A Monitoring Tool for Grid Networks

Mathijs den Burger

22nd February 2002
# Contents

1 Introduction .................................................. 1

2 Related work .................................................. 3
   2.1 Measurement tools ......................................... 3
       2.1.1 Ping and variants ...................................... 4
       2.1.2 Traceroute and Pathchar ............................... 4
       2.1.3 CAIDA and Skitter .................................... 5
       2.1.4 Pathrate ............................................... 5
       2.1.5 SProbe ................................................. 5
   2.2 Grid Monitoring Architecture ........................... 6
       2.2.1 Overview .............................................. 6
       2.2.2 Discussion ............................................ 7
   2.3 Monitoring tools .......................................... 7
       2.3.1 Network Weather Service ............................ 7
       2.3.2 Remos .................................................. 10
       2.3.3 Topology-d ............................................ 14

3 Design .......................................................... 16
   3.1 Functionality .............................................. 16
       3.1.1 Measurements ......................................... 16
       3.1.2 Shared links ........................................... 17
       3.1.3 GMA .................................................... 17
       3.1.4 Combining information ................................. 18
   3.2 Architecture .............................................. 22
       3.2.1 Components ........................................... 22
       3.2.2 Communication ....................................... 26

4 Implementation ............................................... 32
   4.1 TopoMon sensor ............................................ 32
   4.2 Java packages ............................................. 33
       4.2.1 GMA ................................................... 33
       4.2.2 TopoMon Sensors ...................................... 40
       4.2.3 NWS interface ......................................... 41
       4.2.4 Directory service ..................................... 42
4.2.5 TopoMon API ........................................ 46
4.3 Configuration ........................................ 48
  4.3.1 Sensors ........................................... 48
  4.3.2 Producers ........................................ 48

5 Results .................................................. 50
  5.1 Topology visualization ................................ 50
  5.2 Multicast trees ...................................... 52

6 Conclusions and future work ................................ 57
  6.1 Measurements ....................................... 57
  6.2 GMA functionality .................................. 58
  6.3 Scalability .......................................... 58
  6.4 Administration ..................................... 59
  6.5 Multicasting ....................................... 60
  6.6 Visualization ....................................... 60
Abstract

Computational Grids extend today’s Internet by combining globally distributed computers and information sources into a universal source of computing power and information. Grid applications typically need to share the resources available in the Grid. The most predominant shared resource is the network connecting the various sites, something already observed by many Internet users today. To be robust to changing resource availability, Grid applications need to be aware of their execution environment and its changes. This means they need continuous information about resource availability, which can be used as input to application-level models that implement robustness.

The goal of this thesis is to construct a monitoring system for Grid applications that will provide them with the necessary data about their execution environment. We focus on the network and two characteristics of it: latency and bandwidth between the cooperating computers in a Grid environment. We argue that the information about the network topology between these computers has to be provided to applications too, so shared links can be taken into account and performance figures of hierarchical measurement groups can be aggregated. We describe the design and implementation of our monitoring tool: TopoMon, and show results we obtained by running it on a testbed consisting of various geographically distributed sites around the world. Finally we identify further directions of research.
Chapter 1

Introduction

The Internet started a revolution in computer history: after isolated usage of computers and some cooperation in timesharing systems, it became possible to use resources that were geographically distributed by communicating in a uniform way with the IP protocol. As the Internet grew larger and larger, more and more resources (desktop computers, supercomputers, large data stores, etc.) were connected to the same global network. Most resources today are shared explicitly as 'servers' that use a certain protocol (such as FTP for file/storage servers, HTTP for web servers etc.). Remote computing power is typically available in the form of supercomputers that all have their own special procedure to submit a compute job.

All the resources connected to the Internet together have an amount of computing power and storage space that is hard to grasp. That's how the idea of Grid computing [12] was born. By combining these resources and developing a universal way of interacting with them, computing power should ideally be as easy to get as electric power at home. The comparison with the electric power grid gave such systems the name of “Grids”.

A lot of problems have to be solved to make Grids possible. One of these problems is how to get good performance, especially from the interconnecting wide-area network (the Internet). The problem with Grid applications is that the Internet is highly complex, dynamic and heterogeneous. Traditional supercomputers with multiple CPUs are typically connected by a special high-speed network with a regular, known topology and constant performance. A Grid, however, has an irregular and asymmetric network topology where different links have different speeds. Even worse, all this changes over time. Such varying network performance is already observed by many Internet users today.

To be robust to the changing network conditions, Grid applications simply need to be aware of the fact that they run in such an environment. This means they need constant information about the changes of their execution environment so they can adapt their behavior accordingly.

The goal of this thesis is to construct a monitoring system for Grid applications that will provide such information. We focus on two characteristics:
latency and bandwidth between the cooperating computers in a Grid, taking the network topology into account. Our monitoring system gathers information about the network topology for two reasons.

First, multiple end-to-end connections can share parts of the interconnecting network. Such shared links have to handle multiple, simultaneous data flows of the application. Traditional distributed systems often ignore this, and assume the traffic generated by the application is smaller than the concurrent traffic in shared links, so two concurrent application data flows will not interfere with each other. This is not always the case, especially not when applications use collective communication, where many sites communicate with each other simultaneously. If the exchanged messages are large, the chance two data flows will interfere with each other is high. Such interference results in worse performance. The work on MPICH-G (the MPI implementation of Globus [14]) has shown that MagPie [19] (a wide-area collective communication library for MPI) has exactly this problem for shared links [18]. To solve this, information about the network topology has to be available.

A second reason why the network topology is important is related to the scalability of the monitoring tool itself. Measuring end-to-end performance between $N$ computers requires $N \times (N-1)/2$ measurements. To cope with this rapidly increasing number of measurements (which does not scale to large networks), some kind of measurement hierarchy has to be created. This means computers are clustered into groups, and only perform end-to-end measurements within a group. One level higher, 'meta-groups' perform measurements between the groups. This creates a hierarchy of measurement groups that can be extended to more than two levels if the number of computers grows. Existing monitoring systems like the Network Weather Service [34] use such groups, but fail to aggregate the performance figures of multiple hierarchies of groups. To do this, you have to know how packets are routed through this hierarchy, otherwise you cannot know which measurements to use (for details, see Section 2.3.1). The actual implementation of such a measurement hierarchy is beyond the scope of this thesis, but it is clear that information about the network topology is needed for it.

The thesis is structured as follows. In Chapter 2 we look at existing network measurement tools and Grid monitoring systems and the basic concepts behind them. In Chapter 3 we describe and explain the design of TopoMon, our monitoring tool. Chapter 4 describes the implementation details. Chapter 5 shows some results of running TopoMon on a testbed consisting of various geographically distributed sites around the world. Chapter 6 concludes and indicates future directions of research.
Chapter 2

Related work

Many tools that measure and/or monitor certain network characteristics have been developed. They can be divided into two groups:

- **measurement tools** that measure only one or several specific characteristics;

- **monitoring tools** that continuously provide information about certain resource characteristics. They often use measurement tools to perform the actual measurements.

Some standardization efforts have been and are being made. The Global Grid Forum is the central worldwide meeting place of Grid developers. It has several working- and research groups that investigate certain problem areas. For network measurements it is important to identify, characterize and standardize the metrics used. For this reason the Network Measurements Working Group has been created. Its goals are similar to that of the IPPM working group (part of the Internet Engineering Task Force) that develops a set of standard metrics for Internet measurements. The Grid Monitoring Architecture Working Group tries to standardize the architecture of Grid monitoring systems.

We first discuss measurement tools in Section 2.1. In Section 2.2 we describe the GMA (Grid Monitoring Architecture) in detail. In Section 2.3 we describe some monitoring tools and discuss some characteristics of them that influenced the architecture of TopoMon. We also compare the architecture of each system to the GMA to illustrate to power of its abstractions: in almost every systems the GMA parts can be identified.

2.1 Measurement tools

Specific tools to measure certain network characteristics have been around for a long time. We describe some of them in this section. This overview is not meant to be exclusive, but merely gives an impression of previous and current practice.
2.1.1 Ping and variants

The most famous network measurement tool is probably ping, which was invented by Dave Mills as part of the Fuzzball system [22] and implemented for 4.2a BSD UNIX by Mike Muuss in only one evening [23].

The program measures the round-trip time to a target machine by timing ICMP echo packets. The advantage of this method is that the measurements do not require cooperation between both sites. A disadvantage is that the measured latency does not take the application-level latency (for example of the TCP layer) into account.

The Network Weather Service (see Section 2.3.1) uses a variant of ping that does require cooperation, but takes the application-level latency into account by sending TCP segments. The nws_ping program measures the latency between two hosts by sending a small TCP segment that is echoed by the other host. The latency is taken as the round trip time divided by two. nws_ping also measures the bandwidth between two hosts by sending a configurable amount of data in a configurable number of writes. It measures the time it takes to send this, and divides that by size of the data sent to yield the bandwidth.

The PingER project [20] (Ping End-to-end Reporting) logs the results of periodic runs of ping at hundreds of sites and gathers all this data for offline analysis. Currently the PingER analysis includes five metrics: packet loss, round-trip time, unreachability, quiescence (indicating a network is non-busy, detected by the fact that all ping packets are echoed) and unpredictability (which is based the variability of packet loss and round-trip time).

2.1.2 Traceroute and Pathchar

The traceroute tool of Van Jacobson uses a variant of ping (with UDP packets) to discover all the hops in an Internet path. By using increasing time-to-live fields in outgoing UDP packets, intermediate routers reveal themselves by returning an ICMP “time exceeded” reply. Later, Van Jacobson released a more advanced program that uses this strategy, Pathchar [17], which reports better estimations of per-link bandwidth and latency. To do so, it uses more mathematics and probe packets to take the unpredictable queuing time in routers into account. Because both tools need access to the ICMP socket layer, they require special access on UNIX-like machines to let normal users run them. In practice, traceroute is installed and available to normal users on most UNIX systems, but Pathchar is not. This severely limits the use of Pathchar in a Grid monitoring system.

Although the original version of traceroute is written for UNIX, other operating systems typically nowadays have a variant of it. Most Windows versions for example are equipped with the program tracert that does the same job as traceroute. Macintosh computers also contain traceroute since MacOS X.

An alternative for traceroute for discovering nearby routes is the “record route” option of ping. This uses IP’s “record route” option that stores the route a packet follows inside the IP header. Because space there is limited, only eight
hops can be discovered this way. This method uses only one echoed packet to discover the first eight hops, instead of eight separate probes in the case of traceroute. However, not all routers support this option, which makes it less general in practice than traceroute is.

2.1.3 CAIDA and Skitter

The Cooperative Association for Internet Data Analysis (CAIDA) maintains a thorough taxonomy of Internet measurement tools [4]. One of those tools, Skitter, uses a technique similar to that of traceroute to determine the Internet paths and round-trip times between a few sources that run Skitter and many destinations. This world-wide gathered data is combined approximately every day and published for analysis and research purposes.

2.1.4 Pathrate

Pathrate [9, 10] measures the capacity and the available bandwidth of a path. The capacity is the maximum throughput a path can offer in the absence of competing traffic (known as 'cross traffic'). It is also known as the 'bottleneck bandwidth'. The available bandwidth takes also the current cross traffic into account.

To measure this, Pathrate uses the dispersion of packet pairs and packet trains. The packet pair technique uses two packets of the same size. These are sent immediately after each other to a destination, that echoes them back to the sender. The amount of time between the arrival of the echoed packets is the time it took the slowest link in the path between the sender and receiver to process the packets, and is thus an indication of the capacity of a path. Packet trains extend this procedure by sending $N$ packets instead of two, resulting in $N - 1$ dispersion times. This gives a more reliable estimate at the cost of a heavier load on the path.

Pathrate uses many packet pairs (with packets of variable size) and long packet trains and requires cooperation between the source and destination host to echo the packets. Although the estimates are fairly good, its intrusiveness limits its use in a Grid monitoring system.

2.1.5 SProbe

SProbe [27] measures the bottleneck bandwidth of a path. It distinguishes two bandwidth values of a path: the upstream and the downstream bottleneck. The former is the bottleneck of the path from the remote host to the local host, the latter is the bottleneck of the path from the local host to the remote one. To discover the downstream bottleneck it exploits TCP by using SYN packets as a packet-pair. The upstream bottleneck discovery requires the host to run some kind of TCP server (for example a web server). It then issues an application-level request, and uses the arrival time dispersion of the returned packets to estimate the upstream bottleneck bandwidth.
2.2 Grid Monitoring Architecture

In this section we describe the Grid Monitoring Architecture (GMA) that has been specified by the Grid Monitoring Architecture Working Group [2].

