Department of Computing Science and Mathematics
University of Stirling

The ACCENT Policy Server

Stephan Reiff-Marganiec and Kenneth J. Turner

Technical Report CSM-164

ISSN 1460-9673

December 2005

Department of Computing Science and Mathematics
University of Stirling

The ACCENT Policy Server

Stephan Reiff-Marganiec and Kenneth J. Turner

Department of Computing Science and Mathematics
University of Stirling
Stirling FK9 4LA, Scotland

Telephone +44-786-467421, Facsimile +44-786-464551
Email srm13@le.ac.uk, kjt@cs.stir.ac.uk

Technical Report CSM-164

ISSN 1460-9673

December 2005

Abstract

The ACCENT project was concerned with developing a practical and comprehensive policy language for call control. The project therefore studied a number of distinct tasks: the definition of the language, and also a three-layer architecture for deploying and enforcing policies defined in the language. This document focuses on the policy system layer of the three-layer architecture. The policy system layer is concerned with storing, deploying and enforcing policies. It represents the core functionality of the three-layer architecture. This report discusses the prototype implementation at a technical level. It is intended as supporting documentation for developers continuing to enhance the prototype, as well as those wishing to gain an insight into the technical details of the policy server layer.

Relative to the version of November 2003, this Technical Report has been updated as follows:

- The procedure for securing the policy database has been described (section 3.1).
- The *users* policy database table required by the policy wizard has been described (section 3.1).
- More information on running the policy store has been provided (section 3.2).
- The policy server configuration file has been extended (section 3.3).
- More information has been provided on starting the policy server (section 3.3).
- An improved description has been given of the user layer/policy server interface (section 5).
- The policy server QUERY interface has been extended to include the type of information to be returned and the id of the tuple required (section 5.1.1).
- Brief mention has been made of the H.323 and Mitel 7000 ICS protocol message handlers (sections 3.1, 6.7.2 and 6.7.3).

Relative to the version of August 2004, this Technical Report has been updated as follows:

- The description has been brought into line with the current definition of the policy language and the current implementation of the policy server.
- More information has been provided on setting up MySQL and TSpaces (section 3.1).
- Configuration properties have been added and renamed.
- Configuration properties are no longer maintained in the policy store.
- A *ContextHandler* class has been defined in the *protocol* package (section 6.7.4). All protocol classes have been moved to this package.
- QUERY now carries a parameter identifying the policy/variable identifier (section 5.1.1). The identifier '*' now means all policies/variables in a DELETE or QUERY.
- The description of environment variables has been shortened as these are now defined by the policy language Technical Report (section 6.3.2).
- The Mitel 7000 ICS interface code has been described in detail (section 6.7.3).

Relative to the version of May 2005, only minor editorial revisions and technical revisions to the Mitel 7000 Interface have been made.

Contents

List of Figures

Chapter 1

Overview

The ACCENT project is concerned with developing a practical and comprehensive policy language targeted at the call control domain. As such we are concerned with a number of distinct tasks: the definition of the language (discussed in [10]) as well as the definition of a three layer architecture for deploying and enforcing policies defined in the language. The latter has been discussed in [11] and a prototype implementation is underway.

This document focuses on the policy server layer of the three layer architecture. The policy server layer is concerned with storing, deploying and enforcing policies. It represents the core functionality of the three layer architecture and thus is of most interest to us. In this document we discuss the prototype implementation at a technical level. This document is seen as supporting documentation for developers continuing to enhance the prototype as well as those wishing to gain an insight into the technical details of the policy server layer.

In the following sections we consider the context of the software discussed in this document briefly; for a fuller account we refer the reader to [11]. We then discuss startup and shutdown of the policy server, and describe the interfaces to the underlying call control layer as well as the policy definition (or user interface) layer. We then consider implementation details, looking at the individual classes in this project, and discussing their role and interworking. The architecture is open and as such can be extended. In fact we document two foreseen extensions: the addition of further protocols if we wish to use additional call control architectures, and support for further environment variables to be used in policies. Finally we consider further ideas and suggestions for continuation of the work.

The reader should consider other supporting documentation, in particular the papers mentioned before ([10, 11]). Technical reports describe the ACCENT Policy Language ([12]) and the ACCENT Policy Wizard ([14]). Further and related information is available on `http://www.cs.stir.ac.uk/accent`.

Note that much of the following document is related to our current setup, but this does not need to be restrictive. In fact we discuss some extensions which invariably will change the exact class setup (by adding further classes to packages). While we have chosen an IBM tuple space implementation for the policy store and MySQL as a database, though these are not strictly required.

