OverlayGeneration` class simply reads and interprets the topology information defined in this file. The file structure is illustrated in Table 6.4. For each node taking part in the overlay an entry (one per line) specifying the node's direct neighbors is needed. The node and its neighbors are separated by space characters. Moreover, for each neighbor an associated cost value (e.g. the round-trip time) is given, separated from the neighbor address by an at symbol.

152.3.136.2 152.3.136.1@0.234 152.3.136.3@0.222
152.3.136.3 152.3.136.1@0.244 152.3.136.2@0.188

Table 6.4: Extract from an example topology file

The `OverlayGeneration` class provides the *getNeighbors()* and the *getNodeInDomain()* method. Both methods return a list of nodes that fulfill the topology restrictions, and their associated cost values. Furthermore, the class method *isNodeInDomain()* can be used by the

patterns to determine whether a node resides within a given domain identified by a network address and an address mask.

6.3.5 NodeLevelDeployment

The `NodeLevelDeployment` class implements the node-level deployment interface specified in the protocol description. It defines the *testDeployment()* and the *deployServiceComponent()* method. Both methods take a string specifying the service component to be installed as parameter. These methods have to be implemented by the node-level deployment protocol. In our implementation both methods simply return true, since we concentrated on network-level functionality.

6.3.6 NavigationPattern

The abstract `NavigationPattern` class defines the interface between the ASD daemon and the navigation pattern. Each navigation pattern must subclass the abstract `NavigationPattern` class, and thus implement the methods defined by the abstract class. Moreover, navigation patterns inherit the fields and inner classes defined by the `NavigationPattern` class.

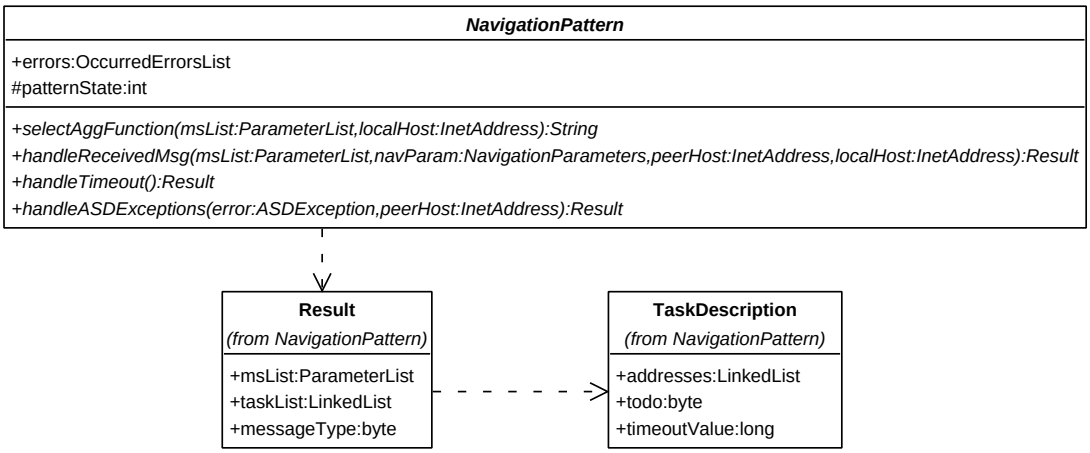

Figure 6.5: Class diagram of the abstract NavigationPattern class

The `NavigationPattern` class defines two instance fields: an errors list and the pattern state. The error list is used to communicate soft errors during pattern execution, which do not cause ASDExceptions, to the ASD daemon. The pattern state is needed, since each navigation pattern represents a finite state machine. Moreover, the `NavigationPattern` class provides a constructor for all its subclasses. When instantiating a concrete subclass of `NavigationPattern`, the actual pattern state is set to the initial pattern state defined by the subclass. An overview of the `NavigationPattern` class is presented in Figure 6.5.

Each pattern must implement the *selectAggregationFunction()* method. This method selects the aggregation function to be executed based on the local pattern state, which is stored in the instance field `patternState`, and the mobile pattern state, which is received in the `msList` and given as parameter to the method. The name of the selected aggregation function is returned as a string. The service provider has to assure that the aggregation pattern implements all aggregation functions requested by the navigation pattern. An empty string signals the `ASDController` that the execution of an aggregation function is not required for the actual state.

The *handleReceivedMessage()* method processes received messages based on the local and mobile pattern state. Parameters given to this method are: the mobile states list (`msList`), the navigation parameters (`navParam`) extracted from the service descriptor, and the address of the local host (`localHost`) and of the peer host (`peerHost`) at the other end of the socket channel. However, for reply messages not all parameters are specified. Since reply messages do not comprise a service descriptor, no navigation parameters are specified. Also the address of the local host is omitted.

The return value of this method is defined by the inner class `Result`. The navigation pattern can specify a mobile states list, a tasks list, a message type, and a timeout value for the message to be sent. The tasks list is defined as a list of `TaskDescription` objects. For each task the address of the node that the task refers to is specified. The task itself is represented by a combination of todo flags (see Table 6.5). When the `SET_TIMEOUT` flag is set, a timeout value must be specified by the navigation pattern. If an empty task list is returned, no actions are taken by the `ASDController`.

`SEND`	= 0x01
`CLOSE_CHANNEL`	= 0x02
`SET_RELEASE_TO`	= 0x04
`CANCEL_TIMEOUT`	= 0x08
`SET_TIMEOUT`	= 0x10

Table 6.5: Todo flags for the TaskDescription

Each navigation pattern must also implement the *handleASDException()* method. This method provides the processing of ASDExceptions for the navigation pattern. It takes the occurred exception (`error`), and the address of the peer host (`peerHost`) at the other end of the channel on which the exception occurred as parameters. Based on this information the navigation pattern can decide how to deal with the timeout, and return a `Result` specifying the actions to be taken by the `ASDController`.

The *handleTimeout()* method informs the navigation pattern that a pattern timeout occurred. Each navigation pattern can implement its own timeout mechanism in this method. The return value is again of the previously described `Result` class. Thus, the pattern can specify actions to be taken by the `ASDController` to handle the timeout.

6.3.7 AggregationPattern

The abstract `AggregationPattern` class defines the interface between the ASD daemon and the aggregation patterns. Each aggregation pattern must subclass the abstract `AggregationPattern` class, and, consequently, implement the methods defined by the abstract class. Moreover, aggregation patterns inherit the fields and inner classes defined by the `AggregationPattern` class.

The `AggregationPattern` class defines one instance field: a list of errors that occurred during aggregation pattern operation. This list can be used by the aggregation patterns to communicate error messages to the ASD daemon. When sending a reply message, these error messages will be included in the reply by the `ASDController`. The same applies for navigation pattern and capability function error lists. The *startAggregationFunction()* method is called by the `ASDController` if the *selectAggregationFunction()* method of the navigation pattern returned a non-empty string. Hence, the aggregation pattern must implement all aggregation functions that can be requested by the navigation pattern it is used with. An overview of the `AggregationPattern` class is presented in Figure 6.6.

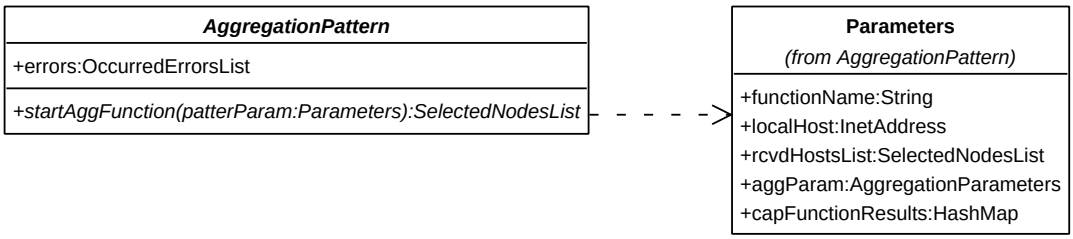

Figure 6.6: Class diagram of the abstract AggregationPattern class

Parameters passed to the *startAggregationFunction()* method are defined by the `Parameters` class. The name of the function to be executed is specified as a string. Moreover, the address of the local host, the selected nodes list received in a reply message, the aggregation parameters extracted from the service descriptor, and the results of the capability functions are specified. Since a selected nodes list is only received in reply messages, this parameter is not specified when a request message was received. In turn, aggregation parameters and capability function results are specified for request messages but not for reply messages. The `AggregationParameters` class is depicted in Figure 6.4. Capability function results are given in a `HashMap` that associates `CapFunctionInfo` objects with a `Short` value representing the score returned by the capability function. The method returns the (eventually) modified `SelectedNodesList`, which is used to update the selected nodes list held by the `ASDController`.

6.3.8 Capability Function

The abstract `CapabilityFunction` class defines the interface between the ASD daemon and the capability functions. Each capability function must subclass the abstract `CapabilityFunction` class, and thus implement the methods defined by the abstract class. Moreover, capability functions inherit the error list field defined by the abstract `CapabilityFunction` class. For a description of the error list field refer to 6.3.6. An overview of the `CapabilityFunction` class is presented in Figure 6.7.

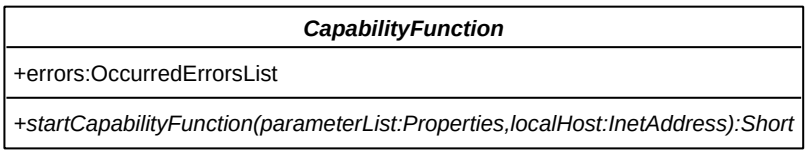

Figure 6.7: Class diagram of the abstract CapabilityFunction class

Each capability function must implement the *startCapabilityFunction()* method. Parameters passed to this method are the capability function parameters (`parameterList`) extracted from service descriptor, and the address of the local host (`localHost`). The `Properties` object holding the capability function parameters contains exactly the key and value pairs defined in the service descriptor. The address of the local host is required to generate error messages, since the *error generator* (see Table 5.3) must be specified for each error message. This applies also to navigation and aggregation patterns. The return value of this method is defined as a `Short` value representing the score for this node. Since this value is used as input for the aggregation pattern, there must be an aggreement between the aggregation pattern and the capability function about the values the score can take.

6.4 ASD Messages

The four ASD message types are implemented in the *asd.messages* package. The structure of this package is illustrated in Figure 6.8. We implemented each message type as a single class, each subclassing the `ASDMessage` class. The `ASDMessage` class provides methods for serializing and de-serializing the ASD header. Moreover, inner classes of `ASDMessage` implement the message parts: mobile states list, selected nodes list, and occurred errors list described in 5.2.1. Also, methods for serializing and de-serialing these message parts are provided. The classes that implement a particular message type utilize this functionality for serialing and de-serializing the messages.

In Table 6.6 we present the constructor of the `ParameterList.Entry` class. A `ParameterList.Entry` object is passed to the *addEntry()* method, which is utilized by the subclasses of `ASDMessage` to construct a parameter list. Since the pattern interface

classes import the *asd.messages* package, this method can also be utilized by the patterns for instance to generate a mobile states list. In this case the `key` parameter specifies the mobile state key for this entry. The `valueLength` parameter specifies the length of the value in bytes, and the `value` parameter defines the corresponding value. Furthermore, the total length of the entry in bytes (`length`) must be specified.