2.2.1 Overview

The GMA consists of three kinds of components:

**Directory Service:** the central access point for consumers to discover and understand the characteristics of the available information in the Grid. Producers publish and update this information to reflect the system state.

**Producer:** makes performance data available to (possibly multiple) consumers. Consumers can either subscribe themselves to a producer’s data or query a producer explicitly. Consumers can also subscribe to a subset of the produced data by specifying the kind of data they’re interested in. A producer can also initiate such a subscription or query itself. The latter only works if the client accepts this action from the producer. An example in which the latter is useful is an ‘archive’ consumer that stores all the events in the system; it is easier to let such a consumer accept incoming requests from producers (‘store this’) than letting a consumer discover new producers to which it should subscribe so its events can be stored.

**Consumer:** receives performance data by subscribing to (possibly multiple) producers, or querying producers explicitly. Consumers can also accept certain producer-initiated subscriptions or queries.

In the GMA, performance data is sent in *events*, of which a separate metadata description is stored in the directory service. Note that the directory service itself does not contain any measurement data, it only contains information about how to get to this data.

The GMA also defines *compound* producer/consumer components. Such components gather data from other producers, do something interesting with it (generate derived performance data such as mean values, predict future values, filter events etc.) and produce the resulting data.

Note that producers are not necessarily the processes that are actively taking measurements; that is typically done by *sensors*. These are separate processes that measure some performance metric, such as CPU load, free memory, the latency between two hosts, etc. Sensors communicate with a producer, which uses the data collected by the sensors to generate the events. The exact distinction between a producer and a sensor is not altogether clear though, and some discussion is still going on within the GMA working group where to draw the boundary between a producer and a sensor.

Components can have built-in security. Whether this is implemented depends on the environment in which the system is used. In an experimental environment, security is not really an issue, but for a real-world production
Grid that possibly involves different organizations security is essential. In such a system, for example, a user has to identify himself before he can search the directory service, so the latter knows which kinds of searches are permitted for this user. The same applies for every action a client can perform on some component. The communication between producers and the sensors they use may also be controlled, so that a malicious user cannot communicate directly with a monitoring process.

2.2.2 Discussion
The notion of producers and consumers provides a nice high-level abstraction. By defining standards for the event types, communication protocols and interfaces of the various components, the interface to the outer world (being users, a higher-level middleware layer, etc.) is standardized. This enables the various systems that have already been implemented to inter-operate with each other by converting their interfaces to the ones of the GMA.

The GMA gets somewhat vague when looking at a lower level than the producers. Behind a producer are typically many sensors, which are currently beyond the GMA specification. This leaves room for various implementations but may introduce inefficiencies. If several producers in a system all want to measure, for example, the CPU load of some hosts, it is less than ideal if every producer installs its own sensors. This means that when designing a producer, one should carefully check what the 'basic' input is, and produce these values as separate events. This makes it possible for other consumers or compound producer/consumers to reuse this data.

Making the components secure is essential for a real-world system. However, this part has not been given much attention yet in working group documents, probably due to the current, experimental status of the GMA.

2.3 Monitoring tools
In this section we describe some monitoring tools that all provide network information to applications. We discuss three such systems: the Network Weather Service, Remos, and Topology-d. Each one has its strengths and weaknesses, which are discussed after the description of the system.

2.3.1 Network Weather Service
Overview
“The Network Weather Service is a distributed system that periodically monitors and dynamically forecasts the performance various network and computational resources can deliver over a given time interval. It operates a distributed set of performance sensors (network monitors, CPU monitors, etc.) from which it gathers readings of the instantaneous conditions. It then uses numerical models to generate forecasts of what the conditions will be for a given time frame.”[31]
The Network Weather Service (NWS) consists of four kinds of components: [34]

**Name Server:** the central contact point for other processes that want to find each other. Registrations follow the LDAP model, and leases are used to deal with expired registrations.

**Sensor:** collects (value, time stamp) pairs and stores that in a Memory process. Currently there exist sensors that measure:

- CPU availability for a new or already running process
- Free disk space
- Free memory
- TCP connection time (the amount of time, in milliseconds, required to establish a TCP connection to a target host)
- Latency and bandwidth between a pair of hosts.

To measure the last two, sensors perform experiments by sending probes. Sensors are organized in groups called *cliques*. In each experiment, a sensor sends probes to all other members of the clique, and uses these probes to calculate the latency and bandwidth. A token protocol is used inside a clique to prevent collisions of experiments. This protocol favors a constant periodicity over ensuring mutual exclusion of experiments[15]. By defining a hierarchical organization of multiple overlapping cliques, the overall number of experiments can be reduced by using separate cliques to measure, for example, a backbone and various sites connected to this backbone. Figure 2.1 shows an example of such a clique organization: three 'local' cliques are connected by three high-speed links that are measured by the higher-level clique consisting of hosts A, B and C. Instead of $15 \times 14 = 210$ measurements between all computers, only $(5 \times 4) \times 3 + (3 \times 2) = 66$ experiments are needed in this way to cover all the end-to-end connections.

**Memory:** stores data in a persistent way. To prevent running out of disk space the size of the storage space is fixed. When it is full, the oldest measurements are overwritten.

**Forecaster:** uses data in memories to generate short-term forecasts. Currently, there is no specific modeling information about a measurement series incorporated. It applies a set of forecasting models to the entire series and chooses the forecasting technique that has been most accurate over a recent set of measurements. For more information about the forecasting models used by the NWS, see [33].

Currently everything is implemented in C and uses UNIX sockets. The whole system consists of several command-line programs and scripts, including administration tools.
Figure 2.1: Example clique organization: three cliques consisting of five hosts each (for example three campus networks), connected by a high-speed backbone. A clique consisting of the gateways in the three clusters measures the backbone.

Discussion

Cliques The notion of a hierarchical organization of network sensors is important. It seems the only way to make a scalable network measurement system that actively measures host-to-host performance between all the hosts in the system, since the number of measurements grows in \( n^2 \) of the number of machines to measure. This organization introduces another problem though: how to calculate, for example, the performance between hosts that reside in different cliques?

We use the latency between two hosts in Figure 2.1 as an example. Currently, the latency between a host in the top-left cluster and the top-right cluster is always taken as the latency between the gateways \( A \) and \( B \). This method is not very accurate. Imagine a hierarchy in which the top level consists of a clique that measures a transatlantic link, and several lower levels that measure for example a European and an American backbone, and two campus networks. To use the latency of the transatlantic link as the latency between any pair of hosts that reside at a different side of the Atlantic Ocean will be very inaccurate, since the latency in the backbones and campus networks will be completely ignored.

Moreover, the method is incomplete: what if the top level clique consists of more than two hosts? Figure 2.1 shows an example of this, with a top level clique consisting of three hosts (\( A, B \) and \( C \)), which are the gateways to the high-speed backbone in each local cluster. The system cannot know which gateway-to-gateway measurement to use as an estimate in this case, unless it has some knowledge about how packets are routed. Packets from \( A \) to \( B \) may for example travel directly, but packets from \( B \) to \( A \) may travel via \( C \).

These problems can only be solved if the system knows the topology of
the underlying network. Unless the route of packets that travel through
the hierarchy of cliques is known, there is no way to combine the measurements
of the various cliques.

**GMA compliance**  The architecture of the NWS resembles the GMA. This
is not surprising, because the authors of the NWS are participating in the de-
velopment of the GMA. The comparison goes as follows:

**Name Server:** serves as a directory service

**Sensor:** similar to a sensor as envisioned in the GMA framework, with the
exception that sensors are also published in the name server (which is not
the case in the GMA)

**Memory:** acts as a producer that gathers data from (possibly multiple) sensors
and produces this data to the Forecaster. Subscription to new measure-
ments is possible and called 'auto-fetch of measurements'.

**Forecaster:** acts as a compound producer/consumer, that consumes the data
from memories by querying them. It produces forecast data to the user,
which can retrieved by using a C API. No subscription to forecasts is
possible, users can only query a forecaster.

### 2.3.2 Remos

**Overview**

"Remos provides resource information to distributed applications. Its design
goals of scalability, flexibility, and portability are achieved through an archite-
tecture that allows components to be positioned across the network, each collecting
information about its local network. To collect information from different types
of networks and from hosts on those networks, Remos provides several collectors
that use different technologies, such as SNMP or benchmarking. By matching
the appropriate collector to each particular network environment and by pro-
viding an architecture for distributing the output of these collectors across all
querying environments, Remos collects appropriately detailed information at
each site and distributes this information where needed in a scalable manner.
Prediction services are integrated at the user-level, allowing history-based data
collected across the network to be used to generate the predictions needed by a
particular user. Remos has been implemented and tested in a variety of networks
and is in use in a number of different environments." [7]

**The API**

Remos started as an API [13]. Its goal is to define a portable, high-level interface
to network information. This comes down to two major abstractions:
**Flows:** application-level connections between pairs of computation nodes. Remos defines three types of flows, according to bandwidth utilization. *Fixed flows* use a specific, limited amount of bandwidth and have the highest priority of the three flow types. *Variable flows* come second, and share all the available bandwidth among each other. *Independent flows* just want the maximum available bandwidth that is left after the fixed and variable flows have taken their share of the bandwidth.

**Network topology:** these network topologies are logical and consist of three types of nodes:

- endpoints, which are the 'original' computation nodes
- switches (routers and switches found during the discovery of the routes between endpoints)
- logical switches, which represent shared bandwidth pools. For example, when two endpoints are on the same Ethernet segment, a logical switch is used to represent that segment and its speed is set to the speed of the Ethernet.

Figure 2.2 shows an example logical topology. Host 1 to 6 are endpoints. A and B can either be switches or logical switches. In the former case, A and B could be two gateways that connect the endpoints to a high-speed link. In the latter case, A and B could represent two Ethetnets that are connected to each other by the high-speed link. In that case, the speed of A and B (not shown in the figure) would represent the speed of the two Ethetnets, respectively.

![Remos network graph representing the structure of a simple network](image)

Applications can obtain information about the network by querying the Remos API. The *flows* abstraction provides a high-level view of the network. The API defines the following function:

```python
remos_flow_info(fixed_flows, variable_flows, independent_flows, timeframe)
```
Applications can specify the flows they want to use, and by calling this function Remos will report the logical available bandwidth to these flows. Because Remos knows all the flows an application needs, it can take the effect of shared links in the logical topology into account when reporting estimates of bandwidth and latency values. The reported bandwidth is only relevant for the specified timeframe. How the actual values are measured on the network depends on the implementation of the API.

The logical topology itself provides somewhat more low-level information about the network. Applications can use the following function:

```python
remos_get_graph(nodes, graph, timeframe)
```

This fills the graph with a logical topology that connects the given nodes. Note that this graph can also contain nodes that are part of the network that connects the specified nodes and the relevant communication links. The nodes contain annotated information (e.g., the internal bandwidth of logical switches) that is relevant for the specified timeframe. Besides these functions, there are a lot more of them that deal with single nodes, list manipulation, etc.

The system

After defining the API, the authors implemented a system accordingly [7, 21]. It consists of three major parts:

Modeler: a library that exports Remos information through Java and C interfaces. The modeler can be linked with applications. Modelers convert the information they get from the Collectors to a form of interest to the application. This means for example that they eliminate unnecessary information in the topology information returned by the collectors, or report only the bottleneck bandwidth of a path although the collectors reported more information about it. The modelers also act as an intermediary between the collectors and the predictors.

Collector: responsible for acquiring the actual information. It can use several methods to do this, such as sensors that perform the actual measurements, SNMP queries etc. Collectors can be organized in a hierarchical way, which enhances scalability. This hierarchy consists of three types of collectors:

- Local collectors, which are responsible for obtaining performance information about their own LAN
- Global collectors, which obtain performance information about the networks connecting LANs
- Master collectors, which maintain a database of the locations of all the other collectors in the part of the network they are responsible for. There can be multiple Master collectors, and some collectors they manage can in turn be other master collectors. In this way a hierarchy of collectors can be created.
**Predictor:** responsible for turning a measurement history into a prediction of future behavior. Currently, the system uses an existing system called RPS (Resource Prediction System) [8] to achieve this. RPS uses sensors on its own to measure host load and flow bandwidth. The former is implemented with special sensors, the latter uses Remos. In other words: the Remos implementation uses the prediction services of RPS and its host sensors, and RPS uses Remos for the network information. As the authors say: “The relationship between RPS and Remos is somewhat complex, as each is an independent system” [7].