Chapter 2

Policies for Call Control

We proposed a three layer architecture consisting of (1) the communications layer, (2) the policy layer and (3) the user interface layer. A three-tier architecture is used in completely different ways for other applications. However, a three-tier policy architecture emerges naturally for this work.

When we consider existing call control architectures, similar architectures have been in use some time. For example, in the IN the three layers are given by the SSP (Service Switching Point), SCP (Service Control Point) and SCE (Service Creation Environment). The functionality of each of these layers is similar to the respective layers in the proposed architecture. Similarly in a SIP [4] environment, the Communications Layer is provided by SIP proxies, while additional functionality can be added to the proxies in form of CPL [8] or CGI scripts (essentially providing the control functionality of the policy server layer). Clearly we expect that there are tools available to create such scripts, which then forms a SIP service creation layer.

However our proposed architecture differs in several key aspects, the details of which we discuss throughout the paper. With reference to Figure 2.1:

- The Policy Servers can communicate with each other to negotiate goals or solutions to detected problems. This is not the case in the IN or SIP.
- The User Interface Layer provides end-users with a mechanism to define functionality. IN does not provide any such mechanism, although this was partially a goal of Parlay [9] and JAIN [7].
- The User Interface Layer and the Policy Servers make use of Context information to provide new capabilities. To our knowledge this has not been done before.
- The Policy Server layer is independent of the underlying call architecture. That means that we can work across several communications architectures if required. Advances at the communications layer will not adversely impact on the higher layers.
- No complex intra-layer signalling is required, as effectively the call path is rerouted through the Policy Servers.

The architecture makes the underlying communications network transparent to the higher layers. It enables end-user configurability and provides communication between policy servers which can be used for interaction handling. We will briefly discuss next the details of each layer.

2.1 The Communications Layer

The communications layer represents the chosen call architecture. For this paper, we assume a general structure consisting of end-devices and a number of switching points. The switching points can be SSPs as in the IN, proxy servers as in SIP, PBXs or any other entity one might expect along a call path. End-devices can be traditional phones, soft-clients or any other communications device.

We impose two crucial requirements on the communications layer. (1) The policy servers must be provided with any message that arrives at a switching point; routing is suspended until the policy server has dealt with the

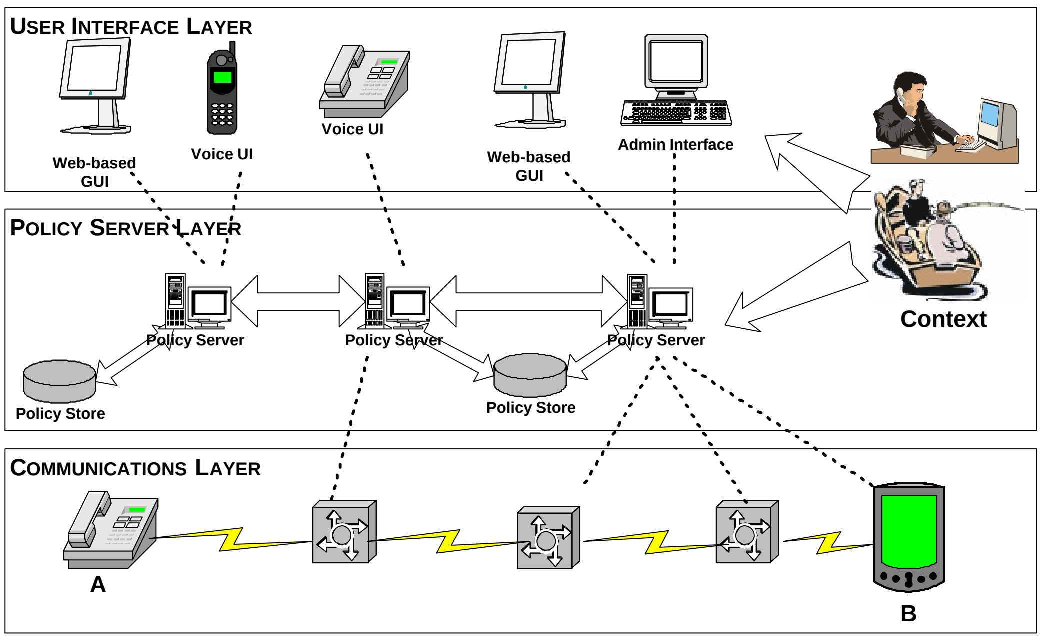


Figure 2.1: Overview of the Policy System Architecture

message. (2) A mapping of low level messages of the communications layer into more abstract policy events must defined.

In this paper we will not be concerned with any further details of this layer and simply assume that both requirements are fulfilled. Our investigations have shown that this is a realistic assumption.