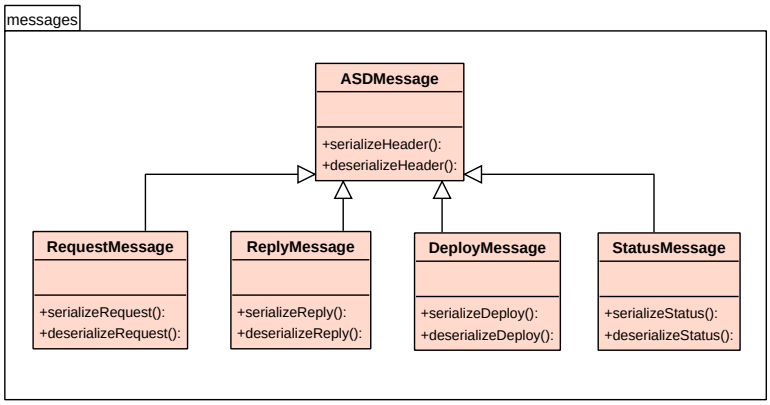

Figure 6.8: Class diagram of the asd.messages package

In Table 6.7 the constructor of the `SelectedNodesList.Entry` class is presented. A `SelectedNodesList.Entry` object is passed to the *addEntry()* and *deleteEntry()* methods which can be used, for instance, by the aggregation pattern to modify a selected nodes list. The `hostAddress` parameter is defined as an array of bytes. To obtain the IP address as an array of bytes for an `InetAddress` object, the *getAddress()* method can be used. The `configList` parameter is defined as a `ParameterList` object. Hence, it can be generated using the *addEntry()* method provided by the `ParameterList` class described above.

```
public Entry (short length, short key,
  byte valueLength, byte[] value) {
  this.byteLength = length;
  this.key = key;
  this.valueLength = valueLength;
  this.value = value;
}
```

Table 6.6: Constructor of the ASDMessage.ParameterList.Entry class

An `ASDException` object can be passed to the *addEntry()* method provided by the `OccurredErrorsList.Entry` class. This method can be utilized by the patterns to generate an occurred errors list. ASDExceptions are introduced in the next section.

```
public Entry (byte[] hostAddress, byte nodeGroup,
  short aggResult, ParameterList configList) {
  this.hostAddress = hostAddress;
  this.nodeGroup = nodeGroup;
  this.aggregationResult = aggResult;
  this.configParameters = configList;
}
```

Table 6.7: Constructor of the ASDMessage.SelectedNodesList.Entry class

6.5 ASD Exceptions

ASD exceptions thrown by the ASD daemon, or the ASD management station are defined in the *asd.exceptions* package. In Table 6.8 all ASD exceptions defined in this package are listed. Only the `MANAGER_TIMEOUT` exception is thrown by the management station, all other ASD exceptions occur at the daemon side. ASD exceptions that occur on the daemon are severe errors that stop the ASD protocol from correct operation. They are handled by the *handleASDExceptions()* method of the `ASDController` (see 6.2.2).

Error ID	Value	Description
`MANAGER_TIMEOUT`	0x01	Timeout on management station
`TTL_EXEEDED`	0x02	Received TTL equals zero
`NTD_PARSING_FAILED`	0x03	Failed to parse the node type descriptor
`NLD_FAILED`	0x04	Node-level deployment failed
`NODE_CONFIG_FAILED`	0x05	Node configuration failed
`CONNECT_FAILED`	0x06	Connection establishment failed
`SSLD_NOT_FOUND`	0x07	No SSL details found for connection
`SOCKET_READ_FAILED`	0x08	Exception during read operation
`UNWRAP_FAILED`	0x09	Exception during unwrap operation
`WRAP_FAILED`	0x0a	Exception during wrap operation
`SOCKET_WRITE_FAILED`	0xab	Exception during write operation
`MSG_TYPE_UNKNOWN`	0x0c	Unknown message type
`INVALID_PRES`	0x0d	Pattern result invalid
`PDG_FAILED`	0x0d	Pattern details generation failed
`SD_PARSING_FAILED`	0x0f	Failed to parse the service descriptor
`CHANNEL_SETUP_FAILED`	0x10	Channel setup failed
`PATTERN_NOT_FOUND`	0X11	Pattern details not found

Table 6.8: Error IDs for the ASD daemon and management station

Patterns can throw ASD exceptions as well. Each exception defined by the patterns is required to subclass the `ASDException` class. Soft errors that do not stop the daemon from normal operation can be caught by the patterns and simply added to the error list with the *addEntry()* method provided by the `ASDMessage.OccurredErrorsList` class. Severe

erorrs, however, must be forwarded to the `ASDController`. ASD exceptions defined by our example patterns will be described in the corresponding sections.

The `ASDException` class also provides the *constructErrorMappings()* method that is used by the `ASDUserInterface` to map the error IDs to strings. These are used for displaying error messages on the management station. Displayed error messages include, additionally to the error string, the ID of the pattern that caused the exception. Exceptions can be caused either by the ASD daemon (0x00), the management station (0x01), the navigation pattern (0x02), the capability function (0x03), or the aggregation pattern (0x04).

6.6 Support Classes

The *asd.support* package defines two support classes that are used on the ASD management station as well as on the ASD daemons. The first class, `SSLDetails`, holds the `SSLEngine` and its associated buffers. The second class, `Defragmenter`, is used to hold incomplete messages received by the management station or an ASD daemon.

The `SSLDetails` constructor requires the following parameters: an `SSLContext`, a host name and port, and two options (`useClientMode` and `requireClientAuth`). The first option specifies whether the client mode is used for the engine, and the second option specifies whether clients are required to authenticate themselves to the server. This option, obviously, makes only sense if the engine is configured to use the server mode. When creating an `SSLDetails` object also the four buffers needed for utilizing the engine (see 6.2.2) are allocated.

The `Defragmenter` class is utilized by the *deployOnNodes()* method of the `ASDManager` on the ASD management station, and by the `ASDController` on the daemons. It is needed in case a message gets fragmented. This can happen when a message is transported over a network which can only handle packets smaller than the packet size. When the message is read from the socket buffer on the receiving node, eventually, only part of the message has already reached the destination. The *getCompleteMessage()* method of the `Defragmenter` is called each time application data is received to assure that the received message is complete. The received message bytes are added to the end of a buffer maintained by the `Defragmenter`. When the message in the buffer is complete, according to the payload length defined in the message header, it is returned by the method. The buffer is compacted to delete the already completed messages.

6.7 Example Patterns

The example patterns we have implemented were briefly introduced in 3.3. The most complex of the example patterns is the `ConstrainedRemoteEcho` navigation pattern. It is implemented as a finite state machine. The `FindBestInform` aggregation pattern matches the

example navigation pattern. Furthermore, the implemented `MaxLoadAvg5minNetwork` capability function is adapted to the example aggregation pattern.

6.7.1 ConstrainedRemoteEcho

The `ConstrainedRemoteEcho` navigation pattern is the most complex pattern of the implemented. It subclasses the abstract `NavigationPattern` class and implements all its methods. In Figure 6.9 the finite state machine, the pattern is based on, is presented.

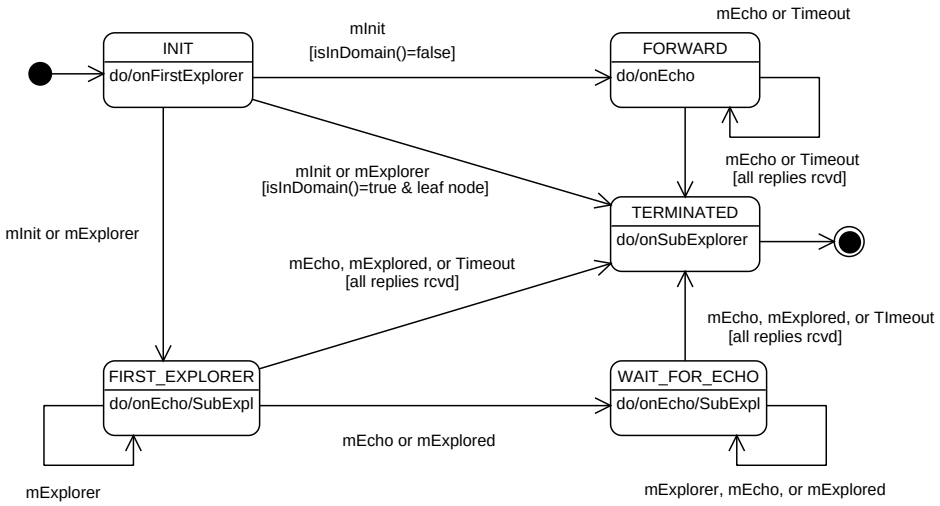

Figure 6.9: Statechart diagram of the ContrainedRemoteEcho navigation pattern

The initial pattern state, which is passed to the navigation pattern constructor, is the `INIT` state. When a service deployment operation is started on the ASD management station all participating nodes are in the `INIT` state. Furthermore, four mobile states are exchanged between nodes: *mInit*, *mExplorer*, *mEcho*, and *mExplored*. As an example we describe the transition from the `INIT` state to the `FORWARD` state. A node in the `INIT` state transits into the `FORWARD` state only if two conditions are fulfilled: an *mInit* message is received, and the node is not part of the target network.

The `ConstrainedRemoteEcho` class implements the *selectAggregationFunction()* method defined by the abstract `NavigationPattern` class. According to the actual pattern state and the mobile pattern state of the received message, this method determines the aggregation function to be executed by the aggregation pattern. In Figure 6.9 these functions are illustrated as do-actions for the particular state. Aggregation functions that must be provided by the aggregation pattern are: *onFirstExplorer*, *onSubExplorer*, and *onEcho*. These functions are described together with the FindBest-Inform aggregation pattern. In the `FIRST_EXPLORER` and `WAIT_FOR_ECHO` state one of two functions is selected according the received mobile state. If an *mExplorer* message was received the *onSubExplorer* aggregation function is

executed. Otherwise, for *mEcho* and *mExplored* messages, the *onEcho* aggregation function is executed.

After execution of the aggregation function, the *handleReceivedMessage()* method is called by the `ASDController`. This method determines the new pattern state and saves it in the `patternState` instance field. Furthermore, actions to be taken by the `ASDController` are determined. Since these leave-actions are not illustrated in Figure 6.9, we will describe them in detail.

Init State

At the beginning of a service deployment operation all nodes should be in the `INIT` state. Messages received from the management station, or from other nodes are processed by the *handleInitState()* method. First, the timeout budget is extracted from the service descriptor (for *mInit* messages), or from the mobile states list (for *mExplorer* messages). Next, for each target network specified in the service descriptor the *isNodeInDomain()* method provided by the `OverlayGeneration` class is used to determine whether this node is part the target network.

If the node does not reside within the target network the *getNeighborInDomain()* method provided by the `OverlayGeneration` class is used to determine the IP address and associated cost of one node within the target network. An *mExplorer* message is send to the identified node. If no node within the target network can be identified an exception is thrown. Since the `ConstrainedRemoteEcho` pattern uses a timeout mechanism, a timer is started, with the received timeout budget as timeout value, when the *mExplorer* message is sent. Moreover, the pattern transits into the `FORWARD` state.

If the node resides within the questioned target network, however, the *getNeighbors()* method provided by the `OverlayGeneration` class is used to determine all direct neighbors within the target network. This method might, in case of a symmetrical overlay topology, return the node that we received the *mExplorer* message from as one of the neighbors. The address of this node was previously saved in the `firstExplorer` instance field. Hence, it can be filtered out from the list of direct neighbors.

The *mExplorer* message is forwarded to all identified direct neighbors, and a pattern timeout is started. All nodes the message is forwarded to are added to the list of `unanwseredExplorers`. Moreover, the timeout bugdet must be decremented. For computing the new timeout budget the maximum cost of all links a message is sent out on (plus an additional margin) is subtracted from the received timeout budget. This is necessary to make sure the timeout occurs first on the node that experienced a problem, and not on some other downstream node. The pattern transits into the `FIRST_EXPLORER` state.

If no direct neighbor, or node within a target domain could be identified, this node is a leaf node in the overlay topology. In this case the received message can not be forwarded, and instead an *mEcho* message is sent immediately to the first explorer node. The pattern

transits directly into the TERMINATED state.

Other States

Messages received in any other pattern state than INIT are processed by the *handleOtherStates()* method. If an *mExplored* or *mEcho* message was received, the node that sent the message is deleted from the list of unansweredExplorers. Furthermore, it is determined whether this message was the last oustanding reply. If this is not the case the pattern waits for the other replies before sending a reply message itself. Moreover, it transits into the WAIT_FOR_ECHO state in case the actual pattern state is FIRST_EXPLORER. In contrast, if all replies have already been received an *mEcho* message is sent to the firstExplorer, and the pattern timeout is canceled. In this case the pattern transits into the TERMINATED state.

On reception of an *mExplorer* message an *mExplored* message is sent back in reply. This signals the sender that the node has already been explored by another node. Moreover, each time a reply message (*mEcho or mExplorer*) is sent the pattern requests the ASDController to close the corresponding channel and set the release timeout.

Timeout Mechanism

Pattern timeouts are handled on behalf of the patterns by the ASDController. When a timeout occurred the ASDController calls the *handleTimeout()* method of the navigation pattern. The ConstrainedRemoteEcho pattern handles timeouts, independent of the actual pattern state, by sending an *mEcho* message to the first explorer node. Moreover, it requests the ASDController to close all channels on which the node waits for outstanding reply messages, and to set the release timeout. When a timeout occurred the pattern transits immediately into the TERMINATED state.