**Discussion**

**The API** Remos is designed top-down: first the API was defined, after which the underlying system was built. This seems a reasonable way of working, since the kind of information a system should provide has to be known before building the actual system. The drawback is that the API is somewhat restrictive: it only reports the flow and topology information, and that’s it. The GMA approach seems more flexible, since new types of information can be added easily. The question remains how applications obtain this information in that case: linking the application with a library might be easier from a programmers point of view than interacting with producers.

**GMA compliance** The authors of Remos compare the system to the GMA themselves:

> “Each Collector is a producer. The Master Collector is a joint consumer/producer, as its responsibility is to contact the other collectors as a consumer, before aggregating the information together and providing it to another layer. Although we view the Modeler as a consumer, it could be another joint consumer/producer, providing end-to-end performance predictions using the component data available from the collectors as a service to other applications. In the Remos architecture, the collectors also implement a limited form of directory service to locate each other. The directory service of the GMA would be natural to use for this purpose.” [7]

**Measurement techniques** The notion to use the best benchmark method per subnet is important: this may be better than just always using probes (as the NWS does). However, SNMP query results have to be converted to network performance as perceived by applications. By using probes this is achieved automatically.

The use of logical topologies has an advantage over just pair-to-pair probing. Imagine three hosts, (A, B and C), between which the bandwidth is measured by using probes. The NWS, for example, tries very hard to keep these probes from interfering with each other. The drawback of this approach is that nothing
can be said when the links A-B and A-C are used simultaneously by an application. When they share, for example, the same gateway, this might introduce a bottleneck. Remos detects such shared links in the logical topology, and is able to report realistic performance data. A drawback of this approach is that Remos cannot know exactly how the sharing affects the bandwidth: this will always be some approximation of reality (as is the case with the SNMP queries).

**The collectors** Remos implements a hierarchical system of collectors. The behavior of the Master Collectors remains a bit vague though: the master collectors form the meta-level between multiple LANs. The authors say that there can be multiple levels of Master Collectors, which are each responsible for a part of the network. But this would introduce the need for another meta-level, between these Master Collectors, otherwise the links between Master Collector domains could never be monitored. Remos solves this problem by letting a Local Collector not only monitor the performance between the desired hosts in the local network, but also between each of the edge nodes of the other subnets that contain nodes included in a query. This means a Local Collector actually measures the performance between local LANs; the Master Collector only identifies which Local Collectors have to measure what. This looks somewhat similar to the NWS approach of hierarchical cliques, in which a Local Collector is a NWS sensor that is contained in a local clique and the 'meta-clique'. The question remains if this 2-level hierarchy is flexible enough; maybe a general N-level hierarchy is more desirable in the long term.

**2.3.3 Topology-d**

**Overview**

"Topology-d estimates the state of the network and networked resources by periodically computing end-to-end latency and available bandwidth. Using its estimates, Topology-d computes a fault tolerant, high bandwidth, low delay topology connecting participating sites. Topologies are periodically re-computed to take into account network and server load dynamics. Network-aware applications can then make use of Topology-d's estimates and logical topologies to ensure they get adequate service from the underlying network and computing infrastructure." [25]

Each machine that is monitored by Topology-d runs the same program. One of the machines is designated as the master. Each machine periodically computes the available bandwidth and latency between itself and all other machines by sending probe packets (similar to the method used by the NWS). The master machine collects the estimates reported by the group members into a cost matrix. The cost of a link is taken as the bandwidth divided by the round trip time. The master then computes the group's logical topology based on this matrix and distributes it among all the group members. The topology calculation is based on that of flood-d, a part of a replication tool that efficiently distributes data among distributed replicas [24].
Topology-d has been integrated with the Globus Meta-computing Directory Service [5], which enables the Globus resource broker to take decisions based on the logical topology.

**Discussion**

The computation of a logical topology is a valuable addition to the per-link statistics. It has already been used in flood-d to implement efficient flooding, but one could think of numerous other applications.

A disadvantage of the computed topology is that the sharing of links in the paths between machines is not taken into account. This may effect the efficiency of the computed topology, since sharing a link might introduce an unexpected bottleneck. Another disadvantage is that only a rough method of prediction is used: each new estimate of e.g. the measured bandwidth is computed as:

\[
\text{new\_estimate} = \alpha \times \text{old\_estimate} + (1 - \alpha) \times \text{current\_estimate}
\]

with \(\alpha\) currently being 0.5. Although this gives some 'damping' effect, the use of longer series (as done by the NWS) combined with a prediction method should produce better estimates.

**GMA compliance**  The master machine in a Topology-d setup can be seen as a producer of bandwidth, latency and topology events. The other machines act as sensors. However, the master machine also acts as a sensor, so the distinction is not very clear. There is no separate consumer process.
Chapter 3

Design

The overview of related work in Chapter 2 indicates several strong and weak points of existing measurement- and monitoring tools. In this chapter we combine the observations we made into arguments for the design of our monitoring tool: TopoMon.

3.1 Functionality

The goal of TopoMon is to provide information to applications about the network that interconnects various endhosts. We focus on two characteristics: the latency and bandwidth between all endhosts as observed by the applications. It will turn out that information about the topology of the network is also needed to take shared links into account.

3.1.1 Measurements

We choose to use the Network Weather Service as the source for bandwidth and latency data in TopoMon for several reasons:

- its use of *forecasters* provides good short-term estimates
- the measurement methods are usable over wide-area links and provide application-level performance information.

Remos also has these qualities, but focuses primarily on SNMP measurements for LANs. Since our interest has primarily to do with wide-area links, using Remos would not be very efficient. Using a separate measurement tool such as Pathrate for the actual measurements could give more accurate measurements. However, these tools are often too intrusive for practical use and focus primarily on measuring the links, not the application-level performance of a connection.
3.1.2 Shared links

The reason for the explicit mutual exclusion of experiments of the NWS was that colliding probes caused the measured performance to be significantly lower than measured by uncollided probes [35]. However, this is precisely the effect shared links can have on end-to-end performance. This means applications cannot rely on the forecasted measurements of the NWS to implement strategies in which multiple links are used simultaneously. Remos addresses this point by discovering the topology of the network and identifying shared links. Another advantage of such a topology is that it can be used like the ones in Topology-d: to calculate efficient application-level topologies for specific purposes. For these reasons, TopoMon contains a subsystem that discovers the topology between all monitored sites.

The actual topology discovery is done by running traceroute from each endhost to all other endhosts. In this way each path is measured twice, which is necessary since Internet paths are often asymmetric. The round-trip time data of the probes reported by traceroute can also be used to estimate the latency between consecutive hosts (see Section 3.1.4).

Pathchar provides better analysis of per-link bandwidth data, but since the NWS already measures end-to-end bandwidth, this extra information is hardly useful. Pathchar is also a lot more intrusive than traceroute. Using SNMP queries to discover the topology (as done by Remos for LANs) is not an option for wide-area topology since public routers generally disable this. Using the BGP router information [26] is also not an option, since they only route between various Autonomous Systems and there may also exist shared links within an Autonomous System.

CAIDA’s Skitter project [28] also uses multiple traceroutes. It could therefore be useful to re-use the data they’ve gathered. The problem is that Skitter does not measure all available Internet links (an impossible job) which means that information about the links between participating Grid sites is often not available. The same applies to other topology repositories CAIDA maintains. TopoMon’s topology subsystem can therefore be seen as a simple Skitter implementation that measures just the links in the monitored Grid.

3.1.3 GMA

To be as flexible as possible, TopoMon implements the GMA. This means that all processes are either producers or consumers of events and locate each other by using a directory service.

For the communication between producers and consumers the GMA working group has defined a simple protocol that uses XML documents as messages [30]. This protocol is described in Section 3.2.2. Using XML has several advantages:

- It is readable for humans, which eases logging and debugging.
- It can be easily parsed by machines too, for which a lot of off-the-shelf parsers are available.
• XML schema’s can be used as meta-descriptions of the messages and events, which creates a very flexible framework. These schemas can be stored in the Directory Service. Producers and consumers can search the directory service for the schema of a particular event and download it. The XML parser can then use the schema to validate the contents and structure of incoming event messages.

A disadvantage of using XML is that the messages get larger than binary message formats. The parsing also introduces extra overhead. A drawback of the GMA protocol itself is that it does not cover security yet.

There exist a number of alternatives for the XML protocol of the GMA working group. Using binary messages gives the smallest possible message size, but this approach does not provide a standard solution for the meta-data in the directory service. Other XML communication approaches use XML descriptions of remote procedure calls, such as XML-RPC [32] and SOAP [3]. This could be used to implement the communication between a producer and consumer, but still does not provide a clear way to describe the meta-information of the events since XML is used to describe information about procedure calls instead of event data.

Since we use the XML producer/consumer protocol, this introduced a little problem for the NWS components. Although its architecture complies with the GMA, the components use a binary protocol instead of XML messages. Furthermore, the forecaster process does not support subscription to forecasts, so we have to poll the forecaster for new forecasts. Therefore a producer process is implemented that acts as a wrapper around a NWS forecaster. This wrapper implements all the functionality of a GMA producer, polls the NWS for new forecasts and produces them as GMA events.

### 3.1.4 Combining information

The ‘raw’ information that TopoMon gathers consists of:

• bandwidth and latency between all endhosts in the monitored Grid (forecasted by the NWS)

• traceroute results between all monitored sites

We have implemented a consumer process (which we will simply refer to as the “TopoMon consumer”) that combines this data to compute the overall topology of the monitored Grid. This is done by combining all the host names in the traceroute results into a graph of the network. Shared intermediate hosts are recognized because they have the same host name or, at least, IP address. In this way a directed graph is generated that represents the logical topology of the network. This graph is then compacted by removing all those hosts that have exactly two neighbors and are not one of the endpoints. This reduces the number of hosts considerably. The resulting graph and other information can be combined for various purposes which we describe in the following.
Visualization

The topology can be visualized by exporting the graph to the GraphViz graph visualization package [11]. This gives Grid application developers insight into the network topology. Each link in the graph can optionally be annotated with either the maximum bandwidth or the minimum latency.

The maximum bandwidth per link is calculated by combining the graph with the end-to-end bandwidth forecasts of the NWS. For each end-to-end path, all the links in that path are annotated with the bandwidth measured by the NWS. If a link is already annotated because it is shared by multiple paths, the maximum of the annotated value and the current value is used as the new value. In this way, each link is annotated with the highest bandwidth of which it should at least be capable of. This gives a conservative estimate for the maximum bandwidth per link, since Internet backbone links may provide higher accumulated bandwidth to multiple, simultaneous transmissions.

To calculate the minimum latency per link, the round-trip times of the probes to the intermediate hosts reported by traceroute are used. Of all reported round-trip times per hosts the smallest one is used, since this represents the link, not disturbed by other, unrelated traffic. Let A and B be host N and N + 1 in a path, respectively, $RTT_A$ and $RTT_B$ the minimum round-trip times to these hosts. The latency of the link from A to B then be calculated as $\frac{RTT_A + RTT_B}{2}$.

However, this method assumes that $RTT_B$ is always greater than $RTT_A$, which is not always true in reality. If $RTT_A$ is greater than $RTT_B$, this means that the time it took a probe packet to reach B, let B generate an ICMP reply, send that back to A and let A forward that reply smaller than the time it took A to generate the ICMP reply to the probe packets that expired there. Apparently A is a bit slow in these cases in creating the reply. We currently assume that this can only happen when A and B are very close to each other (in terms of link speed). Since the latency from A to B then lies within the measurement error, it can safely be modeled as being zero.

The compaction of the graph (as described in the previous section) greatly enhances readability by reducing the number of hosts. If the graph is annotated with latency values, then the values of the consecutive links are added during the compaction. With bandwidth values, one of the values of the links in each direction can be used since they are the same anyway. Figure 3.1 shows two examples of such compaction.

Shared sub-paths

Applications can also ask the TopoMon consumer for the shared parts of two paths between two pairs of endpoints. More precisely: if $A_1$ and $B_1$ are the source and destination endhost of the first path and $A_2$ and $B_2$ the source and destination of the second one, TopoMon will return all the hosts in the path from $A_1$ to $B_1$ that are also part of the path from $A_2$ to $B_2$. Multiple consecutive shared hosts indicate that the links between these hosts are also shared. A 'chain' of one or more shared hosts and links forms a shared sub-path of the
two paths. If two paths share more that one sub-path, TopoMon returns all of them. This is for example often the case with the shared sub-paths of reversed paths between two endpoints (that is, the path from A to B and the other way around), which share the gateways at both sides but are different in between.

