

INFORMATION SOCIETY TECHNOLOGIES (IST)
PROGRAMME



Large Scale Monitoring of BroadBand Internet Infrastructure
Contract No. 004336

D1.2: Common Access Platform Definition

Abstract: This document describes the common access platform, which provides a uniform interface for applications to interact with the distributed monitoring sensors. This platform is realized mainly by the Distributed Monitoring Application Programming Interface (DiMAPI), which enables remote monitoring and enhances the network flow abstraction to include traffic captured at several geographically distributed sensors. In addition, in order to make our design future proof, we try to anticipate possible bottlenecks of the DiMAPI architecture and outline the design of a Distributed Execution Environment (DEE) as part of our Common Access Platform, that would help alleviate the foreseen problems.

Contractual Date of Delivery	30 June 2005
Actual Date of Delivery	5 July 2005
Deliverable Security Class	Public

The LOBSTER Consortium consists of:

FORTH	Coordinator	Greece
VU	Principal Contractor	The Netherlands
CESNET	Principal Contractor	Czech Republic
UNINETT	Principal Contractor	Norway
ENDACE	Principal Contractor	United Kingdom
ALCATEL	Principal Contractor	France
FORTHnet	Principal Contractor	Greece
TNO	Principal Contractor	The Netherlands
TERENA	Principal Contractor	The Netherlands

Contents

1	Introduction	7
2	Monitoring API (MAPI)	9
2.1	The Design of MAPI: A Network Traffic Monitoring API	9
2.1.1	Creating and Terminating Network Flows	10
2.1.2	Applying Functions to Network Flows	11
2.1.3	Reading Packets from a Flow	12
2.1.4	MAPI Example: Simple Packet Count	12
2.2	Software Architecture	14
3	Remote MAPI	17
3.1	MAPI Extensions for Remote Access	17
3.2	Architecture	18
3.2.1	Agent	19
3.2.2	Remote Inter-Process Communication	21
3.2.3	Remote MAPI Stub	23
3.2.4	Support for Multi-Threaded Applications	24
4	Distributed MAPI	27
4.1	Network Flow Scope	27
4.2	Creating Network Flows Using Multiple Remote Sensors	29
4.3	Obtaining Results from Multiple Remote Sensors	30
4.3.1	Results Format	30
4.3.2	Results Retrieval	30
4.4	Fetching Captured Packets	31
4.5	Security Issues	32
4.5.1	Public Key Authentication	32
4.5.2	Secure Communication Channels	33
4.5.3	Denial of Service Attack Countermeasures	34
5	Distributed Execution Environment	37
5.1	Problems with the existing approach	37
5.2	Reducing latency and bandwidth by remote execution	39
5.2.1	Useful features that are not in LOBSTER	40

CONTENTS

5.2.2	Virtual machines	40
5.2.3	Evaluation of different solutions	42
5.3	The LOBSTER distributed execution environment	45

List of Figures

2.1	List of the most frequently used MAPI calls	10
2.2	MAPI architecture	14
3.1	Remote MAPI architecture	18
3.2	Control sequence diagram for <code>mapi_create_flow</code>	20
3.3	Pseudo-code for Remote MAPI function calls	24
3.4	The role of the communication thread	25
3.5	Pseudo-code for Remote MAPI IPC	26
4.1	Network flow scope example	28
4.2	Policy enforcement in LOBSTER	33

LIST OF FIGURES

Chapter 1

Introduction

The main goal of LOBSTER is to deploy an advanced pilot European infrastructure for accurate Internet traffic monitoring. In order to support collaborative passive network monitoring across a large number of geographically distributed—and possibly heterogeneous—sensors, LOBSTER is based on a distributed uniform access platform, which provides a common interface for applications to interact with the distributed monitoring sensors.

This platform is realized mainly by building onto the Monitoring Application Programming Interface (MAPI) [3, 6], which was developed within the SCAMPI project¹. Based on a generalized network flow abstraction, MAPI is flexible enough to capture emerging application needs, and expressive enough to allow the system to exploit specialized monitoring hardware, wherever available. In LOBSTER, MAPI is extended with *remote* monitoring functionality, allowing applications to interact with distant sensors across the Internet. Furthermore, the Distributed MAPI (DiMAPI) introduces the notion of the network flow *scope*, which enables the manipulation of compound network flows that may consist of traffic captured at several geographically distributed monitoring sensors.

This document provides a detailed description of the common access platform in LOBSTER. Section 2 gives an overview of MAPI, including the basic monitoring operation provided by the API, as well as an outline of the software architecture. Section 3 introduces the Remote MAPI, which extends MAPI with remote monitoring functionality, and details its design and implementation, while Section 4 discusses the design of the Distributed MAPI. Finally, Section 5 discusses the distributed execution environment.

¹<http://www.ist-scampi.org/>

CHAPTER 1. INTRODUCTION

Chapter 2

Monitoring Application Programming Interface (MAPI)

This chapter gives an overview of the Monitoring Application Programming Interface (MAPI), as designed within the SCAMPI network monitoring system¹. Briefly, SCAMPI uses programmable hardware to perform computationally intensive operations, while the middleware offers support for running multiple monitoring applications simultaneously. MAPI is a highly *expressive* and flexible network monitoring API which enables users to clearly communicate their monitoring needs to the underlying network traffic monitoring platform. MAPI supports a variety of network monitoring interfaces, including commodity network interfaces and specialized network monitoring hardware, such as DAG cards².

2.1 The Design of MAPI: A Network Traffic Monitoring API

The goal of an application programming interface is to provide a suitable abstraction which is both simple enough for programmers to use, and powerful enough for expressing complex application specifications. A good API should also relieve the programmer from the complexities of the underlying hardware while making sure that hardware features can be properly exploited.

Towards this way, MAPI builds on a simple, yet powerful, abstraction: the *network flow*. In MAPI, a network flow is generally defined as a sequence of packets that satisfy a given set of conditions. These conditions can be arbitrary, ranging from simple header-based filters, to sophisticated protocol analysis and content inspection functions. For example, a very simple flow can be specified to include *all* packets, or all packets directed to a particular web server. A more complex flow

¹<http://www.ist-scampi.org/>

²<http://www.endace.com/>

- `int mapi_create_flow(char *dev)`
Creates a new network flow.
- `int mapi_connect(int fd)`
Connects to the flow `fd` to start receiving information.
- `int mapi_close_flow(int fd)`
Closes the flow defined by descriptor `fd`.
- `int mapi_apply_function(int fd, char * funct, ...)`
Applies the function `funct` to all the packets of the flow `fd`.
- `void * mapi_read_results(int fd, int fid, int type)`
Receives the results computed by the application of the function `fid` in the packets of the flow `fd`.
- `struct mapikt * mapi_get_next_packet(int fd, int fid)`
Reads the next packet of the flow `fd`.
- `int mapi_loop(int fd, int fid, int cnt, mapi_handler callback)`
Invokes the handler `callback` for each of the packets of the flow `fd`.

Figure 2.1: **List of the most frequently used MAPI calls.**

may be composed of all TCP packets between a pair of subnets that contain the string “User-agent: Mozilla/5.0”.

The approach to network flows in MAPI is therefore fundamentally different from existing models, which constrain the definition of a flow to the set of packets with the same source and destination IP address and port numbers within a given time-window. In contrast with existing models, MAPI gives the network flow a *first-class status*: flows are named entities that can be manipulated in similar ways to other programming abstractions such as sockets, pipes, and files. In particular, users may create or destroy (close) flows, read, sample, or count packets of a flow, apply functions to flows, and retrieve other traffic statistics from a flow.

By using first-class network flows, users can express a wide variety of new monitoring operations. For example, MAPI flows allow users to develop simple intrusion detection schemes that require content inspection. In contrast, traditional approaches to traffic/network flows, such as NetFlow, IPFIX, and related systems and proposals, do not have the means of providing the advanced functions required for this task.

In the remainder of this section we present an overview of the main operations provided by MAPI, summarized in Figure 2.1. A complete specification of the MAPI architecture is provided in [7, 8], and a complete list of the available functions, along with their detailed descriptions, is provided in the `mapi(3)` and `mapi_stdlib(3)` man pages, available in [6, 9].

2.1.1 Creating and Terminating Network Flows

Central to the operation of the MAPI is the action of creating a network flow:

```
int mapi_create_flow(char *dev)
```

This call creates a network flow and returns a flow descriptor `fd` that refers to it. This network flow consists of all network packets which go through network device `dev`. The packets of this flow can be further reduced to those which satisfy an appropriate filter or other condition, as described in Section 2.1.2.

Besides creating a network flow, monitoring applications may also close the flow when they are no longer interested in monitoring:

```
int mapi_close_flow(int fd)
```

After closing a flow, all the structures that have been allocated for the flow are released.

2.1.2 Applying Functions to Network Flows

Network flows allow users to treat packets that belong to different flows in different ways. For example, a user may be interested in *logging* all packets of a flow (e.g. to record an intrusion attempt), or in just *counting* the packets and their lengths (e.g. to count the bandwidth usage of an application), or in *sampling* the packets (e.g. to find the IP addresses that generate most of the traffic). The abstraction of the network flow allows the user to clearly communicate to the underlying monitoring system these different monitoring needs. To enable users to communicate these different requirements, MAPI allows the association of functions with network flows:

```
int mapi_apply_function(int fd, char * funct, ...)
```

The above association applies the function `funct` to every packet of the network flow `fd`, and returns a relevant function descriptor `fid`. Depending on the applied function, additional arguments may be passed. Based on the header and payload of the packet, the function will perform some computation, and may optionally discard the packet.

MAPI provides several *predefined* functions that cover some standard monitoring needs through the MAPI Standard Library (`stdlib`). For example, applying the `BPF_FILTER` function with parameter `tcp` and `dst port 80` restricts the packets of the network flow denoted by the flow descriptor `fd` to the TCP packets destined to port 80. Other example functions include: `PKT_COUNTER` which counts all packets in a flow, `SAMPLE` which can be used to sample packets, etc. For a complete list of the available functions in `stdlib` and their description please refer to the `mapi_stdlib(3)` man page.