Exception Handling

The described methods of the ConstrainedRemoteEcho pattern can throw ASD exceptions. These exceptions are caught by the ASDController (see 6.2.2). However, the protocol processing in case of exceptions is pattern-specific, and, consequently, handled by the *handleASDExceptions()* method of the navigation pattern.

Five different ASD exceptions are defined by the ConstrainedRemoteEcho class. A MobilePatternState exception is thrown if the mobile pattern state, extracted from the received mobile states list, is not valid for the actual pattern state. A NeighborDiscovery exception is thrown in case an error occurred during execution of the Overlay Topology Generation protocol. If the initial timeout value specified in the serivce descriptor is smaller than zero a TimeoutValueException is thrown. If the timeout budget, extracted from the mobile states list, is smaller than a minimum amount an TimeoutBudgetException is thrown, because a further reduction would probably result in a negative value. Only the TimeoutException is not thrown by the ConstrainedRemoteEcho class. Instead, it is added to the occurred errors list by the *handleTimeout()* method. This is due to the fact that a

timeout does not interrupt the normal protol operation.

If the `ConstrainedRemoteEcho` pattern is alredy in the `TERMINATED` state ASD exceptions are ignored by the *handleASDExceptions()* method. Otherwise, the peer host is removed from the list of `unansweredExplorers`. In case the list is empty after the peer host has been removed, a reply message (*mEcho*) is sent to the first explorer, and the pattern transits into the `TERMINATED` state. If the list is non-empty nothing needs to be done, since the channel on which the exception occurred is closed automatically by the `ASDController`.

6.7.2 FindBestInform

The `FindBestInform` aggregation pattern subclasses the abstract `AggregationPattern` class, and, consequently, implements the *startAggregationFunction()* method. To match the `ContrainedRemoteEcho` navigation pattern, it provides the *onFirstExplorer*, *onEcho*, and *onSubExplorer* aggregation functions. A list of selected nodes, which is updated and returned by the aggregation functions, is stored in an instance field.

The *onFirstExplorer()* method is executed on reception of the first *mExplorer* message in order to evaluate the capability function results. Each result corresponds to a particular node group. A test deployment of the associated service component is executed for all results with a score greater than zero. Therefore, the *testDeployment()* method provided by the `NodeLevelDeployment` class is utilized. If the test deployment was successful an entry for this node is added to the selected nodes list.

The `FindBestInform` aggregation pattern selects only one (the best) node per node group. Thus, on reception of an *mEcho* or *mExplored* message the local selected nodes list might need to be updated. This is handled by the *onEcho()* method. If the local selected nodes list is empty the received list is simply copied. Otherwise, the scores for each node group are compared and the better node is added to, or rather resides in the local list. The method returns the updated selected nodes list.

The *onSubExplorer()* method is executed when a subsequent *mExplorer* message is received. It returns the local selected nodes list, which is sent on behalf of the navigation pattern by the `ASDController` is an *mExplored* message. The `FindBestInform` aggregation pattern does not define or throw any ASD exceptions.

6.7.3 MaxLoadAvg5minNetwork

The `MaxLoadAvg5minNetwork` capability function subclasses the abstract `CapabilityFunction` class, and thus implements the *startCapabilityFunction()* method defined by its superclass. The capability parameter list, which is passed to this method requires three parameters for the `MaxLoadAvg5minNetwork` class: (i) the maximum allowed load average, (ii) a network or domain address, and (iii) the corresponding address mask.

The method first checkes whether all required parameters have been correctly extracted from the service descriptor. If any of the three parameters specified above is not available an `ParametersNotFoundException` is thrown. Moreover, it is determined whether the node is part of the specified network or domain. An exception thrown by the *isNodeInSubnet()* method of the `OverlayGenreation` class is caught and in turn an `IsNodeInSubnetException` is thrown by the `MaxLoadAvg5minNetwork` capability function. If the node is not part of the network a zero score is returned. Otherwise, the load average in the last 5 minutes is read from the '/proc/loadavg' file. Hence, we anticipate that a linux environment is available on all nodes. If the file cannot be found or any other error occurs during reading an `ReadLoadAvgException` is thrown. Finally, the load average value is be turned into a score, and the score is returned.

Chapter 7

Evaluation

We commence this chapter with a qualitative evaluation of the ASD framework's utility for automated service deployment and resource discovery. In the following sections, system tests with the implemented example patterns are described and a detailed functional and performance evaluation of the results is presented. The resource discovery part of the ASD framework was tested on PlanetLab [24]. The deployment of an example service, the setup of a unidirectional tunnel, was tested in a local testbed for reasons explained in the remainder of this chapter. Finally, the ASD framework is compared to related approaches regarding the provided functionality.

7.1 ASD Framework Utility

The strength of the ASD framework lies in the strict separation of discovery logic provided by the patterns from underlying communication mechanisms. Thus, the service-specific discovery logic (navigation, aggregation, capabilities) can be exchanged very easily by adapting the service descriptor. Only the pattern names and the parameters required by the selected patterns need to be modified. Furthermore, patterns are not required to be available on all nodes participating in the service deployment. A pattern loading mechanism is used to fetch unavailable patterns from a nearby server. This further supports the idea of pattern exchangeability.

The ASD framework is also helpful for the design of service-specific patterns. Developers can concentrate on the complex pattern-logic while the ASD framework provides the required infrastructure. This is on one hand the service descriptor, which provides a format for specifying deployment requests. On the other hand, the distribution of request and reply messages is handled by the ASD framework.

We specified clearly defined interfaces between the patterns and the ASD framework. Thus, patterns can delegate tasks such as sending of messages, and closing of connections, to the framework. Moreover, pattern-relevant information is extracted from received messages by the framework and passed to the patterns. Security is also provided by the ASD framework, since TLS authentication and encryption are used for all connections.

7.2 Functional Resource Discovery Tests

For the functional tests we deployed the ASD framework on four PlanetLab nodes at the ETH Zurich site, and two local nodes. The overlay configuration for each test was generated according to the test objective and transferred to each ASD daemon. In test (I) the general operation of the implementation is tested. whereas in tests (II) and (III) error situations are investigated. Finally, in test (IV) the pattern is tested on a fully connected network graph.

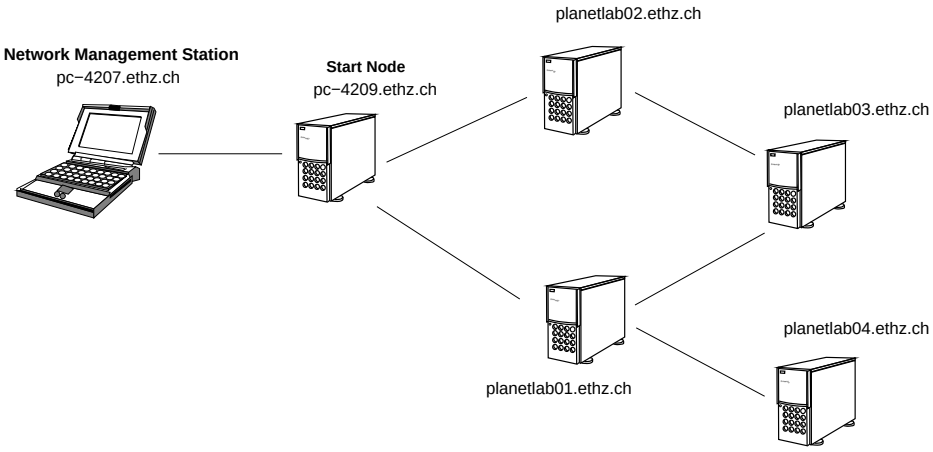

Figure 7.1: Testbed setup for test (I)-(III)

The testbed setup for tests (I)-(III) is depicted in figure 7.1. The ASD management station resides on local node pc-4207.ethz.ch, all other nodes run ASD daemons. As start node serves the local node pc-4209.ethz.ch.

7.2.1 General Functionality

Objective: The objective for test (I) is to verify the general functionality of the ASD framework under normal conditions.

Assumptions: We made the following assumptions regarding the navigation parameters: We defined one target network (129.132.57.0) which comprises all PlanetLab nodes in the testbed that run ASD daemons. The initial timeout value is set to 30 seconds, in order to prevent timeouts under normal operation (no errors). Regarding the aggregation parameters, we assume that only one node group is requested. Furthermore, only one node of this node group is required. As we do resource discovery tests, no service component and no configuration information is specified. Assumptions about capability function parameters include: a maximum load average in the last 5 minutes of less than 5.0, and a network address and mask identical with the target network. On the management station pc-4207.ethz.ch was selected as start node. In the daemon start-up arguments the start node is specified as pattern server.

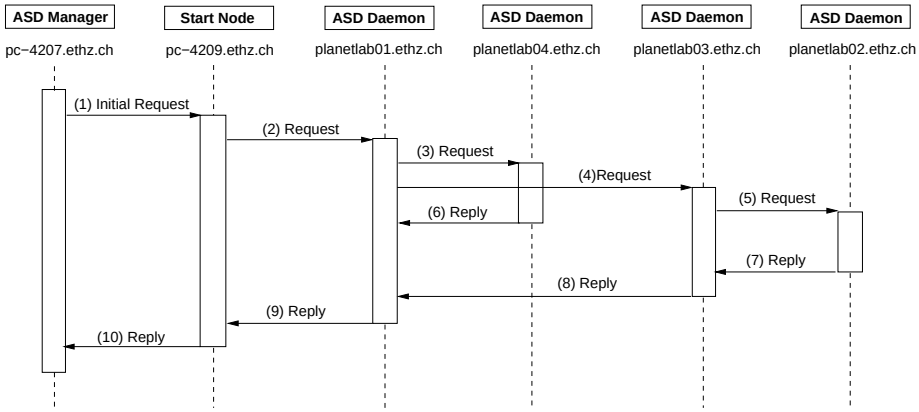

Figure 7.2: Sequence diagram for test setup (I)

Progression: The test progression is summarized in a sequence diagram (Fig. 7.2). On starting the test, the management station sends an initial request (1) to the start node, which is processed by the ASD daemon running on that node. As the start node itself is not within the target network, it forwards the ASD request (2) to one node within the target domain (planetlab01.ethz.ch). This node forwards the ASD request (3, 4) to all its neighbor nodes within the target network (planetlab03.ethz.ch and planetlab04.ethz.ch). Node planetlab04.ethz.ch has no further neighbors in the target network. Thus, it immediately generates a reply message (6) and sends it back to planetlab01.ethz.ch. Node planetlab03.ethz.ch forwards the ASD request (5) to its neighbor node planetlab02.ethz.ch, which in turn sends an immediate reply (7) back since it has no further neighbors within the target network. On reception of this reply message, node planetlab03.ethz.ch generates itself a reply (8) message and sends it back to node planetlab01.ethz.ch. Then, node planetlab01.ethz.ch has received all outstanding reply messages, and sends its reply message (9) to the start node. Finally, the start node has received replies from all nodes it forwarded a request message to, and thus it sends a reply (10) message back to the ASD management station. The resource discovery result is displayed on the management station.

Results: No exceptions occurred during the test. One suitable node was identified and displayed to the user. Thus, the basic operation of the ASD framework and the example patterns, as well as the interaction between them, is verified. Furthermore, the remote pattern loading mechanism worked correctly on all nodes.

7.2.2 Node Failure Tolerance

Objective: The objective of test (II) was to verify the correct behavior of the ASD framework in case some of the nodes are not responding or down. This scenario is considered very common in real-world networks.

Assumptions: The ASD daemons on node planetlab04.ethz.ch and planetlab02.ethz.ch

are not started in order to simulate unreachable nodes. Apart from that, assumptions are identical with those in test (I).