**Multicast spanning trees**

The TopoMon consumer uses the end-to-end forecasts of the NWS to generate simple application-level multicast spanning trees. This requires some explanation.

Multicasting means: sending a message to a group of destination hosts. A naive multicasting method sends a separate message to each destination. This method is simple to implement, but wastes bandwidth if there are a lot of destinations. Using broadcasting is another option, but not if the hosts are scattered across the Internet (as is the case in a Grid).

A better approach is to send the message according to the discovered routes in a spanning tree. A spanning tree represents the shortest routes from one vertex in a graph to all the other ones. The vertices in the spanning tree represent the hosts, and the edges the links between them. At each branching in the tree, the incoming message is duplicated by the host and forwarded to all neighbors in the spanning tree.

Multicasting in the Internet, also known as *IP multicasting*, was first introduced in Steve Deering's PhD dissertation [6] and has led to several protocols and 'virtual network' efforts such as the MBone [1]. In IP multicasting, the intermediate routers are actively generating multicast trees. An endpoint can send a message to a special *multicast address*, after which the routers take care
of delivering the message to all endpoints that are part of the group with this address. The main problem with IP multicasting, however, is that it is still not supported by all routers.

Since a multicast tree can only contain branching vertices that 'know' they are part of the tree and will duplicate and forward incoming messages, this means that the only possible splitting points a Grid multicast tree can always use are the endpoints. This is called application-level multicasting, since the application handles the multicasting and acts as a router for the multicasted message at each endpoint.

The TopoMon consumer calculates two types of spanning trees: a \textit{minimal-latency} and a \textit{maximal-bandwidth} tree. The first one optimizes the time to multicast a small message, which mostly depends on the latency of the links used. The second one optimizes the time to multicast a large message, since in that case the bandwidth of the links is the most predominant bottleneck.

The minimal-latency spanning tree is calculated by using only the NWS end-to-end forecasts. These are used to create a completely connected graph that contains only the endpoints. The weight of each link in this graph is its forecasted latency value. Currently, Dijkstra's algorithm is used to compute the tree of shortest paths from the initiator of the multicast to all other endpoints in this graph, which corresponds to the minimal-latency spanning tree.

The maximal-bandwidth spanning tree is calculated by the same procedure, but uses the reciprocals of the forecasted bandwidth values as distance metric. This is only a temporary solution, since it simplifies a lot of things and makes several assumptions about the multicasting application:

- **Endpoints do not stream messages**
  Each endpoint receives an incoming message entirely before it forwards it to its neighbors in the multicast tree.

- **Forwarding messages is (conceptually) done in parallel**
  An incoming message is forwarded to all neighbors in the multicast tree in parallel, at least conceptually. That means the time to send a message to the first neighbor does not influence the time it takes to send it to the second one.

- **The forwarding time of a message is zero**
  The application-level latency for handling the message is ignored.

- **Shared links can be ignored**
  The shared links in the paths used in the multicast tree can be ignored.

All these assumptions are valid for the minimal-latency tree, since then it is assumed that only small messages are used which do not take much time to send or forward. Forwarding of messages can therefore be done sequentially, which takes as much time as sending them in parallel. The messages are unlikely to collide in shared links since they are very small.
However, for the maximal-bandwidth tree these assumptions are not very realistic. First, it will be more efficient to forward messages in a streaming way, since it may take a while to receive the whole message completely. Second, real parallel forwarding may cause collisions on the first few hops in each path, since this typically contains a route to the outside world that is the same in many paths. The alternative of sequential forwarding, however, will take some time to send each message, so this time should be taken into account. The application-level latency may also be relevant for large messages. Finally, the ignorance of shared links can effect the usefulness of the spanning tree. We already mentioned this problem for the first few hops. In general, links may be shared by any path in the tree. This is illustrated in Figure 3.2, which shows a theoretical example topology.

The topology in Figure 3.2a shows the reciprocals of the bandwidth for each end-to-end connection between the five endpoints. To calculate the spanning tree rooted in A, Dijkstra’s algorithm generates the shortest path tree shown in Figure 3.2c. However, the end-to-end connections in this topology may contain shared links. Figure 3.2b shows the ‘real’ topology, which includes a shared link between the connections from B to D and C to E. The end-to-end bandwidths in Figure 3.2a are actually the bottleneck bandwidths of the complete path. If an application multicasts a large message according to the tree in Figure 3.2c, the bandwidth of the link between X and Y will be shared by the streams from B to D and C to E. We assume that the bandwidth of this link will approximately be equally divided between the two streams, which causes the reciprocal of its bandwidth to become 4 instead of 2. In this case the spanning tree shown in Figure 3.2d will therefore provide better overall performance since it contains no shared links. However, this tree still contains the assumption that nodes do not use ‘streaming’ (otherwise the link from D to Y would become a bottleneck).

The ‘ideal’ algorithm would incorporate all assumptions about streaming, forwarding time, etc.; simply using Dijkstra’s algorithm will probably produce suboptimal solutions. The development of such an algorithm is beyond the scope of this thesis, but will be the subject of future work. Until that time, TopoMon uses Dijkstra’s algorithm as a temporary solution for maximal-bandwidth trees.

### 3.2 Architecture

We now describe the architecture of TopoMon. First we give a birds-eye overview of the various components. Then we describe how the communication between these components takes place.

#### 3.2.1 Components

Figure 3.3 shows a global overview of TopoMon’s architecture. The parts of the NWS are shown as dotted boxes. We will now describe the various parts.
(a) An example topology, annotated with the reciprocals of the band of each end-to-end link. The dotted links all have a reciprocal of 10. All links are bidirectional for clarity of presentation.

(b) The 'real' topology, which contains two intermediate routers, X and Y, and a shared link between them.

(c) The spanning tree rooted in A as the current algorithm would generate it.

(d) The spanning tree rooted in A that takes shared links into account.

Figure 3.2: An example topology that shows some problems of the current algorithm that generates maximum-bandwidth spanning trees.
Sensors

TopoMon monitors a specific set of computers that are connected to the Internet. We call these monitored computers the endpoints (contrary to the routers that are part of the Internet, which we'll call hosts). Each endpoint runs two programs: an NWS sensor and a TopoMon sensor.

Each NWS sensor measures the latency and bandwidth between itself and all the other endpoints. The NWS sensors interact with each other and the other components of the NWS (represented in the figure by the dotted box 'NWS') by sending messages. The NWS stores the measurements internally in a Memory process.

The TopoMon sensors periodically run traceroute to all other endpoints, and forward the output of these runs to the TopoProducer.

Producers

TopoMon contains two producers: the NWS producer and the TopoProducer. The NWS producer uses the NWS API to periodically poll for new forecasts. These forecasts are forwarded as events to all subscribed consumers. A consumer can also query the NWS producer for a specific forecast. The NWS producer then immediately queries the NWS and returns the result.

The TopoProducer retrieves all the traceroute data sent by the TopoMon
Table 3.1: trade-offs of query response policies

<table>
<thead>
<tr>
<th>Policy Description</th>
<th>Response time</th>
<th>Staleness of information</th>
<th>Storage space</th>
<th>System complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Queue requests and return the next available measurement</td>
<td>high</td>
<td>young</td>
<td>medium</td>
<td>low</td>
</tr>
<tr>
<td>Take new measurements for each query</td>
<td>low</td>
<td>young</td>
<td>small</td>
<td>high</td>
</tr>
<tr>
<td>Return the last measurement</td>
<td>low</td>
<td>old</td>
<td>large</td>
<td>low</td>
</tr>
</tbody>
</table>

sensors, parses them and forwards them to all subscribed Consumer processes as events. Consumers can also query the TopoProducer for a single traceroute event. The TopoProducer currently queues such requests and returns the next available traceroute event. Alternatives would be to take a new measurement for each query or to return the last measured one. The former policy would make the communication between the TopoProducer and the TopoMon sensors and the architecture of the TopoMon sensors considerably more complex. The latter policy would require the TopoProducer to store all traceroute data, which grows in \( n^2 \) of the number of endpoints. Each policy assumes a certain usage pattern, and has a trade-off between query response time, staleness of the information, required storage space and complexity of the system components. Table 3.1 summarizes these trade-offs.

Since the query functionality of the TopoProducer is not used by the TopoMon consumer, the queueing of requests was the simplest method to implement with the lowest storage space requirements. However, other policies might be implemented in the future to enhance response time. The NWS producer uses the second policy in the table to answer queries.

**Consumer**

There's currently only one consumer process, which provides information about a subset of the endpoints to interested applications. The consumer interacts with the two producers with the XML producer/consumer protocol defined the GMA working group (see Section 3.1.3). It subscribes itself to all events of both producers for the endpoints it monitors (see Section 3.2.2). The consumer combines these data to generate the information described in Section 3.1.4. Applications can retrieve this information by using the Java interface defined in Section 4.2.5.

**Directory Service**

Both producers store information about their location and the events they produce in the Directory Service. The consumer searches this directory to retrieve the location of the producers of the various events.
3.2.2 Communication

Traceroute

The result of each run of traceroute to another endpoint is captured by the TopoMon sensor. This output is sent over the TCP connection to the TopoProducer, followed by a special character string that indicates the end of the message. An example of such a traceroute result is shown in Figure 3.4.

```
traceroute to erebor.ics.muni.cz (147.251.3.18), 30 hops max, 40 byte packets
1 colossus (130.37.26.1) 0.838 ms 0.776 ms
2 wsk71-2-001-223 (130.37.14.1) 0.818 ms 0.757 ms
3 hka16-1-d01.backbone.vu.nl (130.37.5.1) 0.983 ms 0.896 ms
4 G15-2-27.ARS.Amsterdam1.surf.net (145.145.18.57) 1.434 ms 0.935 ms
5 PO6-0.CMI.Amsterdam1.surf.net (145.145.162.4) 1.270 ms 1.187 ms
6 PO0-0.BR1.Amsterdam1.surf.net (145.145.166.2) 1.160 ms 1.053 ms
7 surfnet.nl.nl.geant.net (62.40.103.97) 1.465 ms 1.407 ms
8 nl.de1.de.geant.net (62.40.96.53) 7.637 ms 7.739 ms
9 de.cz1.cz.geant.net (62.40.96.37) 16.007 ms 15.952 ms
11 r43km-pee9-0-stm16.cesnet.cz (195.113.156.118) 19.728 ms 19.282 ms
12 routier-accl.bmo.cesnet.cz (147.251.19.68) 20.402 ms 23.471 ms
13 erebor.ics.muni.cz (147.251.3.18) 21.313 ms 21.128 ms
<END OF MESSAGE>
```

Figure 3.4: Example traceroute data sent to the TopoProducer

Producer/Consumer protocol

The protocol between producers and consumers is defined by the GMA Working Group [30]. It is a simple protocol that uses XML documents as messages. The name of the root tag of an XML document identifies the message type. The message data is contained as XML within the root tag. Messages are sent by first sending the number of bytes of the message as a 32-bit integer in network byte order, followed by the XML document.

The protocol defines three types of interaction between a producer and consumer: consumer-initiated subscription, producer-initiated subscription and request/reply interaction. These interaction types are shown in Figure 3.5.

With consumer-initiated subscription, the consumer subscribes itself to the producer for a certain event type. From that moment on the producer will send all these events to the consumer until the consumer unsubscribes itself. This is probably the most common kind of interaction, and currently the only one used by the TopoMon components. The other two are supported, however, since they might be used in the future.

In producer-initiated subscription, the producer subscribes itself to a consumer. The producer will then send events to the consumer until it unsubscribes itself. This kind of interaction is for example useful for archival purposes, as described in Section 2.2.1.

The request/reply interaction covers two cases. First, a consumer can query a producer for a single event, which provides a fast alternative for the sub-
scription method if a consumer only wants to receive a single event. Second, a consumer can ask a producer to send the names of all the events it produces. This functionality is somewhat awkward, since the directory service can also be used to do this, so it will probably never be used.

The actual communication generally takes the form of a request message to which the other side returns a reply. For example: if a consumer wants to subscribe itself to a producer, it sends a SubscribeRequest message to the producer and waits for the SubscribeReply message to return. Unsubscription, event queries and event name queries follow the same structure. The only messages that are never replied are Event messages.

All request messages contain a request ID, so the reply to the request can be identified. With a subscription, both sides generate a subscription ID that uniquely identifies the connection. Events always contain this subscription ID, so a consumer can identify the subscription they belong to. These subscription IDs are also used in an UnsubscribeRequest message to identify the subscription to terminate.