Although these functions enable users to process packets, and compute the network traffic metrics they are interested in without receiving the packets in their own address space, they must somehow communicate their results to the interested users. For example, a user that will define that the function `PKT_COUNTER` will be applied to a flow, will be interested in reading what is the number of packets that have been counted so far. This can be achieved by allocating a small amount of memory or a data structure to each network flow. The functions that will be applied to the packets of the flow will write their results into this data structure. The user who is interested in reading the results will read the data structure using:

```
void * mapi_read_results(int fd, int fid, int type)
```

The above call receives the results computed by the function denoted by the function descriptor `fid`, which has been applied to the network flow `fd`. When `type` is set to `MAPI_REF`, it returns a pointer to the memory where the results are stored. Thus, the application may read future values by dereferencing this pointer, without having to invoke `mapi_read_results()` again. Otherwise, if `type` is set to `MAPI_COPY`, it returns a pointer to a copy of the results.

2.1.3 Reading Packets from a Flow

Once a flow is established, the user will probably want to read packets from the flow. Packets can be read one-at-a-time using the following *blocking* call:

```
struct mapipkt * mapi_get_next_pkt(int fd, int fid)
```

which reads the next packet that belongs to flow `fd`. In order to read packets, the function `TO_BUFFER` (which returns the relevant `fid` parameter) must have previously been applied to the flow. If the user does not want to read one packet at-a-time and possibly block, (s)he may register a callback function that will be called when a packet to the specific flow is available:

```
int mapi_loop(int fd, int fid, int cnt, mapi_handler callback)
```

The above call makes sure that the handler `callback` will be invoked for each of the next `cnt` packets that will arrive in the flow `fd`.

2.1.4 MAPI Example: Simple Packet Count

This section presents a simple example program that introduces the concept of the network flow, and demonstrates the basic steps that must be taken in order to create and use a network flow. In this example, a network flow is used for counting the number of packets destined to a web server during a certain time period.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include "mapi.h"
5
6 int main() {
7
8     int fd, fid;
9     unsigned long long *cnt;
10
11     /* create a flow using the eth0 interface */
12     fd = mapi_create_flow("eth0");
13     if (fd < 0) {
14         printf("Could not create flow\n");
```

2.1. THE DESIGN OF MAPI: A NETWORK TRAFFIC MONITORING API

```
15     exit(EXIT_FAILURE);
16 }
17
18 /* keep only the packets directed to the web server */
19 mapi_apply_function(fd, "BPF_FILTER", "tcp and dst port 80");
20
21 /* and just count them */
22 fid = mapi_apply_function(fd, "PKT_COUNTER");
23
24 /* connect to the flow */
25 if(mapi_connect(fd) < 0) {
26     printf("Could not connect to flow %d\n", fd);
27     exit(EXIT_FAILURE);
28 }
29
30 sleep(10);
31
32 /* read the results of the applied PKT_COUNTER function */
33 cnt = (unsigned long long *)mapi_read_results(fd, fid, MAPI_COPY);
34 printf("pkts:%llu\n", *cnt);
35
36 mapi_close_flow(fd);
37 return 0;
38 }
```

The control flow of the code is as follows: We begin by creating a network flow (line 12) that will receive the packets we are interested in. We specify that we are going to use the `eth0` network interface for monitoring the traffic. For a different monitoring adapter we would use something like `/dev/scampi/0` for a Scampi adapter, or `/dev/dag0` for a DAG card, depending on the configuration. We store the returned flow descriptor in the variable `fd` for future reference to the flow.

In the next step we restrict the packets of the newly created flow to those destined to our web server by applying the function `BPF_FILTER` (line 19) using the filter `tcp and dst port 80`. The filtering expression is written using the `tcpdump(8)` syntax. Since we are interested in just counting the packets, we also apply the `PKT_COUNTER` function (line 22). In order to later read the results of that function, we store the returned function descriptor in `fid`.

The final step is to start the operation of the network flow by connecting to it (line 25). The call to `mapi_connect()` actually activates the flow inside the MAPI daemon (`mapi_d`), which then starts processing the monitored traffic according to the specifications of the flow. In our case, it just keeps a count of the packets that match the filtering condition.

After 10 seconds, we read the packet count by passing the relevant flow descriptor `fd` and function descriptor `fid` to `mapi_read_results()` (line 33).

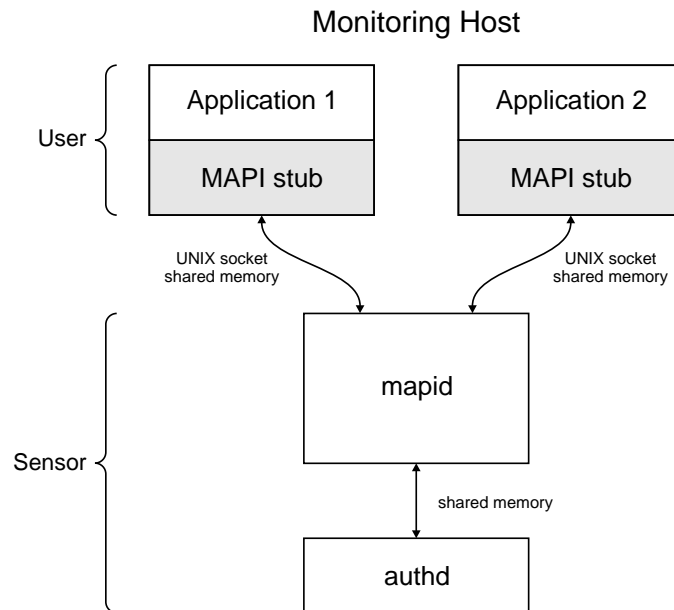


Figure 2.2: **MAPI architecture.** User applications, `mapid`, and `authd`, run on the same machine.

Our work is done, so we close the network flow in order to free the resources allocated in `mapid` (line 36).

2.2 Software Architecture

Figure 2.2 illustrates the basic architecture of MAPI. On the top of the figure we see a set of monitoring applications that, via the MAPI stub, communicate with the MAPI daemon, called `mapid`: a user-level process running in a separate address space. The MAPI daemon, which has exclusive access to the captured packets, is the most complex part of the software, as this is where all the processing of the monitoring requirements of each user application is done. `mapid` consists of several modules:

- `mapidcom`: an IPC communication module that interacts with all the active applications and passes the messages generated by the MAPI calls along to the correct driver. All control communication between the monitoring applications and `mapid` is made through UNIX sockets. The applications can also retrieve the results of the applied functions and the captured packets directly from `mapid` through shared memory.
- `mapidrv`: it is inside the `mapid` drivers that all the packet processing is performed. The driver can either implement functions for processing packets

specially designed for this specific hardware adapter, or use generic software only functions in the `mapidlib`.

- `mapidlib`: this is a common library available to all `mapiddrivers` that contains generic functions for packet processing and the organizing of flows and functions applied to flows.
- low level drivers: the `mapid` drivers will normally use a library to communicate with the low level kernel drivers. This library can be an existing hardware specific library such as the `daglib` for DAG cards, or a more generic library like `libpcap` for normal NICs.

Applications that connect to the MAPI daemon must authenticate themselves and provide the necessary credentials. Before the activation of a newly created flow in `mapid` the system has to check whether the user has the appropriate privileges to create a flow with the specified configuration, i.e., combination of applied functions and options, based on his/her credentials. In MAPI, this is the responsibility of the Admission Control daemon, which has been designed as an independent module known as `authd` [4]. `authd` is a daemon user-level process that runs standalone and uses shared memory IPC to communicate with `mapid`, as shown on the bottom of Figure 2.2. `authd` is responsible for user authentication, as well as for checking their privileges against the specifications of a flow before activating it. If the application does not have the appropriate privileges, or *credentials*, to create a flow with the specified characteristics, then the flow is rejected.

Chapter 3

Remote MAPI

MAPI supports the operation of monitoring applications that run on the same computer that hosts the monitoring hardware. MAPI within the SCAMPI project did not support remote monitoring, since in that implementation `mapid` and user applications communicate via local calls, and thus, must reside on the same monitoring host. This chapter describes the design of **Remote MAPI**, which extends MAPI with remote monitoring functionality, allowing user applications to utilize monitoring sensors located at different hosts across the Internet.

3.1 MAPI Extensions for Remote Access

Monitoring applications address the monitoring platform throughout MAPI. An application uses the functions defined in the MAPI interface to configure the monitoring daemon and retrieve results from it. Whether this daemon is running locally, in the same host with the application, or remotely at a different host, should be of no concern to the functionality of the application.

In order to ensure backwards compatibility with existing MAPI applications, Remote MAPI should preserve the semantics of all MAPI calls, while extending certain calls to allow the remote monitoring functionality. In Remote MAPI, the core MAPI function `mapi_create_flow`, which is used to create a new network flow on a local monitoring interface, is extended to support network flows associated with interfaces hosted at remote sensors. For example, the following MAPI call creates a network flow associated with the local interface `eth0`:

```
local_fd = mapi_create_flow("eth0");
```

In order to specify the interface of a remote sensor, `mapi_create_flow` is extended in Remote MAPI to support the following notation: the interface name can be prefixed with the host name or IP address of the remote sensor, along with a colon which separates it from the interface name. For example, any of the two following calls can be used to create a flow at the sensor `mon1.ics.forth.gr` with IP address `138.90.34.22` using its `/dev/dag0` monitoring interface:

```
remote_fd = mapi_create_flow("mon1.ics.forth.gr:/dev/dag0");  
remote_fd = mapi_create_flow("138.90.34.22:/dev/dag0");
```

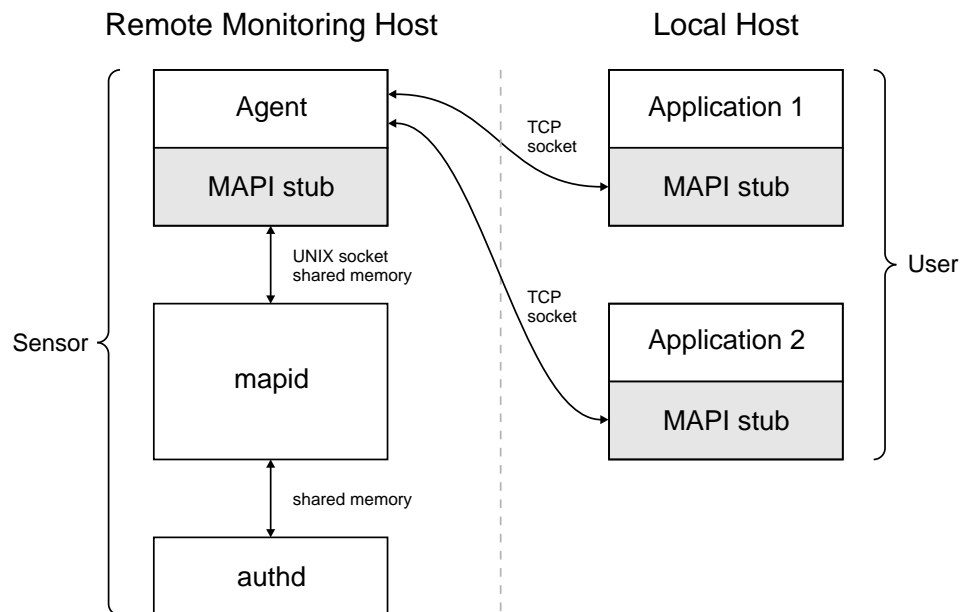


Figure 3.1: **Remote MAPI architecture.** User applications can run on different hosts than the monitoring sensor. In contrast with Figure 2.2, user applications interact with `mapid` through the agent, instead of communicating directly with `mapid`.

3.2 Architecture

The straightforward solution to implement MAPI with remote monitoring functionality would be to modify any local communication between the MAPI stub and `mapid` so that it is performed through some other remote interaction mechanism, such as TCP sockets. This design maintains the basic architecture of MAPI, since the MAPI stub still communicates directly with `mapid`, but requires changes to be made to both `mapid` and the MAPI stub.

Although modifying the MAPI stub is unavoidable, since any communication with the monitoring sensor must be changed to utilize a remote interaction mechanism, modifying `mapid` to interact directly with remote clients poses several shortcomings. Indeed, `mapid` is a complex part of the MAPI software and is already responsible to handle important “heavy-duty” tasks, as this is where all the processing of the monitoring requirements of the user applications takes place. Besides increasing software complexity, extending `mapid` to handle communication with remote clients would probably introduce additional performance overhead. Furthermore, allowing remote clients to connect directly to `mapid`, which has exclusive access to the captured packets, may introduce significant security risks.

An alternative design that avoids any modifications to `mapid` is possible by introducing an *intermediate* process between `mapid` and the remote applications

that runs on the same host with `mapid`. This “agent” process acts like a proxy for the remote applications, forwarding their monitoring requests to `mapid` and sending back to them the computed results. The presence of the `agent` is completely transparent to the applications, which continue their operation as if they were interacting directly with `mapid`—only the Remote MAPI stub is aware of the presence of the agent.

Figure 3.1 illustrates the Remote MAPI architecture. On the right we see a set of monitoring applications that communicate with a monitoring sensor located at a different host across the Internet. On the monitoring host, the `agent` that runs along with `mapid` and `authd`, constantly listens for requests from remote applications and forwards them to `mapid`. While the `agent` communicates remotely with the user applications, it communicates locally with `mapid` using IPC, in the same way that “traditional” MAPI applications communicate with a local `mapid`. Hence, applications interact *indirectly* with `mapid` with the intervention of the agent. As expected, the agent can serve multiple remote applications concurrently. Together, the `agent`, `mapid`, and `authd`, constitute the monitoring software of a remote sensor.

The main benefit of this design approach is that it does not require any changes to `mapid`. The only existing code that needs modifications is that of the MAPI stub, which needs to be extended to support a remote interaction mechanism. At the monitoring host, the Remote MAPI functionality is solely implemented by the `agent`, which is built as a “traditional” MAPI application, which solely interacts locally with the `mapid` through its MAPI stub. This allows for a cleaner implementation with shorter debugging cycles, and to a more robust system due to fault isolation. Furthermore, in case that the remote monitoring functionality is not required by a sensor, it can be easily left out by simply not starting the `agent`.

In the following sections we describe in more detail the implementation of the Remote MAPI, with respect to the `agent`, the Remote MAPI stub, and the communication mechanism between them.

3.2.1 Agent

As described in Section 3.2, communication between a user application and the remote MAPI daemon is performed through the `agent`, an intermediate program that runs in the same host with `mapid` and acts like a proxy server for client requests. Upon the reception of a monitoring request, it forwards the relevant MAPI call to the local `mapid` and sends back to the user the computed results.

The `agent` is implemented as a user-level process on top of MAPI. To put it in other words, it looks like an ordinary MAPI-based application. Its key characteristic, however, is that it can receive monitoring requests from *other* monitoring applications that run on different hosts and are written on top of the Remote MAPI. This is achieved by directly handling the control messages sent by the Remote MAPI stub, and transforming them to the relevant local MAPI calls.

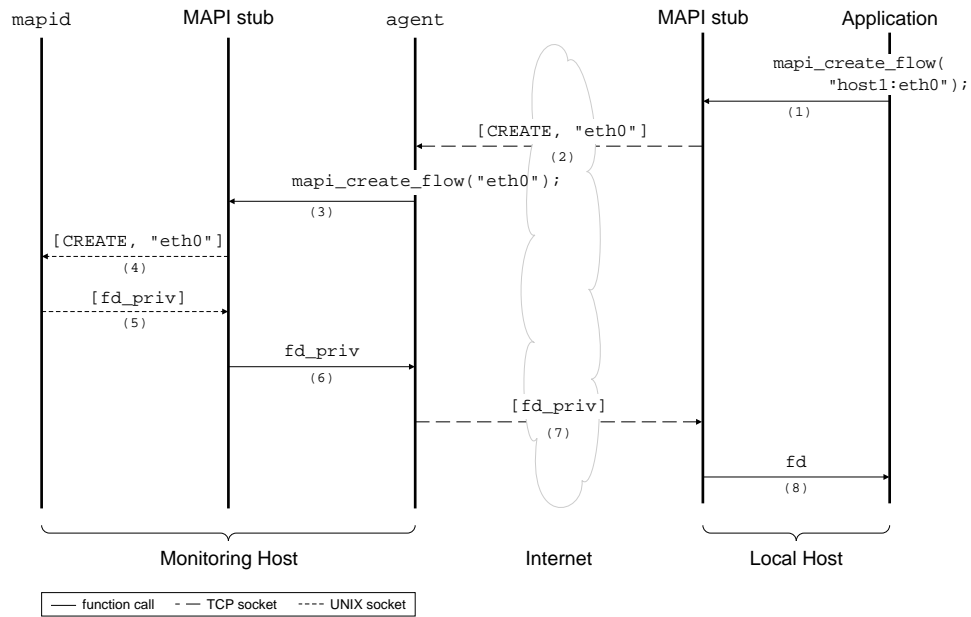


Figure 3.2: Control sequence diagram for `mapi_create_flow`.

During startup, the `agent` creates a TCP socket and binds it to a known predefined port. Then, it runs into an infinite loop that constantly waits for connections from remote applications. A new thread is spawned for each new remote application, which thereafter handles all the communication between the application and the `agent`. Complete information for each MAPI call that is made from the remote application is contained in the control messages sent by the Remote MAPI stub of the application. Based on that information, the `agent` can repeat the MAPI call, though this time the MAPI stub of the `agent` will interact directly with the `mapid` running on the same host. The `agent` then returns the result to the stub of the remote application, which in turn returns it to the user.

Note that the `agent` relies only on the local MAPI functionality, since it never manipulates flows at remote sensors. Thus, recalling the pseudo-code of Figure 3.3, it can be compiled without defining the `RMAPI` flag, setting aside altogether the unnecessary code that implements the Remote MAPI functionality.

The operation of the `agent` is described in more detail in Figure 3.2, which illustrates the control sequence for the implementation of the `mapi_create_flow` call. Initially, a user application calls `mapi_create_flow` in order to create a network flow at a remote sensor (step 1). The Remote MAPI stub of the application extracts the host name of the remote sensor (`host1`) from the function argument, and sends a respective message to the `agent` running on that host (step 2) through a TCP socket. This message contains the type of the MAPI call to be executed (`CREATE`), along with the monitoring interface that will be used. The `agent`, upon the receipt of the message and based on the information it contains, repeats

the call to `mapi_create_flow` (step 3). However, this time the call is destined to the local `mapid`, thus the stub of the agent sends the respective message through a UNIX socket (step 4). Assuming a successful creation of the flow, `mapid` returns through the same UNIX socket the flow descriptor `fd_priv` of the newly created flow to the stub of the agent (step 5), which in turn finishes the execution of the `mapi_create_flow` call by returning `fd_priv` to the agent (step 6). The agent constructs a corresponding reply message that contains the flow descriptor, and sends it back to the Remote MAPI stub of the user application through the TCP socket (step 7).

Finally, the stub of the application has to return the flow descriptor back to the user. However, this is not as trivial as it seems, since there is a subtle issue that the stub must take care of. Consider for example an application that creates two network flows associated with different remote monitoring sensors. Assume also that the MAPI daemons of the two sensors have served exactly the same number of flows so far, and are occupied by exactly the same number of active flows at that moment. Since `mapid` generates the values for the flow descriptors sequentially, starting from the same predefined number, the agents of the two sensors will return to the Remote MAPI stub of the application the same flow descriptor for each of the two flows. Clearly this is not acceptable, since the flow descriptor becomes ambiguous, and does not identify uniquely the network flow anymore. Although each descriptor is unique within the scope of the `mapid` that generated it, it is not unique within the scope of the remote application.

The Remote MAPI stub solves this issue by generating its own flow descriptor for each remote network flow, and storing the mapping between the received and the generated descriptor. Thus, even if two or more remote sensors return the same flow descriptor, the stub always returns to the user a unique identifier for each flow. In our example, the stub of the user application receives the descriptor `fd_priv`, generates the new unique flow descriptor `fd`, saves the mapping between `fd_priv` and `fd`, and finally returns `fd` to the user (step 8).

Although it may seem that the overhead of a single MAPI call is quite high, since it results to a rather complex sequence of control flow transitions, we should stress that most of the above steps are function calls or IPC that take place on the same host, and thus incur very small overhead. The operations responsible for the largest part of the overhead are the send and receive operations through the TCP socket (steps 2 and 7).

3.2.2 Remote Inter-Process Communication

MAPI is designed for local use and does not provide access to remote monitoring sensors. In the SCAMPI architecture, user applications and `mapid` both run on the same monitoring sensor, and communicate locally through UNIX sockets and shared memory.

In LOBSTER, however, applications have to interact remotely with monitoring sensors at different locations across Europe. As a consequence, any control

communication between the user application and the monitoring daemon, which in SCAMPI is done through UNIX sockets, should be performed through some other mechanism, such as TCP sockets. Shared memory communication for retrieving results is also not applicable any more. Instead, results retrieval in LOBSTER is done through TCP sockets.

The Remote MAPI library is compiled into every application and encapsulates the communication with the remote monitoring sensor, and specifically with the agent. In Remote MAPI, all communication between the monitoring applications and the agent is implemented through TCP sockets.

MAPI function calls operate by sending a control message that describes the function to be executed. Each message contains all the necessary information for the execution of each function instance. The format of the messages exchanged between the agent and the stub is the following:

```
struct remote_ipc_message {
    int length;
    mapiipcMsg cmd;
    int fd;
    int fid;
    unsigned long long ts;
    unsigned char data[MAX_DATA_SIZE];
}
```

- `length` is the total message length.
- `cmd` contains the type of request sent by the stub to the agent, or an acknowledgment value for a request that the agent processed. It takes values from an enumeration of all message types that can be sent between the stub and the agent. For example, for a call to `mapi_create_flow()` the relevant message sent from the stub will have a `cmd` value of `CREATE_FLOW`, for a call to `mapi_apply_function()` `cmd` will be `APPLY_FUNCTION`, and so on.
- `fd` is the descriptor of the network flow being manipulated.
- `fid` is the descriptor of an applied function instance being manipulated (not always set).
- `ts` is a timestamp of the moment that the result was produced, represented as a 64-bit unsigned fixed-point number, in seconds since 1 January 1970. The integer part is in the first 32 bits and the fraction part in the last 32 bits.
- `data` is the only field of variable size. It serves several purposes: It contains the results of an applied function sent from the agent to the stub, after a call to `mapi_read_results()`. For requests sent from the stub, it may contain the arguments of the relevant mapi call, e.g., in a call to `mapi_apply_function()` it contains the name of the applied function and its arguments.

After sending a request, the stub waits for a corresponding acknowledgement from `mapid`, always through the `agent`, indicating the successful completion of the requested action, or a specific error in case of failure.

This implementation is similar to the existing IPC mechanism between local applications and `mapid`. Indeed, whether the MAPI daemon is running locally or remotely is of no concern to the application. The details of the underlying communication mechanism are hidden from the end-user, making Remote MAPI completely transparent to the application.

3.2.3 Remote MAPI Stub

Besides the necessary changes to the interfaces of the MAPI functions that are affected by the additional operations of the Remote MAPI, as discussed in Section 3.1, several other modifications are required for the MAPI stub in order to implement the remote monitoring functionality.

Applications may interact with a local or a remote sensor without changes or recompilation. Depending on the argument of `mapi_create_flow`, the stub will direct the creation of the flow to a local or a remote MAPI daemon, and internally store whether the flow descriptor returned by the call refers to a local or a remote network flow. The MAPI stub is then responsible for identifying whether subsequent MAPI calls manipulate local or remote network flows, in order to send the request to the locally running `mapid` or to the `agent` of the remote sensor.

Depending on whether the flow being manipulated by each MAPI call is local or remote, the stub takes appropriate action. If the flow has been created at a `mapid` running on the same host with the application, the Remote MAPI stub sends the relevant control message through the UNIX socket that is used for control communication with the local `mapid`. In case of results or captured packets retrieval, the application reads directly the data from `mapid` through shared memory. However, the stub must take different action if the flow has been created in a remote sensor. Although sending the control messages through a TCP socket instead of the UNIX socket requires relatively few code modifications, any result retrieval should also be implemented through TCP communication.

Going back to the example of Section 3.1, MAPI calls that manipulate `local_fd` will be passed directly to the local `mapid` through a UNIX socket, while calls that manipulate `remote_fd` will be passed to the `agent` of the remote sensor through a TCP socket. Thus, applications written on top of the Remote MAPI can interact transparently with a remote MAPI daemon as if it was running locally.

The pseudo-code in Figure 3.3 outlines how the implementation of each MAPI call has been extended in Remote MAPI, using `mapi_apply_function` as a driving example. Before executing the main body of the call, the stub identifies the type of the flow. Then, depending on whether the flow is local or remote, the appropriate code path is taken. In case that the remote monitoring functionality is not needed, i.e., for applications that operate solely with a local `mapid`, the code of the MAPI stub can be optimized by setting aside altogether the remote monitoring

```
int mapi_apply_function(int fd, char *funct, ...) {  
  
    flow_t *flow = list_get(flowlist, fd);  
  
#ifdef RMAPI  
    if (flow->type == REMOTE) {  
  
        /* Remote MAPI implementation */  
    }  
    else  
#endif  
    {  
        /* MAPI implementation */  
    }  
}
```

Figure 3.3: **Pseudo-code for the implementation of the Remote MAPI function calls.** The Remote MAPI functionality can be easily set aside by not defining the `RMAPI` flag during the compilation of the Remote MAPI stub.

functionality.¹ This can be easily achieved by not defining the `RMAPI` flag during the compilation of the stub.

3.2.4 Support for Multi-Threaded Applications

The mechanism of sending and receiving control messages has also been reconsidered in Remote MAPI. Since the two endpoints are located at different hosts across the Internet, the time interval between the dispatch of a control message and the receipt of the corresponding reply is not constant, and may be several milliseconds long. This makes the exchange of control messages challenging for multi-threaded user applications, which may call several MAPI functions at the same time.

Consider for example a user application consisting of two threads, each of which at a given point of time reads the result of a `PKT_COUNTER` function applied at a different network flow. Both flows have been created on the same remote monitoring sensor, and each flow is private to each thread. The stub will send to the `agent` of the remote sensor two control messages, one for each thread, requesting the value of the counter. However, there is no guarantee that the replies from the `agent` will arrive in the same order that the corresponding requests were made, due to the arbitrary delays and routes of the network packets that carry the reply messages. Thus, a reply may be delivered to the wrong call that did not

¹As discussed in Section 4.1, where the notion of the network flow scope is introduced, there are cases where this optimization is not applicable even for a single monitoring host, and specifically when it has more than one monitoring interfaces. In such a configuration, a network flow may be associated with several interfaces of a single monitoring host, which requires the above functionality.

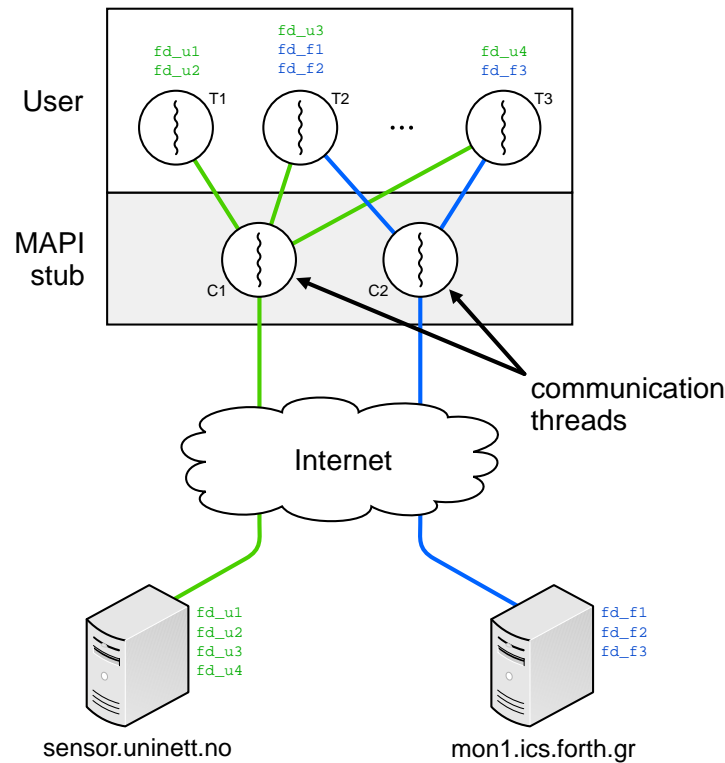


Figure 3.4: **The role of the communication thread in a multi-threaded application.** Each communication thread is responsible for the communication with one remote sensor.

make the corresponding request. This introduces the need for demultiplexing of the incoming messages at the Remote MAPI stub.

The receipt of incoming messages in Remote MAPI is implemented using a separate “communication thread.” This thread is responsible for receiving the replies of pending MAPI calls from remote sensors, and delivering them to the appropriate function. One such thread is created for each remote sensor used by the application (i.e., for each TCP socket created by the stub). For instance, Figure 3.4 illustrates a multi-threaded application that has created several network flows at two remote monitoring sensors. Specifically, thread T1 has created the flows `fd_u1` and `fd_u2` at `sensor.uninett.no`, thread T2 has created the flow `fd_u3` at `sensor.uninett.no` and `fd_f1` and `fd_f2` at `mon1.ics.forth.gr`, and thread T3 has created the flows `fd_u4` and `fd_f3`, both at different sensors. Messages related to the flows `fd_f1`, `fd_f2`, `fd_f3`, and `fd_f4` are handled by the communication thread C1, which delivers them to the appropriate thread (T1, T2, or T3). Correspondingly, messages for flows `fd_u1`, `fd_u2`, and `fd_u3` are handled by the communication thread C2, which delivers them to T2 or T3, depending on the flow.