Progression: The sequence diagram in Fig. 7.3 depicts the test progression. The progression is identical to the test under normal operation until node planetlab01.ethz.ch forwards the request (3) to node planetlab04.ethz.ch, which simulates one of the failed nodes. When node planetlab01.ethz.ch attempts to establish the connection, an *ASDSocketConnectException* is thrown. Nevertheless, the request (4) is forwarded to node planetlab03.ethz.ch as under normal operation. On node planetlab03.ethz.ch the same exception occurs (5) when the second inactive node, planetlab02.ethz.ch, is contacted. Since planetlab03.ethz.ch has no further neighbors within the target domain, it sends its reply message (6) immediately after the exception. On reception of this reply message node planetlab01.ethz.ch can send its reply (7) to the start node. The further progression is again identical to the test under normal operation. But this time, the resource discovery result displayed on the management station includes two error messages.

```
planetlab01.ethz.ch:  Exception occurred during connection establishment
planetlab03.ethz.ch:  Exception occurred during connection establishment
```

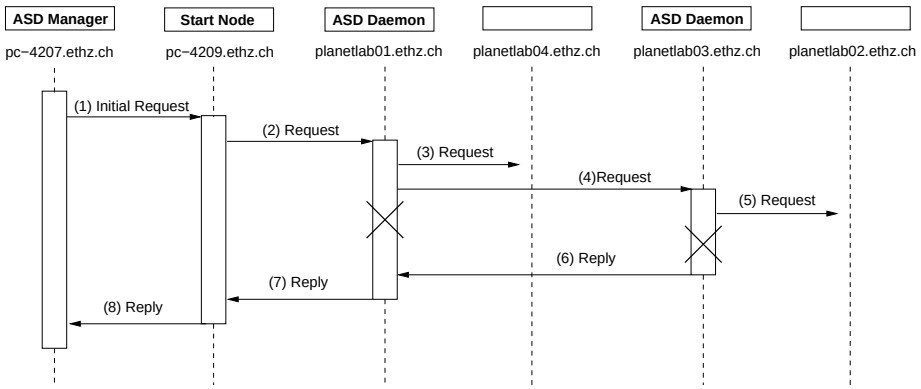

Figure 7.3: Sequence diagram for test setup (II)

Results: The test showed that the ASD framework can handle node failures. *ASDSocket-ConnectExceptions* are processed by the ASD daemon. The *handleASDException()* method, which handles all ASD exceptions, successfully generated an error message and called the *ConstrainedRemoteEcho* navigation pattern's exception handling method. Furthermore, is was verified that the exception handling method of the *ConstrainedRemoteEcho* pattern works correctly. A reply message is sent only if all outstanding replies have been received. The test also showed that error messages are displayed correctly to the user on the management station.

7.2.3 Timeout Handling

Objective: The objective of test (III) was to verify the timeout handling of the ASD framework and the example patterns.

Assumptions: We decreased the initial timeout value for the *ConstrainedRemoteEcho* navigation pattern to 5000 ms to force the occurrence of timeouts. Apart from that, the same assumptions as in test (I) apply.

Progression: Depending on the actual load of the network, and on the particular nodes the timeout occurs on different nodes. If the time between start of the timer on pc-4209.ethz.ch and start of the timer on planetlab01.ethz.ch is greater than a certain value the timer on pc-4209.ethz.ch expires first. If the time is smaller than this value the timer on planetlab01.ethz.ch expires first. Assuming the timeout occurs first on pc-4209.ethz.ch, see Fig. 7.4a, it causes a reply message (3) to be sent to the ASD management station immediately. Furthermore, the connection to planetlab01.ethz.ch is closed which automatically cancels the timeout on this node. Eventually, the timeout on planetlab01.ethz.ch occurs before the connection is closed. In this case a reply message is sent to pc-4209. This reply message, however is ignored by the ASD daemon on pc-4209. Assuming the timeout expires first on node planetlab01.ethz.ch, see Fig. 7.4b, the timeout causes a reply message (3) to be sent to pc-4209.ethz.ch immediately. If the timer on pc-4207.ethz.ch is not already expired it is canceled on reception of the reply message, and a reply message (4) is sent to the ASD management station. In both cases, an error message is displayed to the user on the ASD management station.

a) `pc-4209.ethz.ch: Timeout occurred.`
b) `planetlab01.ethz.ch: Timeout occurred.`

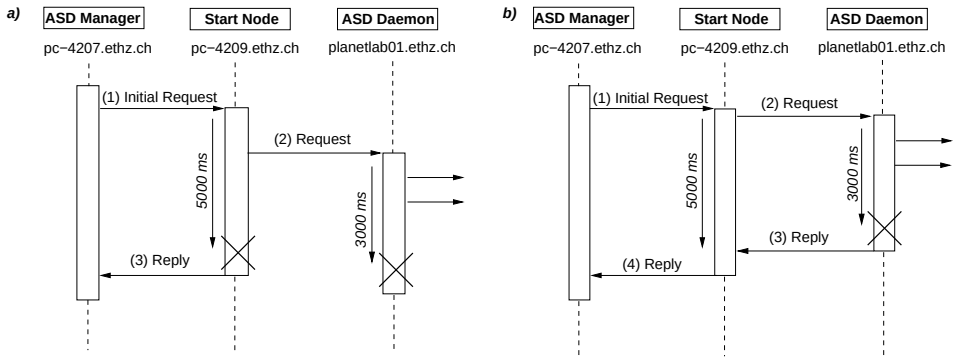

Figure 7.4: Sequence diagrams for test setup (III)

Results: Starting, canceling, and expiration of timeouts is handled correctly by the ASD daemon. As expected, the *ConstrainedRemoteEcho* navigation pattern sends an immediate reply

message in case of a timeout. Moreover, the timeout budget is decreased correctly each time a request is forwarded by the *ConstrainedRemoteEcho* pattern. The mapping of error IDs to error messages for patterns, which is based on configuration files, was verified.

7.2.4 Fully Connected Graphs

Objective: The objective of test (IV) was to verify the correct operation of the ASD framework and the example patterns on a fully connected network graph.

Assumptions: The overlay configuration for this test, depicted in Fig. 7.5, is based on a fully connected graph. We restricted the overlay to three ASD daemons because otherwise the sequence diagrams would get too complex. In contrast to the preceeding tests, we specified a node that is part of the target network (planetlab01.ethz.ch) as start node.

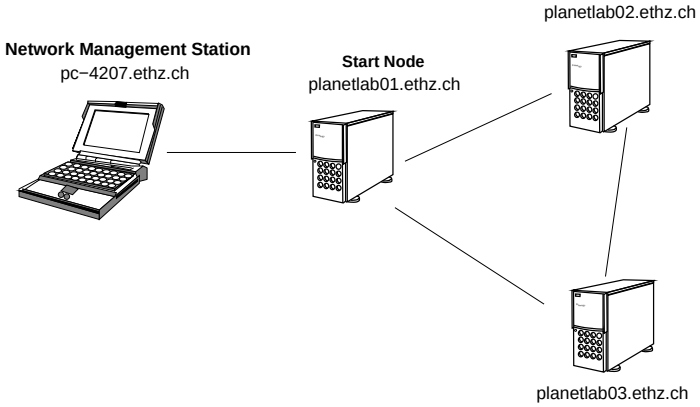

Figure 7.5: Testbed setup for test (IV)

Progression: The management station sends a request to the start node planetlab01.ethz.ch, which is part of the target network. Thus, it forwards the request to its neighbor nodes planetlab02.ethz.ch and planetlab03.ethz.ch. According to the fully connected graph topology, these nodes are neighbor nodes as well. Thus, each node is forwarding the request to the other node, respectively. When a request is received by one of the nodes, this node replies with sending an *mExplored* message back. This indicates that already a request message from another node was received. Changing loads on the individual nodes and in the network result in different message orders. In Fig. 7.6 one of the observed progressions is is depicted representatively.

Results: The test showed that the ASD daemon is capable of handling multiple connections to the same node. Furthermore, it was verified that the *ConstrainedRemoteEcho* navigation pattern handles fully connected graphs correctly by sending *mExplored* messages to subsequent explorers.

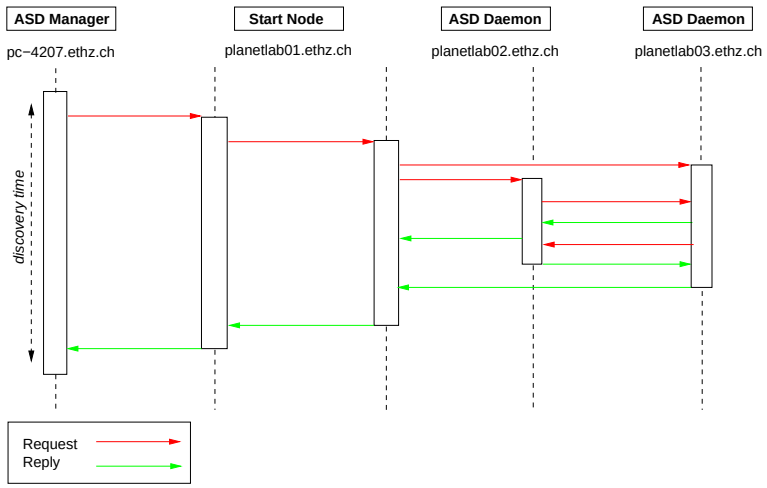

Figure 7.6: Example sequence diagram for test setup (IV)

7.3 Resource Discovery Performance Tests

To evaluate the large-scale functionality of the ASD framework and the implemented example patterns, we used the PlanetLab platform. PlanetLab is a globally distributed platform for developing and testing network services under conditions as experiences in the real Internet.

For testing the ASD framework we have to generate an overlay topology on top of PlanetLab. The idea is to generate the overlay based on round-trip-times between the individual PlanetLab nodes. That means, two nodes become direct neighbors in the overlay topology only if the round-trip-time for these nodes is smaller than a specified maximum. Hence, PlanetLab network characteristics such as the delay between nodes, influence the generated overlay topology. We analyze this influence of PlanetLab network characteristics on the overlay topology in order to draw conclusions for the test setup.

7.3.1 PlanetLab Network Analysis

Scriptroute [25] is a general network measurement interface that provides access to the underlying raw sockets on PlanetLab. A great number of PlanetLab nodes is running Scriptroute servers. These servers host measurement scripts that can be executed remotely to send network probes (e.g. ICMP requests) to a number of specified nodes. Hence, Scriptroute can be used to measure the round-trip-times between PlanetLab nodes.

Round-trip-time (RTT) measurements with the Scriptroute tool are continuously (every half hour) conducted for all PlanetLab nodes by Jeremy Stribling. This All-Pairs-Pings data is published at [26]. Naturally, we use this data for generating the overlay topology. Round-trip-times are measured between all PlanetLab nodes that run Scriptroute servers resulting in a $N \times N$ matrix (where N is the number of Scriptroute servers). Since PlanetLab provides real Internet conditions, some of the theoretically available servers might be unreachable at

the time measurements are conducted. Hence, measurement results are reduced to an *n x m* matrix (where *n* is the number of Scriptroute servers that send ICMP probes, and *m* the number of Scriptroute servers that answer ICMP probes).

We analyze the All-Pairs-Pings data for the week 07/01 to 07/07 to observe how the matrix varies over time. For each data set (in total 48 sets per day) the number of Scriptroute servers (*n*), that executed the measurement script successfully, is determined. Results are illustrated in Figure 7.7.

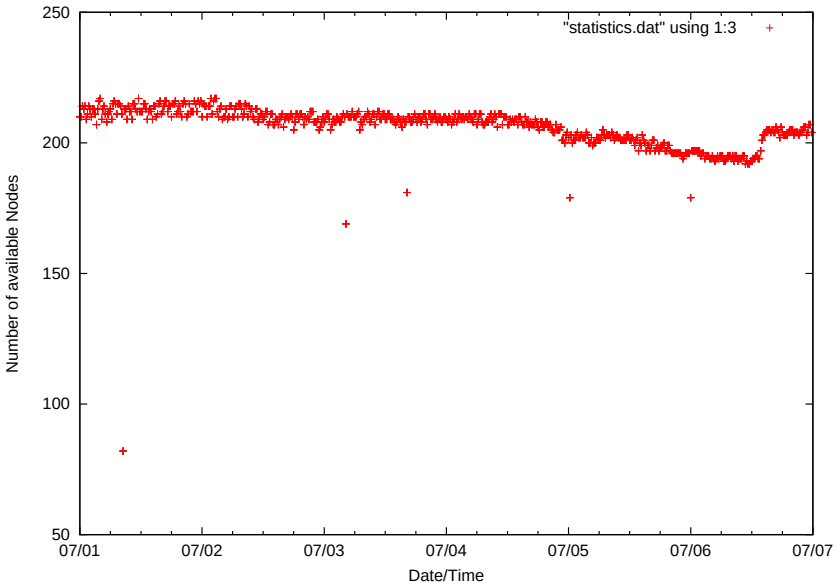

Figure 7.7: Number of available Scriptroute servers (n) from 07/01 to 07/07

On an average 210 Scriptroute servers, out of 280 deployed on PlanetLab, executed the measurements scripts. The number of servers available for the measurements has a variation of about 5% during one day. Considering the whole week, the number of available servers decreases between 07/03 and 07/04 reaching a minimum of 192 available servers. Moreover, for 5 data sets, distributed over the week, even less than 180 servers yielded results. Consequences for the overlay topology are examined at the end of this section.