**Communication example**

To illustrate the producer/consumer protocol, we illustrate the communication of the TopoMon consumer subscribing itself to Traceroute events of the TopoProducer. First, the consumer will send a SubscribeRequest message to the TopoProducer, indicating the type of events it wants to receive. Currently, the consumer subscribes to each traceroute between two endpoints separately to enable subscription to only a subset of the events generated by the TopoProducer. One of the subscription requests will for example be:
This will subscribe the consumer to all the traceroutes between the endpoints 
das0fs.cs.vu.nl and das2fs.wins.uva.nl, as specified by the \textit{event parameters} of an event. Currently TopoMon only uses the host name of the source and 
destination machine as possible parameters for events. This makes it possible, 
for example, for a consumer to only subscribe to the \textit{NWSLatencyTcp} events of the 
NWS producer between only two endpoints. The TopoMon consumer subscribes itself to all \textit{Traceroute}, \textit{NWSBandwidthTcp} and \textit{NWSLatencyTcp} events between all pairs of endpoints it has to monitor.

The \texttt{xmlns} attributes stand for XML name space, and indicate the name space the tag is in. The names of the producer/consumer XML messages are all in the name space \texttt{http://www.gridforum.org/Performance/Protocol}. The tags of the TopoMon events and their content are all in the name space \texttt{http://www.cs.vu.nl/albatross/TopoMon}.

The TopoProducer will, for example, reply to the above shown example subscribe request with the message:

```
<SubscribeReply xmlns="http://www.gridforum.org/Performance/Protocol"
requestID="1">
  <Return>Success</Return>
  <SubscriptionID>18937</SubscriptionID>
</SubscribeReply>
```

The subscription is now known by the TopoMon consumer as ID 15433 and by the TopoProducer as ID 18937. After this the TopoMon consumer can expect \textit{Traceroute} events from the TopoProducer that match the event parameters, for example:

```
<Event xmlns="http://www.gridforum.org/Performance/Protocol"
subscriptionID="15433"/>
  <Traceroute xmlns="http://www.cs.vu.nl/albatross/TopoMon"
    probeSize="40" maxHops="30">
    ...event data...
  </Traceroute>
</Event>
```

The exact contents of a \textit{Traceroute} event is described in Section 3.2.2. Finally, the consumer will unsubscribe itself from the TopoProducer for this event by sending:
The TopoProducer will reply:

```
<UnsubscribeReply xmlns="http://www.gridforum.org/Performance/Protocol"
    requestId="4">
    <Return>Success</Return>
</UnsubscribeRequest>
```

**Events**

TopoMon uses three types of events. The NWS producer generates *NWSBandwidthTcp* and *NWSLatencyTcp* events, which contain information about the bandwidth and latency between two endpoints as forecasted by the NWS, respectively. Examples of those events are shown in Figure 3.6. Only the event data are shown.

```
<NWSBandwidthTcp xmlns="http://www.cs.vu.nl/albatross/TopoMon">
    <SourceHostName>das0fs.cs.vu.nl</SourceHostName>
    <DestinationHostName>das2fs.cs.vu.nl</DestinationHostName>
    <NWSForecast>
        <Value unit="Mb/s">2.31</Value>
        <Error>0.010219</Error>
    </NWSForecast>
    <Timestamp>2001-11-25T16:14:38Z</Timestamp>
</NWSBandwidthTcp>

<NWSLatencyTcp xmlns="http://www.cs.vu.nl/albatross/TopoMon">
    <SourceHostName>das0fs.cs.vu.nl</SourceHostName>
    <DestinationHostName>bc3m.cs.unh.edu</DestinationHostName>
    <NWSForecast>
        <Value unit="ms">12.3</Value>
        <Error>0.00354</Error>
    </NWSForecast>
    <Timestamp>2001-11-25T16:23:01Z</Timestamp>
</NWSLatencyTcp>
```

Figure 3.6: example XML descriptions of *NWSBandwidthTcp* and *NWSLatencyTcp* events

The TopoProducer converts the *traceroute* output received from the TopoMon sensors into XML descriptions. The output shown in Figure 3.4 will, for example, be converted to the *Traceroute* event shown in Figure 3.7.

The attentive reader will notice that the host names of the first two hops were reported by *traceroute* in Figure 3.4 as being “colossus” and “wsk71-2-d01-223” respectively, but in the XML version in Figure 3.7 they both got the
Figure 3.7: The traceroute output of Figure 3.4 as Traceroute event. Some hops are left out for readability.
suffix ".cs.vu.nl" added. The traceroute parser in the TopoProducer is responsible for this, because shared hosts in the topology are recognized by the TopoMon consumer by having the same host name (which provides better results than IP addresses since one host can have multiple addresses, especially routers). However, sometimes hosts report their host name without the domain suffix when they are queried from a host within the same domain, which is probably caused by some local configuration of the host name lookup. This poses a recognition problem for the TopoMon consumer, which is solved by automatically adding the domain of the source host of the traceroute to all hosts that have not got a domain suffix.
Chapter 4

Implementation

In this chapter, we look more closely at the components of TopoMon. We describe how they are implemented and work internally. We also explain how the various components can be configured and how applications can extract information from TopoMon. We start with the TopoMon sensor, followed by the TopoProducer, NWS producer and the Directory Service. Then we describe the TopoMon consumer and its interface to applications. Finally, we explain how the sensors and components can be configured.

4.1 TopoMon sensor

The TopoMon sensor is written in C to load the monitored machine as little as possible. Currently only a UNIX version has been programmed since the only monitored hosts so far all run some variant of UNIX.

Once a sensor is started, it establishes a TCP connection to the TopoProducer by using Internet sockets. The TopoProducer uses the created connection to send a message back to the connected sensor that specifies the work the sensor has to do. This 'to-do list' contains a list of the host names of the other endpoints, the command to execute and the number of seconds to wait between consecutive measurement sessions. The command is a string that contains one parameter. A typical command is for example "traceroute -w 3 -q 2 %s 40 2>&1", which executes traceroute to the host name substituted for the %s.

After the sensor has received the to-do list it enters a loop. First it executes the command to with each endpoint as a parameter and sends the output to the TopoProducer over the created connection. After that it sleeps for the number of seconds specified in the to-do list minus the time it took to gather all the tracertoes. Then the loop starts again. A sensor stays in this loop until either the sensor is killed (e.g. by sending a signal) or the TopoProducer terminates. The latter case can be recognized by the sensor by the fact that the TCP connection to the TopoProducer gets closed down.
4.2 Java packages

The GMA components of TopoMon are implemented in Java. Although Java has a larger running overhead than C or C++, it has the great advantage of being platform-independent. From version 1.4 it also has built-in support for XML parsing, which is useful for the parsing of the various messages in the producer/consumer protocol.

The Java classes of TopoMon are grouped into the following packages:

topomon: contains the implementation of the TopoProducer, the NWS producer and the TopoMon consumer.

topomon.gma: contains all the functionality to implement a producer or consumer in Java that communicates with the simple XML producer/consumer protocol

topomon.sensors: the interface to the TopoMon sensors

topomon.nws: the Java interface to the NWS

topomon.directory: contains the implementation of the directory service

topomon.xml: classes to represent XML documents as objects, which provides simple object-oriented manipulation of XML documents.

topomon.data: classes that represent data generated by TopoMon (traceroutes, NWS forecasts, graphs, etc.)

topomon.util: contains various utility classes (for example for logging)

We now describe the structure and functionality of these packages in detail.

4.2.1 GMA

The logical layers in the Java GMA implementation are shown in Figure 4.1.

<table>
<thead>
<tr>
<th>producer and consumer implementations</th>
</tr>
</thead>
<tbody>
<tr>
<td>GMAClient</td>
</tr>
<tr>
<td>GMAClientControl</td>
</tr>
<tr>
<td>GMACommunicator</td>
</tr>
<tr>
<td>Java sockets</td>
</tr>
</tbody>
</table>

Figure 4.1: Logical layers in the package topomon.gma

The functionality of a producer and consumer can be split in two parts: a client and a server part. The client part is the active one, and involves sending requests, sending queries, etc. The server part is the passive side that waits for incoming subscribe requests, query requests, etc. Although it seems
that the client is actually the consumer and the server the producer, this is only true with consumer-initiated subscription (compare Section 3.2.2). With producer-initiated subscription, the consumer also has a server part that listens for incoming requests, and the producer has an active client part that initiates connections. This requires careful division of functionality for the client and server part. Even worse, if a consumer also runs a server part, it has to be ready for events it subscribed to itself and events that arrive because a producer subscribed to the consumer and pushes its events to the consumer. This means that a GMA component (being a consumer or producer) has to be able to specify whether it has a client part, a server part or both of them, and what to do with incoming events.

Client side

A client starts by creating a GMAClient object, of which the interface is shown in Figure 4.2. This object provides two functions, `connect()` and `disconnect()`, which allows the client to setup a TCP connection to another host and terminate it, respectively. The `connect()` call returns a GMAControl object, which provides the client side functionality. Its interface is show in Figure 4.3.

```
GMAClientControl connect(String hostName, int port) throws IOException
    Connects to the given machine. If there already existed a connection to this machine,
    that connection is reused. For each connection the number of times it is reused is
    remembered. If an error occurred during connection setup an IOException is thrown.

boolean disconnect(String hostName, int port)
    Disconnects from the given machine. If the connection was reused, only the number of
    times it remains reused is decremented. Only if a connection is used no more at all it
    is really closed. If there existed no connection to the given machine nothing happens.
    Returns true if the connection existed, false otherwise.
```

Figure 4.2: Interface of a GMAClient object

Once a client subscribed to events, the incoming events are dispatched to the event handler specified upon subscription. Such an object has to implement the interface GMAEventHandler, shown in figure 4.4. For every incoming event the method eventReceived() is called with the local subscription identifier associated with the event and the event data themselves.

Server side

A server will start by creating a GMAServer object. This is a Runnable object, so the server can create a new thread that runs the object. The constructor of a GMAServer object is:

```
GMAServer(int port, GMAServerMessageHandler serverMessageHandler, Log log)
    throws IOException
```
Identifier subscribe(XMLNode eventParameters, GMAEventHandler eventHandler)
  throws SubscribeFailedException
  Subscribes to the the event type specified by the event parameters. Future incoming
  events will be handled by the given event handler. Returns the local identifier
  associated with this subscription. If the subscription failed for some reason, a
  SubscribeFailedException is thrown.

void unsubscribe(Identifier localSubscriptionId) throws UnsubscribeFailedException
  Unsubscribes from the subscription with the given local identifier. If the unsubscribe
  failed for some reason or there was no subscription associated with the given local
  identifier, an UnsubscribeFailedException is thrown.

XMLNode query(XMLNode eventParameters) throws QueryFailedException
  Queries the remote side for an event specified by the given parameters. Returns the
  returned event. If the query failed for some reason, a QueryFailedException is thrown.

XMLNode[] eventNames() throws QueryFailedException
  Asks the remote side to return the names of all the events it generates or accepts. Each
  name is modeled as an XML element with the attributes params and events that
  specify the name space of the parameter tags and the event tags, respectively. The
  contents of the XML element is a single String that forms the name of the event.
  If the query for the event names failed for some reason a QueryFailedException is
  thrown.

void close()
  Closes the control handle. It is the responsibility of the application to make sure that
  all subscriptions are correctly unsubscribed before calling this function.

Figure 4.3: Interface of a GMAClientControl object

public interface GMAEventHandler {

  public void eventReceived(Identifier localSubscriptionId,
                             XMLNode event);

}

Figure 4.4: The GMAEventHandler interface
The *port* variable specifies the port number to which the server listens for incoming connections. The *serverMessageHandler* is the object that will handle incoming messages, and will be explained in a moment. The *log* variable is a utility object that provides a uniform way of logging to all TopoMon components. If the creation of a passive server socket fails, an *IOException* is thrown.

