Reto Zürcher, Andreas Mühlemann

Pattern-based Service Deployment in Active Networks

Semester Thesis SA-2003.13
November 2002 to February 2003

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

Abstract

The functionality of current networks concentrates mainly on routing and forwarding. Active networks are a novel approach to network architecture in which the nodes allow to have distributed computation on them. This rapidly evokes new services, which are not possible in traditional networks. The usage of these services requires firstly the search of suitable nodes and secondly the installation on these nodes. This is definitively an error-prone and expensive work. We therefore thought about the following questions: How can we find appropriate nodes? How can we do this efficiently? And, since we not only have to find the nodes, but also to install the services on them: Is it possible to automate the whole service deployment?

That led us to the concept of deployment patterns from Koon-Seng Lim and Rolf Stadler. These patterns allow to classify the deployment of active services based on their deployment requirements. On the one hand, we provide a survey of services in active networks and their classification. On the other hand, we implemented some patterns and compared them to a central approach in different network structures. The simulation testifies that deployment patterns reduce the amount of sent data in the network compared to a traditional approach.

Zusammenfassung

Heutige Netzwerke zeichnen sich vor allem dadurch aus, dass sich ihre Hauptfunktionalität auf einfaches Routing und Forwarding beschränkt. Aktive Netzwerke sind ein neuer Forschungsansatz, der es erlaubt, auf den aktiven Knoten verteilte Programme auszuführen. Das führt zu neuen Services, die in traditionellen Netzen nicht möglich sind.
Die Nutzung dieser Services erfordert die Installation und Konfiguration auf geeigneten, aktiven Router. Diese Suche und die anschliessende Installation, was Deployment genannt wird, ist eine aufwendige und fehlerbehaftete Arbeit. Deshalb stellen sich unweigerlich die folgenden Fragen: Wie können diese geeigneten, aktiven Router gefunden werden? Wie kann Effizienz erreicht werden? Und da nicht nur geeignete Knoten gefunden werden müssen, sondern auch noch eine Installation erforderlich ist: Wie kann das gesamte Deployment von Services automatisiert werden?
Das führte uns zum Konzept der Deployment Pattern von Koon-Seng Lim und Rolf Stadler. Diese Pattern erlauben es, Services nach ihren Deployment Anforderungen zu klassifizieren. Pattern wurden in Software implementiert und mit einem Simulator getestet. Es wurde ein zentralistischer Lösungsansatz in verschiedenen Netzwerken mit einem Pattern basierten Ansatz verglichen. Es konnte gezeigt werden, dass Deployment Pattern die Menge versendeter Daten stark reduzieren können und somit die Netzwerkressourcen möglichst nachhaltig verwenden.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter presents the introduction and motivation for the research described in this thesis. Section 1.1 provides first the motivation for thinking about a classification of active services and developing specific navigation and aggregator patterns on this base class . It furthermore introduces into the advantages of active networks and describes the pattern-based service deployment approach used in this thesis.
The last section sketches the organization of this thesis.

1.1 Motivation

The installation, configuration and setup of software is often a boring, repetitive, expensive and even error-prone task. Lot of effort is payed on automation and classes are identified to solve similar tasks with one solution.
That is exactly the main goal for the deployment of active services in this thesis. The task of deployment is simply reduced by applying one pattern, if general patterns for each class of active services are implemented.
Before describing our survey of these services and their classification, an overview of active networks and the pattern-based service deployment is given.

1.1.1 Active Networks

Traditional computer networks are limited in their functionality. The major task of routers is mainly to deliver packets from one host to another. Processing within the network is limited basically to passive congestion control and quality of service (QOS) [1]. Other tasks like transcoding, active congestion control and fusion of data, which could be computed more efficiently on the nodes between, have to be performed on one of the end hosts. Such "passive" networks lead to some difficulties of integrating new technologies and standards into the shared network infrastructure, poor performance due to redundant operations at several protocol layers and difficulty in accommodating new services in the existing architectural model [1].

Active networks (AN) are a new approach to network architecture. The services are deployed either in-band or out-of-band. In-band deployment on the one hand refers to a system, where the service logic is distributed in the same packet as the payload data. Out-of-band deployment, on the other hand, refers to an architecture where both deployment and payload data use distinct communication channels [2]. Since the deployment in the former is given by the use of the service itself, this thesis considers solely services, which are deployed out-of-band.

Routers can perform computations on active packets, modify the packets content and users are able to include new programs into the network. Today, routers may modify a packet's header, but pass the user data opaquely without examination or modification [3].

Active Routers are based on a classical router. On top of this router exist one or more execution environments (EE) to execute the code or programs of the active packets [2]. If a passive packet arrives at an active node, it will be routed and forwarded in the common way. Only active packets are checked if they contain any program or data to be executed. Active packets should be routeable by passive nodes too, to combine active and passive networks.

1.1.2 Pattern-based Service Deployment

Koon-Seng Lim and Rolf Stadler proposed a pattern-based network management. It is based on the methodical use of distributed control schemes [4]. They specify the key concept of "navigation and aggregator patterns", which describes the flow of control during the execution of a distributed management program. A navigation pattern determines the degree of parallelism and internal synchronization of a distributed management operation. This concept has two main benefits.

- It allows the analysis of management operations with respect to performance and scalability
- Two patterns which belong together allow to separate the semantics of a management operation from the control flow of the operation

Navigation patterns should be as generic as possible to be used for different management tasks, according to different performance objectives. Lim and Stadler developed an approach that will free the management application programmer from developing distributed algorithms, allowing to focus on the specific management task. He only has to select a navigation pattern that captures the requirements for that task [4].
This work is directly influenced by the work of Kim and Stadler because of the idea of the pattern-based approach and the simulation program to develop these patterns.

The structure of a pattern-based management program
A pattern-based management program consists of three abstract object classes.

- A **navigation pattern,** which describes how the messages get from node to node during its execution.
- An **aggregator,** which specifies the operations executed on each node and how the results are aggregated.
- An **operator,** which provides a platform independent interface to state and control variables of a network node.

Our classification uses the navigation and aggregator patterns. The operator provides information of the nodes and is used as a utility.

1.2 Organization

This thesis is organized as follows:
Chapter 2 presents a survey of active services to show the differences to nowadays standards in networking.
Chapter 3 describes our navigation and aggregator patterns, which classify

all the active services described in the last chapter.
In Chapter 4, the simulation and the gained results are shown, which were built competing the patterns against a central approach.
Chapter 5 gives a summary of this thesis and the conclusions. It describes the future work as well.

Chapter 2

Survey of Active Services

This chapter provides a survey of services in active networks. It will be useful to understand the gained flexibility in using active networks. Furthermore, it acts as a base for the classification in the next chapter. This thesis focuses on the deployment of the services and not on the services themselves. Therefore, this chapter solely presents a collection of summaries of other works.

2.1 Active Reliable Multicast

Lehmann, Garland and Tennenhouse describe Active Reliable Multicast (ARM) as follows. "Provide a reliable service for multicast, avoiding NACK explosion and multiple retransmission of the same packet from the source." [5]
The reliable multicast service is provided on a best-effort internetwork. If a NACK occurs in a passive network, it climbs up the multicast tree to the root. On lossy or congested networks, multiple hosts in the same branch may send a NACK for the same packet. All NACK's are then separately sent to the root node. In big multicast trees, that can result in an overload of the source.
An ARM network handles NACK packets differently. If the sequence number of the NACK is not known, it is saved and propagated back to the source. When the requested packet arrives, it is sent to the destination and saved on the node as well. The advantage comes out if another NACK with the same sequence number arrives. It is not propagated because the packet can be immediately resent. This reduces the total traffic in the network. In the downstream direction, active routers reduce the traffic of resent packets and they reduce duplicate NACK's in the upstream direction.

As a result of the last paragraph, the best places to install an ARM service are fork nodes on the multicast tree. Lehman, Garland and Tennenhouse estimated in their paper [5] good results if at least 35% of the nodes, randomly selected, are active.

2.2 Web caching

The idea of Active Reliable Multicast can be generalized. Not only the data of the ARM, but the whole data of the web communication can be stored within the network. This results in a caching of web contents. Data is stored and retransmitted directly from there. Active nodes act like proxy servers in passive networks. Proxy servers are used in big LAN's to control the web traffic that is entering or leaving them. They cache often used websites to reduce the outgoing and incoming traffic [6]. Web caches provide shorter response times, less network traffic, no source overloading and load distribution. They have to hold time stamps next to the data to have a criteria of the up-to-date-ness of the required data. Active routers need a big cache and a sophisticated caching policy.
The placement of active web caches is a sophisticated task, since it is hard to determine the flux of the web traffic. Although supported by the existence of smart places. Imaging a local network with many webservers inside, the overall traffic within can significantly be reduced by placing active web caches at the border.

2.3 Wave Video

Distributing video streams over the Internet poses many difficulties. Two important factors for video transportation are bandwidth and maximum delay [7]. Usual networks can't guarantee neither the one nor the other. In case of congestion, packets are dropped and have to be retransmitted again, without any regards if they will arrive too late or not.
In the case of congestion in an active network, some packets are not only dropped, but the wavelet-based active router plug-in is also able to scale the data rate in a sophisticated manner. It reduces the bandwidth of the video stream by dropping the unimportant information of the video just at the

place where the congestion occurs. The rest of the distribution tree will not be affected by this reduction. The resulting movie is not jerky, but reduced in quality of the details.
To achieve this quality of service, every node where congestion may occur, must be able to reduce the bandwidth of the video stream. Therefore, every node on the path or in the multicast tree, must be active.

2.4 Monitoring

Monitoring is one of the most important services in networks. Someone can not decide how good a network is, if he does not have any valuable information about it. Controlling of each node is necessary to identify the possible point of failures. In general, monitoring is an application that collects some statistical data and publishes it on one single, central machine. To reduce the transmission overhead of the data collection, it is necessary that the application is not polling its sensors. Useless data has to be filtered out as soon as possible and the rest of the data should be as compact as possible. Active nodes have to perform some data fusion and filtering before sending them compactly to the central management station.
To achieve a good compensation between centralization and decentralization, active nodes should collect and collate data by themselves, filtering uninteresting data out and sending a compact summary to the management host. Active nodes can monitor themselves as well as their active or passive neighbors. Best results should be achieved, if as many as possible nodes are active.
The best places for this service are the fork nodes of the established tree.

2.5 Information fusion

Information fusion is similar to monitoring. Information is collected from various nodes and sent to one specific node. The difference lies in the origin of the data. While monitoring controls more the behavior of the network itself, information fusion supervises primarily the applications or services running in this network, e.g. on line auctions: lower, delayed bids can be proceeded directly in the network to free up server resources for competitive bids [8].

Uninteresting, useless data or data that is out of date, has to be filtered out as soon as possible. Information fusion nodes try to reduce the used bandwidth, destination overload and offer a wide range of scalability. As in monitoring, information fusion scales better the more nodes are part of this service. The service is best placed on fork nodes of the established tree.

2.6 Congestion control

Congestion is a big problem in networks. Bottleneck links overfill, the flow of data is congested and results in large delays or interruption of communication [9]. The prediction of congestion is very difficult or impossible and only fast reaction to a congested situation can prevent communication links from interruption.
Nowadays, routing algorithms react quite slowly to congestion. They are optimized to react on topology changes. It is not possible to prevent links from congestion. The only thing you can do is to design your network with adequate bandwidth for the next years and hope it will be enough.
Congestion control in an active network discovers the aggregation of data flows and prevents bottleneck links from overfilling. New routes through a network are chosen fastly and allow communication links to persist. Therefore it is possible to prevent communication failures and reduce the delay for time sensitive transmissions. Active nodes are used as triggers that indicate congestion control, flow state is examined for advice about how to reduce the quantity of data. They aggregate traffic information about themselves and their neighbor nodes. According to reduction techniques (i.e. wavevideo) they prevent links from congestion and choose new routes for data flows. If a network must be prevented from congestion, all routers need to participate in the active service to be efficient.