Another interesting network topology characteristic is the average vertices degree. Considering the All-Pairs-Pings data, the vertices degree corresponds to the average number of Scriptroute servers that responded to ICMP requests. Hence, *N-1* equals the maximum, theoretically possible, vertices degree. We determined the average vertices degree for different maximum RTT values. That means, the number of *n* servers responding to the requests is reduced to servers that answered with an RTT less than the specified maximum. Naturally, the average vertices degree is proportional to the maximum RTT value. Figure 7.8 depicts the average vertices degree vs. the maximum round-trip-time (RTTmax) over

the week from 07/01 to 07/07. We randomly selected the All-Pairs-Ping data set measured at 11:00 Eastern Time for the analysis. Thus, for each RTTmax value seven average degree values (one value per day) are computed.

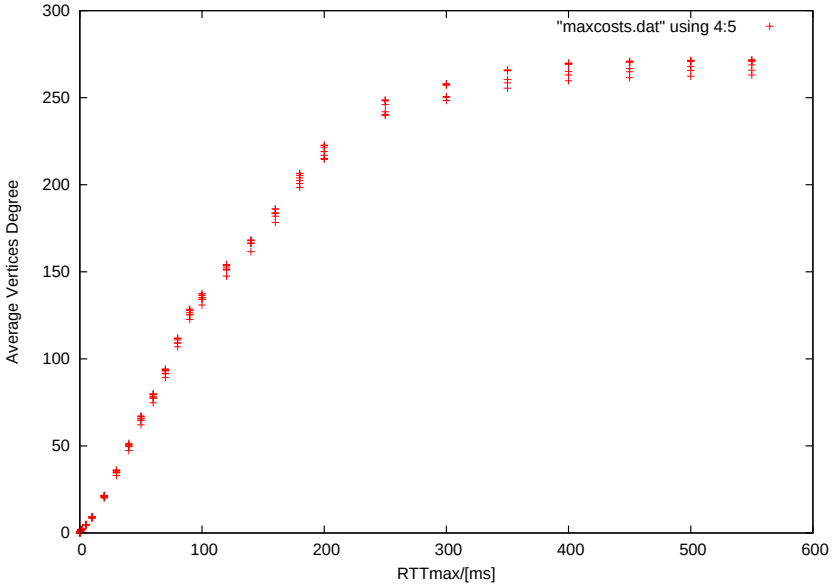

Figure 7.8: Average vertices degree vs. RTTmax from 07/01 to 07/07 at 11:00

As we expected, the resulting curve shows a proportional relation between the average vertices degree and the maximum RTT value. For RTTmax values greater than approximately 300 msec the curve flattens, and the maximum vertices degree averages around 270 reachable servers. Hence, the average maximum number of servers (*m*) that answer ICMP requests sent by another server is larger than the maximum number of servers (*n*) that actually send requests. According to Jeremy Stribling this is due to login-problems on Scriptroute servers when starting the measurement scripts. However, for smaller RTTmax values the number of sending servers (*n*) is larger than the number of responding servers (*m*).

We will generate overlay topologies based on All-Pairs-Pings data for varying RTTmax values. Since the measurement result matrix is asymmetrical, the generated overlay topology will be asymmetrical as well. The maximum number of nodes participating in the overlay is determined either by *n* or by *m*, which ever is larger for a particular RTTmax value and data set. Moreover, each node participating in the overlay generated from a particular data set can have up to *m-1* neighbors. To determine whether the resulting network graph is connected, or to determine the number of subgraphs, is non-trivial. However, it is not required for this work.

7.3.2 Test Setup

For the performance measurements we deployed the ASD daemon on all practically available (reachable via SSH) PlanetLab nodes. To achieve this in an efficient way, we utilized the Nixes Tool Set [27] developed at the Northwestern University. Nixes provides a set of bash scripts to install, maintain, control and monitor applications on PlanetLab. All scripts take a as argument a list of nodes. The ASD management station was deployed one PlanetLab node at the ETH Zurich site.

We used one All-Pairs-Pings data set, generated on 07/27 at 05:15 Eastern Time, for all measurements. For this data set we generated overlay topologies for varying (from 10 msec to 90 msec with an interval of 10 msec) RTTmax values. A detailed description of the overlay generation mechanism is given in 7.3. The topology was stored in a configuration file with the structure required by the OTG protocol (refer to 6.3.3). We transfered the file to our repository, and utilized the *plcmd* script provided by Nixes to download it to all nodes participating in the overlay.

Depending on the maximum round-trip-time value used to generate the overlay topology, the number of nodes that can be reached from one point (start node) within the network can vary. Moreover, the number of reachable nodes varies for different start nodes, since the topology is not fully connected. We selected planetlab1.inria.fr, a PlanetLab node at the INRIA Sophia Antipolis site, as start node, because it was the closest node that Scriptroute data was available for. The number of reachable nodes from planetlab1.inria.fr for the generated overlay topologies vs. the maximum round-trip-time used for the generation is depicted in Figure 7.9.

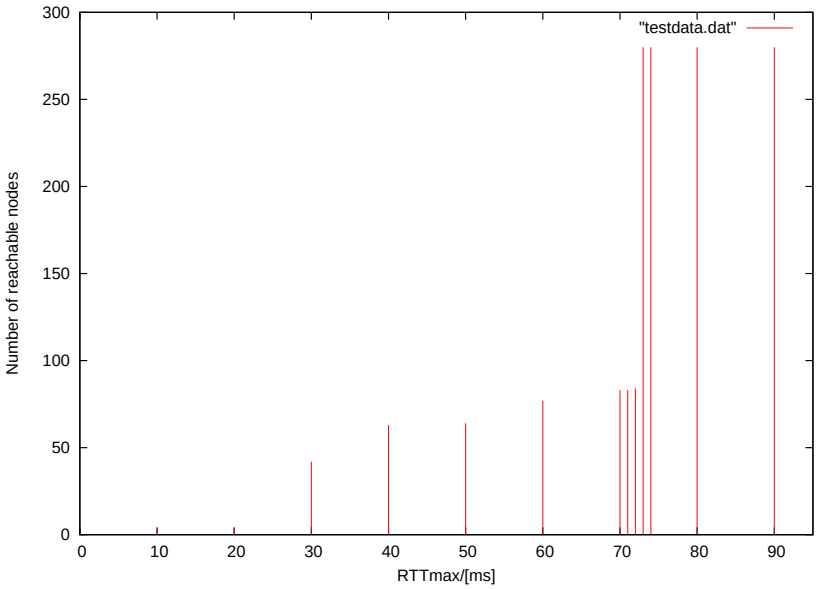

Figure 7.9: Number of reachable nodes vs. RTTmax for start node planetlab1.inria.fr

As we expected, the number of nodes that can be reached from the start node increases with RTTmax. Four nodes can be reached with a round-trip-time of less than 10 msec. These nodes probably reside at the same or a nearby PlanetLab site. The maximum number of nodes, indeed all 280 nodes that run Scriptroute servers, can be reached with an RTTmax value of 73 msec or more. The gradient of the curve is largest between 72 msec and 73 msec. We reason that for RTTmax values smaller than 72 msec only nodes at European sites can be reached. Whereas for RTTmax values larger than 73 msec all nodes on other continents (most of them at North American sites) can be reached as well. This requires only one oversea link, since most of the nodes will be connected among each other as well. We expect that the number of reachable nodes has an influence on the measurement results.

We can summarize that each test setup is characterized by static parameters such as the used All-Pairs-Pings data set, and the start node, as well as varying parameters such as the RTTmax value used to generate the overlay topology. For each test setup we measured the

- total discovery latency on the ASD management station
- total traffic generated for one resource discovery

Based on the analysis of PlanetLab network characteristics (Fig. 7.8), and the exploration of reachable nodes statistics (Fig. 7.9), we decided to vary RTTmax in the range from 20 msec to 70 msec with an interval of 10 msec. Moreover, test results are averaged over 10 measurements, and for each result the standard deviation is given.

ASD daemons were restarted for each test setup, since the topology configuration file had to be updated on all daemons. For starting the daemons we utilized the Nixes *plcmd* script again. The service descriptor was adapted to resource discovery tasks. Thus, node configuration information was omitted, since it is not needed for resource discovery. The initial timeout budget was set to 30 seconds, since this was about the expected time for a resource discovery over a maximum of 83 nodes to complete if no timeouts occur.

Moreover, the *ConstrainedRemoteEcho* navigation pattern requires a target network that is used as a restriction for the discovery. The network in which all PlanetLab nodes participate, indeed, is the Internet, as nodes are hosted by globally distributed institutions each using its own address space. Thus, we have to use a 'pseudo' target address (0.0.0.0) and address mask (0.0.0.0) to be able to reach all PlanetLab nodes using the *ConstrainedRemoteEcho* navigation pattern.

7.3.3 Test Results and Evaluation

First, we present the test results for the discovery latency measurements. In Fig. 7.10 measured resource discovery latencies as a function of RTTmax are depicted. The values for mean, and standard deviation (rounded to one digit) are given in table 7.1.

In contrast to our expectations, the resource discovery latency is not proportional to RTTmax. We anticipated that the discovery latency increases with RTTmax, since an

increasing RTTmax value results in an increasing number of visited nodes, and an increasing average degree of these nodes. We expected the average degree to have an influence on the discovery latencies (and on the generated traffic), because the *ConstrainedRemoteEcho* pattern forwards discovery requests to all direct neighbor nodes in order to flood the target network.

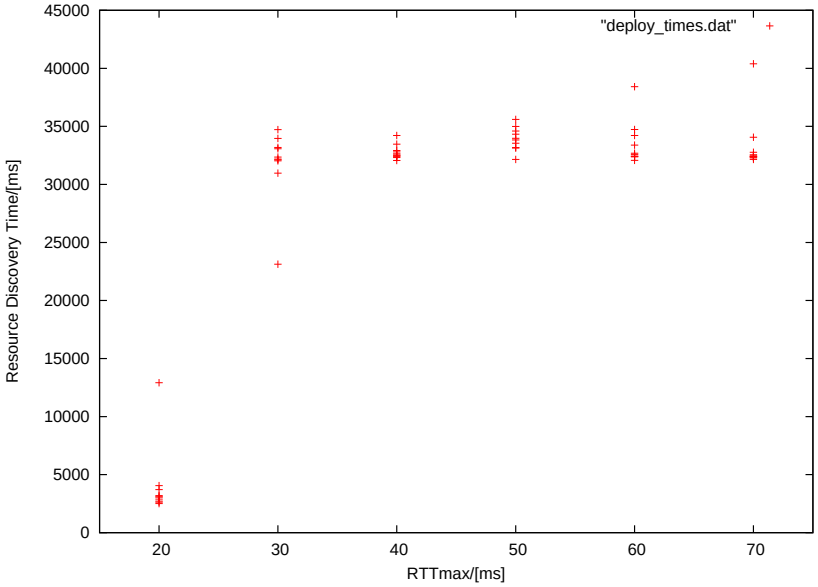

Figure 7.10: Resource discovery latencies vs. RTT max

For all measurements with an RTTmax value larger than 20 msec, except one, resource discovery completed in between 30 and 41 seconds. This can be explained with the more realistic conditions (node failures, high loads etc.) in the PlanetLab network, and the larger number of nodes participating in the performance tests compared to the functional tests. Thus, the probability of timeouts is much higher than in the functional tests. Indeed, only one resource discovery with an RTTmax value larger than 20 msec completed before a timeout occurred. This was due to the fact that during this test a large number of nodes was unreachable via SSH, and the discovery request could only be forwarded to a small subset of the theoretically reachable nodes.