Once the GMAServer is created, it listens to the given port and creates a new *GMAServerControl* object for each incoming connection. This object also implements the *Runnable* interface and is started in a new thread. It listens continuously to the created connection. If a message arrives, it is dispatched to the corresponding call-back method in the server message handler. A server message handler implements the *GMAServerMessageHandler* interface, which is shown in Figure 4.5. Implementing the four methods specified by this interface is all one has to do to create a working server side of a GMA component.

```java
void subscribeRequest(Identifier requestId, Identifier subscriptionId, 
                     XML.Element eventParameters, GMACommunicator replyChannel);
```

Called every time a *SubscribeRequest* message arrives. The *subscriptionId* is the identifier the other side uses for the subscription. The event parameters specify the type of event the other side wants to subscribe to. The reply channel can be used to send a *SubscribeReply* message, which should contain the given *requestId*.

```java
void unsubscribeRequest(Identifier requestId, Identifier subscriptionId, 
                       GMACommunicator replyChannel);
```

Called every time an *UnsubscribeRequest* message arrives. The *subscriptionId* is the identifier the other side uses for the subscription. The reply channel can be used to send an *UnsubscribeReply* message, which should contain the given *requestId*.

```java
void queryRequest(Identifier requestId, XML.Element eventParameters, 
                  GMACommunicator replyChannel);
```

Called every time an *QueryRequest* message arrives. The event parameters specify the type of event the other side wants to receive. The reply channel can be used to send a *QueryReply* message, which should contain the given *requestId*.

```java
void eventNamesRequest(Identifier requestId, GMACommunicator replyChannel);
```

Called every time an *EventNamesRequest* message arrives. The reply channel can be used to send an *EventNamesReply* message, which should contain the given *requestId*.

Figure 4.5: The *GMAServerMessageHandler* interface

**Communication**

The *GMACommunicator* object that appears in the *GMAServerMessageHandler* interface is responsible for the actual communication between GMA components and dispatching of messages to the respective handler. It is used by both a *GMAClientControlHandler* and the *GMAServerControlHandler*. Its interface is shown in Figure 4.6. The *send()* and *receive()* methods use XML documents as messages. The time to wait for an incoming message is bound by a maximum number of milliseconds to prevent a deadlock when for example the remote side crashed. Higher layers can register event handlers in an *GMACommunicator*,

36
which will from then on take care of the correct dispatching of Event messages. The GMAClientCommunicator uses this for example in the subscribe() and unsubscribe() methods to activate the supplied event handler.

```java
void send(XMLDocument document) throws IOException
   Sends the given XML document to the other side. If something went wrong while sending the document, an IOException is thrown.

XMLDocument receive(long maxWaitTime) throws IOException
   Waits at most maxWaitTime milliseconds for the next XML document to arrive. Returns the first received XML document. If something went wrong while reading from the socket, or the maximum number of milliseconds to wait elapsed, an IOException is thrown.

XMLDocument receive(String rootElementName, long maxWaitTime)
   Waits at most maxWaitTime milliseconds for the next XML document to arrive of which the root element's name corresponds to rootElementName. Other XML documents that arrive before the specified one are discarded. Returns the first received XML document with the specified root element's name. If something went wrong while reading from the socket, or the maximum number of milliseconds to wait elapsed, an IOException is thrown.

void addEventListener(Identifier localSubscriptionId, GMAEventHandler eventHandler)
   Adds an event handler that will be called when Event messages for the subscription associated with given local identifier arrive.

void deleteEventHandler(Identifier localSubscriptionId)
   Removes the registered event handler that handles events of the subscription associated with the given local identifier. If there exists no such event handler nothing happens.

void close()
   Closes this communicator. All observers are notified about this before the communication channels are actually closed, so this communicator can still be used by them to send or receive urgent last messages.

String getHostId()
   Returns a unique identification string of the remote host, which consists of the host name and port number of the remote host, separated by a colon.
```

Figure 4.6: Interface of a GMACommunicator

To make sure that Event messages are delivered immediately to the registered event handlers, a GMACommunicator internally starts another thread that runs a MessageListener object. This object continuously listens for incoming messages. If an Event message arrives, the message listener calls the eventReceived() method of the associated event handler. Other messages are put in a message queue. When a higher layer calls a receive() method of a GMACommunicator it checks the message queue for messages, and blocks when the queue is empty until the message listener has put a new message into the queue. This internal structure is illustrated in Figure 4.7.

The creation of the various message in the producer/consumer protocol
is handled by the singleton GMAMessageFactory. This object provides a builder method for every type of message to ease the creation of messages.

Example producers and consumers

It is easy to create all the types of consumers and producers possible in the GMA with the classes described in the previous sections.

A consumer that only supports consumer-initiated subscription will create a GMAClient object that is used to open connections to the producers it has interest in. For each opened connection it gets a GMAClientControl object that provides all the functionality to interact with each of the producers. The consumer also has to implement the GMAServerEventHandler interface so it can process the incoming events. This is how the GMA part of the TopoMon consumer process is implemented.

A more advanced consumer that also accepts producer-initiated subscriptions will, besides the GMAClient object, create a GMAServer object that listens for incoming connections from producers. It will implement the GMAServer-MessageHandler interface to handle the request messages sent by producers. Once a producer sends a SubscribeRequest message, the consumer can add the event handler that will process the events the producer is about to send to the GMACommunicator that is used as the reply channel. This direct addition and removal of the event handler in the GMACommunicator can also be used to deactivate the event handler once the producer has sent an UnsubscribeRequest message.

A producer will always use a GMAServer object to accept incoming connections from consumers. For each incoming subscription it will have to remember
the event parameters that specify the type of event the consumer is interested in, the subscription identifier the consumer uses and the GMACommunicator that can be used to communicate with the consumer. To ease this bookkeeping, a ConnectionTable object can be used. Its interface is shown in Figure 4.8. Once an event is available, the producer can walk through all the subscriptions to see if the event matches with the event parameters a consumer specified. If that is the case, the event, which has to be tagged with the remote identifier the consumer specified, can be sent to the consumer by using the GMACommunicator stored in the connection table. In both the TopoProducer and the NWS producer the GMA part is implemented in this way.

A compound producer/consumer is simply created by using a GMAClient object for the consumer part and a GMAServer for the producer part.

```java
Identifier add(Identifier remoteId, XML Element eventParameters,
                      GMACommunicator channel)
      Adds information about a connection to the table (the identifier of the connection that is used by the remote side, the parameters that specify the type of event the remote side is interested in and the channel that can be used to communicate with the remote side). Returns the local identifier under which the information is stored.

void remove(Identifier localSubscriptionId)
      Removes the connection information stored under the given local identifier. If this identifier is unknown nothing happens.

Enumeration getAllSubscriptionIDs()
      Returns an enumeration of all the local identifiers under which connection information is stored.

Identifier getRemoteId(Identifier localSubscriptionId)
      Returns the remote identifier of the connection associated with the given local identifier, or null if the local identifier is unknown.

XML Element getEventParameters(Identifier localSubscriptionId)
      Returns the event parameters of the connection associated with the given local identifier, or null if the local identifier is unknown.

GMACommunicator getChannel(Identifier localSubscriptionId)
      Returns the channel of the connection associated with the given local identifier, or null if the local identifier is unknown.

boolean contains(Identifier localSubscriptionId)
      Indicates whether connection information is stored under the given local identifier.

void clear()
      Removes all connection information in the table.

int size()
      Returns the number of connections that are stored in this table.
```

Figure 4.8: Interface of a ConnectionTable
4.2.2 TopoMon Sensors

The package topomon.sensors contains interface to the TopoMon sensors for the TopoProducer. Figure 4.9 shows the logical layers in this package and how it relates to the TopoProducer. The latter uses the package topomon.gma to implement its GMA functionality, and the topomon.sensor package to communicate with the TopoMon sensors.

<table>
<thead>
<tr>
<th>TopoProducer</th>
<th>package topomon.gma</th>
</tr>
</thead>
<tbody>
<tr>
<td>TopomonSensorServer</td>
<td></td>
</tr>
<tr>
<td>TopomonSensorControl</td>
<td></td>
</tr>
<tr>
<td>TopomonSensorCommunicator</td>
<td></td>
</tr>
<tr>
<td>Java sockets</td>
<td></td>
</tr>
</tbody>
</table>

Figure 4.9: The logical layers in the package topomon.sensors

When the TopoProducer is started it creates a TopomonSensorServer object. This object is similar to a GMAServer object, as is its constructor:

```java
TopomonSensorServer(int port, TopomonSensorEventHandler sensorEventHandler,
                    Log log) throws NOException
```

The `port` variable specifies the port to which the server listens for incoming connections. The sensor event handler is the object that will handle all the traceroute results generated by the sensors. Such an event handler implements the TopomonSensorEventHandler interface, which is shown in Figure 4.10. The `log` object is used for logging.

```java
public interface TopomonSensorEventHandler {

    public void tracerouteReceived(TracerouteResult traceroute);
}
```

Figure 4.10: The TopomonSensorEventHandler interface

The TopomonSensorServer listens for incoming connections from Topomon sensors and spawns a new TopomonSensorControl object in a new thread for each incoming connection. This is only done if the machine from which the connection attempt originates is known by the TopoProducer. In other words: only known sensors are allowed to connect. This provides a minimal form of security. Which sensors are known is specified in the configuration file, which is described in Section 4.3.

The TopoMon sensor control object takes care of sending the to-do list back to the connected sensor by using a TopomonSensorCommunicator object. Its interface is shown in Figure 4.11.
void sendToDoList(int interval, String[] hostNames,
      String tracerouteCommand) throws IOException
      Sends a to-do list to the TopoMon sensor managed by this object. This consists of the
      interval (in seconds) the TopoMon sensor has to perform measurement sessions with,
      the host names to which the TopoMon sensor should run a command and the command
      to run. Use Xs in the latter string to indicate the place of the destination host name
      in the command. If an error occurred while sending the to-do list, an IOException is
      thrown.

     TracerouteResult receive() throws IOException
     Waits for the next traceroute result to arrive, parses it and returns the resulting object.
     This is a blocking call. If an error occurred during the waiting, an IOException is
     thrown. It is safe to assume that when this happens, the connection with the TopoMon
     sensor is lost.

Figure 4.11: Interface of a TopomonSensorCommunicator

After sending the to-do list the TopoMon sensor control object just listens
for incoming traceroute results and forwards the results to the TopomonSensor-
EventHandler by calling the method tracerouteReceived().

The actual parsing of the traceroute output is done by the singleton
TracerouteEventFactory in the package topomon.data. This object uses re-
cursive descent as the parsing technique, and creates a TracerouteResult ob-
ject that represents the traceroute data as a Java object. Since the traceroute
tool does not give exactly the same output on every platform it is up to the
parser to be as flexible as possible to understand all these output formats. So
far, the encountered differences are minor, such as a slightly different header
text. Coping with these difference is relatively easy.

4.2.3 NWS interface

To let the NWS producer extract forecasts out of the NWS, the C API of the
NWS is used in combination with the Java Native Interface. To build a bridge
between the Java interface and the NWS C API, a C library called libnws.so
is created that acts as a wrapper around the NWS API. This structure is shown
in Figure 4.12.

<table>
<thead>
<tr>
<th>NWS producer</th>
</tr>
</thead>
<tbody>
<tr>
<td>NWSInterface</td>
</tr>
<tr>
<td>libnws.so</td>
</tr>
<tr>
<td>NWS API</td>
</tr>
<tr>
<td>NWS components</td>
</tr>
</tbody>
</table>

Figure 4.12: Structure of the NWS wrapper

The class NWSInterface in the topomon.nws package provides the Java in-
terface to the NWS. Figure shows the interface of this class.
Figure 4.13: Interface of the NWSInterface object

The NWSInterface uses the following external methods:

- public native boolean setNameServer(String hostname, int port)
- public native NWSForecast getForecast(String source, String destination, String resource, double sinceWhen)

These methods are implemented in C in the library libnws.so in the JNI syntax:

- JNIEXPORT jobject JNICALL Java_topomon_nws_NWSInterface_setNameServer(JNIEnv* env, jobject obj, jstring hostname, jint port)
- JNIEXPORT jobject JNICALL Java_topomon_nws_NWSInterface_getForecast(JNIEnv* env, jobject obj, jstring sourceHostname, jstring destinationHostname, jstring resource, jdouble sinceWhen)

The libnws.so library is linked with the NWS API and uses the functions available there to implement the retrieval of forecasts.

4.2.4 Directory service

The interface to the directory service is shown in Figure 4.14. Each producer and consumer publishes information about itself by adding Searchable objects to the directory service that contain the information. The package topomon.directory contains four Searchable objects that resemble the information envisioned by the GMA working group [29].

- EventType, which specifies a type of event. Each event type has a unique name. TopoMon currently uses three event types, with the names “Trace-route”, “NWBanderwidthTcp” and “NWSLatencyTcp".
DirectoryService getInstance()
Returns the unique DirectoryService object.

void add(Searchable info)
Adds information about an GMA component to the directory service. A deep copy of the information is made before it is added.

void remove(Searchable info)
Removes the first information information that equals the given one from the directory service. The given information is compared to existing information with the equals() method. If no such information exists nothing happens.

void replace(Searchable oldInfo, Searchable newInfo)
Replaces existing information in the directory service with new information. This is actually an atomic combination of a remove() followed by an add().