2.7 Application security gateway

There are two different ways to gain security in communication networks, securing the communication link itself like VPN, or securing the application (i.e. SSH). VPNs will be discussed later in this work.
Application security gateways (ASG) are specialized nodes in a network that handle all traffic for a single application. Each application uses its own

ASG which can exist on the same node as the one for another application. They filter all the outgoing and incoming traffic and decide what to do with it. Such gateways should be as transient as possible for the user, not to complicate the use of the application. A node must be capable to serve as an ASG and there are normally just a few of them, because they are equipped with security hardware. Therefore the service is set up on just one node within the network.

2.8 Transcoding

The sending hosts have to send data in different ways, to allow different systems to access the same data. The network only transmits the provided data packets. If a website should present its content for both HTML and WAP, the same content has to be stored in both formats. The same problem occurs for different video or sound formats (i.e. MJPEG, H.261 and other). Transcoding can solve this problem by converting data somewhere within the network. The gained flexibility can be used by programmers to develop their own transcoding schemes for new services or formats and install them directly in the active network. Transcoding enables to accommodate fixed and mobile network users without the need of storing data in all possible formats.
Active routers between the end hosts only need to know the transcoding scheme from one format or protocol to another. They can choose an appropriate location to transcode the data packets, together with the network status information (bandwidth, delay, free resources). If the needed transcoding scheme is not known on the transcoding node, it can be downloaded from another node or a scheme database and installed just in time.
E. Amir, S. McCanne and R. Katz described such a node as a “Media Gateway” (MeGa) [10].

2.9 Firewall

In the time of daily attacks to computer networks, firewalls have become more and more a must for network security. They implement filters that determine which packets should pass [3]. Firewalls can be implemented in that way they are transparent to the user or not.

While a firewall can be set up quite fast for simple and small networks, their installation and maintenance is very complicated for large networks like a LAN of a university or a multinational company. As the user asks for more flexibility, the administrator will complain about the difficulty in the firewall support. If more than one firewall is used to secure the local network from the Internet, it is very important to hold the firewall rules actual and to check them regularly.
The whole scenario becomes even more complicated, if someone does not want to check only data packets but also complete data streams. Fragmentation divides data into multiple packets, of which each alone is safe, but all together can be dangerous (i.e. viruses). Firewalls need a cache to store incoming packets and check all other packets for the same session. The more traffic a firewall has to handle, the bigger the cache must be.
Active nodes can offer this service perfectly. They offer a cache and also computing power in their EE. As they are placed on usual routers for passive networks, there is no need of extra hardware. Temporary firewalls become also possible and are set up in a very short time, without any interruption of the communication. They can update the implemented rules automatically (with the appropriate security standards) without any operation of an administrator. Any update for new protocols or automated adaption (by allowing applications from approved vendors to authenticate themselves to the firewall and inject the appropriate modules into it) can be performed right in time and not only after a long process of standardization.
A firewall has to be placed and configured at the border of a domain.

2.10 Tunnel

Tunnels are point-to-point connections and used very often to enable secure transactions between two hosts. They have to be set up and closed in a minimum of time to offer flexibility to the user. Until now, the user has to define the two hosts, where he will put up the tunnel. This work could be done by the active network itself, which searches two appropriate nodes to establish a connection between. The only thing the user still has to do, is to define between which networks the tunnel has to be set up and provide some additional parameters (i.e. maximum hops in between, security standard, maximum delay). The active network will automatically search two possible nodes in two different networks and set up the tunnel.

2.11 Overlay

Overlay networks are similar to tunnels. They are placed on top of a usual network, but it seems to be a different network with a different configuration (i.e. grade of networking, number of nodes). The setup is very difficult because the user has to choose the hosts for this network and to send them their neighbor identity.
Active networks install overlay networks automatically. The user has to define network regions where overlay nodes must be placed (i.e. company LANs) and some additional parameters (i.e. grade of networking). The deployment of the service will search for appropriate nodes in the specified areas and connect them.

2.12 VPN

VPNs are a kind of overlay networks or tunnels. They offer a grade of security for data transmission. Users can apply them by authenticating people or hosts. All data transactions are encrypted in between the end points of this virtual network (which is sitting on top of a communication network).
Active nodes can be used very easily to implement a VPN. The authentication is done as in traditional VPNs, but offering more flexibility, especially for mobile users (routers could hand over connections transparently to the user). To set up a fixed or temporary VPN, no new hardware is needed. Only enough free resources on the active nodes must be provided to use the appropriate standard of security and the service is ready to run.

Chapter 3

Service deployment patterns

This chapter describes service deployment patterns, which classify the services listed in the last chapter. The first section describes the representation used to describe navigation and aggregator pattern. The next section introduces the classification scheme, and the last sections finally describe all navigation and aggregator patterns and assign appropriate services to them. The table at the end of the chapter summarizes the classification.

3.1 Representation of patterns

The key concept of pattern-based service deployment was introduced in section 1.1.2 and is developed in [4]. This section presents some parts which are needed to understand the way of of how navigation and aggregator pattern are described in this thesis in detail.

3.1.1 Navigation patterns

The navigation pattern describes the flow of messages. Therefore, it needs to know its location. As otherwise, it can not deliver messages properly. Furthermore, it must take care of messages that have visited its location. To support these two requirements, a navigation pattern can be best implemented on a network node as a Finite State Machine (FSM), which is triggered by messages. This was shown by Kim and Stadler in [4].
This chapter presents a FSM for each navigation pattern and the parameters needed for the proper navigation.

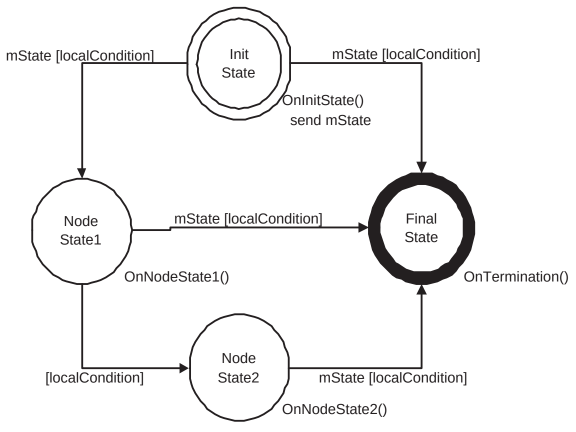


Figure 3.1: The type of state transition

Figure 3.1 shows an example of such a FSM. The boundaries of the states specify their type as initial, if there are two circles, as final with a bold circle or as normal with a single narrow circle. The transition from one state to another depends on the *mState*, the mobile state of the arriving message and local conditions which are indicated in square parentheses afterwards. If there is only a condition at the edge, it acts as a guard. This means that the transition to that state takes place if the condition is true and no message is needed for triggering.
If a message arrives at a node in a state where no handling for that *mState* message is given, no transition occures and the node remains in the same state.
We developed base navigation patterns and patterns which are inherited from those. The meaning of inheritance in this navigation pattern-based view is the joint of all states into one FSM, except **XXXPatternTerminated** and **XXXBegin**. They are private states and **XXX** indicates a certain pattern. Each pattern has them as its own private states.
At inheritance, transitions and conditions between states from parent patterns have to be added or adapted. The merging of the states is explained in the corresponding navigation pattern description. The resulting hierarchy is shown in Figure 3.2.

On the right bottom of each state, the aggregator function is indicated. The FSM's shows all possible aggregator functions. This aggregator functions are executed solely on entering or reentering the state by a transition.

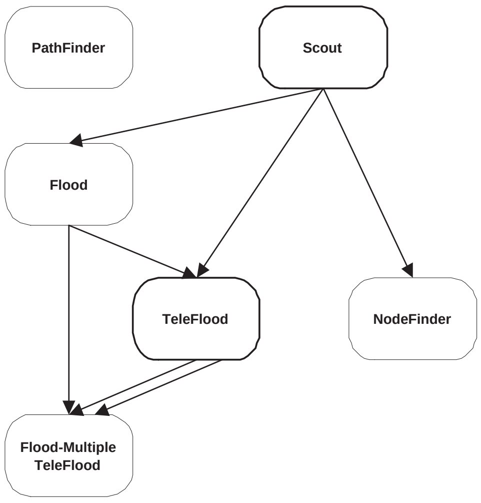


Figure 3.2: The hierarchy of the navigation patterns

The actions of the navigation pattern is indicated below the aggregator function.

3.1.2 Aggregator patterns

Aggregator patterns define the operations executed on each state and the way how the information is aggregated. This makes them dependent on their corresponding navigator, although the pattern hierarchy allows an inheriting navigation pattern to use the aggregator of its parents.
There is also a inheritance tree of aggregator patterns, as shown in Figure 3.3. Inheriting methods in an aggregator based view means, that all functions keep their functionality. Furthermore, every function is inherited or overridden, except XXXPatternTermination() and XXXBegin(), for the same reason as explained above.
The description of aggregator patterns in this chapter includes identifying states with their functions, which must be implemented, and describing their scope.

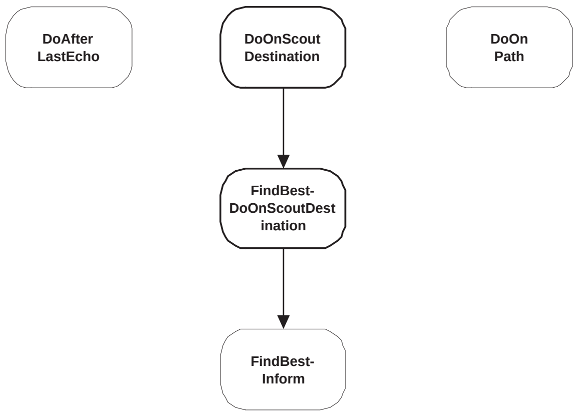


Figure 3.3: The hierarchy of the aggregator patterns

3.2 Classification of active services

This section introduces our classification scheme, which groups services according to their type of deployment in the network.
Since two different patterns have to be implemented, it results in a two dimensional classification.
On the one hand, there is the navigation classification. It groups the services according to the topology of the nodes involved. There are e.g. services which need one node in the local domain and a second one in another domain. This must be distinguished from a further service, which requires all nodes in between for the execution of the service.
On the other hand, the aggregator dimension of the classification classifies what the deployment has to do on the nodes. For example, it puts a service that needs execution on each node into another class, than a deployment that executes a method just on one single node.
The following patterns build the classification of all introduced services in the previous chapter. Each pattern description lists its class of deployment. A list of all services and their corresponding pattern classification is given at the end of this chapter in table 3.1.

3.3 Scout pattern

The first pattern to be described is the Scout navigation pattern. Its FSM is depicted in Figure 3.4. The main goal of this pattern is travelling from a

start node to a destination node and returning back to the origin. The work to be done on each node, to collect and distribute information, is the scope of a suitable aggregator. It is described in the next section.
At the very begin, each node is in the **Init** state. On an arriving message, the FSM changes into one of the four states. The two states with the underlined names are used by nodes where the travelling message passes. They have no impact on the pattern and are only shown for the reason of completeness. They can be passive nodes as well and are omitted in the following.
The pattern starts as soon as the first message *mScoutBegin* reaches a node, which becomes the start and administration node of the pattern. It jumps into **ScoutBegin** and sends out the Scout message with the mobile state *mScout*. This message goes its way towards the destination according to the routing algorithm in the network. It brings each visited node into the state **Travel** as long as it has not reached the destination node and the node is active. Reaching the target causes the node to jump into **Scout-Destination**, which invokes the return trip. Back on the start node, the *mScoutBack* triggers the FSM from **ScoutBegin** into **ScoutTerminated**. This state is used from an aggregator to clean up the pattern and to obtain the ability of inheritance. Without any conditions, the final and private state **ScoutPatternTermination** is triggered and the pattern ends.
The pattern must be started with a parameter, that specifies the destination of the *mScout*. In nowadays networks, that is an IP address.
This class of services has two nodes involved. On on side, it is the start node and on the other side it is a distant destination node. The path between has no relevance to the service and may consist of active or passive nodes. Actually, no services are assigned to this class. Although, it is a navigation class, because it is used for several inheritances.