2.2 The Policy Server Layer

The policy server layer is probably the most interesting layer. This layer contains a number of policy servers that interact with the underlying call architecture. It also contains a number of policy stores, that is database or tuple space servers where policies can be stored and retrieved by the policy servers as required. We assume that several policy servers might share a policy store, and also that each policy server might control more than one switching point or apply to more than one end device.

The policy servers interact with the user interfaces in the policy creation process as will be discussed in section 5. They also interact with the communications layer where policies are enforced; this will be discussed in detail in section 4.

A further role of the policy server is to detect and resolve (or suggest resolutions of) conflict among policies. To enable richer resolution mechanisms, communication between policy servers for information exchange is permitted.

Further, the policy servers have access to up-to-date information about the user's context details which are used to influence call functionality.

2.3 The User Interface Layer

The user interface layer allows users to create new policies and deploy these in the network. A number of interfaces can be expected here: graphical or web-based interfaces, voice-controlled mechanisms, or administrative interfaces, each providing functionality targeted at certain types of users and devices.

We would assume the normal user to use a web-based interface for most functions. However mobile users might not have the capability to use such technology, so voice controlled interfaces are more appropriate. These also suit users that simply want to activate or deactivate policies. A voice interface is essential for disabled or partially sighted users. Both web and voice interfaces should guide the user in an intuitive way, preferably in natural language or in a graphical fashion. We also envision libraries of policies that users can simply adapt to their requirements and combine to obtain the functionality required, in a similar way that for example clip-art libraries are common today.

The administrative interfaces are system-oriented and exist mainly for system administrators to manage more complex functionality.

The policy definition environment provides access to contextual variables which are instantiated at runtime by the policy servers with actual context information. For example, ‘secretary’ in a forwarding policy could be filled in from a company organisation chart in conjunction with holiday and absence information. Also, the location of a user could be determined from an active badge system.

2.4 Language Support and System Limitations

The policy server supports nearly all of the APPEL language described in [13]; the communication server modules that underlie the policy server are rather more limited. The following observations refer to the current version (1.5) of the policy server:

- Policies must be explicitly triggered (i.e. by an external call, presence or availability event). Although a policy can defined without a trigger (e.g. a policy that depends on time only), it will be activated only if an external event causes it to be considered.
- The *and* operator always applies actions in the same order as the policy. The *or* and *or else* operators always choose the first action in the policy.
- Presence and availability may be checked only for another user on the same policy server. Injudicious or malicious use of *note_availability* or *note_presence* may lock the policy server up by causing a direct or indirect loop of checks.
- The *send_message* action is implemented only for an email address parameter and a plain text message (though variable substitution is allowed in this).
- Static detection of policy conflicts has not been implemented. Dynamic detection of policy conflicts is limited to a single server. A single level of resolution checking is supported; conflicts among resolutions are not detected.
- The policy server does not verify the source of any call or policy information. Thus any Internet system may maliciously provide incorrect information or may access private policies.
- The policy database stores policy wizard passwords in clear, so it is essential that only authorised users can read the accent database. Passwords are also stored in various policy system configuration files, so again these should be readable only by authorised users. Passwords are passed between system elements in clear, so there is no protection against eavesdropping or spoofing.
- The policy wizard does not support resolution policies. Resolution policies must be prepared manually in XML. They can then be entered using the ‘Add Policies’ button on the policy server GUI. However they cannot subsequently be edited within the policy system (except by adding a replacement in the same way). More seriously, they cannot be deleted (except by emptying the entire policy store); a work-around is to add a replacement that is not enabled.
- If it is planned to run the policy database on a Mitel 7000, note that the GNU *librwap* library provided in the current version (3.0 SP2) has a bug that prevents MySQL working over a network connection. Correct this by running *yum upgrade glibc*, which will typically upgrade *glibc* and *glibc-common* from version 2.2.5-40 to version 2.2.5-44.
- If it is planned to run the policy database on a Mitel 7000, note that the current version (3.0 SP2) requires some changes to the firewall settings if MySQL needs to be accessed from outside the Mitel 7000 (e.g. by the policy wizard).

The SER module supports only the following:

triggers: *connect*, *connect_incoming*, *connect_outgoing*, *disconnect*, *disconnect_incoming*, *disconnect_outgoing*

actions: *add_medium* for audio and video, *connect_to*, *forward_to*, *reject_call*.

The GNU GK module supports only the following:

triggers: *bandwidth_request*, *connect*, *connect_incoming*, *connect_outgoing*, *disconnect disconnect_incoming*, *disconnect_outgoing*, *no_answer*

actions: *confirm_bandwidth*, *forward_to*, *reject_bandwidth*, *reject_call*.