Latency statistics	20 msec	30 msec	40 msec	50 msec	60 msec	70 msec
Mean	4076.4	31772.5	32783.9	33932	33533.6	33382.5
Standard deviation	2980.1	3052.9	598.8	950.3	1825.9	2392.9

Table 7.1: Mean and standard deviation for resource discovery latencies

We reason a decreased initial timeout budget would be more efficient. Ideally, the timeout occurs after all reachable nodes have sent their reply messages. The generated traffic for

different initial timeout budgets could be analyzed to determine the optimum timeout value. This, however, is left for further research.

Moreover, we observed that the first resource discovery lasted significantly longer than subsequent discoveries (see Fig. 7.10). This effect causes a high standard deviation value for a maximum RTT value of 20 msec. For measurements with an RTTmax value larger than 20 msec the standard deviation is not effected by this because the resource discovery is stopped after a timeout has occurred. However, standard deviation values are high due to inconsistent network conditions. Since ASD daemons had to be restarted before the first measurement with a particular setup, we believe this difference is due to the just-in-time (JIT) compilation mechanism of the Java Virtual Machine (VM). JIT compilation is used to increase the Java VM performance by speeding up the execution of code that is run repeatedly. However, if the simple OTG protocol we implemented, which requires static topology data in form of a configuration file, is replaced with a dynamic approach, ASD daemons do not need to be restarted anymore in case the overlay topology changes.

Second, we present the traffic measurement results. We differentiate between the total generated traffic, the TLS authentication traffic, and the ASD-related traffic (ASD messages including TLS encryption header). Measurement results are presented in Fig. 7.11. The three curves show the generated ASD traffic (blue), TLS authentication traffic (green), and total traffic (ASD + TLS, red) for varying maximum RTT values. In table 7.2 the mean and standard deviation for the overall traffic, and the average number of opened TLS connections (rounded to one digit) is given for each measurement.

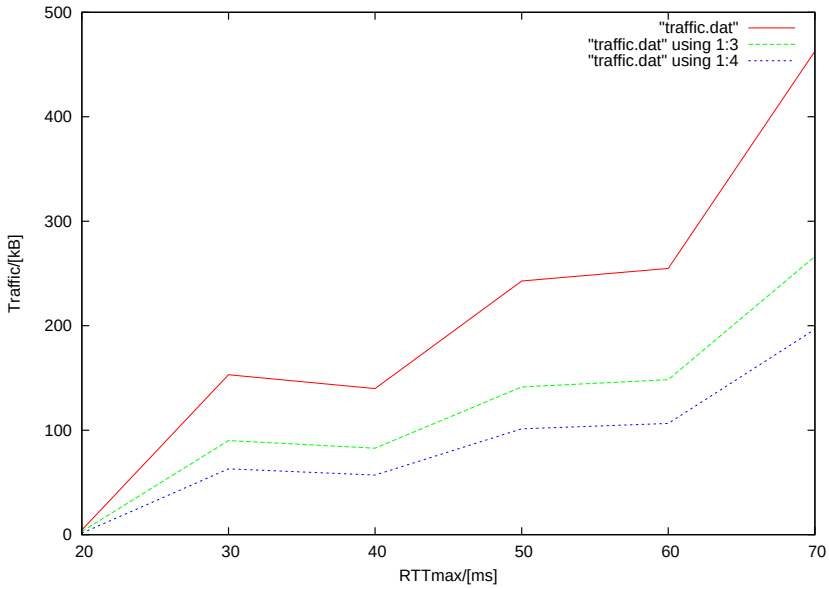

Figure 7.11: Generated traffic (total, ASD, TLS) vs. RTT max

TLS authentication traffic, ASD traffic, and consequently the overall traffic increases with RTTmax. This is due to the fact that with an increasing number of visited nodes, and an increasing average degree of nodes more TLS connections are to be opened, and more ASD messages are to be send. TLS authentication traffic makes up approximately 60% of the overall traffic. Only the remaining 40% are TLS encrypted ASD message bytes. TLS and ASD traffic curves have a very similar progression. In fact, TLS as well as ASD traffic increases linearly with the average number of opened connections (see Table 7.2).

Traffi c statistics	20 msec	30 msec	40 msec	50 msec	60 msec	70 msec
Mean	4.5	153.1	139.8	242.8	254.8	462.8
Standard deviation	0	12.8	4.5	48.3	82.6	78.6
Opened connections	1	44.7	47.8	79.1	77.1	172.4

Table 7.2: Mean, standard deviation, and opened connections for traffic measurements

For the *ConstrainedRemoteEcho* pattern the number of opened connections, in addition to the number of visited nodes, also depends on the average vertices degree of nodes. This is due to the fact, that each node forwards resource discovery requests to all direct neighbor nodes in order to flood the network. Hence, to further evaluate the large-scale characteristics of the *ConstrainedRemoteEcho* pattern, the number of opened connections as a function the average vertices degree of the visited nodes can be interesting to analyze. Remark that the average vertices degree of the visited nodes does not equal the average degree of all nodes participating in the overlay. This is due to the fact that the overlay topology can consist of several subgraphs, and only nodes in the same subgraph as the start node can be visited. Moreover, the number of visited nodes does not equal to the number of theoretically reachable nodes, since node failures are very common. If one node fails, and this node represents the only link to a subnetwork of *N* nodes, the number of visited nodes is decreased by *N+1*.

The conducted tests produced first performance measurement results for the *ConstrainedRemoteEcho* navigation pattern. In a test network that experiences real Internet conditions like PlanetLab, however, measurement results are highly dependent on factors such as network latency, and node CPU loads. Nevertheless, the tests have shown that a flooding-based navigation pattern such as the *ConstrainedRemoteEcho* pattern runs into scalability problems for large networks. Further analysis of the generated traffic for different network sizes and average degrees is required in order to make more precise statements about the scalability of flooding-based navigation patterns. The strength of the ASD framework is the support for different patterns. Thus, patterns tailored to specific service needs and network conditions can be developed, and used together with the ASD framework.

7.4 Service Deployment Tests

Since the ASD framework was not integrated with a node-level deployment service such as *Chameleon*, the deployment of service components could not be tested. Instead, we implemented node configuration functionality within the ASD daemon to deploy an example

service that does not require the installation of service components on nodes. This example service is a unidirectional tunnel between two nodes residing in different networks.

Moreover, we could not utilize the PlanetLab platform for this test. PlanetLab provides, for security reasons, only a very restricted environment to its users. This restricted environment does not allow the setup of tunnels on PlanetLab. Thus, the deployment of the unidirectional tunnel was tested in a local testbed. The test scenario is illustrated in Figure 7.12.

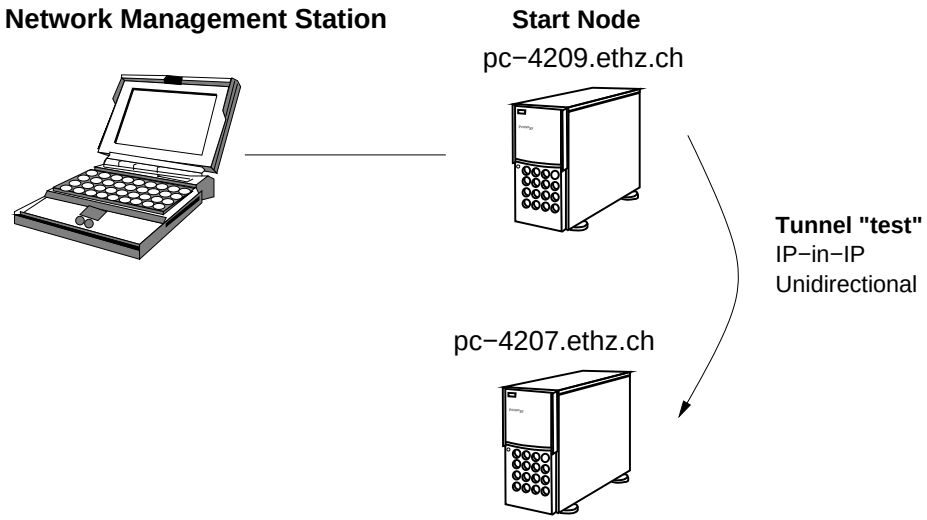

Figure 7.12: Service deployment test scenario

We decided to use a very simple overlay topology configuration, since resource discovery functionality has already been tested extensively. Only one node (pc-4209.ethz.ch) participates in the overlay. This node is also selected as start node. The objective of the test is to setup a unidirectional ip-in-ip tunnel from the start node to another node residing in a different network, which is specified in the service descriptor. The service descriptor contains additional configuration information required for automatically deploying a tunnel. In Table 7.3 an extract from the service descriptor is presented that illustrates the specified configuration information.

Configuration information is specified as a list of key and value pairs, which we call the *configuration parameter list*. Refer to Chapter 3 for details on the service description format. The configuration parameter list used for the service deployment test specifies seven keys and the corresponding values. Mandatory for all configuration parameter lists is only the *Script* key. This key specifies the name of the setuid-perl script that is to be executed in order to configure a node. The *Name* key specifies the name of the tunnel that is to be deployed, and the *Mode* key the tunnel mode. The tunnel mode for this tunnel is ip-in-ip. The *RemoteAddress* key specifies the remote tunnel endpoint, which is node pc-4207.ethz.ch

for this test. Furthermore, an IP address can be assigned to the tunnel interface using the *AssignedAddress* key. Which traffic will be directed through the tunnel has to be specified with the *Route* key. Finally, the *LocalInterface* key is used to explicitely force the tunnel to connect to the specified local interface.

```
<configurationParameterList>
  <key>Script</key>
  <value>tunnel.pl</value>
  <key>Name</key>
  <value>test</value>
  <key>Mode</key>
  <value>ipip</value>
  <key>RemoteAddress</key>
  <value>129.132.57.107</value>
  <key>AssignedAddress</key>
  <value>192.168.1.1</value>
  <key>Route</key>
  <value>129.132.57.0/24</value>
  <key>LocalInterface</key>
  <value>eth0</value>
<configurationParameterList>
```

Table 7.3: Configuration parameter list for the tunnel service

The configuration script, in this case the *tunnel.pl* file, has to be available on the node that is to be configured. Moreover, the script is required to be setuid by the root of this node. This is necessary because commands executed in the script such as *ip tunnel*, require root privileges. Note that all scripts, in order to be executed by the ASD daemon, must enable Perl‘s taint mode [28]. We have to admit that this is not suitable in case a larger number of nodes needs to be configured. However, node configuration is node-level specific functionality. Consequently, it should be handled by a node-level service deployment protocol.

Finally, we briefly describe test progression. Since node pc-4209.ethz.ch is the only node participating in the overlay, it is selected as suitable node and returned in a reply message to the ASD management station. On the management station we select node pc-4209.ethz.ch for deployment of the unidirectional tunnel service. A deploy message, containing the node-group-specific service descriptor, is sent to the selected node. On pc-4209.ethz.ch the *tunnel.pl* script is executed and the tunnel is setup. Node pc-4209.ethz.ch sends a status message back to the management station, and the deployment result *“Service deployment on node pc-4209.ethz.ch succeeded”* is displayed to the user. However, whether the tunnel has been setup correctly has to be verified manually on node pc-4209.ethz.ch.

7.5 Comparison with related Approaches

To give a qualitative evaluation of our work we compare the ASD framework with four competing approaches targeting the same or related areas. SWORD and Sophia have in common that they do not provide means for service deployment. SWORD targets resource discovery in large-scale networks, and Sophia proposes an approach for distributed management. HiGCS and Self-Configuring Active Services (SCAS) provide automatic service deployment in programmable networks. In Table 7.4 resource discovery functionalities of the different approaches are explored. The SCAS approach is not listed, since its resource discovery mechanism shows characteristics similar to SWORD (see Chapter 2).