```
int mapi_apply_function(int fd, char *funct, ...) {  
  
    flow_t *flow = list_get(flowlist, fd);  
  
    /* prepare control message */  
    /* ... */  
  
    send(host, msg);  
    sem_wait(flow->sem);  
  
    /* read reply */  
    /* ... */  
}  
  
void comm_thread(void *host) {  
  
    while(1) {  
  
        recv(reply);  
        flow = list_get(pendinglist, reply.fd);  
        copy_result(fd->res, reply);  
        sem_up(flow->sem);  
    }  
}
```

Figure 3.5: **Pseudo-code for the Remote MAPI IPC implementation.**

The pseudo-code of Figure 3.5 outlines the implementation of the generic IPC code for Remote MAPI calls and the communication thread. A Remote MAPI call—`mapi_apply_function` in our example—prepares and sends the control message, and then blocks by pushing down a semaphore variable, as a result of calling `sem_wait`. The communication thread waits infinitely in a loop for incoming replies from the agent. When such a reply message arrives, the communication thread looks up the flow for which it is destined, copies the result in a flow-specific buffer, and “wakes up” the blocked MAPI call by calling `sem_up`. When the execution of the blocked call resumes, it retrieves the result from the buffer and processes it accordingly. This implementation guarantees that the incoming messages are always delivered to the call that sent the relevant request.

Chapter 4

Distributed MAPI

The LOBSTER infrastructure consists of several geographically distributed monitoring sensors. This monitoring infrastructure is shared among many distributed applications, which, at the same time, need to access several monitoring interfaces across different sensors. While Remote MAPI enables applications to perform remote monitoring, network flows are still limited to a single network interface. This chapter describes the design of the Distributed MAPI (DiMAPI), which enables the creation of network flows that receive traffic from many monitoring interfaces, and introduces the notion of the **network flow scope**.

4.1 Network Flow Scope

Both MAPI and Remote MAPI, as described in Sections 2 and 3 respectively, support the creation of network flows that use a *single* monitoring interface, which may be located either on the same host with the application, or at a different host across the Internet. Although an application may create several flows at different, and possibly remote, monitoring sensors, each network flow is associated with only one of the available network interfaces at each sensor. Thus, a network flow receives network packets that are always captured at a single monitoring point.

Applications in LOBSTER need concurrent access to several distributed sensors. DiMAPI fulfils this requirement by facilitating the manipulation of many remote monitoring interfaces from a single application. One of the main novelties of DiMAPI is the introduction of the **network flow scope**. Generally, a network flow is defined as a sequence of packets that satisfy a given set of conditions. In MAPI and Remote MAPI, this stream of packets normally comes from a single monitoring interface. The notion of scope allows a network flow to receive packets from several monitoring interfaces. The abstraction of the network flow remains intact—a network flow with scope is still a sequence of packets. However, in contrast with MAPI, the traffic that constitutes the network flow may come from more than one monitoring points.

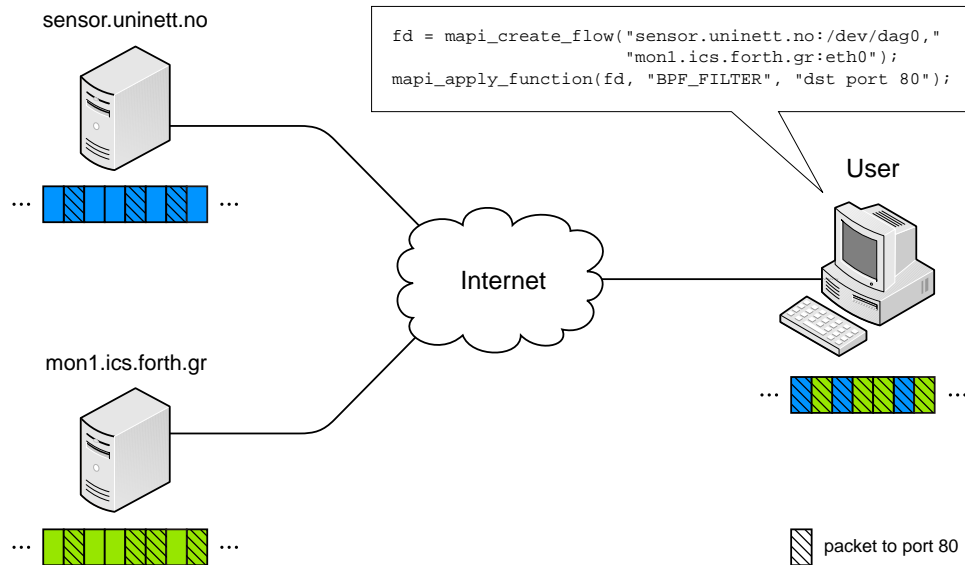


Figure 4.1: **Network flow scope example.** A user application manipulates a network flow associated with two remote monitoring interfaces.

In order to support the abstraction of scope in DiMAPI, the interface of the `mapi_create_flow` function is further extended to support the definition of multiple monitoring interfaces. As discussed in Section 3.1, Remote MAPI supports the notation `host:interface`, which denotes a monitoring interface located at a remote sensor. DiMAPI extends this notation by allowing several comma-separated `host:interface` pairs as an argument to `mapi_create_flow`. For example, the following call creates a network flow associated with two monitoring interfaces located at different hosts across the Internet:

```
fd = mapi_create_flow("sensor.uninett.no:/dev/dag0,"
                    "mon1.ics.forth.gr:eth0");
```

The notion of scope is illustrated in the example of Figure 4.1. A user application creates a network flow using the above `mapi_create_flow` call, and then applies the function `BPF_FILTER` in order to restrict the packets of the flow to only those that are destined to some web server, i.e., the packets with destination port 80. As a result, the network flow consists of packets with destination port 80 that are captured from both Uninett's and FORTH's sensors.

Note that the scope abstraction also allows the creation of flows associated with interfaces located at the same host. For example, the following call creates a network flow associated with a simple Ethernet interface and a DAG card, both installed at the host `mon2.ics.forth.gr`:

```
fd = mapi_create_flow("mon2.ics.forth.gr:eth1,"
                    "mon2.ics.forth.gr:/dev/dag0");
```

4.2 Creating Network Flows Using Multiple Remote Sensors

To support the scope functionality, the Remote MAPI stub has to be extended in order to handle communication with many remote sensors concurrently. Consider for example the following call, which creates a network flow at three different remote sensors:

```
fd = mapi_create_flow("sensor.uninett.no:/dev/dag0,"
                    "monitor.cesnet.cz:/dev/scampi/0,"
                    "mon1.ics.forth.gr:eth0");
```

In order to implement this call, the DiMAPI stub has to communicate with the agents that run at each of the three remote sensors. This is achieved by sending three control messages, one to each agent, through three different TCP sockets. Thus, the following calls will be made by the three agents:

```
Uninett: fd_uninett = mapi_create_flow("/dev/dag0");
Cesnet:  fd_cesnet = mapi_create_flow("/dev/scampi/0");
FORTH:  fd_forth = mapi_create_flow("eth0");
```

In this example, the creation of one *distributed* network flow from the user application resulted in the creation of three *local* network flows, one at each of the three remote sensors. Assuming that the three flows were created successfully, each agent will send back to the DiMAPI stub an acknowledgment message containing the flow descriptor of the flow that it created remotely (`fd_uninett`, `fd_cesnet`, and `fd_forth`, respectively). However, in the client side, the call to `mapi_create_flow` has to return back to the user a single unique identifier for the newly created distributed network flow (`fd` in the above example). The DiMAPI stub is responsible for generating such a unique flow identifier, and internally storing the remote flow descriptors that it corresponds with. In the above example, the stub will store the mapping between `fd` and [`fd_uninett`, `fd_cesnet`, `fd_forth`]. For subsequent calls that manipulate `fd`, such as the following:

```
int fid = mapi_apply_function(fd, "PKT_COUNTER");
```

the DiMAPI stub will send to the agents of the three sensors the following messages:

```
Uninett: [APPLY_FUNCTION, PKT_COUNTER, fd_uninett]
Cesnet:  [APPLY_FUNCTION, PKT_COUNTER, fd_cesnet]
FORTH:  [APPLY_FUNCTION, PKT_COUNTER, fd_forth]
```

Since the stub knows each of the remote flow descriptors that constitute `fd`, it can send targeted control messages with the appropriate flow descriptor for each agent. Note that the DiMAPI stub stores a similar mapping for the function identifier `fid`, and acts in a similar fashion whenever it is manipulated.

4.3 Obtaining Results from Multiple Remote Sensors

4.3.1 Results Format

In MAPI and Remote MAPI, network flows are associated with a single monitoring interface, and thus, any results produced by an applied function are related to that interface. For example, the result of the `PKT_COUNTER` function is the number of packets that matched the specification of the flow and were captured from one specific interface. In DiMAPI, however, since a network flow is usually associated with multiple monitoring interfaces, the obtained results are not anymore related to a single interface.

Going back to the above example, the result of a `PKT_COUNTER` function applied to a network flow associated with many remote sensors can be viewed in two ways: a user could expect to receive the *aggregate* number of packets captured from all interfaces, while another user could expect to receive the number of packets captured from *each* interface separately. Although from the latter result it is possible to obtain the aggregate count, the inverse is not possible. Therefore, in order to provide maximum flexibility to the end user, results in DiMAPI are retrieved in a vector format with one entry for each monitoring interface that the network flow uses. Each entry contains the partial result for a particular interface, the `host:interface` pair for that interface, and two timestamps: the time that the result was produced (set by the relevant `mapid`), and the time that the stub received the result (set by the stub).

4.3.2 Results Retrieval

In local MAPI, a user who is interested in reading the results of an applied function will use the following MAPI call:

```
void * mapi_read_results(int fd, int fid, int type)
```

The above call receives the results computed by the function denoted by the function descriptor `fid`, which has been applied to the network flow `fd`. When `type` is set to `MAPI_REF`, it returns a pointer to the memory where the results are stored. Thus, the application may read future values by dereferencing this pointer, without having to invoke `mapi_read_results()` again. Otherwise, if `type` is set to `MAPI_COPY`, it returns a pointer to a copy of the results.

However, given that in DiMAPI a user cannot access `mapid` directly, we are facing two contradictory issues: the semantics of the `type` field should be retained for compatibility with legacy MAPI applications, but user applications cannot share a memory segment with each `mapid` or `agent`, since they run on different hosts, making the `MAPI_REF` option nonfunctional.

DiMAPI retains the original semantics of the `type` option in the following way: When `type` is set to `MAPI_COPY`, `mapi_read_results()` acts in the same way as before, returning a pointer to a copy of the results. Note that this

incurs a round-trip latency between the time that the call is made and the time the results are returned, since the stub has to send a request for the result to the remote sensor and then wait for the reply.

When `type` is set to `MAPI_REF`, the stub reserves a shared memory segment accessible from the user application, where the received results are stored. Thus, as before, user applications can call `map_read_results()` once, in order to get a reference to the memory where the results are stored, and then read directly the results. However, there is a subtle difference with the `MAPI_REF` option in local MAPI. When `mapid` and the MAPI stub run on the same host, `mapid` writes directly the updated value of the result into the shared memory segment, right upon the new value is derived. Since this is not feasible in DiMAPI, the new value of the result must be sent to the stub through a TCP socket. However, this update cannot happen every time a new value for a certain result is available at the sensor. Consider for example a simple program that counts the number of packets of a 10 Gigabit link. The synchronization of the result value at the stub each time a new value is available would incur a tremendous amount of control traffic. Thus, the update is performed at a user-defined interval, with a minimum value of one second.

Note that the two options for the `type` field have different impact on the delay between the request of the result and its actual delivery and the control traffic they incur. Indeed, upon the request of a result, with `MAPI_COPY` the result is delivered after a round-trip delay, while with `MAPI_REF` it is delivered immediately. On the other hand, `MAPI_REF` results to considerable constant control traffic in order to keep the local result value updated, while `MAPI_COPY` results to a single control message each time the user reads the result.

4.4 Fetching Captured Packets

In a traditional MAPI application, a user could get a packet from the daemon through the