Searchable[] search(Filter filter, int maxResults)
Searches in the directory service for information that matches the given filter. Maximal maxResults results are returned. If multiple copies of the same information exists, only one of them is returned.

Figure 4.14: Interface of the DirectoryService

- **EventInstance**, which is a combination of an EventType object and the event parameters of an event (which is a also Searchable object). We will give an example of an event instance to clarify this. The type of an event could, for example, be “Traceroute” for a Traceroute event. The package topomon.data contains an TracerouteEventParameters object that implements the Searchable interface and contain the possible event parameters of a Traceroute event: the host names of the source and destination endpoint. The combination of an EventType object with the name “Traceroute” and an TracerouteEventParameters object with, for example, the host names “das06.cs.vu.nl” and “erebor.ics.muni.cz” creates an event instance of a Traceroute event between these two endpoints.

- **ProducerInfo**, which represent information about a producer in a Grid:
  - the name of the producer
  - the name of the host it runs on
  - the port is listens to
  - a human-readable description of the producer
  - the event instances it produces (as EventInstance objects)
  - the protocols it supports (currently only the XML producer/consumer protocol)

- **ConsumerInfo**, which represent information about a consumer in a Grid:
  - the name of the consumer

43
– the name of the host it runs on
– the port it listens to (currently unused)
– a human-readable description of the consumer
– the protocols it supports (currently only the XML producer/consumer protocol)

These objects are all subclasses of Searchable. The interface of the abstract object Searchable is shown in Figure 4.15. Any object that inherits Searchable can be stored in the directory service. In this way all information entries in the directory service are modeled as Java objects.

```
Comparable getSimpleFilterField(int id)
    Returns the simple field with the given identifier, or null if the identifier is unknown.

Searchable getComplexFilterField(int id)
    Returns the complex field with the given identifier, or null if the identifier is unknown.

Iterator getCompoundFilterField(int id)
    Returns the compound field with the given identifier, or null if the identifier is unknown.

void addedToDirectory()
    This method is called every time this object is added to the directory service. Subclasses can use this method to add searchable components of themselves to the directory service, so they can be found separately. The default implementation does nothing.

void removedFromDirectory()
    This method is called every time this object is removed from the directory service. Subclasses can use this method to remove components from the directory service that were recursively added by addedToDirectory(). The default implementation does nothing.

boolean matches(Filter filter)
    Checks if this object matches the given filter. Returns true if it does, false if not.
```

Figure 4.15: Interface of the abstract class Searchable

The directory service can examine the contents of each entry by using the methods of a Searchable object. From the point of view of the directory service, a Searchable object can contain three types of fields:

– simple fields, that implement Java's Comparable interface (such as Strings, Integers etc.).
– complex fields, which are other Searchable objects
– compound fields, which implement a linear structure of simple or complex fields (such as an array of Strings or a Vector of Searchable objects).
Depending on the type of fields a class actually contains, it overrides one or more of the first three functions of the Searchable object it inherits. Each field is given an index (which is typically available as a public constant in the object), so it can be retrieved in a uniform way.

Each Searchable can be matched against a Filter object. A filter object specifies a number of tests to perform. A Searchable object matches a filter if either all tests in the filter are successful or at least one of them is. Which behavior is used is specified when the filter is constructed.

A filter can contain three types of tests:

- simple tests, that test if a simple field is smaller, equal to or greater than a certain value
- complex tests, that test if a complex field matches a certain filter
- compound tests, that tries to match elements of a compound field with a certain filter. The type of test depends on the quantifier, which specifies if either all elements should match the filter, or at least one of them should.

Besides these tests a filter always tests if the Searchable object to which it is matched has the Java class type that was specified when the filter was constructed. This prevents comparing objects that are incomparable. For example, trying to match a filter that tests certain fields in a ProducerInfo object cannot be used to match a ConsumerInfo object since the field identifiers of the former will have an undefined meaning in the latter.

If a producer or consumer wants to search the directory service for certain types of information, it creates a filter that matches the information wanted, and gives this to the search method in the directory service. The latter will then try to match the filter with all entries in the directory service. The matching entries will be returned.

The use of filters to match Searchable hides the technique behind the directory service interface completely. Which method is used internally to store, search and manage the information is completely hidden from the user. Currently, the information is simply stored in a central file by serializing the stored Java objects. A central lock file is used to prevent race conditions. This is only a temporary solution, since it does not provide a 'real' directory service that is accessible from anywhere in the world. The use of a central file requires that all the producers and consumers run on machines that share the same file system. This will be replaced in the future with a technique that does not have this limitation.

Finally, to clarify the use of filters, we give some code examples:

- Find all event instances in the directory service (maximal 100):

  ```java
  Filter f = new Filter(EventInstance.class);
  Searchable[] result = DirectoryService.getInstance().search(f, 100);
  ```

- Find a producer that generates Traceroute events:
Filter(Class objectClass, int testCombine)

Creates a new filter that filters objects of the given object class. If testCombine is
Filter.AND, the results of the tests added to this filter will be combined by ANDing
the results. The alternative Filter.OR, which will OR the results.

void init()

Removes all tests.

void addSimpleTest(int id, int operation, Comparable value)

Adds a simple test that will compare the field with the given identifier with the
given value. Possible comparing operations are Filter.LESS_THAN, Filter.EQUALS and
Filter.GREATER_THAN, which will check if the field with the given identifier is less than,
equal to or greater than the given value, respectively.

void addComplexTest(int id, Filter filter)

Adds a complex test that will check if the field with the given identifier matches the
given filter.

void addCompoundTest(int id, int quantifier, Filter filter)

Adds a compound test that will check if the elements of the field with the given identifier
match the given filter. If the quantifier is Filter.ALL_LIKE, all elements have to match.
If it is Filter.EXISTS_LIKE, only one element has to match.

Figure 4.16: Interface of a Filter

Filter filterEventType = new Filter(EventType.class);
filterEventType.addSimpleTest(EventType_FILTER_NAME, Filter.EQUALS, "Traceroute");

Filter filterEventInstance = new Filter(EventInstance.class);
filterEventInstance.addComplexTest(EventInstance.TYPE, filterEventType);

Filter f = new Filter(ProducerInfo.class);
f.addCompoundTest(ProducerInfo_FILTER_EVENT_INSTANCES, Filter.EXISTS_LIKE, filterEventInstance);
Searchable[] result = DirectoryService.getInstance().search(f, 1);

4.2.5 TopoMon API

The TopoMon consumer uses the GMA functionality and directory service to
retrieve Traceroute, NWSBandwidthTcp and NWSLatencyTcp events. Only the most recent traceroute results between all the machines are stored. The same
is true for the NWS forecasts: only the most recent forecast for each pair of
endpoints is stored.

Applications can use the TopoMon object in the package topomon to retrieve the ‘combined information’ described in Section 3.1.4. Figure 4.17 shows the
interface of TopoMon. The interface uses two additional objects. A Host object
is simply a combination of a DNS name and one or more IP addresses. A
DirectedGraph represents a directed graph that can be manipulated. Both
object are all part of the topomon.data package.

Note that during startup, it takes some time before TopoMon has collected all
the information. This means that if, for example, the topology is exported
TopoMon(String[] hostNames) throws ConnectException
Constructs a TopoMon object that provides information about the Internet topology
between the given hosts. When some of the given host names are not valid, one of
them does not run a TopoMon sensor and a Network Weather Service sensor or the
TopoProducer and NWS producer could not be reached, a ConnectException is thrown.

void exportToGraphviz(String fileName, int annotateLinks) throws IOException
Writes the current Internet topology between the observed endpoints into the given file
in the format as defined by the GraphViz program. The annotateLinks variable indicates
what kind of information is annotated to the links in the graph. Possible values are
the constants ANNOTATE_NONE, ANNOTATE_LATENCY and ANNOTATE_BANDWIDTH. Any other
value has the same effect as ANNOTATE_NONE. If the specified file is not writable or
something goes wrong while writing, an IOException is thrown.

Host[] getPath(String sourceHostName, String destinationHost)  
throws UnknownHostException
Returns the Internet path between two endpoints. This is a list of hosts/routers through
which a packet is routed when sent from sourceHostName to destinationHost. The
first host is always sourceHostName. The last host should be destinationHost,
but can be another host if the destination host was not reachable. In that case the path
is returned as far as TopoMon could reconstruct it. Unknown hops in the path are
represented as null. If the given hosts are not known endpoints an UnknownHostException
is thrown. If the path is not known yet, null is returned.

double getLatency(String sourceHostName, String destinationHostName)
throws UnknownHostException
Returns the latency in milliseconds between two endpoints, as predicted by the Network
Weather Service. If the given hosts are not known endpoints an UnknownHostException
is thrown. If the latency is not known yet, -1.0 is returned.

double getBandwidth(String sourceHostName, String destinationHostName)
throws UnknownHostException
Returns the bandwidth in megabits per second between two endpoints, as predicted by
the Network Weather Service. If the given hosts are not known endpoints an
UnknownHostException is thrown. If the bandwidth is not known yet, -1.0 is returned.

Host[][] getSharedSubpaths(String sourceHostName1, String destinationHostName1,
String sourceHostName2, String destinationHostName2)
throws UnknownHostException
Returns the shared part[s] of the paths between sourceHostName1 and
destinationHostName1 and between sourceHostName2 and destinationHostName2.
Note that this may consist of several distinct sub-paths. The first dimension of the
returned array indicates the sub-path. The second dimension contains the actual sub-
path. Each sub-path consists of one or more hosts. If the given hosts are not known
endpoints an UnknownHostException is thrown. If the two paths are not known yet,
null is returned.

DirectedGraph getMinimumLatencyTree(String hostNameRoot)
throws UnknownHostException
Returns a spanning tree that originates in hostNameRoot and connects it with a minimal
latency to all the other hosts TopoMon observes. If the root is not a known endpoints
an UnknownHostException is thrown.

DirectedGraph getMaximumBandwidthTree(String hostNameRoot)
throws UnknownHostException
Returns a spanning tree that originates in hostNameRoot and connects it with a max-
imum bandwidth to all the other hosts TopoMon observes. If the root is not a known
endpoints an UnknownHostException is thrown.

Figure 4.17: Interface of a TopoMon object
immediately after the creation of the TopoMon object, it will contain no or only a few endpoints.

4.3 Configuration

The components of TopoMon can be divided in two types: system components and user components. User components are controlled by normal users, system component by the administrator of a TopoMon system. Currently, the only user component is the Topomon consumer. The system consists of the TopoMon sensors, the TopoProducer and the NWS producer. In this section we describe how the system components can be configured.

4.3.1 Sensors

Each TopoMon sensor has to know the host name of the machine on which the TopoProducer runs and the port it listens to. This is either specified as command line arguments when a TopoMon sensor is started or by setting the environment variables TOPO_PRODUCER and TOPO_PRODUCER_PORT to the host name and port number of the TopoProducer, respectively. This is typically done in a shell configuration file that is initialized during startup.

4.3.2 Producers

The producers of TopoMon can be configured with a configuration file called .topomonrc, which should reside in the home directory of the user that starts the producers. Figure 4.18 shows an example of a configuration file for a TopoMon system that monitors three endpoints.

```
# TopoMon sensor configuration file

# TopoProducer
topo_server flite.cs.vu.nl:2309

# NMS name server
nms_name_server das0fs.cs.vu.nl:8090

# endpoints
topo_sensor das0fs.cs.vu.nl  "tracert -q 2 -w 3 %s 40 2>&1" 900
topo_sensor erebor.ics.suni.cz  "tracert -q 2 -w 2 %s 40 2>&1" 900
topo_sensor xomui1.inria.fr  "tracert -q 2 -w 3 -a %s 40 2>&1" 900
```

Figure 4.18: An example configuration file of a TopoMon system with three endpoints

Each entry starts with the name of a TopoMon component. A topo_server line contains the host name of the machine on which a TopoProducer runs and
the port number it uses. When a TopoProducer is started it reads the configuration file and tries to find its current host name. If the name is present the TopoProducer knows which port number to use. If it is not, the TopoProducer quits with an error telling the user it was started on a wrong machine. In this way the system checks that it is setup as specified by the administrator, and prevents obvious human errors. Although there's currently only one TopoProducer, there can be multiple ones, each with its own line in the configuration file. It is then up to the administrator to decide which TopoMon sensors contact which TopoProducer.

The NWS producer reads the host name and port number of the NWS name server in the configuration file in the line starting with nws_name_server. This information is given to the NWS API, so it can contact the name server to find the forecaster process that is used to generate the forecasts needed by the NWS producer.