Functionality	ASD	SWORD	Sophia	HiGCS
Distributed evaluation	distributed	DHT	distributed	distributed
On-demand collection	on-demand	continuous	on-demand	on-demand
Query language	XML	XML	Prolog	not specified
Range queries	supported	supported	not supported	not supported
Node groups	supported	supported	not supported	not supported
Exchangeable strategies	supported	not supported	not supported	simulated
Extensibility	req./strategies	requirements	requirements	requirements
Failure handling	fault reports	fault tolerance	fault tolerance	not specified
Service-specific queries	supported	not supported	not supported	not supported
Service-specific strategies	supported	not supported	not supported	supported

Table 7.4: Comparison of resource discovery functionalities

All listed approaches use *distributed evaluation* mechanisms. For ASD, Sophia evaluation is distributed over all queried nodes. SWORD, in contrast, uses distributed hash tables to store node status data and evaluation results on so-called DHT server nodes. Hence, SWORD does not apply *on-demand collection* of node status data. Instead, reporting nodes continuously send node status data reports to the DHT server nodes. For ASD, Sophia, and HiGCS node status data is collected on-demand.

ASD and SWORD use XML as *query language*. Moreover, both approaches support *range queries* and *node groups*. Sophia uses Prolog to express information about actual and desired system state. Since Sophia targets network management, its query language does not offer special resource discovery support such as range queries or node groups. HiGCS suggests the usage of XML as query language, but a particular query format is not specified.

Exchangeable strategies for resource discovery are only supported by ASD. Navigation and aggregation patterns, implementing different discovery strategies, can be exchanged very easily. SWORD implements three different range search techniques, but exchangeability is not supported. Sophia supports only one discovery strategy applying flooding mechanisms. The idea behind HiGCS is to use optimized hierarchies for querying nodes. To the best of our knowledge, different algorithms for creating these hierarchies have been simulated.

All approaches support the *extension of node requirements* by specifying additional query parameters in one or the other way. However, as described in the last paragraph, only ASD allows the *extension of discovery strategies* by applying different navigation and aggregation patterns. *Failure handling* mechanisms are supported by ASD, Sophia, and SWORD. The HiGCS approach does not specify failure handling mechanisms. Whereas, Sophia and SWORD provide fault tolerance, which means the system survives node failures, ASD additionally provides fault reports to inform a network manager about failures during discovery.

Service-specific query formats are solely supported by ASD. Service-specific templates, implemented as XML schemes or documents, provide means for defining service-specific query formats. As mentioned beforehand, three approaches (ASD, SWORD, and HiGCS) support multiple discovery strategies. However, only for ASD and HiGCS the strategies are defined under consideration of service-specific requirements.

A comparison of the ASD framework with the HiGCS and the SCAS approach regarding the provided service deployment functionality is presented in Table 7.5. SWORD and Sophia are not listed, since they do not provide service deployment functionality.

Functionality	ASD	HiGCS	SCAS
Node-level interface	implemented	not specified	implemented
Node configuration	implemented	implemented	implemented
Deployment interruption	supported	not specified	not specified
Security/Privacy	TLS auth/enc.	not specified	not specified

Table 7.5: Comparison of service deployment functionalities

The SCAS approach provides node-level functionality such as installation of code and *node configuration*. It provides a *node-level interface* to the plugin loader that implements the node-level functionality together with the PromethOS plugin framework. The HiGCS approach also provides node-level functionality. An Intermediate Representation (IR) is proposed to map Linux netlink configuration commands to the programming model provided by network processors. An IR prototype providing node configuration is implemented. However, to the best of our knowledge a node-level interface, linking resource discovery and node-level deployment functionality is not specified. ASD provides functionality for executing node configuration scripts on selected nodes. Installation of service components on selected nodes can be initiated using the provided node-level interface.

Deployment interruption is implemented by ASD in order enable human control of the otherwise automated service deployment process. Resource discovery results are displayed on the user interface, and the final node selection, or interruption of the deployment operation is executed by the service provider. HiGCS and SCAS do not specify means for external control. ASD uses TLS authentication and encryption to provide *security and privacy* for its

users. ASCS and HiGCS do not specify security measurements at all.

Chapter 8

Conclusion

8.1 Review of Claims

The aim of this work was to provide a flexible, and extensible framework for the automatic, on-demand deployment of network services in programmable networks. Two main tasks are to be performed by the framework. First, suitable nodes to deploy a service have to be identified. Second, the selected nodes have to be configured, and the installation of distributed service logic has to be initiated.

We have introduced a comprehensive service deployment architecture that automatically, performs the network-level service deployment tasks described above. The developed architecture features on-demand deployment, since the individual deployment steps are executed sequentially at deployment time. The idea of service-specific deployment is reflected (i) by the service description model, and (ii) by the defined deployment protocol. Service-specific description templates define the static service requirements such as the requested number of nodes and the node capabilities to be evaluated. We specified a general deployment protocol that distributes discovery requests to all candidate nodes, and installation requests to all nodes selected to deploy a service. This general protocol is extended with service-specific discovery mechanisms that are implemented as patterns, and can be loaded into the framework at runtime. Moreover, we have implemented a prototype of the developed architecture, which has been tested extensively in a large-scale network.

We have introduced a secure, flexible, and modular deployment protocol that supports service-specific deployment mechanisms. Security is achieved by using TLS encryption and authentication. Flexibility is addressed by supporting different discovery strategies which are implemented as patterns, as well as a mechanism to dynamically load these patterns. By composing service-specific deployment protocols from reusable patterns we address modularity. We defined three pattern types implementing different discovery functionality. Navigation patterns are responsible for the exploration of candidate nodes. Capability functions define rules for the evaluation of node capabilities on candidate nodes, and aggregation patterns provide functionality for aggregating the evaluation results.

We defined a layered service description model that supports service-specific discovery requests. This includes a (i) generic description template for specifying service deployment requests, a (ii) service-specific description template that defines static service requirements, and (iii) the instance-specific service descriptor that defines values for the parameters specified by the corresponding template. Moreover, the introduced generic description format supports all network-level deployment steps. Hence, the solicitation, summarization, dissemination, and advertisement step are handled autonomously by the ASD framework.

8.2 Critical Assessment

One objective for implementing the ASD framework was to facilitate the development and testing of service-specific resource discovery mechanisms. To achieve this, we compose deployment protocols from reusable components, namely navigation patterns, aggregation patterns, and capability functions. When developing an example discovery mechanism we considered this separation very helpful, since the ASD framework provides a clearly defined interface for each pattern. For the implemented example patterns these interfaces offer sufficient flexibility. However, we cannot exclude that other patterns might be constrained by the defined interfaces. This we will only find out when further patterns are developed.

We experienced that the navigation pattern implements the most complex functionality of the three patterns. It must be designed to explore a network in an efficient way, tailored to the service needs and network conditions such as network size or connectivity. Additionally, node and network failures must be handled by the navigation pattern. We have implemented a flooding-based navigation pattern, which is suitable for exploring smaller overlay networks. Node failures are handled by the pattern using a timeout mechanisms. However, the pattern can be improved to offer better performance, i.e. lower latencies, by optimizing the timeout mechanisms for a particular network setup.

Since our implementation is a prototype, the user interface serves only the most essential needs. The user interacts with the ASD framework (i) when specifying the service requirements, (ii) when defining the overlay topology, and (iii) when selecting the nodes. The ASD framework in its current version does not offer any support to the user regarding (i) and (ii). The service descriptor XML document has to be generated manually by the user according to the defined templates (XML schemes). Also the overlay topology used for the automatic deployment has to be generated, and distributed to all participating nodes manually.

8.3 Future Work

The most interesting and challenging part of future work will be the development of service-specific patterns to support the deployment of a broad range of services with a reasonable performance. In order to further facilitate the testing of different patterns with the ASD framework, the PlanetLab test environment should be extended. It would be very convenient

to have diagnostic tools that automatically observe the state of the distributed service. Also a sophisticated mechanism for collecting measurement data from the distributed nodes would enhance the testing comfort.

Moreover, the ASD protocol interfaces can be enhanced to provide more comfort to the user. A graphical user interface should be implemented that supports service-specific description templates. That means, the GUI constrains the parameters to be specified by the user according to the service-specific description templates. Also, the implemented Overlay Topology Generation protocol can be replaced with a more sophisticated approach. For instance, the round-trip-times between all nodes participating in an overlay can be measured locally on the nodes, and based on that information each node can generate its own topology view. In this case, a mechanism for distributing the overlay topology is not required, since it is generated in a distributed fashion. Moreover, the user would only have to specify the maximum round-trip-time for two nodes to be connected by a direct link in the generated overlay topology.

Appendix A

Task Description

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

TIK Institut für Technische Informatik und Kommunikationsnetze

Winter 2003/2004

Diplomarbeit

für

Daniela Brauckhoff

Supervisor: Matthias Bossardt
Co-Supervisor: Lukas Ruf

Ausgabe: 21.10.2003
Abgabe: 31.8.2004

Patterns for Service Deployment in Programmable Networks

1 Introduction

Programmable or active networks contain routers, which allow users to dynamically install programs. Such programs are typically part of a distributed service deployed in the network. Programmable routers enable the customization of packet handling in a very flexible way. Possible services include packet filtering, Web caches, as well as specialised multicasting protocols, load balancing and video scaling.

At TIK, we pursue an approach that composes services from service components. As a consequence, service components must be deployed on one or more programmable routers. We developed a service deployment architecture, which performs node level service deployment. That is, it allows to identify and deploy service components that must be executed on a specific router [1, 2].

The goal of this thesis is to extend the service deployment architecture to the network level. That is, programmable routers being able to execute service components must be identified according to multiple criteria, such as available bandwidth, processing power, location, etc. The most appropriate set of routers is then selected, and node level service deployment is triggered.

Analyzing existing services for active and programmable networks, patterns that guide the deployment of the services can be identified. Services may be classified according to their requirements on service topology, location, required resources, etc. As a consequence, service deployment must only deal with the different classes of services, instead of each service individually. With each class of service, a pattern may be developed that guides the details of deployment on the network level.

This project investigates a pattern based approach to gathering the necessary information and triggering the deployment of service components on the programmable routers. Service deployment is performed in a distributed way on behalf of the network manager. A preceding study yielded encouraging results [3]. Such an approach may be useful for automatizing the service deployment task, which is less error-prone and possibly more efficient in terms of deployment time and signalling protocol overhead than the traditional, centralized and human-controlled procedure.

2 Assignment

2.1 Objectives

The following objectives are expected to be met by this thesis.

- A network level service descriptor is to be defined allowing a formal description of the service deployment. It may specify topological and resource requirements of the service. A distinction of static requirements and configuration information known at deployment time should be considered.
- A basic protocol defining the interaction among the distributed deployment agents is to be specified and implemented. Using patterns, the protocol must allow to identify appropriate routers being able to execute the service components.
- The protocol should be based on the pattern based service deployment approach. That is, the protocol should be executed among the programmable routers, using their processing capabilities. Relevant deployment information should be aggregated whenever reasonable.
- The protocol should be executed in a management execution environment that allows to extend the protocol functionality by downloading code. This is necessary, if new deployment patterns are introduced and must be supported.
- An evaluation of the protocol includes a comparison with other approaches described in the literature and discusses its advantages and drawbacks.

2.2 Tasks

- Get familiar with active networks and service deployment concepts.
- Study the properties of active services relevant for their deployment.
- Deduce a list of requirements based on your findings from above.

- Get familiar with the concepts of pattern based management and pattern based service deployment.
- Define the format of a service descriptor that allows to specify deployment requirements. Get familiar with similar approaches (e.g. for Web services, IST-FAIN project, etc.).
- Based on pattern based service deployment concepts, describe the architecture and design of the system.
- Implement the basic protocol and management execution environment.
- Identify and implement at least one deployment pattern. Consider previous work on that subject [3].
- Compare the pattern-based approach to other approaches found in literature.
- Document your work in a detailed and comprehensive way. We suggest you to continually update your documentation. New concepts and investigated variants must be described. Decisions for a particular variant must be justified.

3 Deliverables and Organisation

- If possible, students and advisor meet on a weekly basis to discuss progress of work and next steps. If problems/questions arise that can not be solved independently, the students may contact the advisor anytime.
- At the end of the third week, a detailed time schedule of the semester thesis must be given and discussed with the advisor.
- At half time of the diploma thesis, a short discussion of 15 minutes with the professor and the advisor will take place. The student has to talk about the major aspects of the ongoing work. At this point, the student should already have a preliminary version of the table of contents of the final report. This preliminary version should be brought along to the short discussion.
- At the end of the diploma thesis, a presentation of 15 minutes must be given during the TIK or the communication systems group meeting. It should give an overview as well as the most important details of the work. Furthermore, it should include a small demo of the project.
- The final report may be written in English or German. It must contain a summary written, the assignment and the time schedule. Its structure should include an introduction, analysis of related work, and a complete documentation of the developed software. Related work must be correctly referenced. See http://www.tik.ee.ethz.ch/.ury/tips.html for more tips. Three copies of the final report must be delivered to TIK.
- Documentation, presentations and software must be delivered on a CDROM.