3.4 DoOnScoutDestination pattern

The preceding section described the Scout navigation pattern. This section introduces a corresponding aggregator.
As it is shown in the following patterns, there are a lot of situations where some computed data has to be displaced to a far node, or a node has to be notified about a decision another node had made. This is best solved with a message that triggers a function with some parameters on that node. The

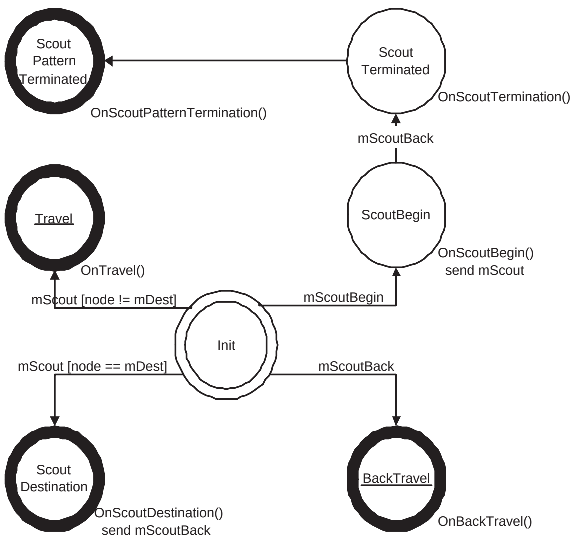


Figure 3.4: FSM of the Scout navigation pattern

following methods are implemented by this pattern:

OnScoutBegin() This method is executed at the beginning of the Scout pattern. Therefore, it identifies the number and destination of all *mScout* messages that it has to send. In addition, it sets a unique pattern identification to differentiate it from other patterns.

OnScoutDestination() This function is used to execute a piece of code on the destination node or to let the node know, that the arrived message contains data which has to be handled. It is also possible to start an external application with it.

OnScoutTermination() This function is called when the *mScoutBack* returns to the start node. It is responsible to handle the content of the message e.g. evaluating the success of the deployment.

OnScoutPatternTermination() Since this method is executed on the start node after the pattern has terminated, it is used to communicate with the application which called the deployment pattern. It is intended for implementation dependent resource freeing.

This pattern, being in its not-inherited appearance, is not used by any services. It is listed anyway, because it tightly fits to the Scout navigation pattern and is used for inheritance. It classifies services deployments which need to notify or get some information about other nodes in distance.

3.5 Flood pattern

The Flood pattern is a navigation pattern and inherits the states from the Scout pattern.
It's outlet is divided into three steps. First, a group of nodes is flooded. This group is either a restricted network like a domain or an overlay network of a multicast tree. The second step lets messages gathering together in the shape of a tree. This makes sure to visit all nodes and an aggregator is able to gather information properly. The last step is the inherited Scout pattern and aims to one or more nodes selected by the aggregator. This proceeding is called gather-compute-scatter and was taken out of the work of Y. Chae, S. Merugu et al. in [11]. It is used in several of the following deployment patterns.
The flood pattern is implemented in the FSM in Figure 3.5. At the begin, the start node changes into **FloodBegin** and sends out a *mExplorer* message to each connected link if they are within the group. This is actually the job of every node when the first *mExplorer* arrives and it jumps into **FirstExplorer**. The first *mEcho* triggers the node into **EchoOnNode**, where the node waits for all *mEcho* messages from its neighbors of the group, except the explored link. The explored link is defined as the link where *mExplorer* arrived. If a subsequent *mExplorer* visits the node, it has to notify the sender that it has already been explored by sending a *mExplored*. This is in the state **SubsequentExplorerOnNode**.
The nodes have their jobs done and terminate at **NodeTerminated** in two different cases. Either if all *mEcho* messages have arrived or if they are leaf nodes. That is the case if they have no neighbors in the same domain. In this termination state, the node has to send a *mEcho* on the explored link, which leads the pattern to gather in the shape of the desired tree.
If the start node has collected all *mEcho* messages in **EchoOnStartNode** as well, it changes into **FloodTerminated**. A corresponding aggregator pattern sets the boolean NoScout, which decides whether or not to send

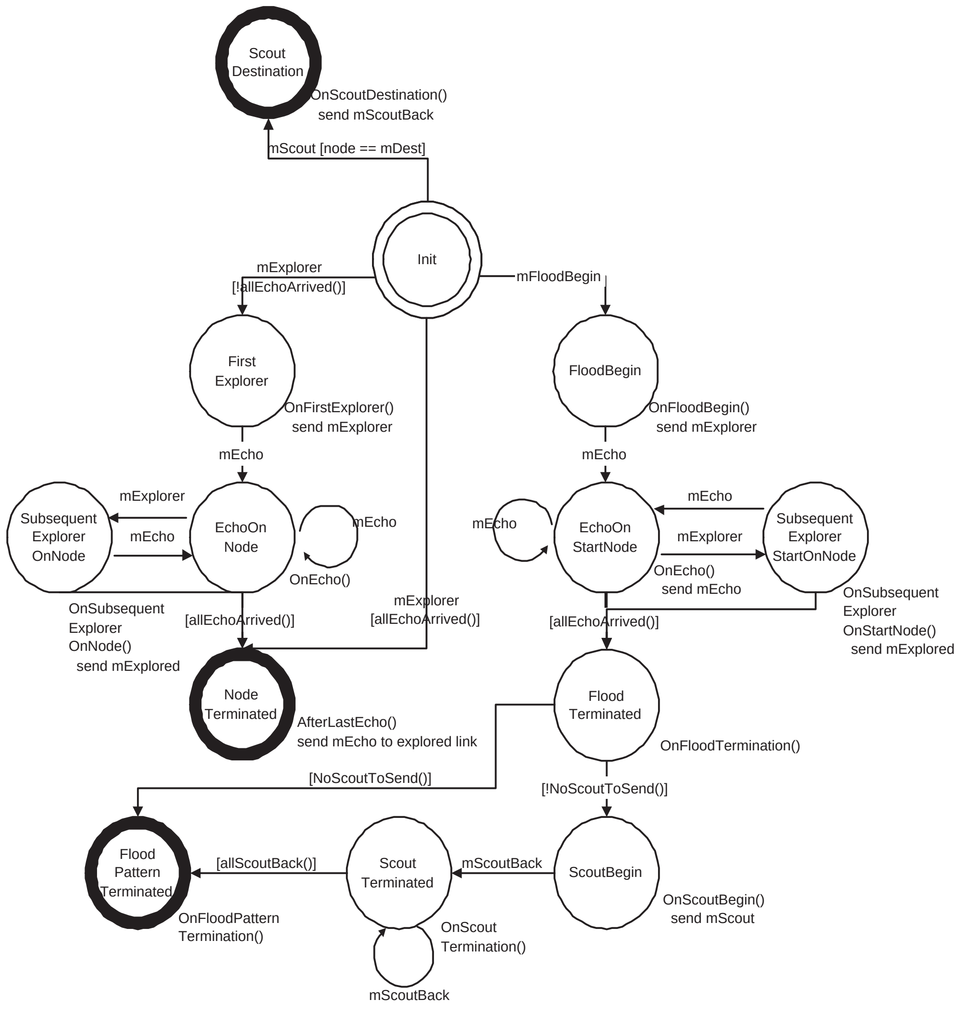


Figure 3.5: FSM of the Flood navigation pattern

mScout messages. If it is TRUE, *mScout's* are sent as the last step of the Flood pattern. As soon as the Scout pattern comes to the end, the FSM on the start node terminates in **FloodPatternTerminated**. At this point, every FSM on each node has terminated. The kind of information and how it is collected from each node is within the scope of the aggregator and described in the next section.
The only parameter that must be passed to the Flood pattern is a domain identity. It is used by the nodes to identify their neighbors, since *mExplorer* messages are only sent within the desired area. Furthermore, all nodes have to calculate the numbers of *mEcho* they are waiting for, based on the domain identity of their neighbors.
This pattern classifies the deployment of the services in the navigation dimension. It characterizes its members due to the fact that it floods an entire domain and visits every node. Therefore all service deployments in this class search or use specific nodes in a group.
A member is e.g. Active Reliable Multicast. The use of the flood pattern is based on the assumption that the multicast tree has already been established and acts as an overlay network.
Another active service is Congestion Control. Since an administrator of a local network wants to deploy this service in his network, the flood pattern is used to visit all nodes for starting and configuring.
The installation or configuration of an Application Security Gateway (ASG) also needs to check each node for its ability to act as an ASG. Furthermore it finally has to install the application on one able node.
Each of this deployment examples uses the Flood pattern as the navigation deployment scheme. The entire list of all services is provided in the table 3.1 at the end of this chapter.

3.6 FindBest-DoOnScoutDestination pattern

The FindBest-DoOnScoutDestination pattern is an aggregator and fits all navigation patterns that are inherited from the Flood pattern. Moreover, it inherits all the functions of the DoOnScoutDestination aggregator.
Its purpose is finding appropriate nodes in the flooded area and deploy some information to them.
The pattern needs a simple procedure how to rate all competing nodes

against each other. Proposed is a local method on the platform independent operator, which can be executed by the aggregator and returns a scalar or just a binary. Indeed, multiple parameters may be passed to the pattern which are forwarded to this function. In the following, they will be called property.
The following methods are needed from the Flood pattern above:

OnEcho() This function is called on an arriving *mEcho*, which either contains references to the searched nodes of its part of the branch or is empty. Therefore, OnEcho() has to store this value plus the references to the nodes.

AfterLastEcho() This method is called after collecting all *mEcho* messages. It is responsible for evaluating all nodes from the collected *mEcho* messages and itself. The result is attached to the new *mEcho* message, which is sent down to the root by the navigation pattern. This function may contain implementation specific code. This is freeing allocated resources.

OnFloodTermination() When each branch reported its best node to the start node, this method is called. It is used to do the last evaluation and to select nodes to pass them to the Scout pattern.

OnFloodPatternTermination() This function is called when the Flood pattern terminates. Therefore, it reports the termination of the pattern or unlocks the application which is waiting for the full establishment of the pattern.

Inherited functions Function OnScoutDestination(), OnScoutBegin() and OnScoutTermination() are inherited from the Scout pattern and suffice the same requirements.

Administrative functions All functions above contain implementation specific code. In special, OnFirstExplorer() and OnFloodBegin() may prepare the node for the forthcoming evaluation and the arriving of *mEcho* messages.

The only parameters, that must be passed to the aggregator are the properties of the node to search. That may be requirements for installed software, plugged hardware components or processor usage of the nodes. This list is not complete and depends on the application.

3.7 DoAfterLastEcho

DoAfterLastEcho is also an aggregator and fits with all children of the Flood navigation pattern. Its purpose is to execute a method on well specified nodes in the flood area. It does not make use of the Scout pattern as the third step mentioned above. Therefore the boolean NoScout is constantly set to TRUE. The nodes, on which to execute some code, are set by a parameter, that is passed to the aggregator. Possible values are ALLNODES, BORDERNODES and FORKNODES. ALLNODES executes the method on each node and BORDERNODES on all nodes that have at least one node in another domain. Fork nodes are identified as nodes which have received at least two *mEcho* messages from other nodes and are specified with the parameter FORKNODES. This parameter is passed further to the AfterLastEcho() function.
The following methods need to be implemented:

AfterLastEcho() The Flood navigation pattern calls this function either just after the last *mEcho* arrived at the node or, if the node has only neighbors from other domains, on the arrival of the first *mExplorer* message. Therefore the function can distinguish whether it is a leave node, whether it is a fork node in a multicast tree, or if it is a border router of a specified network. These tree specifications are the possible parameters mentioned above.