```
map_get_next_pkt(int fd, int fid)
```

function call, where `fd` is the flow descriptor and `fid` is the descriptor of the “TO_BUFFER” function applied on the flow. When requesting packets from multiple daemons, retrieving packets and delivering them to the application needs careful treatment. To ensure fairness among packets from different daemons, an internal mechanism for handling incoming packets takes place as follows. Upon the first `map_get_next_pkt` call, the request is forwarded to all daemons of the scope. Each daemon is mapped to a slot in an internal buffer that stores incoming packets. If we have three daemons, for example, a buffer of three slots is created. Packets from first daemon go to slot 1, from second daemon to slot 2 and so on. The first packet that arrives is delivered to the application. When a packet is delivered to the application, the slot from which it was picked up is now considered

empty. Upon consequent `mapi_get_next_pkt` requests, all slots are checked in a round-robin way. The round-robin check begins from the slot that was emptied in the previous call to avoid starvation of slots. If the round-robin check is started from the first slot and the daemon of the first slot sends packet in faster rate than others, then only packets from first slot will be picked up and rest of the slots will starve. In our proposed way, the check will check all slots. Furthermore, a request for next packet is sent to the daemon whose slot was emptied in the previous call. This request ensures that all slots will be always full or at least have one pending request and that no single daemon will halt the `mapi_get_next_pkt` request.

4.5 Security Issues

The LOBSTER infrastructure consists of many distributed monitoring sensors that cooperate through the Internet. Being exposed to the Internet, each monitoring sensor should have a rigorous security configuration in order to preserve the confidentiality of the monitored network and resist to attacks that aim to compromise it. In the following sections we describe some important security considerations for the LOBSTER infrastructure.

4.5.1 Public Key Authentication

The administrator of each LOBSTER sensor is responsible for issuing credentials, which specify the usage policy for that sensor, as well as delegate authority to use that sensor. Figure 4.2 illustrates the steps that take place during the authorization of a user. Given that the administrator of a node has specified the usage policy (step 1), a user that wants to use the monitoring sensor must first acquire the necessary credentials for that sensor. Credentials delegate authority to a user (or a user group) identified by a public key (or a set of public keys). Thus, the user first has to deliver his/her public key to the administrator (step 2), which is added into the credentials. The administrator then signs the credentials and stores them in the Policy Repository (step 3). Since the credentials are digitally signed, they can be easily distributed over untrusted networks, so the user can safely download them from the Policy Repository (step 4), in order to use them for accessing the sensor(s).

The user then configures the network flows in his/her application according to the obtained credentials (step 5). Before creating a new network flow, the user has to call `mapi_set_authdata()`, which informs the sensor about the credentials of the user, and which public key should be used to identify him/her. The authentication in order to prove that he/she is really the user that corresponds to the public key is achieved by supplying a nonce value (an integer number), together with the encrypted version of this integer using the users private key. If the encrypted nonce decrypted with the supplied public key equals the original value of the nonce, the users request is authenticated. By using the flow descriptor as the nonce value, the authentication is also tied to this particular request. When the configuration of

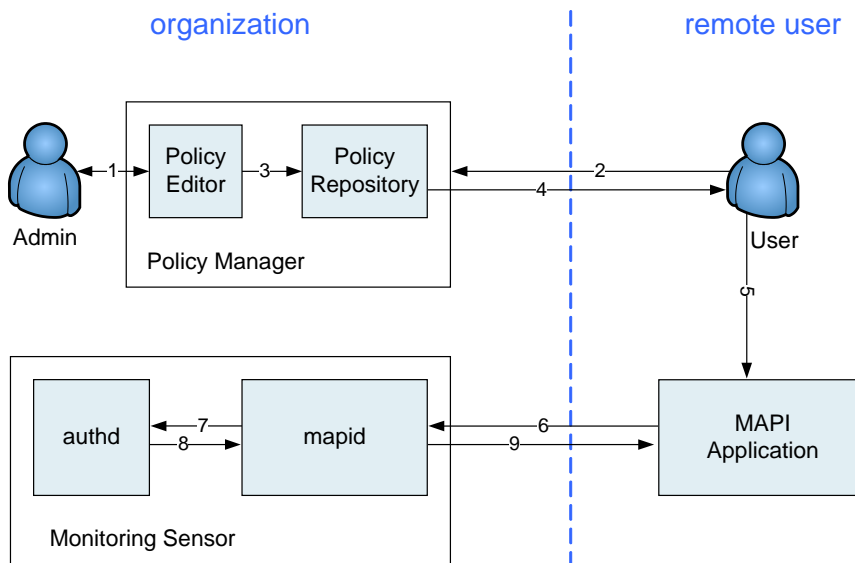


Figure 4.2: **Policy enforcement in LOBSTER.** (1) The Administrator specifies a usage policy, including anonymization. (2) The user delivers his/her public key. (3) Signed credentials are stored in the Policy Repository. (4) The user obtains his/her credentials. (5) User's network flows are configured to conform with the policy. (6) Each network flow is instantiated. (7-8) `authd` evaluates the specification of the flow for compliance with the supplied credentials and authenticates the user. (9) The flow is activated and the relevant flow descriptor is returned.

the flow is completed, the user instantiates the flow by calling `mapid_connect()` (step 6).

At this point, `mapid` (through the `agent`) has complete information about the flow in question. The user has provided his/her public key and credentials, while `mapid` is aware of the specification of the flow, as a result of the consecutive `mapid_apply_function()` calls that the user has issued to configure it. All this information is forwarded to `authd` (step 7), which checks the authentication and evaluates the specification of the flow for compliance with the supplied credentials. `authd` then returns the result of the evaluation to `mapid` (step 8). If the flow specification complies with the credentials and the authentication is successful, `mapid` activates the flow and returns the relevant flow descriptor to the user (step 9), otherwise the flow is rejected. Steps 6–9 are repeated each time the user wants to create a new flow.

4.5.2 Secure Communication Channels

Since all communication between user applications and the remote sensors will be through public networks across the Internet, special measures must be taken in order to ensure the confidentiality of the transferred data. Indeed, data transfers

through TCP are unprotected against third-parties that have access to the tcp packets, since they can reconstruct the TCP stream and recover the transferred data. This would allow an adversary to record control MAPI messages, forge them, and replay them in order to access a LOBSTER sensor and impersonate a legitimate user. For protection against such threats, any communication between the MAPI stub and the remote sensors will be encrypted, using mechanisms such as SSL. For intra-organization applications, where an adversary cannot have access to the internal traffic, encrypted communication may not be necessary, depending on the policy of the organization, and could be replaced by plain TCP, for increased performance.

4.5.3 Denial of Service Attack Countermeasures

Being exposed to the Internet, each monitoring sensor could be a potential target of Denial of Service (DoS) attacks. The aim of DoS attacks is to consume a large amount of resources, thus preventing legitimate users from receiving service with some minimum performance. For example, an adversary could initiate a SYN flooding attack targeting the agent of a monitoring sensor.

TCP SYN flooding exploits TCP's three-way handshake mechanism and its limitation in maintaining half-open connections. Any system connected to the Internet that provides TCP-based network services, such as a Web server, an FTP server, or a LOBSTER monitoring node, is potentially subject to this attack. A TCP connection starts with the client sending a SYN message to the server, indicating the client's intention to establish a TCP connection. The server replies with a SYN/ACK message to acknowledge that it has received the initial SYN message, and at the same time reserves an entry in its connection table and buffer space. After this exchange, the TCP connection is considered to be half open. To complete the TCP connection establishment, the client must reply to the server with an ACK message. In a TCP SYN flooding attack, an attacker sends many SYN messages, possibly from a large number of compromised clients in case of Distributed DoS (DDoS) attacks, to a single server (victim). Although the server replies with SYN/ACK messages, these messages are never acknowledged by the client. As a result, the server is left with many half-open connections, consuming its resources. This continues until the server has consumed all its resources, hence can no longer accept new TCP connection requests.

It is clear that (D)DoS attacks are a major threat against distributed infrastructures such as LOBSTER, thus we plan to take strict measures in order to protect the monitoring sensors against them. Each lobster node will be equipped with a firewall configured with a conservative policy that will selectively allow inbound traffic from selected IP addresses only—the addresses of legitimate LOBSTER users. Inbound traffic from any other source will be dropped. Since the software architecture of a LOBSTER node is based on Linux, such a policy can be easily implemented using `iptables` or some other equivalent filtering mechanism, depending on the OS version.

However, besides DoS attacks from external sources, a sensor could suffer a DoS attack due to a misconfiguration of a legitimate user's application or some internal malicious user. For protection against such threats, monitoring sensors will be equipped with resource control mechanism that will limit the resources that a user can allocate.

Chapter 5

Distributed Execution Environment

In this chapter we consider the issue of distributed monitoring applications from a different angle. In particular, we explore potential problems with the architecture as described so far, as well as a possible solution. The problems are fairly well-known in the field of distributed systems, e.g., latency, bandwidth, and security. Often they are solved by decreasing the distance between the data and the code that processes the data. Whether or not these problems and their solutions are important in practice depends on the type of LOBSTER applications that will be developed. This is information that cannot easily be obtained by means of interviewing stakeholders, as the problems crop up only when an advanced monitoring infrastructure like LOBSTER's is available, which is currently not the case. However, to make our design future proof, we try to anticipate possible bottlenecks and outline the design of a distributed execution environment that would help alleviate the problems.

5.1 Problems with the existing approach

Moving the monitoring application away from the monitoring node, as done in the distributed MAPI as well as in the remote MAPI, introduces a few nasty problems. In particular, there may be issues with:

- latency - any reply-response interaction will take at least a round trip time (RTT);
- bandwidth - if a client application requires accessing large volumes of network data from multiple sites, sending all data to the client is not a practical solution, as it would lead to excessive network loads;

- firewalls and packet manglers - in a wide area network devices like firewalls and network address translators (NATs) render communication with remote sites located behind them difficult or even impossible;
- security - moving information across a network rather than taking it from a local device introduces problems for sensitive data as the communication can be intercepted.

The latency problem is made particularly poignant because of the way we handle results, as will be explained presently. A common method for hiding latency involved in interacting with a remote site is to use the principle that writes should be remote, so that reads are local. According to this principle, a sensor that reads packets that are needed by a remote application should transmit these packets across the network and store them directly in a remote buffer. This adds relatively little latency, as write operations do not expect results and, hence, incur a RTT latency. The read operations may then proceed locally and no longer incur a RTT latency either.

In remote MAPI, we have chosen not to implement such latency hiding techniques, for several reasons. Firstly, it would severely complicate the design of the MAPI daemon, while it is by no means certain that the problem is important for many applications. Secondly, it is not a practical solution in the context of LOBSTER, as transmitting all these packets may create a huge amount of traffic. This is particularly the case in the distributed MAPI where results may be sent by many sensors. In that case, the receiver and the links to the receiver may quickly become a bottleneck. In summary, for remote communication we have to accept reasonably high latencies.

A question that should be asked is whether high latency is important. The answer to that question depends on the nature of the application. For instance, an application that is interested in periodically reading a counter from a remote site, is not expected to suffer from this problem. On the other hand, an application that should respond immediately to network events may break if the latency goes up. For instance, suppose we have an application that monitors a peer-to-peer system that uses dynamic ports. It may do so by inspecting all *control* traffic corresponding to the peer-to-peer protocol and take action immediately when it finds a control packet containing the details of a new *data* connection. If the interaction with the monitoring application has to travel across the network using some form of remote polling, the application would miss a lot of the peer-to-peer data traffic.

One way of dealing with this problem is by encoding all required functionality in the MAPI functions. In that case, the monitoring application would simply run on the sensor in the fastest possible manner. However, it is doubtful that the full set of MAPI functions will cater to all applications, present and future.

The bandwidth problem is related to the latency issue. If the monitoring application has to perform some processing on a large number of network packets and the function to be performed is not available as a MAPI function, there is no

5.2. REDUCING LATENCY AND BANDWIDTH BY REMOTE EXECUTION

alternative to simply transmitting the packets across the network to the client site. Worse, with the current design it will have to do so using a poll model. This may well introduce unacceptable latency and bandwidth bottlenecks, especially if the application monitors multiple sites.

Firewalls and NATs are likewise devices with which we will have to deal. This is not an easy problem, as normally both of them make it difficult for outside parties to connect to a host within the protected domain. On the other hand, it is often easy for a host inside the protected domain to connect to a host on the outside. The problem occurs when both hosts are protected by firewalls and/or NATs. How to establish communication?

Several approaches have been suggested in the literature, most of which use some form of TCP splicing (also known as simultaneous SYN, or simultaneous initiation [2]). TCP splicing changes the common three-way handshake of TCP into one that is more like a 2×2 handshake that is initiated from both ends. An example of how TCP splicing is used to deal with firewalls and NATs is described in [1].

The final issue, security of communication lines, is a fairly well-studied domain. Rather than sending raw data across the lines, it may be prudent to encrypt such information first. Doing so, however, only exacerbates the latency issue, as encryption and decryption are relatively expensive operations.

5.2 Reducing latency and bandwidth by remote execution

A well-known approach to reducing latency is to process data remotely. Rather than moving the data to the processing node, the idea is to push the code to process the data to the remote sensor. Once the code is at the remote sensor, it can perform all subsequent processing locally. This is attractive in the LOBSTER context also, as processing may involve large volumes of data and the latency across a wide area network may be fairly high.

However, we may expect for security and performance reasons that network operators are loathe to permit foreign code to run on the sensor itself. What is more acceptable is to run the code on a machine that is 'close' to the sensor. Note that such code is not part of the MAPI functions themselves and all communication still takes place across a (single) network link. For this reason, such code will be inherently slow, relative to the packet processing code on the sensor itself. What this suggests is that it will not be feasible to implement functions that have to process every packet on such a node if the link rate is high. Controlling the sensor (e.g., using the MAPI), and responding to events, on the other hand, is a task that is more suited to this type of code.