A TopoProducer also has to know the host names of all the endpoints, the command each TopoMon sensor at these endpoints has to execute and the frequency with which it has to do that. This is used to generate the to-do list for each sensor, and to check which hosts are allowed to connect (as described in Section 4.1). Each endpoint has its own line in the configuration file, starting with topo_sensor. For example: the configuration file in Figure 4.18 specifies that the TopoMon sensor running on the endpoint das0fs.cs.vu.nl will be instructed to execute the command "traceroute -q 2 -w 3 %s 40 2>&1" to the hosts erelbor.ics.muni.cz and ronbul.imria.fr every 15 minutes. The %s in the parameters indicates the place of the host name of the destination. The 2>&1 makes sure that the output to standard error is also captured, since for some reason traceroute sends the first line of the output to standard error instead of standard out. The separate commands per endpoint makes it possible to take advantage of specific options of different versions of traceroute on the endpoints.
Chapter 5

Results

We tested TopoMon on a testbed consisting of seven sites around the world\footnote{We like to thank the following people for making this testbed possible: Ludek Matyska and Mirek Ruda at Masaryk University, Phil Hatcher at the University of New Hampshire, Graziano Oberti at UC Santa Barbara, Thomas Kunz and John Knox at Carleton University and Guillaume Pierre at the Vrije Universiteit. Also thanks to Martin Swany for his help with the Network Weather Service.}: 

- \texttt{das0fs.cs.vu.nl} at the Vrije Universiteit, the Netherlands
- \texttt{das2fs.wins.uva.nl} at the University of Amsterdam, the Netherlands
- \texttt{erebor.ics.muni.cz} at the Masaryk University Brno, Czech Republic
- \texttt{huron.cs.unh.edu} at the University of New Hampshire, USA
- \texttt{raven.cs.ucsb.edu} at the UC Santa Barbara in California, USA
- \texttt{wmc-04.sce.carleton.ca} at Carleton University in Ottowa, Canada
- \texttt{xombul.inria.fr} at INRIA in Rocquencourt, France

The sites were selected for their geographical dispersion, so together they are representative for a world-wide Grid environment.

5.1 Topology visualization

The size of the complete network topology between a number of sites grows in \(n^2\) of the number of endpoints. This makes it hard to visualize the topology; even with only seven endpoints in our testbed, it is hard to keep a visualization of this topology human-readable.

Figure 5.1 shows the complete topology of our testbed, as it was on February 20th, 2002. It shows all intermediate nodes, and each link is annotated with an estimate of its minimal latency in milliseconds. The seven endpoints are shown as black boxes.
Figure 5.1: Uncompact topology of the testbed, annotated with the estimated minimal latency (in ms) per link
By compacting the topology, as described in Section 3.1.4, its readability increases considerably. Figure 5.2 shows the compacted version of the testbed topology. The seven endpoints are now clearly visible. Figure 5.3 shows the same compacted topology, now annotated with an estimate of the maximum bandwidth per link. These compacted topologies essentially show all shared links of the paths between all endpoints.

The topologies reveal some practical difficulties related to topology monitoring. First, not all hosts are discovered by traceroute. Sometimes routers just do not reply to traceroute's probes. Sometimes routers are confused. Sometimes it is just completely unclear what the problem is. These unknown hosts are represented as a single ellipse-shaped host with the name “UNKNOWN”. The latency of the links from and to this 'host' is not known of course, so these links are annotated with question marks. This does not apply to the maximum bandwidth of these links, since this is the maximum of all the end-to-end bandwidths of the connections that use the links.

Second, not all traceroute runs reach the destination host. This may be caused by a gateway that drops traceroute probes, buggy router software or strange network configurations. An example of this problem is the endpoint wmc-04.sce.carleton.ca. All traceroute runs to this host stop at the host filter-onet.carleton.ca which reports the endpoint is unreachable. Such 'dead ends' in the topology are also visualized as ellipses instead of boxes.

These visualizations can for example be used for logging and debug purposes. One could for example write a simple topology-logging program that exports the Grid network topology every hour, and makes this available to users of the Grid for informational purposes. It can also be used by network administrators to identify problems.

### 5.2 Multicast trees

The spanning tree functionality in TopoMon has been used to generate application-level spanning trees with either minimal-latency or maximum-bandwidth for the example topology.

Figure 5.4 shows the logical end-to-end topology of the testbed. Each end-to-end connection is annotated with the NWS prediction of its latency. This topology is used by TopoMon to calculate spanning trees that optimize the multicasting of a small message to all the other endhosts. Figure 5.5 shows how the spanning trees would look in this case.

The end-to-end topology annotated with bandwidth predictions (shown in Figure 5.6) is used to calculate spanning trees for each endpoint with optimal bandwidth, shown in Figure 5.7. Recall from Section 3.1.4 that these trees currently make assumptions about the application and ignore the shared links.
Figure 5.2: Compact topology of the testbed, annotated with the estimated minimal latency (in ms) per link
Figure 5.3: Compacted topology of the testbed, annotated with an estimate of the maximum bandwidth per link.
Figure 5.4: End-to-end topology showing the latency (in ms) of the application-level connections.

Figure 5.5: Spanning trees with minimal latency for each endpoint. The links are annotated with their predicted end-to-end latency (in ms).
Figure 5.6: End-to-end topology with the prediction of the application-level bandwidth between all endpoints

(a) huron.cs.unh.edu

(b) wmc-04.sce.carleton.ca

(c) erebor.ics.muni.cz

(d) raven.cs.ucsb.edu

(e) das2fs.cs.vu.nl

(f) das2fs.wins.uva.nl

(g) xombul.inria.fr

Figure 5.7: Spanning trees with maximum bandwidth for each endpoint. The links are annotated with their predicted end-to-end bandwidth (in Mb/s)
Chapter 6

Conclusions and future work

We have presented TopoMon, a monitoring tool for Grid networks that reports the predicted end-to-end latency and bandwidth between all monitored end-points and can identify the shared links in the given topology. This information can be used by higher-level applications to predict their communication performance and to avoid congestion on links that are shared by multiple, simultaneous data streams.

TopoMon reports simple multicast spanning trees based on shortest path trees that minimize either overall latency (for multicasting short messages) or overall bandwidth (for long messages). TopoMon’s information can be easily used by other applications by implementing a custom consumer process.

The current implementation of TopoMon can be optimized and extended in several ways, which we will discuss in the following sections.

6.1 Measurements

The periodicity of performing measurements is currently fixed: both NWS and TopoMon let the user specify periodicities at the start the monitoring process, after which it cannot be changed. For topology discovery, it might be better to use a more dynamic scheme. A sensor could for instance keep track of how often a route to a certain endpoint changes, and adapt the measurement frequency accordingly. This would reduce the staleness of the topology information. To express this more clearly, TopoMon should be able to report how old the information is, something that is currently missing.

Another problem is whether concurrent measurements influence each other. The NWS uses a token protocol to avoid concurrent measurements. We argue that for topology discovery such mutual exclusion is not necessary: the probes used by traceroute are primarily used to discover intermediate hosts, not to measure link statistics. The link-latency estimates based on the probe round-trip times are only used in the visualization process, and use only the minimum value. If concurrent measurements cause this value to be higher than normal,
TopoMon simply uses a too conservative estimate.

6.2 GMA functionality

Currently, a consumer has to subscribe to every measurement event between a source and destination separately. This does not scale to large testbeds, since it requires $N \times (N-1)$ subscriptions to receive all measurements between $N$ endpoints. A more efficient way would be to allow multiple source and destination hosts as event parameters, which subscribes the user to all possible combinations.

The GMA is still in an experimental stage. Which directory service technique or meta-information descriptions to use is an open question. Currently, XML seems a promising technique to express the data, which is the reason TopoMon uses it. XML schema’s can provide the necessary meta-information, although this is currently unused since there are only two producers and one consumer.

6.3 Scalability

The TopoMon sensors have the same scalability problem as the NWS: the number of measurements grows exponentially. This introduces two problems:

- **Route discovery takes too much time**
  The inherent parallelism of each sensor solves a part of this problem: sensors perform traceroute runs concurrently. However, if the number of endpoints grows to several hundreds, the measurements would already take a lot of time. The most time-consuming part of each run is the 'discovery' of unknown hops. This requires a timeout of a probe, which can be set to a minimum of 2 seconds in most traceroute implementations. Several optimizations can be thought of to solve this, such as dynamically tweaking the number of probes to use (possible on a per-route basis) or detecting fixed paths of routes (e.g. if the first two hops are always the same two gateways to the outside world, the initial probe can be sent with a TTL of 3).

- **Route discovery introduces too much overhead**
  This is a difficult problem with every monitoring system: how to prevent the system itself from becoming too intrusive. The measurements are supposed to increase the overall performance of the system, but if they introduce too much overhead, the overall system performance may decrease. The problem with the traceroute runs is that they generate quite a lot of network traffic. To limit the amount of data sent over the network, the sensors should try to use as less probes as possible.

In the long run, a hierarchical architecture such as the cliques in the NWS or Collectors in Remos may be the only solution to enhance scalability, but how to
use that while still discovering all shared links is an open research question. Even worse, the clique hierarchy currently used by the NWS is inherently incomplete when higher-level cliques contain more than two hosts (recall Section 2.3.1). To aggregate the NWS measurements of multiple cliques, routing information has to be combined with the clique layout. TopoMon already gathers this information. How to combine it with the aggregation of NWS measurements in multiple cliques will be the subject of future research.

However, making only the sensors scale is not the only problem. If the number of endpoints grows, the producers of the events become the bottleneck. This can be solved by using multiple producers of the same type. This is already possible in the GMA, but would require some changes to the behavior of a TopoProducer. Instead of measuring the paths between all endpoint pairs, it would measure only the paths from a few sources to all endpoints. A consumer would then subscribe to several producers separately for all traceroute results from certain endpoints. The smallest granularity would be to have a separate TopoProducer for each endpoint, which essentially reduces it to a TopoMon sensor to which consumers can subscribe.

Eventually, the real bottleneck becomes the consumer: all events will have to be sent to a single process that computes the overall topology. When the system scales up to several hundreds of endpoints, each endpoint will simply be flooded with monitoring data. To solve this, the TopoMon consumer should be converted to a compound producer/consumer that generates “topology” events to which applications can subscribe.

### 6.4 Administration

A practical problem with monitoring systems related to scalability is the administration of the system. When the number of endpoints and processes increases, the management of all these components becomes a tedious task. Currently, every TopoMon sensor has to be started by hand. This was useful in our testbed with seven endpoints, but would be a maintenance nightmare in a real Grid environment. An administration tool that starts, stops and checks certain TopoMon sensors would be a valuable addition to TopoMon. For this reason, the NWS already contains a shell script called `nws-hostadmin` to control NWS components. The administration tool could completely replace all the current configuration methods described in section 4.3.

Another administrative enhancement would be the possibility to change the monitored endpoints at runtime. This prevents restarting all the sensors when the group of monitored endpoints changes. This functionality could also be integrated into the administration tool.
6.5 Multicasting

As explained in Section 3.1.4, the maximum-bandwidth trees currently generated by TopoMon can be suboptimal because they do not take shared links into account. However, the importance of shared links on the overall network performance is the reason TopoMon discovers shared links in the first place. Using this information to generate better maximal-bandwidth spanning trees will be the subject of future research. The 'ideal' algorithm for spanning trees would also take into account the time it takes to send messages (instead of only using the link performance as a distance metric) and whether splitting points forward messages in a streaming fashion.

A related question is what the impact of each shared link on the overall performance actually is. Some links will be able to handle concurrent data streams better than others, so general assumptions as “the available bandwidth is always equally shared between all concurrent streams” may not be true in reality. Even worse, the bandwidth of shared links is not even known exactly, since the NWS carefully avoids concurrent measurements. It will therefore probably be necessary to let the NWS perform additional concurrent measurements when needed.

6.6 Visualization

It turns out that the visualization of a network topology of reasonable size is hard. Large graphs are no problem for computers to reason with, but making them understandable for humans is difficult. The example topologies in Chapter 5 are good examples of this problem. To let the visualization be of use when the topologies get larger, better visualization techniques have to be designed. The question is for which purpose the visualization is used. In our case, it gave a reasonable overview of the testbed which could be printed on paper. However, if the number of endpoints increases, the resulting topology will just be too big to fit on any sheet of paper. It will then be better to let applications use their own visualization technique, which leads to some kind of visualization tool. Such a tool could use the information gathered by TopoMon to not only visualize the network topology, but also the data streams in it and where they collide.
Bibliography


[16] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.