OnEcho() This method must be implemented if the success of the execution on each node has to be reported. In this case, it is the same method as in the previous pattern.

OnFirstExplorer() This method provides a speed optimization if the execution takes a long time. One disadvantage of code execution in this state is, that the node does not know yet which part of the tree it will become.

All other functions need no implementation and return without any computation.

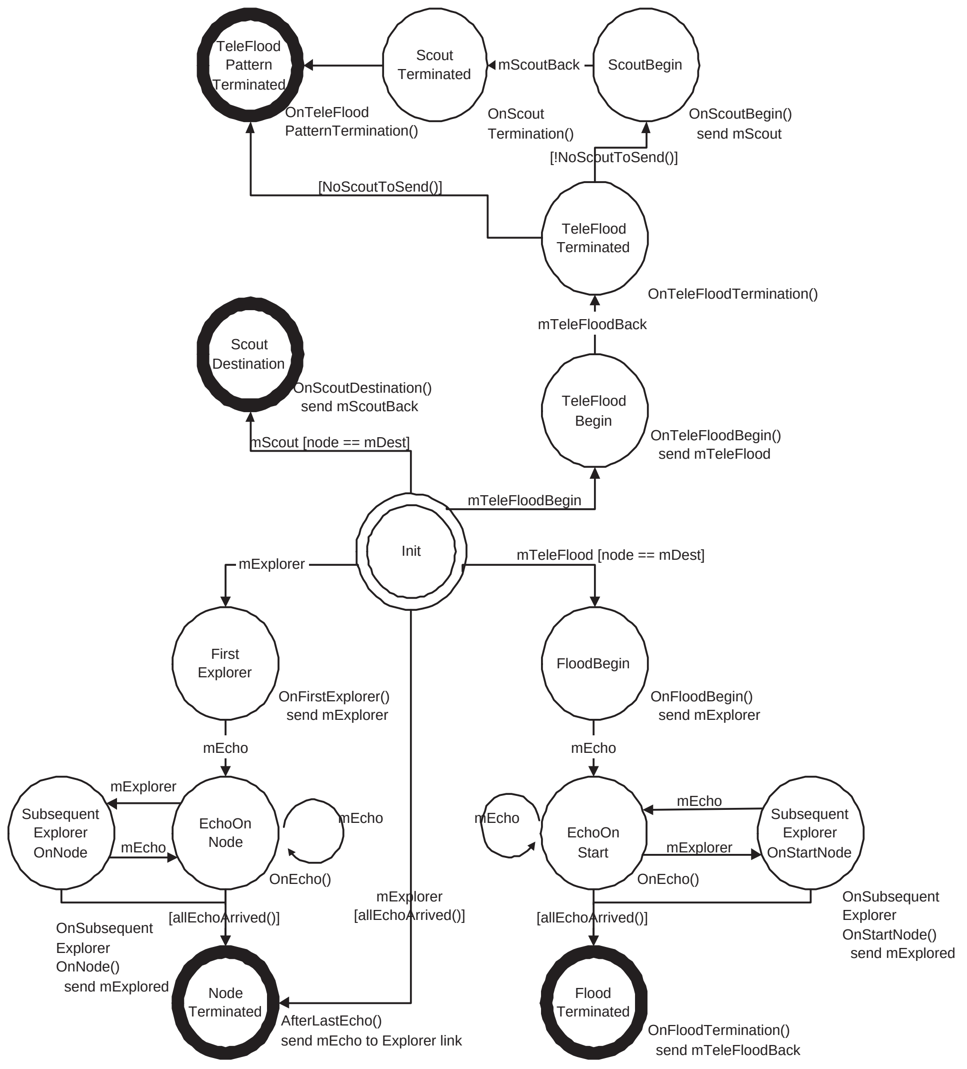
Service deployment patterns performing a task on each node, depending on its location in the network, take part of this class. Nowadays, active services need to decide between border and all nodes. In the case of multicast, good algorithms exist to build a tree to act as an overlay network. This makes all multicast service deployment to be treated as a normal network.
This is for example ARM, which ideally starts the ARM service on each fork node in the multicast tree. In contrast, the deployment of the active Congestion Control is not constructed upon an overlay and needs to be deployed to each node on the network.

3.8 TeleFlood pattern

The TeleFlood pattern is a navigation pattern. It is similar to the flood pattern, except that the start node is not part of the flood area. This pattern does not classify the service deployment, but leads to the background of the following navigation pattern.
The fact that the start node is not in the flood area, but still must be the origin of a possible Scout pattern, makes some changes necessary. The FSM is depicted in Figure 3.6. The termination of the flooding in the far domain must trigger an *mTeleFloodBack* message to the pattern start node with the aggregated information. Therefore, the FSM terminates there on **Flood-Terminated**. The message travels back to the start node and triggers it into **TeleFloodTerminated**. At this state, NoScout() is called. If it returns TRUE, Scouts are sent and on FALSE, the pattern terminates without a Scout. Hence, the moving of the deciding node is the only change from the Flood pattern and the rest of the functionality of the FSM remains. The new state **TeleFloodPatternTerminated** comes to the place of **Flood-PatternTerminated** and is triggered as soon as the *mScoutBack* returns to the start node. The only parameter that must be passed to the pattern is the foreign domain identity to specify the destination of the *mTeleFlood* and to restrict the flooding.

3.9 Flood-MultipleTeleFlood pattern

Flood-MultipleTeleFlood is also a navigation pattern. As its name already betrays, two or more domains are flooded. Both, the local and, as already



TeleFlood Pattern Terminated
Scout Terminated
mScoutBack
ScoutBegin
OnTeleFlood PatternTermination()
OnScout Termination()
OnScoutBegin() send mScout
[!NoScoutToSend()]
[NoScoutToSend()]
TeleFlood Terminated
OnTeleFloodTermination()
mTeleFloodBack
Scout Destination
OnScoutDestination() send mScoutBack
TeleFlood Begin
OnTeleFloodBegin() send mTeleFlood
mScout [node == mDest]
mTeleFloodBegin
Init
mExplorer
mTeleFlood [node == mDest]
First Explorer
OnFirstExplorer() send mExplorer
FloodBegin
OnFloodBegin() send mExplorer
mEcho
mEcho
mExplorer
Subsequent Explorer OnNode
mEcho
EchoOn Node
mEcho
OnEcho()
mEcho
EchoOn Start
mEcho
mExplorer
Subsequent Explorer OnStartNode
OnEcho()
OnSubsequent Explorer OnStartNode() send mExplored
mExplorer [allEchoArrived()]
[allEchoArrived()]
[allEchoArrived()]
OnSubsequent Explorer OnNode() send mExplored
Node Terminated
AfterLastEcho() send mEcho to Explorer link
Flood Terminated
OnFloodTermination() send mTeleFloodBack

Figure 3.6: FSM of the TeleFlood navigation pattern

introduced in the previous section, the distant domains reached by a *mTele-Flood* message. Since these two parts have already been introduced, the new pattern can be built by inheritance.
The resulting FSM, as shown in Figure 3.7, is described in the following and corresponds in general to the joining of all parent states. Three new termination states and one adapted for the beginning have been added.
FloodMultipleTeleFloodBegin is the new beginning state on the pattern start node and activates the deployment. All *mTeleFlood* and *mExplorer* are sent out. Whenever a local Flood message or a TeleFlood response returns to the start node, it is triggered into the corresponding state taken from its parent pattern. **FloodTerminated** is triggered when the local flooding has terminated and **MultipleTeleFloodTerminated** when all *mTeleFloodBack* have flooded their domains. Both states need to be visited to cause the FSM to jump into **Flood-MultipleTeleFloodTerminated**. At this state, every node of all desired domains have been flooded and a corresponding aggregator is able to decide whether and where to send *mScout* messages. The entire FSM terminates in **Flood-MultipleTeleFloodPatternTerminated** when all these *mScoutBack* messages have returned.
Parameters to be passed to this navigation pattern are the domain identities solely.
This navigation pattern classifies all service deployments, which need to gather information on each node of more than one domain and after that, revisit some picked nodes. This is for example a construction of an overlay between several domains. The service deployment pattern needs to figure out the nodes and to inform each about its neighbors. More examples are depicted in the table.

3.10 FindBest-Inform pattern

The FindBest-Inform aggregator pattern corresponds to the Flood-Multiple-TeleFlood navigation pattern. It classifies deployment patterns of services in the information flow dimension. Multiple nodes selected out of domains, are notified about their peers. One service in a classical manner is the deployment of an overlay network.
The following methods need to be implemented:

OnFlood-MultipleTeleFloodBegin() At the beginning, this function iden-

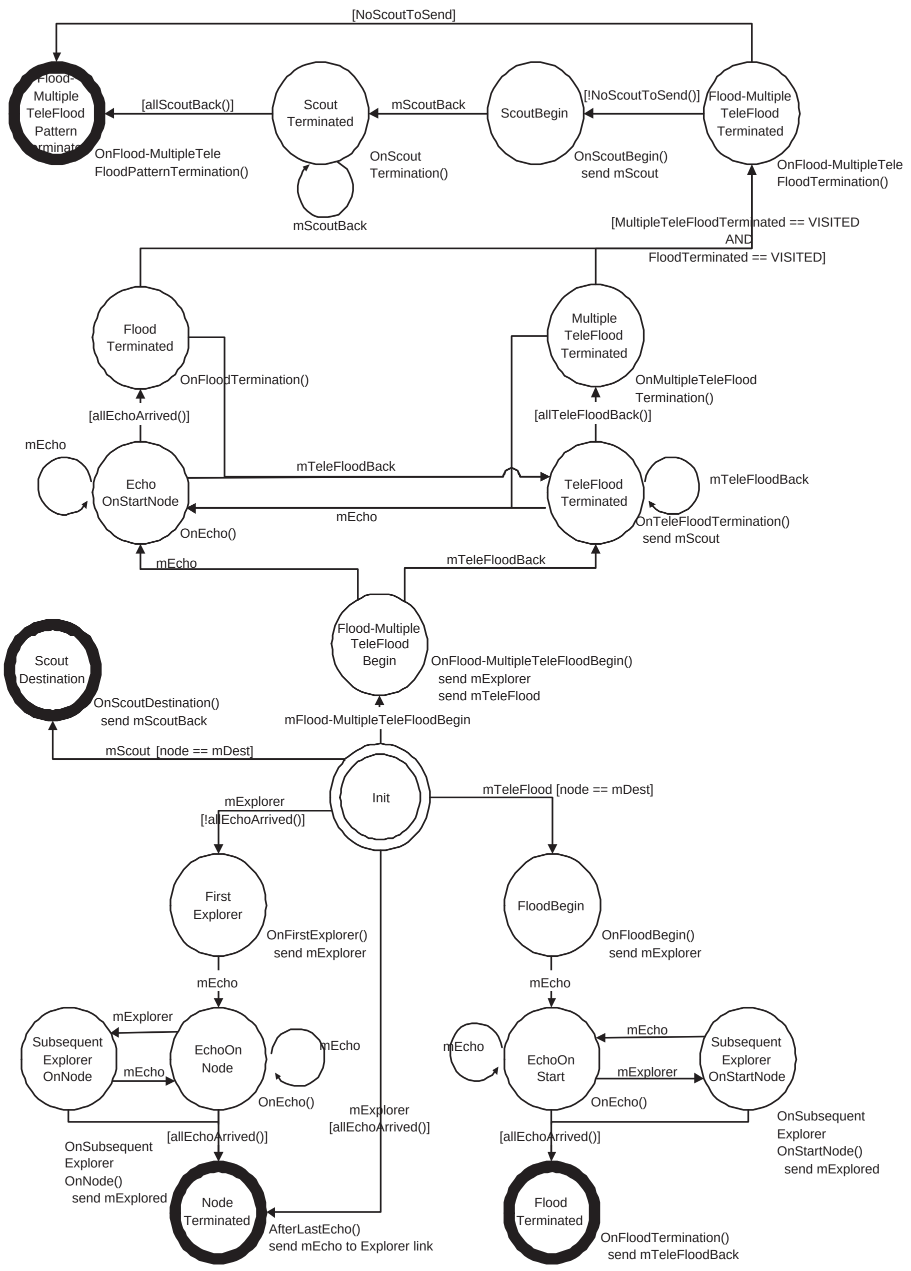

[NoScoutToSend]
Flood-Multiple TeleFlood Pattern Terminated
[allScoutBack()]
Scout Terminated
mScoutBack
ScoutBegin
[!NoScoutToSend()]
Flood-Multiple TeleFlood Terminated
OnFlood-MultipleTele FloodPatternTermination()
OnScout Termination()
OnScoutBegin() send mScout
OnFlood-MultipleTele FloodTermination()
mScoutBack
[MultipleTeleFloodTerminated == VISITED AND FloodTerminated == VISITED]
Flood Terminated
Multiple TeleFlood Terminated
OnFloodTermination()
OnMultipleTeleFlood Termination()
[allEchoArrived()]
[allTeleFloodBack()]
mEcho
Echo OnStartNode
mTeleFloodBack
TeleFlood Terminated
mTeleFloodBack
mEcho
OnEcho()
OnTeleFloodTermination() send mScout
mEcho
mTeleFloodBack
Scout Destination
Flood-Multiple TeleFlood Begin
OnFlood-MultipleTeleFloodBegin() send mExplorer send mTeleFlood
OnScoutDestination() send mScoutBack
mFlood-MultipleTeleFloodBegin
mScout [node == mDest]
Init
mTeleFlood [node == mDest]
mExplorer [!allEchoArrived()]
First Explorer
FloodBegin
OnFirstExplorer() send mExplorer
OnFloodBegin() send mExplorer
mEcho
mEcho
mExplorer
Subsequent Explorer OnNode
EchoOn Node
mEcho
mEcho
mEcho
EchoOn Start
mEcho
Subsequent Explorer OnStartNode
mExplorer
OnEcho()
mExplorer [allEchoArrived()]
OnEcho()
OnSubsequent Explorer OnStartNode() send mExplored
[allEchoArrived()]
[allEchoArrived()]
OnSubsequent Explorer OnNode() send mExplored
Node Terminated
AfterLastEcho() send mEcho to Explorer link
Flood Terminated
OnFloodTermination() send mTeleFloodBack

Figure 3.7: FSM of the Flood-MultipleTeleFlood navigation pattern

tifies where to send all the *mTeleFlood* messages. It allocates resources as well to handle the returned messages.

OnTeleFloodTermination() It evaluates the arrived *mTeleFloodBack* and makes them readable for the following method.

OnFlood-MultipleTeleFloodTermination() This function is called when the local flooding and all teleflooding information has been aggregated. It therefore prepares the information to be sent out with the Scout pattern.

OnFlood-MultipleTeleFloodPatternTermination() Since this function is a PatternTermination function, it is responsible to communicate with the application which called the pattern.

Inherited functions All methods from the FindBest-DoOnScoutDestination are inherited: OnEcho(), AfterLastEcho(), OnFloodTermination(), OnScoutDestination(), OnScoutTermination(), OnScoutBegin() OnFirstExplorer() and OnFloodBegin().

3.11 PathFinder pattern

Until this point, navigation patterns were used for investigating nodes in one or more closed domains. That forced the navigation pattern to visit each node and the aggregator to carry that information back to the start node. This was done by building a tree out of the network topology. The PathFinder works different in the way that it does not collect information of each node, but searches a path from one node to another.
The criteria how the path has to be searched is given by the local operator method **Ability()**. It must be implemented to return TRUE if the node can become a member of the path.
The FSM of the PathFinder pattern is shown in Figure 3.8. The pattern starts by sending *mPathFinderBegin* to the start node, which begins to find a way through the network by sending a *mTrial* message into the direction of the destination node. This forces a node to check its ability and to send a *mFlop* back on FALSE. On TRUE, it propagates the finding of the path by passing the *mTrial* on. A negative *mFlop* message tells the prior node there is no way on that link and triggers a new *mTrial* on another link. *mFlop*

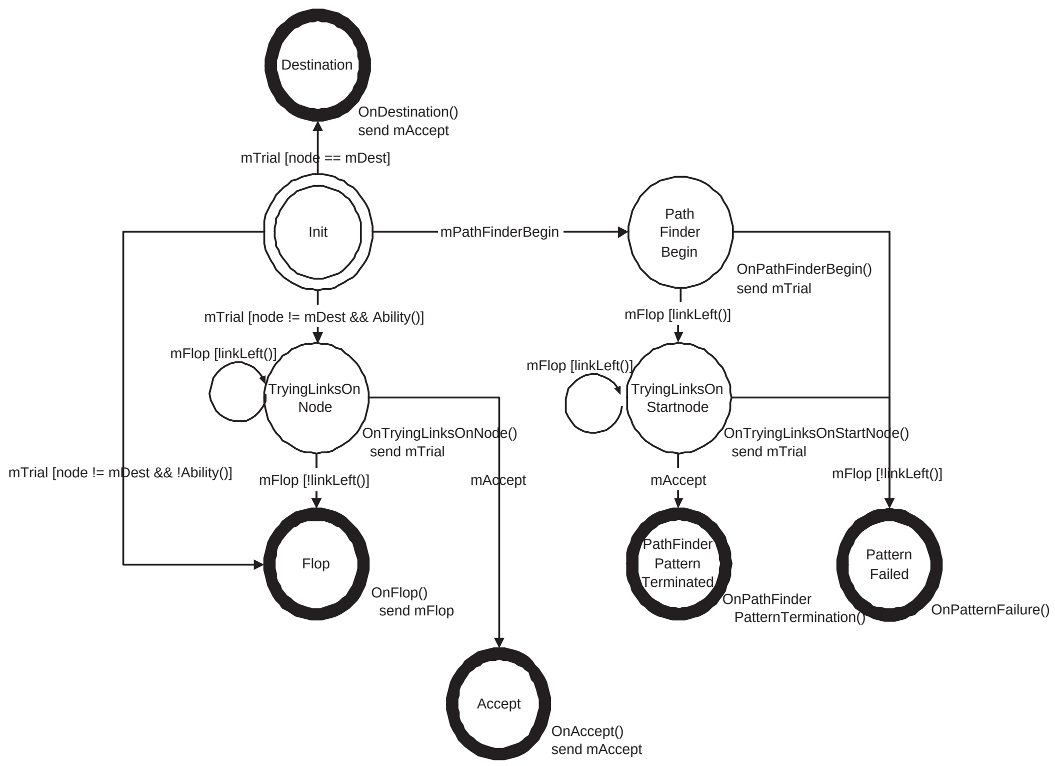


Figure 3.8: FSM of the Pathfinder navigation pattern

messages are also sent if there are no more links available, except the link where the *mTrial* cames in.

As soon as *mTrial* has reached the Destination node, the pattern has found a way corresponding to the previously defined **Accept()** method. All the nodes on the path are still in the state **TryingLinksOnNode** and must be informed about the success. Therefore, a *mAccept* message is sent to propagate back to the start node and trigger each node into **Accept** state. The PathFinder navigates to a destination which must be passed as a parameter. Since the PathFinder is a navigation pattern, it classifies on the navigation dimension. Its members need to have a path through the network, on which each node satisfies a given criteria.

An example is the deployment of the WaveVideo application. It needs to run its service on each node from the source to the destination.

3.12 DoOnPath pattern

The aggregator that fits to the previous navigation pattern is DoOnPath. If the PathFinder travels to the destination, it leaves each node in **TryingLinksOnNode**. To build the path, each node gets the knowledge about its neighbors, when the *mAccept* propagates back. On one side, it is where the message came from and on the other side, it is the link which triggered it into **TryingLinksOnNode**.
It implements the following methods of the previous navigation pattern:

OnAccept() is called from the Scout navigation pattern when it travels back to the start node. It may reserve some resources needed for the service and establish a firm connection to its neighbors on the path. Furthermore it must attach a reference to itself on the outgoing *mAccept*.

TryingLinksOnNode()/TryingLinksOnStartNode() They are called if the pattern tries to make this node a part of the path. Therefore, these functions reserve some resources, if needed by the service.

OnFlop() This method is called if there exists no way through the network including this node. It is used to free possibly reserved resources.

OnPatternFailure() This method is called when the pattern was not able to construct a path to a given destination. It reports a failure to the application, from which the pattern was called.

OnPathFinderPatternTermination() If the pattern successfully ends, this function is called. It reports this to the application.

The parameter, which must be passed to the PathFinder is the constraint of the nodes. It is the parameter of the Ability() function as mentioned above. This pattern is the only aggregator that corresponds to the PathFinder and is used for the same services.

3.13 NodeFinder pattern

The NodeFinder navigation pattern is similar to the one discussed in the preceding section. The PathFinder is searching for a path through the network, where each node must have a special ability. In contrast, this pattern

is searching for a number of nodes with defined abilities, which do not have to be directly connected. This may be the case if the desired node ability is very rare and no direct connection of the nodes is necessary. This is used in the deployment of an active transcoding service. If, for example, a webserver sends data to a PDA and both do not speak the same protocol, a node in the middle must be found to transcode the data. The final result of the NodeFinder navigation pattern is therefore an overlay network from a starting node to the destination node, where each node has a previously defined ability. A parameter that must be passed is the address of the destination node.
Before going into detail, the next paragraph shows the concept of this pattern.
Supposing the aim to have an overlay network from the start node to the end node over another node, which has to convert the traffic data to make it readable for the destination. A way to find such a middle node with the required ability, is to send a message to a node in the middle and flood the environment from there. The flood scope must be constantly increased, till it finds an adequate node. This action is called Bubbling due to the fact, it looks like the middle node is building increasing bubbles. The algorithm for this is taken from the Scope navigation pattern of the tutorial of the SIMPSON simulator [12] and is based on the Chang algorithm. Finally, after reporting the result to the two other nodes, the overlay can be established.
This is exactly how the FSM in Figure 3.9 works. As well as each navigation pattern, it is started by sending *mNodeFinderBegin* to the start node. It does an evaluation where to send *mScout* messages and waits for their *mScoutBack*. There are many possible destinations to send the Scout pattern to. It depends on the application and also on the network topology. A general solution is to divide the number of hops through the number of desired nodes. This can be done with the Scout navigation pattern and a counting aggregator.
If the *mScout* messages reach their destinations, they trigger the node into **IncreaseScope**, which becomes the administrator node of the Bubbling. The node remains in this state untill all *mEcho* got back. Depending on whether a node was found or not, it either increases the mobile variable *m_MaxScope* and continues the Bubbling or terminates and reports its decision to all reserved nodes and to the start node.

The rest of the FSM is handling the Bubbling and the reservation of the nodes. It is inspired by the Flood navigation pattern in section 3.5. If an *mExplorer* arrives and the mobile *m_Scope* variable is bigger than *m_Max-Scope*, the scope is reached and an *mEcho* is sent back on the *mExplorer* link. Otherwise it increases *m_Scope* and continues the flooding. The pattern gathers like the Flood pattern in the type of a tree. Each node checks his **Ability()**, which the pattern is looking for, before it reports its part of the tree back to the root node. In case of FALSE, the node has no more to do as to send *mEcho*. On TRUE, the node jumps into **PreReservation** and waits for the decision of the Bubbling administration node. This is either a *mForget*, if it decided to take another node for this task, or a *mReserve*, if this node has been selected. In this case, the node still has to wait for the *mScout* message of the initial start node. It contains information of the neighbors, necessarily to establish an overlay network.