A distributed execution environment, would permit applications to spread the load over a distributed system. While this adds significantly to the flexibility, it also increases complexity. To control the complexity, we require the distributed execution environment to be a simple extension to the MAPI (remote and distrib-

uted) discussed in the previous chapters. In addition, the distributed environment should (a) provide secure connections, (b) be simple to set up and maintain, and (c) be portable (given the heterogeneity of systems employed by potential users of the LOBSTER architecture). For convenience, it would be helpful if the distributed execution environment had explicit support for building distributed applications. For instance, if it were possible to send message to a group of processing nodes, or if firewalls could be easily (and safely) circumvented. In the next section we discuss some of these desirable properties.

5.2.1 Useful features that are not in LOBSTER

Besides actual problems that may inhibit the functioning of the monitoring infrastructure, as described in the previous section, there are features that would simply be ‘nice to have’ and that are currently lacking in the architecture. Again, most of these issues occur only when moving towards highly distributed monitoring applications.

In this section, we identify such ‘useful but not essential’ features for the common access platform’s distributed execution environment. In essence, the common access platform is expected to execute the control code for distributed monitoring applications. As the code is not likely to be executed on the fast path on a monitoring node, it is unrealistic and indeed unnecessary to expect it to cope with multi-gigabit speeds. High-speed processing should be performed on the monitoring node. It may be *controlled* via a flexible API by relatively slow control code, probably running on a different machine. Such code may process results and possibly some of the packets, but certainly not every packet of a fast link.

Different distributed execution environments provide different features. We will mention only a few that may be particularly useful:

1. ability to run code remotely;
2. ability to schedule, transfer and stage files and applications
3. elaborate communication APIs (PVM, MPI, different APIs), allowing for useful primitives such as group communication;
4. sandboxing;
5. authentication and authorisation;

5.2.2 Virtual machines

‘Sandboxing’ as mentioned in the list above these day commonly refers to running the code in a virtual machine (VM). For VMs there are many options. For example:

- Java Virtual Machine (JVM);
- VMware;

5.2. REDUCING LATENCY AND BANDWIDTH BY REMOTE EXECUTION

- Xen;
- Others:
 - .Net platform: not an obvious choice as most of the potential users are used to UNIX-like environment;
 - PlanetLab: potentially as attractive as Xen or VMware, but there is not much information about how to build a private PlanetLab. At any rate, most advantages and disadvantages for PlanetLab coincide with those of VMware or Xen.
 - User Mode Linux: performance is a problem;
 - Bochs: performance is a problem;
 - QEMU: performance is a problem.

However, we need much more than just a VM. In fact, we need a VM plus a distributed environment, preferably one that provides communication primitives and other features needed or useful for our distributed network monitor. The number of options for the distributed environment is almost endless and we will not evaluate them in this document. Examples include:

- Ibis
- Globus
- CORBA
- Webservices
- environments that support MPI/PVM

In principle, any environment will do, as long as it satisfies the constraints mentioned earlier. In this chapter we will identify various options of how the distributed execution environment could be implemented. As the distributed execution environment is an extension to the core of the project, there may even be multiple different implementations, just like there are multiple implementations of CORBA.

Note that of all the systems mentioned in the VM list, only Ibis offers a distributed processing environment. For all others, we will have to look for something that suits our needs. As mentioned above, there are many options to choose from in distributed processing environments.

We should also mention that the list of VMs is compiled of rather distinct types of VMs. For instance, in virtualization terms, the JVM is known as a ‘high-level-language process VM’ [5]. A process VM is a virtual platform that executes an individual process. Xen and VMware on the other hand are known as ‘system VMs’, i.e., VMs that provide a complete, persistent system environment that supports an operating system along with its many user processes. Currently, system VMs suffer

from the fact that CPUs are not very suitable for virtualization. However, this will be fixed in the next generation of x86-based processors, so that virtualization will no longer incur significant overhead and the implementation of virtual machine monitors will be much simplified [10].

5.2.3 Evaluation of different solutions

In this section we briefly compare the various solutions to virtualization and assess their suitability for the Loster environment. For all solutions, but in particular the system VMs an issue that needs to be solved is making sure the code running in the VM does not misbehave in a way that, although it does not jeopardise the integrity of the host system, may damage the hosting organisation's reputation. For instance, the code should be prevented from spamming or launching attacks on other machines. Whether technical or legal solutions are best suited for this purpose is beyond the scope of this chapter.

5.2.3.1 VMware

VMware is perhaps the best-known virtualization platform. Its fast binary rewrite technique offers the *appearance* of pure virtualization (i.e., guest operating systems will run without modification) and is both mature (easy to install) and reasonably efficient (the code runs fairly fast, although slower than code without virtualization). It is fast already and may become faster yet as when VMware take advantage of Intel's future chip-level virtualization technologies (Vanderpool). Unfortunately, this is not expected to be in time for this project.

Pros:

- Safe, low-level partitioning, no interference between two VMs;
- Efficient code (mostly native);
- Authentication is taken care of with the granularity and strength of the guest OS.

Cons:

- *Ease-of use.* VMware is commercial software (while price may not be an issue, it raises the threshold for downloading and evaluation our work);
- *Ease-of use.* While VMware is not particularly hard to install, it implies that multiple machines must be administered;
- *Communication.* Provides no solution in itself for communication primitives or indeed for the common access platform.

5.2. REDUCING LATENCY AND BANDWIDTH BY REMOTE EXECUTION

- Performance and system requirements. We probably need multiple VMs (one for each group of clients) - multiple VMs tend to increase significantly the requirements for the system (especially memory), and often slows down the system considerably;
- *Firewalls*. No simple solution for firewalls or NATs;
- *Portability*. We may use VMware with any guest OS;
- *Security*. No direct support for secure communication (although most guest OSs provide this in one way or another);
- *Security*. Provides no solutions for authorisation, etc. Some of these issues can be solved by something like the `authd` daemon developed in SCAMPI, but it is certainly not provided in the system.

5.2.3.2 Xen

Xen has been picked up the open source community and will find its way in major distribution (interested parties include Red Hat and manufacturers like Intel and HP have also expressed interest). Xen consists of a hypervisor that partitions resources and provides an idealised hardware interface to the guest OSs. In the words of its makers: think of Xen as a next generation BIOS - Xen is a minimally invasive manager that shares physical resources (such as memory, CPU, network, and disks) among a set of operating systems. Effectively, Xen is transparent, as each operating system believes it has full control of its own physical machine. In fact, each operating system can be managed completely independent of one another.

Xen uses *paravirtualization*, rather than pure virtualization. In other words, the interfaces offered to the guest OS is different from that of the raw hardware. It means that the OSs need to be modified in order to run on Xen. Xen is a new architecture, slightly different from x86, that operating systems must be ported to.

Moreover, Xen divides resources very rigidly: it's impossible for a misbehaving guest (an operating system that runs on a Xen host) to deny service to other guests. Simultaneous yet discrete operation is incredibly valuable.

One problem with Xen is that it may not (yet) very simple to use and maintain:

"There's obviously a lot of work to be done in making Xen friendlier to install, getting more tools around the administration of Xen, etc." [Nathan Torkington of the Open Source Convention, OSCON Europe.]

Pros:

- Safe, low-level partitioning, little interference between two VMs;
- Very efficient code (native) - faster than VMWare;
- Authentication is taken care of with the granularity and strength of the guest OS.

Cons:

- *Ease-of use.* Open source, but not terribly user friendly;
- *Ease-of use.* Xen also implies that multiple machines must be administered;
- *Communication.* Provides no solution in itself for communication primitives or indeed for the common access platform.
- *Portability.* Xen works with Linux and some version(s) of BSD - current work is on incrementing the number of supported OSs, but Windows OSs will not be supported;
- *Firewalls.* No simple solution for firewalls or NATs;
- *Security* No direct support for secure communication;
- *Security.* Provides no solutions for authorisation, etc. Some of these issues can be solved by something like the `authd` daemon developed in SCAMPI, but it is certainly not provided in the system.

5.2.3.3 Ibis

The goal of the Ibis project at the Vrije Universiteit Amsterdam is to design and implement an efficient and flexible Java-based programming environment for distributed computing. Java has some advantages for distributed applications. Foremost, by being based on a high-level virtual machine concept that is implemented on a huge number of operating systems, it is inherently more portable than traditional, statically compiled languages. This makes it much easier to execute Java applications in a heterogeneous environment. Also, Java is based on a high-level, object-oriented, type-safe programming model and it has built-in support for multithreading and distributed computing. A disadvantage is raw speed as Java is still significantly slower than native code. Techniques like JIT improve things considerably. Recent reports even suggest that Java with a good JIT compares on a par with C++ code.

It seems that Ibis cannot be compared directly with VMware or Xen. It is not a VM in itself. Rather, it is an environment for distributed computing based on the JVM. At best, we can compare it with a more specific environment on Xen or VMware, e.g., VMware/Xen + Linux + Corba. Indeed, there is nothing to prevent one from running Ibis for instance on top of Xen/linux.

Pros:

- Safe (i.e., as safe as the JVM);
- Support for many communication primitives and models (e.g., multicast, group);

- Highly portable;
- Fairly efficient code with just-in-time compilation, but not so fast as native code;
- Support for coping with firewalls and NATs;
- Direct support from developers;
- Support for secure communication;
- Open source (BSD-like license).

Cons:

- *Performance and system requirements.* Fast for a high-level VM, but not as fast as native code.
- *Security.* As secure as the JVM. Whether this is good enough, depends on your point of view. We should bear in mind that the JVM is used in many existing public services. Moreover, as the code runs on a different machine (as seems likely), the integrity of the JVM itself may be somewhat less of an issue.
- authorisation is currently not handled

5.2.3.4 Summary

For most points in section 5.1, Ibis seems to have an advantage. Performance-wise VMware and Xen may hold an advantage. On the other hand, both of these only provide a platform to host an OS. We still need to identify a common access platform to run on these systems. If the common access platform is used for slow-path code, the performance may be of less importance. None of these solutions has sufficient support for authentication and authorisation, so some authorisation is required for all of them. In the SCAMPI project, an authentication and authorisation daemon was developed that is already used in the LOBSTER project and that seems quite suitable for this task.

5.3 The LOBSTER distributed execution environment

As the distributed execution environment is an extension to the LOBSTER architecture, the consortium will describe only the minimal subset of requirements that should be satisfied. In particular, we do not describe how nodes in the distributed system communicate and what sort of code they run. Instead, we only prescribe a small number of primitives that are needed to set up a VM, load data and code in the VM and execute code in the VM. With these primitives any type of distributed execution environment can be built:

CHAPTER 5. DISTRIBUTED EXECUTION ENVIRONMENT

- `vm_handle_t vm_create (client_id_t type, enum vm_type type, vm_param_t *param);`

Create and start a virtual machine of a specific type for a specific client and with specific initialisation parameters and return a handle to this VM.

- `int vm_delete (vm_handle_t vm);`

Delete a VM.

- `int vm_scp_to (vm_handle_t vm, char *src_filename, char *dst_filename);`

Copy a file to a specific VM.

- `int vm_scp_from (vm_handle_t vm, char *src_filename, char *dst_filename);`

Copy a file from a specific VM.

- `int vm_execute (vm_handle_t vm, char *executable, vm_param_t *params);`

Execute a program in an existing VM.

- `int vm_ping (vm_handle_t vm);`

Check whether the VM is still alive.

Using these primitives, it is possible to build a VM-based infrastructure on top of Xen, VMware, Ibis, or any other virtual machine. Within the project's time-frame, we plan to implement a basic infrastructure using one of the above VMs.

Bibliography

- [1] Alexandre Denis, Olivier Aumage, Rutger F. H. Hofman, Kees Verstoep, Thilo Kielmann, and Henri E. Bal. Wide-area communication for grids: An integrated solution to connectivity, performance and security problems. In *HPDC*, pages 97–106. IEEE Computer Society, 2004.
- [2] Information Sciences Institute. Transmission Control Protocol RFC 793, 1981. Edited by Jon Postel. Available at <http://rfc.sunsite.dk/rfc/rfc793.html>.
- [3] Michalis Polychronakis, Kostas G. Anagnostakis, Evangelos P. Markatos, and Arne Øslebø. Design of an Application Programming Interface for IP Network Monitoring. In *Proceedings of the 9th IFIP/IEEE Network Operations and Management Symposium (NOMS'04)*, pages 483–496, April 2004.
- [4] Georgios Portokalidis and Herbert Bos. Admission control daemon, 2004. <http://admctrl.sourceforge.net>.
- [5] James E. Smith and Ravi Nair. The architecture of virtual machines. *IEEE Computer*, 38(5):32–38, July 2005.
- [6] The SCAMPI Consortium. MAPI Public Release. http://mapi.uninett.no/download/mapi_1.0b1.tgz.
- [7] The SCAMPI Consortium. Deliverable D1.3: Final Architecture Design, November 2003. <http://www.ist-scampi.org/publications/deliverables/D1.3.pdf>.
- [8] The SCAMPI Consortium. Deliverable D2.3: Enhanced SCAMPI Implementation and Applications, April 2004. <http://www.ist-scampi.org/publications/deliverables/D2.3.pdf>.
- [9] The SCAMPI Consortium. A Tutorial Introduction to MAPI, April 2005. <http://mapi.uninett.no/doc/mapitutor.pdf>.
- [10] Rich Uhlig, Gil Neiger, Dion Rodgers, Amy L. Santoni, Fernando C.M. Martins, Andrew V. Anderson, Steven M. Bennett, Alain Kagi, Felix H. Leung, and Larry Smith. "intel virtualization technology". *IEEE Computer*, 38(5):48–56, July 2005.