Literatur

[1] Bossardt, Matthias and Ruf, Lukas and Stadler, Rolf and Plattner, Bernhard. A Service Deployment Architecture for Heterogeneous Active Network Nodes. In *IFIP International Conference on Intelligence in Networks (SmartNet)*, Kluwer Academic Publishers, Saariselka, Finnland, April 2002.

[2] Bossardt, Matthias and Hoog Antink, Roman and Moser, Andreas and Plattner, Bernhard. Chameleon: Realizing Automatic Service Composition for Extensible Active Routers. In *Proceedings of the Fifth Annual International Working Conference on Active Networks, IWAN 2003*, Springer Verlag, Lecture Notes in Computer Science 2982, Kyoto, Japan, December 2004.

[3] Bossardt, Matthias and Muehlemann, Andreas and Zuercher, Reto and Plattner, Bernhard. Pattern Based Service Deployment for Active Networks. In *Proceedings of the Second International Workshop on Active Network Technologies and Applications, ANTA 2003*, Osaka, Japan, May, 2003.

Zürich, den 24.10.2003

Appendix B

Generic Description Template

```
<xs:schema
  xmlns:xs="http://www.w3.org/2001/ XMLSchema"
  elementFormDefault="qualified">

  <xs:element name="genericDescriptionTemplate">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="xs:serviceName"/>
        <xs:element ref="xs:navigationPattern"/>
        <xs:element ref="xs:aggregationPattern"/>
        <xs:sequence maxOccurs="unbounded">
          <xs:element ref="xs:navigationParameters"/>
        </xs:sequence>
        <xs:sequence maxOccurs="unbounded">
          <xs:element ref="xs:aggregationParameters"/>
        </xs:sequence>
      </xs:sequence>
    </xs:complexType>
  </xs:element

  <xs:element name="serviceName" type="xs:string"/>

  <xs:element name="navigationPattern" type="xs:string"/>

  <xs:element name="aggregationPattern" type="xs:string"/>
```

```
<xs:element name="navigationParameters">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="xs:targetAddress"/>
      <xs:sequence minOccurs="0">
        <xs:element ref="xs:parameterList"/>
      </xs:sequence>
    </xs:sequence>
  </xs:complexType>
</xs:element

<xs:element name="aggregationParameters">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="xs:nodeGroup"/>
      <xs:element ref="xs:serviceComponentName"/>
      <xs:element ref="xs:numberOfNodes"/>
      <xs:element ref="xs:capabilityFunction"/>
      <xs:element ref="xs:configurationParameterList"/>
    </xs:sequence>
  </xs:complexType>
</xs:element

<xs:element name="targetAddress">
  <xs:complexType>
    <xs:simpleContent>
      <xs:restriction base="xs:string">
        <xs:enumeration value="Router"/>
        <xs:enumeration value="Tree"/>
        <xs:enumeration value="Network"/>
      </xs:restriction>
    </xs:simpleContent>
  </xs:complexType>
</xs:element

<xs:element name="parameterList">
  <xs:complexType>
    <xs:sequence maxOccurs="unbounded">
      <xs:element ref="xs:key"/>
      <xs:element ref="xs:value"/>
    </xs:sequence>
  </xs:complexType>
</xs:element
```

```
  <xs:element name="nodeGroup" type="xs:short"/>

  <xs:element name="serviceComponentName" type="xs:string"/>

  <xs:element name="numberofNodes" type="xs:integer"/>

  <xs:element name="capabilityFunction">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="xs:capFunctionName"/>
        <xs:element ref="xs:parameterList"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element

  <xs:element name="configurationParameterList">
    <xs:complexType>
      <xs:sequence maxOccurs="unbounded">
        <xs:element ref="xs:key"/>
        <xs:element ref="xs:value"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element

  <xs:element name="capFunctionName" type="xs:string"/>

  <xs:element name="key" type="xs:string"/>

  <xs:element name="value" type="xs:string"/>

</xs:schema
```

Appendix C

Example Configuration Files

The *config* file must be available in the <ASD_HOME>/bin/ directory. It defines the ASD install directory and the home directory of a JDK 1.5.0 VM or higher.

The *managerarg.txt* file must be available in the <ASD_HOME>/input/txt/ directory on the ASD management station. It defines the startup arguments for the ASD management station.

The *daemonarg.txt* file can be available in the <ASD_HOME>/input/txt/ directory on the ASD daemon in case the default settings need to be changed.

```
############################################################
#  Install Configuration File
############################################################
#
#  install directory
   ASD_HOME=/home/ethz_chameleon/asd
#
#  location of JDK 1.5 or higher
   JAVA_HOME=/usr/java/jdk1.5.0
#
############################################################
```

Table C.1: /home/ethz_chameleon/asd/bin/config

```
############################################################
#  Management Station Configuration Parameters
############################################################
#
#  The default settings are changed
#  for all given parameters!
#
#  startNode and serviceID must always be specified!
#
   startNode = planetlab04.ethz.ch
   serviceID = 4545
#
#  descriptorFile =
#  port =
#
############################################################
```

Table C.2: /home/ethz_chameleon/asd/input/txt/managerarg.txt

```
############################################################
#  Daemon Configuration Parameters
############################################################
#
#  The default settings are changed
#  for all given parameters!
#
#  port =
#  localPatternDirectory =
#  remotePatternDirectory =
#  patternServerIPAddress =
#  overlayConfigFile =
#
############################################################
```

Table C.3: /home/ethz_chameleon/asd/input/txt/daemonarg.txt

Appendix D

ASD Directory Structure

asd/	- ASD home directory
asd/src/	- ASD implementation source code
asd/make/	- Makefiles for ASD daemon and management station
asd/jars/	- Jar files (asd-1.0.0.jar and jdom.jar)
asd/input/	- Input files for the ASD daemon and management station
asd/bin/	- ASD daemon start script
asd/daemon/	- Temporary daemon files
asd/manager/	- Temporary management station files
asd/docs	- Java documentation files
asd/templates	- Generic and service-specific description templates
asd/input/txt	- Configuration files and pattern error ID mappings
asd/input/cert/	- TLS certificates and keys
asd/input/xml/	- Service description XML documents
asd/input/pl/	- Node configuration Perl scripts

Bibliography

[1] Matthias Bossardt, Takashi Egawa, Hideki Otsuki, and Bernhard Plattner. *Integrated Service Deployment for Active Networks*. In Proceedings of the Fourth Annual International Working Conference on Active Networks IWAN, number 2546 in Lecture Notes in Computer Science, Zurich, Switzerland, December 2002. Springer Verlag.

[2] Matthias Bossardt, Andreas Mühlemann, Reto Zürcher, and Bernhard Plattner. *Pattern Based Service Deployment for Active Networks*. In Proceedings of the Second International Workshop on Active Network Technologies and Applications (ANTA 2003), Osaka, Japan, May 2003.

[3] Ralph Keller, Bernhard Plattner. *Self-Configuring Active Services for Programmable Networks*. In Proceedings of Fifth Annual International Working Conference on Active Networks (IWAN 2003), Kyoto, Japan, December 2003.

[4] Matthias Bossardt, Lukas Ruf, Rolf Stadler, Bernhard Plattner. *A Service Deployment Architecture for Heterogeneous Active Network Nodes*. Kluwer Academic Publishers, In Proceedings of 7th Conference on Intelligence in Networks (IFIP SmartNet 2002), Saariselkä, Finland, April 2002.

[5] T. Egawa, K. Hino, and Y. Hasegawa. *Fast and Secure Packet Processing Environment for Per-Packet QoS Customization*. In Proceedings of the IFIP-TC6 Third International Working Conference (IWAN 2001), September 2001.

[6] R. Haas, P.Droz, and B.Stiller. *Distributed Service Deployment over Programmable Networks*. In DSOM 2001, Nancy, France, 2001.

[7] D. Oppenheimer, J. Albrecht, D. Patterson, and A. Vahdat. *Scalable Wide-Area Resource Discovery*. UC Berkeley Technical Report UCB//CSD-04-1334, July 2004.

[8] Ralph Keller. *Dissemination of Application-Specific Information using the OSPF Routing Protocol*. Technical Report Nr. 181, TIK, ETH Zurich, Switzerland, November 2003.

[9] Joe Touch and Steve Holtz. *The X-Bone*. Third Global Internet Mini-Conference at Globecom'98, Sydney, Australia, November 1998.

[10] Yu-Shun Wang and Joe Touch. *Application Deployment in Virtual Networks using the X-Bone*. DARPA Active Networks Conference and Exposition, May 2002.

[11] Mike Wawrzoniak and Larry Peterson. *Sophia: An Information Plane for Networked Systems*. In Proceedings of HotNets-II, Cambridge, MA, USA, November 2003.

[12] H. de Meer, K. Tutschku. *Dynamic Operation of Peer-to-Peer Overlays*. In Proceedings Fourth Annual International Working Conference on Active Networks IWAN, Zürich, Switzerland, December 2002.

[13] L.H. Lehman, S.J. Garland, D.L. Tennenhouse. *Active Reliable Multicast*. In Proceedings of IEEE Infocom'98, San Francisco, CA, March 1998.

[14] K.L. Calvert, J. Griffioen, B. Mullins, A. Sehgal, and S. Wen. *Concast: Design and Implementation of a New Network Service*. In Proceedings of 1999 International Conference on Network Protocols, Toronto, Ontario. 1999.

[15] R. Keller, S. Choi, M. Dasen, D. Decasper, G. Fankhauser, and B. Plattner. *An Active Router Architecture for Multicast Video Distribution*. In Proceedings of IEEE Infocom'2000, Tel Aviv, Israel, March 2000.

[16] S. Bhattacharjee, K.L. Calvert and E.W. Zegura. *An Architecture for Active Networking*. In Proceedings of High Performance Networking (HPN'97), White Plains, NY, April 1997.

[17] D. Wetherall, U. Legedza, and J. Guttag. *Introducing New Internet Services: Why and How*. IEEE Network, Special Issue on Active and Programmable Networks, July 1998.

[18] M. Yamamoto, H. Miura, K. Nishimura, and H. Ikeda. *A Network Supported Server Load Balancing Method: Active Anycast*. IEICE Transactions on Communications, Special Issue on New Development on QoS Technologies of Information Networks, June 2001.

[19] E. Amir, S. McCanne, and R. Katz. *An Active Service Framework and its Application to Real-time Multimedia Transcoding*. In Proceedings of ACM SIGCOMM, Vancouver, Canada, August 1998.

[20] Koon-Seng Lim and Rolf Stadler. *Developing pattern-based management programs*. In Proceedings of 4th IFIP/IEEE International Conference on Management of Multimedia Networks and Services (MMNS 2001), Chicago, USA, Novemeber 2001.

[21] T. Dierks and C. Allen. *The TLS Protocol Version 1.0*. RFC 2246, January 1999.

[22] S. Kent, R. Atkinson. *Security Architecture for the Internet Protocol* RFC 2401, November 1998.

[23] Java 2 Platform Standard Edition 5.0 API Specification. SSLEngine Class. http://java.sun.com/j2se/1.5.0/docs/api.

[24] A. Bavier, M. Bowman, B. Chun, D. Culler, S. Karlin, S. Muir, L. Peterson, T. Roscoe, T. Spalink, and M. Wawrzoniak. *Operating System Support for Planetary-Scale Services*. In Proceedings of the First Symposium on Network Systems Design and Implementation (NSDI), March 2004.

[25] Neil Spring, David Wetherall, and Tom Anderson. *Scriptroute: A Public Internet Measurement Facility.* USENIX Symposium on Internet Technologies and Systems (USITS), 2003.

[26] Jeremy Stribling. All-Pairs-Pings for PlanetLab. http://www.pdos.lcs.mit.edu/.

[27] Stefan Birrer. The Nixes Tool Set. http:///www.aqualab.cs.northwestern.edu/.

[28] Nicolas Clark. Perl Security. http://search.cpan.org/∼nwclark/perl/