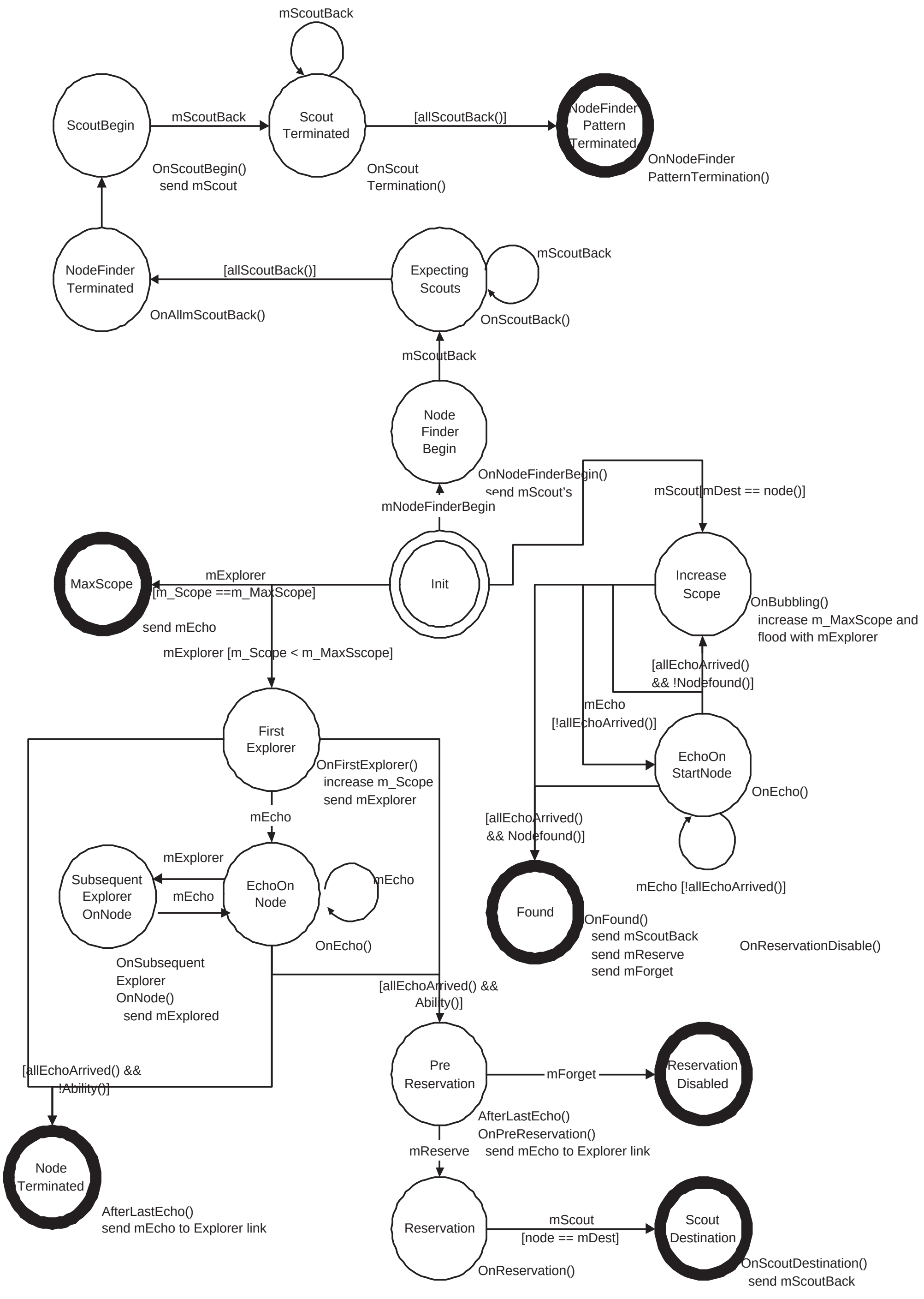

mScoutBack
ScoutBegin
mScoutBack
Scout Terminated
[allScoutBack()]
NodeFinder Pattern Terminated
OnScoutBegin() send mScout
OnScout Termination()
OnNodeFinder PatternTermination()
NodeFinder Terminated
[allScoutBack()]
Expecting Scouts
mScoutBack
OnAllmScoutBack()
OnScoutBack()
mScoutBack
Node Finder Begin
OnNodeFinderBegin() send mScout's
mNodeFinderBegin
mScout[mDest == node()]
MaxScope
mExplorer [m_Scope ==m_MaxScope]
Init
Increase Scope
send mEcho
OnBubbling() increase m_MaxScope and flood with mExplorer
mExplorer [m_Scope < m_MaxSscope]
[allEchoArrived() && !Nodefound()]
mEcho [!allEchoArrived()]
First Explorer
OnFirstExplorer() increase m_Scope send mExplorer
EchoOn StartNode
OnEcho()
mEcho
[allEchoArrived() && Nodefound()]
mExplorer
Subsequent Explorer OnNode
mEcho
EchoOn Node
mEcho
mEcho [!allEchoArrived()]
Found
OnFound() send mScoutBack send mReserve send mForget
OnReservationDisable()
OnEcho()
OnSubsequent Explorer OnNode() send mExplored
[allEchoArrived() && Ability()]
Pre Reservation
mForget
Reservation Disabled
[allEchoArrived() && !Ability()]
AfterLastEcho() OnPreReservation() send mEcho to Explorer link
mReserve
Node Terminated
AfterLastEcho() send mEcho to Explorer link
Reservation
mScout [node == mDest]
Scout Destination
OnReservation()
OnScoutDestination() send mScoutBack

Figure 3.9: FSM of the NodeFinder navigation pattern

Service	Navigation Pattern	Parameters	Aggregator Pattern	Parameters
Active Reliable Multicast	Flood	Multicast ID	DoAfterLastEcho	FORKNODES
Concast	Flood	Domain ID	DoAfterLastEcho	FORKNODES
WaveVideo				
- as multicast	Flood	Multicast ID	DoAfterLastEcho	ALLNODES
- as unicast	PathFinder	Destination	DoOnAccept	ALLNODES
Web Caching	Flood	Domain ID	DoAfterLastEcho	BORDERNODES
Monitoring	Flood	Multicast ID	DoAfterLastEcho	FORKNODES
Congestion Control	Flood	Domain ID	DoAfterLastEcho	ALLNODES
Firewall	Flood	Domain ID	DoAfterLastEcho	BORDERNODES
Application security gateway	Flood	Domain ID	FindBest-DoOnScoutDest	property
Transcoding				
- between networks	Flood-MultipleTeleFlood	Domain ID	DoAfterLastEcho	BORDERNODES
- all nodes on a path	PathFinder	Destination	DoOnAccept	property
- on a few nodes	NodeFinder	Destination	FindBest-DoOnScoutDest	array of properties
Tunnel	Flood-MultipleTeleFlood	Domain IDs	FindBest-Inform	property
Overlay	Flood-MultipleTeleFlood	Domain IDs	FindBest-Inform	property
VPN	Flood-MultipleTeleFlood	Domain IDs	FindBest-Inform	property

Table 3.1: Overview of classified services

Chapter 4

Simulation Results

In this chapter we describe the simulation of the Flood-MultipleTeleFlood pattern compared to a central approach. After describing services and the appropriate deployment patterns, the task was to measure the effort to deploy a service in different networks.

4.1 Description of the simulation environment

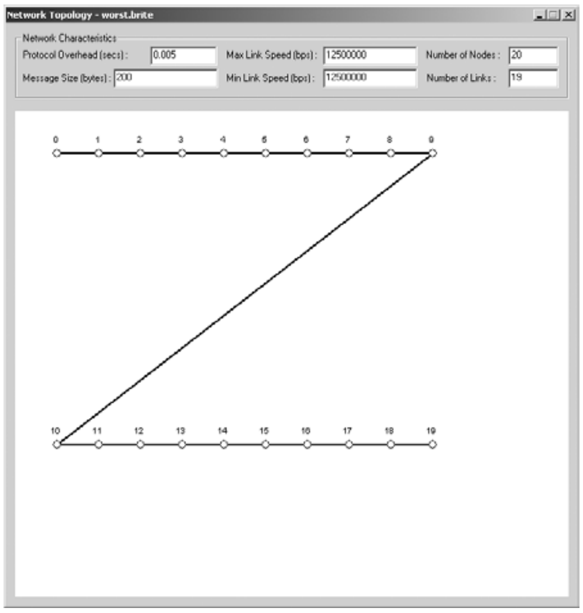
The simulation is based on the SIMPSON simulator by Lim and Stadler [12]. This handy tool offers an environment to simulate the performance of navigation and aggregator patterns.
Because SIMPSON doesn't offer any routing algorithms, it was necessary to implement own routing algorithms to study the patterns in the network.
SIMPSON measures the used execution time and counts the messages. It does not count the number of messages from the source to the destination, but all messages that are sent over a link. E.g. a data packet has to traverse an intermediate node, SIMPSON will display a message count of 2. One from the start node to the intermediate node and one from the intermediate node to the end node. The traffic complexity shows the message count multiplied by the message size.
The main goal of the simulation was to compare the pattern based approach, whose nodes are active, against a traditional central approach in a passive network. Our patterns implement a strategic aggregation of node information, while the central approach just sends a query-message to each node, since it has no possibility for computation.
To compare the performance of each pattern, two different network struc-

tures were chosen. The two structures represent the best and the worst case. In the best case, each node is directly connected to the central node, where the pattern is started. In the worst case, all nodes are aligned, so the pattern has to cross each node in the network.
All links were defined as 100Mbit/s (12500000 bytes/s) links, to simulate uncongested Local Area Networks. First, it was planned to define a wide area link between the two domains with a lower bandwidth. In contrary to the expected results, SIMPSON does not handle congested links. All messages can pass in parallel, while in a real congested network messages pass one after the other.
The chosen size of messages is 200 bytes in the central approach and 400 bytes for the patterns. The messages in the central approach contain only the destination address, the source address, the requested information and the corresponding values. A message in the pattern contains mobile states and the entire aggregated information. This messages have to be bigger than the messages from the central approach. Ethernet-, IP- and TCP-headers generate an overhead of 74 Bytes. 200 (or 400) bytes should be a good average size to exchange some useful information and not to waste too much bandwidth on messages. TCP is needed for reliable transmission and cannot be displaced by UDP. In reality the message size is flexible, dependent on the chosen service, number of nodes, and number of requested parameters.

4.2 The simulation

The main goal of the simulation was to compare the amount of sent messages during the execution of the two different approaches. The Flood-MultipleTeleFlood pattern was expected to send less messages than the centralistic approach, which sends a unique message to each single node in the local and the distant domain. The two network structures are shown in figure 4.1 and 4.2.
The patterns were started on all nodes in both scenarios. Therefore it is possible to analyze the dependence on the network structure and to see the changes in the amount of sent messages depending to the start node. In the worst and best-case scenario, the patterns explored both domains and returned the best nodes, since all nodes were rated with a scalar value. A polling approach congests the link even more, because it sends messages all

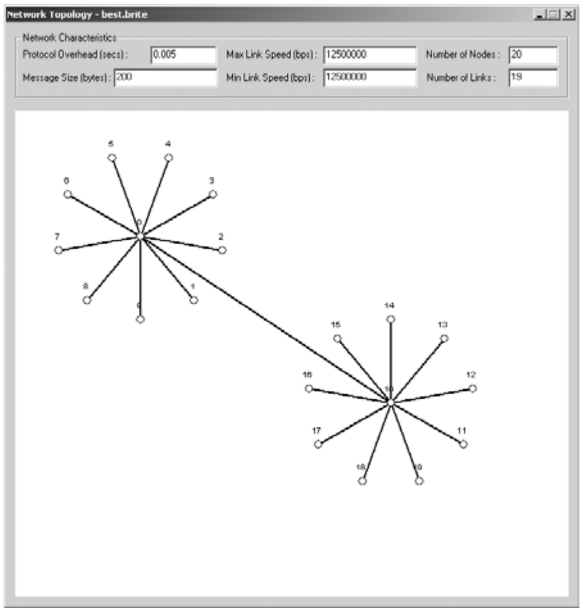
the time, even if the deployment of this service is not requested. As SIMPSON doesn't support polling patterns, it was not possible to simulate this approach.
Because the two simulated networks are only a special case and far from reality, a second simulation should compare the Flood-MultipleTeleFlood and the central pattern in a more realistic network. As SIMPSON doesn't offer any routing algorithm and it is not the scope of this work to do so, a regular network as seen in figure 4.3 was chosen. The Grid allows a simple routing algorithm and can give a good impression of the behavior in a bigger network. The routing algorithm calculates the number of links in vertical and horizontal direction and chooses randomly one of this two directions. This routing algorithm obviously doesn't simulate a real IP-roting, but it was not possible to implement a routing algorithm like OSPF. In this scenario the interesting point is mainly the message count itself, than it's dependence from the network structure. As both patterns use the same routing algorithm, the comparison will provide fair results.
The patterns deploy an overlay network between 3 domains. The minimum distance from the upper left to the lower left is 11 hops. From the lower left to the lower right 9 hops and from the upper left to the lower right 20 hops. the number of hops for 3 different simulation runs are listed in table B.1. This shows that the routing algorithm finds the shortest path from the start node to the destination. In this scenario, called "the Grid", the patterns had to send the best nodes an initial message to start the overlay network. In each domain, one node was defined as the "best", by assigning the highest value to it. These nodes were number 63, 324 and 358.
The main goal of the patterns is to reach the domains, find the "best" node (with the highest value set) and send its address back to the start node. The start node aggregates all values and chooses the node with the highest. Afterwards it sends them a message to establish the overlay. The two distant nodes send an ACK back to the start node, to confirm the success.


Network Topology - worst.brite

Network Characteristics

Protocol Overhead (secs) : 0.005
Max Link Speed (bps) : 12500000
Number of Nodes : 20
Message Size (bytes) : 200
Min Link Speed (bps) : 12500000
Number of Links : 19

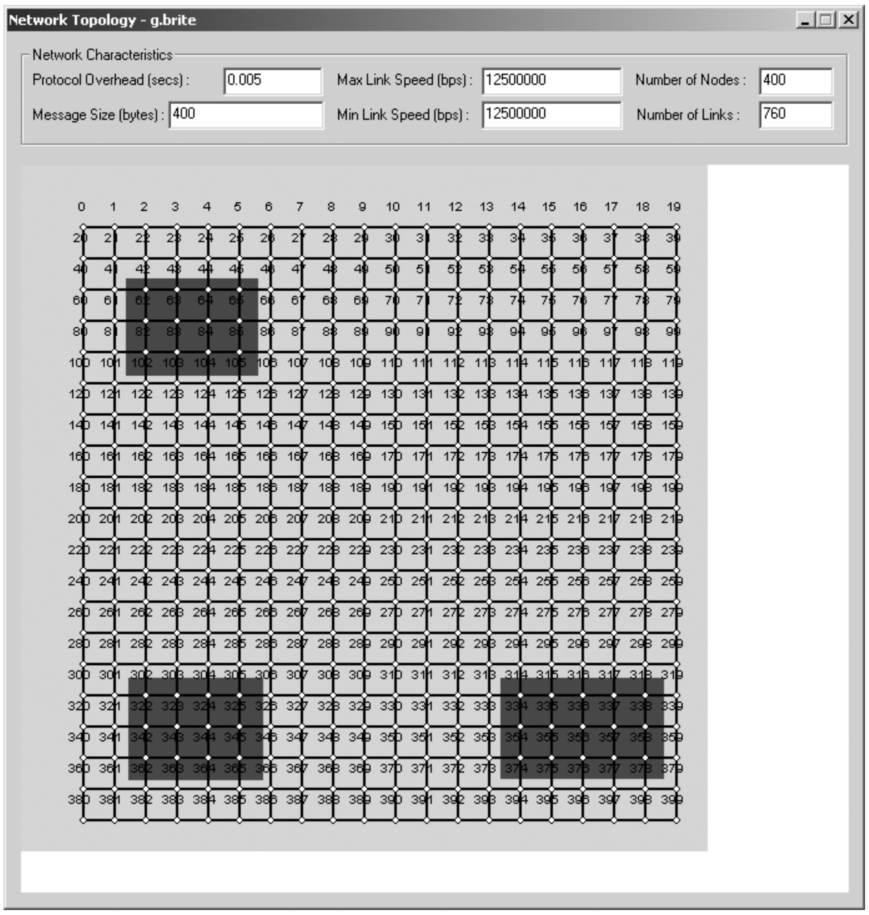
Figure 4.1: The network for the worst-case scenario


Network Topology - best.brite

Network Characteristics

Protocol Overhead (secs) : 0.005
Max Link Speed (bps) : 12500000
Number of Nodes : 20
Message Size (bytes) : 200
Min Link Speed (bps) : 12500000
Number of Links : 19

Figure 4.2: The network for the best-case scenario



Network Characteristics

Protocol Overhead (secs) : 0.005

Max Link Speed (bps) : 12500000

Number of Nodes : 400

Message Size (bytes) : 400

Min Link Speed (bps) : 12500000

Number of Links : 760

Figure 4.3: The network for "the Grid"

4.3 The simulation Results

To simplify the reading of the simulation results, the central approach is called "central" and the Flood-MultipleTeleFlood pattern "FMTF". The regular network is named "the Grid".

start node	worst - case				best - case			
	central		FMTF		central		FMTF	
	msg	bytes	msg	bytes	msg	bytes	msg	bytes
0	380	76000	56	22400	56	11200	38	15200
1	344	68800	54	21600	92	18400	40	16000
2	312	62400	52	20800	92	18400	40	16000
3	284	56800	50	20000	92	18400	40	16000
4	260	52000	48	19200	92	18400	40	16000
5	240	48000	46	18400	92	18400	40	16000
6	224	44800	44	17600	92	18400	40	16000
7	212	42400	42	16800	92	18400	40	16000
8	204	40800	40	16000	92	18400	40	16000
9	200	40000	38	15200	92	18400	40	16000
10	200	40000	38	15200	56	11200	38	15200
11	204	40800	40	16000	92	18400	40	16000
12	212	42400	42	16800	92	18400	40	16000
13	224	44800	44	17600	92	18400	40	16000
14	240	48000	46	18400	92	18400	40	16000
15	260	52000	48	19200	92	18400	40	16000
16	284	56800	50	20000	92	18400	40	16000
17	312	62400	52	20800	92	18400	40	16000
18	344	68800	54	21600	92	18400	40	16000
19	380	76000	56	22400	92	18400	40	16000

Table 4.1: The number of sent messages and bytes in the worst- and best-case scenario

Table 4.1 shows the number of sent messages and the traffic complexity in the worst-case and best-case for all start nodes. Figures 4.4 and 4.5 illus-

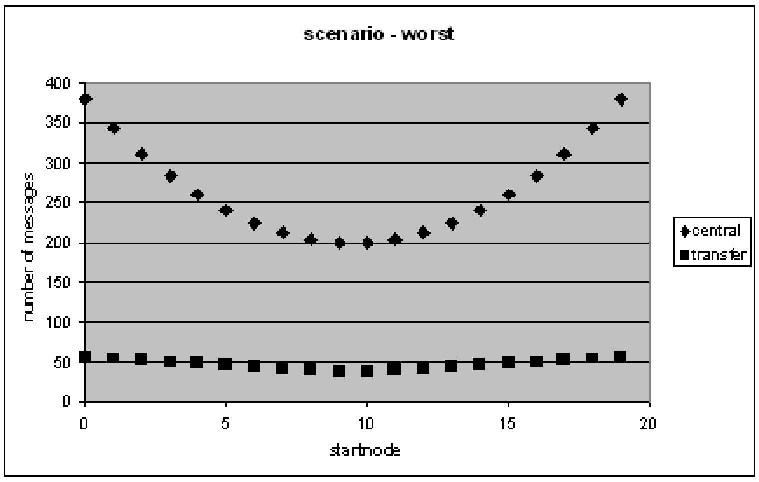


Figure 4.4: The number of sent messages in the worst-case scenario

trate the gain in using the FMTF pattern for deploying a network service. In the worst-case scenario, the central pattern decreases the number of sent messages by a factor of approximately 2, by shifting the start node from the end to the center of the domain, while the FMTF patterns message count rests approximately constant. The difference is smaller in the best-case scenario. Here the central approach generates only 16 messages more than the FMTF pattern on start node 0 and 10. For all other start nodes, the difference of sent messages is 52.

As expected, the central approach generates much more network traffic than the FMTF pattern, which sends only one explorer to the distant domain, aggregates the information of all nodes and sends one message back to the start node. Except in the best-case scenario, where the central approach transmits less data because of the smaller message size, the FMTF pattern uses less network resources than its centralized competitor. Only on start nodes 0 and 10 in the best-case scenario, the difference of sent messages is small enough to make the central approach look better. As soon as the network structure differs from the best-case, the FMTF pattern transmits less data, e.g. if the pattern is started on one of the nodes on the circle around the central node.
The number of messages depends strongly on the network structure. If the degree of cross-linking is high, most messages can be sent directly to the

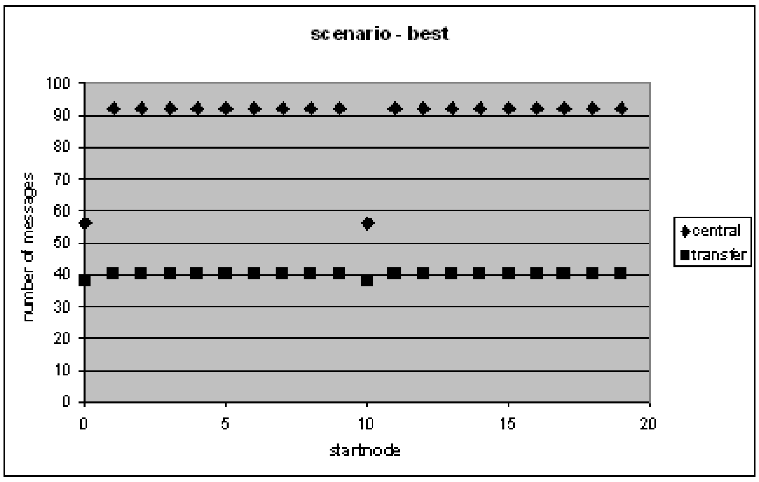


Figure 4.5: The number of sent messages in the best-case scenario

destination node, while in a sparely-linked network all messages have to be forwarded by one or more intermediate nodes. The amount of data is proportional to the number of messages. The more messages are generated, the more data has to be transported.
Most benefit will be achieved when the pattern has to travel over a congested wide area link (i.e. the Internet). Instead of sending a separate message for each single node, the FMTF pattern sends one message to the distant domain, aggregates the information from all requested nodes and sends the complete report in a single message back through the congested link to the start node. In the central approach, all messages have to wait in front of the congested link, which will end in a huge delay for the execution time and result in a even higher grade of congestion on this link.

Table 4.2 shows the number of sent messages in "the Grid". Figure 4.6 shows the number of sent messages graphically for both patterns. As in the two preceding scenarios, the FMTF pattern generates less traffic in the regular network. In a congested network, or on a congested wide area link, the central pattern would have a big delay in the execution time.
Pattern based service deployment in active networks can simplify the deployment process significantly. On the one side it is possible to save time, because of the flexibility of the patterns. On the other side, it allows you to save bandwidth, because of the lower amount of sent data, especially over

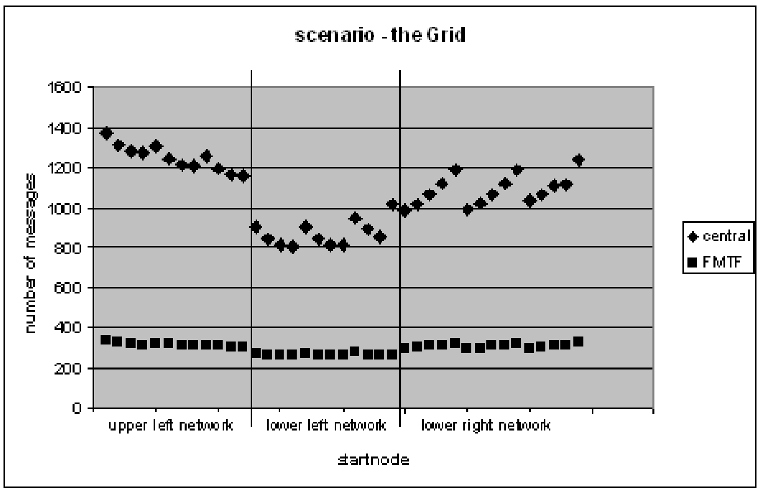


Figure 4.6: The number of sent messages in the “the Grid” scenario

congested wide area links.

start node	central		FMTF		start node	central		FMTF	
	msg	bytes	msg	bytes		msg	bytes	msg	bytes
42	1370	274000	332	132800	342	950	190000	274	109600
43	1310	262000	324	129600	343	890	178000	262	104800
44	1278	255600	318	127200	344	858	171600	260	104000
45	1274	254800	314	125600	345	854	170800	264	105600
62	1320	260400	322	128800	314	986	197200	298	119200
63	1242	248400	316	126400	315	1018	203600	302	120800
64	1210	242000	308	123200	316	1062	212400	308	123200
65	1206	241200	306	122400	317	1118	223600	312	124800
82	1254	250800	316	126400	318	1186	237200	320	128000
83	1194	238800	308	123200	334	990	198000	294	117600
84	1162	232400	300	120000	335	1022	204400	298	119200
85	1158	231600	300	120000	336	1066	213200	308	123200
302	902	180400	266	106400	337	1122	224400	312	124800
303	842	168400	258	103200	338	1190	238000	318	127200
304	810	162000	258	103200	354	1034	206800	298	119200
305	806	161200	260	104000	355	1066	213200	304	121600
322	906	181200	268	107200	356	1110	222000	310	124000
323	846	169200	258	103200	357	1116	223200	316	126400
324	814	162800	260	104000	358	1234	246800	324	129600
325	810	162000	258	103200					

Table 4.2: The number of sent messages and bytes in “the Grid”

Chapter 5

Conclusion and future work

At the end of this thesis, the conclusion is developed and future work is indicated.

5.1 Conclusion

Active Networks are a new and very interesting research area. Future network-technology can profit a lot by introducing new concepts from active networks. Most will result in a better usability for the end user and a more sparing handling with the existing resources.
The benefit of pattern-based service deployment is, that it allows to develop classes of the same deployment scheme. The fact that navigation patterns do not contain any service specific operation makes this possible. In this thesis, we have developed navigation and aggregator patterns for each class of service deployment and assigned existing services to them. The navigation patterns are FSM and therefore completely specified. Otherwise the aggregators. There are also classes of aggregators in general, but they can not be completely implemented. They all need a service specific implementation of its class. At the first glimpse this seems to be against our main goal of this thesis. But aggregators are also significantly less complex than navigation patterns and therefore do not need a lot of knowledge to be implemented.
In the simulation it was possible to show the differences between a deployment-pattern and a central approach. Instead of sending a message to each node in a distant network, the deployment pattern generates less traffic by sending a scout to the network and start the flooding from there.
Pattern-based service deployment is still in development, but it can be one

of the key concepts in future networks.

5.2 Future work

Although this thesis had to come to an end, the research on pattern-based service deployment is not finished at all. SIMPSON offers some nice features to implement and simulate deployment patterns. Unfortunately we struggled with some points that could have simplified the handling. An extensive tool to simulate and test deployed patterns can simplify and set some standards for future research.

- **Adding routing algorithms to SIMPSON:** active nodes are based on standard routers. Therefore they must offer the functionality of a common router in today's networks. If SIMPSON offered routing algorithms from modern networks, it would be possible to concentrate more on the pattern itself than on its distribution in the network.

- **Adding queuing to SIMPSON:** SIMPSON does not queue patterns in front of congested links. Therefore it was not possible to measure exactly the execution time. In a realistic network, data packets would have to wait in front of congested links to be able to pass. This would result in a delay of the pattern during it's execution.

- **Implementing other deployment-patterns:** Not all deployment patterns for the described services were implemented. So it was not possible to compare all of them in the simulation. To receive a good overview on service deployment in active networks it would be necessary to have them all implemented.

- **Comparing to a polling-approach:** As SIMPSON doesn't support polling patterns, the deployment-patterns couldn't be compared to a polling approach. The difference in the number of messages should be even bigger than in a central approach. This would result in a even higher degree of congestion.

- **Describing new services for active networks:** The given survey doesn't enclose all possible future services in active networks. New services will be invented and may result in the development of new patterns. Taking them into consideration will ask for new deployment-patterns.

- **Porting the patterns into the network:** The management programs are written in such a manner that they can be executed on the simulator as well as on a planned prototype platform. It is a future task to have the services deployed in reality.

Appendix A

Appendix

A.1 The SIMPSON Simulator

SIMPSON stands for “a SIMple Pattern Simulator fOr Networks”. It was developed by Koon-Seng Lim and Rolf Stadler to serve as a workbench for the construction and testing of management programs, as well as the observation of their performance characteristics [12].
It is possible to define an own network with all definitions for a simulation (i.e. bandwidth per link, message size, protocol/OS delay, network delay per hop) and also include a background image to present a realistic scenario. The navigation pattern and aggregator are included and loaded as dynamic-link libraries (DLL’s in MS Windows). The simulation can be paused and resumed during run-time, to get debugging information from the debug console. SIMPSON prints all information from the patterns on the debug console (i.e. execution time, message count, traffic complexity and number of faults) and also all information printed out by the patterns. The start node, the delay of each event and the pattern arguments have to be defined at the beginning of a simulation. The simulator then calculates the needed execution time in the network. To check the flow of the navigator pattern, the color of the visited nodes can be changed by executing a *log_node_event()* in the navigation pattern. For full description see also the SIMPSON tutorials [13].

A.2 Problems with the SIMPSON Simulator

The SIMPSON Simulator is a handy tool to develop patterns for active networks. But it can't solve all problems you're running across.

- SIMPSON doesn't provide any routing algorithms. Therefore own routing algorithms have to be implemented into the navigation pattern. In a real active network, the patterns would choose the next hop by selecting the appropriate IP-address and the routing would be done by the network.
- SIMPSON only can handle patterns. It is not possible to implement a program which runs all the time on a node (i.e. for a polling approach). The patterns are activated by incoming messages and they finish by sending out new messages. So it wasn't possible to compare the pattern based approach to a polling one. The central pattern used in the simulation is based on a navigation pattern which implements a central approach.

Appendix B

Appendix

B.1 The number of hops

To approve the used routing algorithm in “the grid”, the pattern from the central approach was modified to count the number of hops that a message needs to travel from the start node to the destination and back.
Therefore it was possible to check, if the routing algorithm found the shortest path to the destination, even if not all messages took the same path from one network to another. In traditional networks, all messages for the same network usually travel the same path, which is given by the routing algorithm. Table B.1 shows the number of hops for messages for three different start nodes. E.g. from node 63 to node 337: 14 necessary hops in horizontal and 13 hops in vertical. This makes a total of 27 hops or 54 hops back and forth. From node 302 to 358, 16 hops in horizontal and 2 hops in vertical direction makes a total of 18 hops or 36 hops back and forth.

destination	start node			destination	start node		
	63	302	315		63	302	315
42	4	26	52	342	30	4	30
43	2	28	50	343	28	6	28
44	4	30	48	344	30	8	26
45	6	32	46	345	32	10	24
62	2	24	50	314	46	24	2
63	0	26	48	315	48	26	0
64	2	28	46	316	50	28	2
65	4	30	44	317	52	30	4
82	4	22	48	318	54	32	6
83	2	24	46	334	48	26	4
84	4	26	44	335	50	28	2
85	6	28	42	336	52	30	4
302	26	0	26	337	54	32	6
303	24	2	24	338	56	34	8
304	26	4	22	354	50	28	6
305	28	6	20	355	52	30	4
322	28	2	28	356	54	32	6
323	26	4	26	357	56	34	8
324	28	6	24	358	58	36	10
325	30	8	22				

Table B.1: The number of hops in the central approach

Appendix C

Appendix

C.1 Enclosed CD-Rom

The enclosed CD-Rom contains the following directories:

Directory / File	**Content**
/readme	Description and usage of the content of this CD
/programs	Used Programs
/programs/simpson	SIMPSON Simulator v16a SIMPSON Bugfix SIMPSON User Guide SIMPSON Programming Tutorial
/programs/compiler	Digital Mars Compiler
/paper	Pdf version of this thesis
/src	Source Code of our patterns
/dll	Dynamic link libraries of our patterns
/scenarios	Our scenario files

Table C.1: Content of the CD-Rom

Bibliography

[1] Konstantinos Psounis. Active Networks: Applications, Security, Safety and Architectures. *IEEE Communications Surveys*, (First Quarter), 1999.

[2] Matthias Bossardt, Takashi Egawa, Hideki Otsuki, and Bernhard Plattner. Integrated Service Deployment for Active Networks. In James Sterbenz, Osamu Takada, Christian Tschudin, and Bernhard Plattner, editors, *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.

[3] David L. Tennenhouse, Jonathan M. Smith, W. David Sincoskie, David J. Wetherall, and Gary J. Minden. A Survey of Active Network Research. *IEEE Communications Magazine*, 35(1):80–86, 1997.

[4] Koon-Seng Lim and Rolf Stadler. Developing Pattern-based Management Programs. Technical Report 503-01-01, CTR, 2001.

[5] Li-Wei H. Lehman, Stephen J. Garland, and David L. Tennenhouse. Active Reliable Multicast. In *IEEE INFOCOM*, March, April 1998.

[6] David Wetherall, Ulana Legedza, and John Guttag. Introducing New Internet Services: Why and How. *IEEE Network Magazine*, Special Issue on Active and Programmable Networks, July 1998.

[7] Ralph Keller, Suomi Choi, Marcel Dasen, Dan Decasper, George Frankhauser, and Bernhard Plattner. An Active Router Architecture for Multicast Video Distribution. In *IEEE INFOCOM*, March 2000.

[8] Ulana Legedza, David Wetherall, and John Guttag. Improving the Performance of Distributed Applications Using Active Networks. In *IEEE INFOCOM*, March, April 1998.

[9] Samrat Bhattacharjee, Kenneth L. Calvert, and Ellen W. Zegura. An Architecture for Active Networking. In *HPN*, pages 265–279, 1997.

[10] Elan Amir, Steven McCanne, and Randy H. Katz. An Active Service Framework and Its Application to Real-Time Multimedia Transcoding. In *SIGCOMM*, pages 178–189, 1998.

[11] Y. Chae, S. Merugu, E. Zegura, and S. Bhattacharjee. Exposing the Network: Support for Topology Sensitive Applications, 2000.

[12] Koon-Seng Lim and Rolf Stadler. SIMPSON - A Simple Pattern Simulator for Networks. `http://comet.ctr.columbia.edu/adm/software.htm`.

[13] Koon-Seng Lim and Rolf Stadler. Distributed Management Project. `http://2g1331.imit.kth.se/`.