The Mitel 7000 module supports only the following:

triggers: *connect*, *connect_incoming*, *connect_outgoing*, *disconnect*, *disconnect_incoming*, *disconnect_outgoing*

actions: *forward_to*, *play_clip*, *reject_call*: *play_clip* must currently be the last (external) action of a policy; *reject_call* takes an autoattendant number as its argument.

Chapter 3

Running The Policy System

The policy server is written entirely in Java. It makes use of a properties file to establish local settings. The policy server uses a policy database and a policy store. The policy server requires Java 1.4 (or above), as some String handling functions are not available in older Java versions. The policy server makes use of a relational database server as a policy database (currently MySQL) to store static policy information such as protocol terms and user information. The policy server also makes use of a tuple space server as a policy store (currently IBM TSpaces) to hold dynamic policy information, policies and policy variables.

3.1 Running The Policy Database

The policy database is a conventional relational database, currently using MySQL (http://www.mysql.com/). In principle, any SQL database could be used. It is assumed that a system administrator has already installed and configured MySQL. This might be used only by the policy system, but may also be shared with unrelated applications. Everything needed by the policy system is in the *accent* database.

Typically a system should be configured to start the policy database automatically on boot-up. The policy database *must* be running before the policy server is started. If the database is started and stopped manually, database-specific commands should be used (e.g. *mysqld* or *mysqladmin* for MySQL).

Since the policy database contains user passwords, it should be configured to allow only authorised users. A typical setup would allow user *accent*, with a specified password on all remote hosts and the local host, to read and modify the *accent* database. (There is no necessity for the policy user or database to be called accent as these are configurable settings.)

The database administrator might typically do this as follows. The file *lib/mysql_setup.sql* contains this code. The *accent* password will need to be placed in other policy system configuration files.

```
CREATE DATABASE accent;

REVOKE ALL PRIVILEGES ON accent.* FROM "%"@"%";

GRANT ALL PRIVILEGES ON accent.* TO accent@"%" IDENTIFIED BY 'password';

GRANT ALL PRIVILEGES ON accent.* TO accent@localhost IDENTIFIED BY 'password';

FLUSH PRIVILEGES;
```

The database provides information to the policy server as to which protocols are currently supported and crucially about the mappings between the policy terminology and the respective meaning in protocol terms. Currently the database contains values for the SIP, H.323 and Mitel 7000 ICS protocols.

The protocols table specifies a list of protocol names indexed by a numeric id. The file *lib/protocol_setup.sql* contains this code.

```
CREATE TABLE protocols (
  id int AUTO_INCREMENT,
  protocol text NOT NULL,
  PRIMARY KEY (id)
);

INSERT INTO protocols VALUES(NULL,'SIP');
INSERT INTO protocols VALUES(NULL,'H323');
INSERT INTO protocols VALUES(NULL,'Mt7000');
...
```

The terminology mapping is the more interesting table, as it specifies a list of terminology mappings indexed by a numeric id. Each row in this table contains a protocol name, a policy term and the related protocol term. Policy terms are exactly the text that will occur in the XML of the policy (e.g. *add_medium(video)* or '*connect_outgoing*'). For details of the policy terms, we refer the reader to the documentation of the APPEL policy language [13]. Protocol terms can be longer, for example adding a video channel might be more than a simple operation. It is the role of the protocol specific protocol handler to evaluate the link between policy and protocol terms and arrange for appropriate actions to be initiated. The *isAction* field should contain 'no' or 'yes' to indicate an input to the policy server or an output from the policy server. It determines how the data found in the field will be used: actions are translated from policy terms to protocol actions, non-actions are translated from protocol to policy terms (e.g. to identify triggers). The file *lib/terminology_setup.sql* contains the following code:

```
  CREATE TABLE terminology_mapping (
    no int AUTO_INCREMENT,
    protocol text NOT NULL,
    PolicyTerm text NOT NULL,
    ProtoTerm text NOT NULL,
    isAction text NOT NULL,
    PRIMARY KEY (no)
  );

  INSERT INTO terminology_mapping
    VALUES(NULL,'SIP','connect_incoming','INVITE:in','no');
  INSERT INTO terminology_mapping
    VALUES(NULL,'SIP','disconnect_incoming','BYE:in','no');
  ...
```

Before starting the policy server, the user should ensure that these two tables exist and are filled with the required values. The policy server reads the contents of the two tables only at startup and maintains its own copy. During the runtime of the policy server no further mappings can be added.

Note: It is not fundamental to the architecture that mappings are only added at startup time. This is simply a restriction of the current implementation. It would be conceptually possible to modify and add mappings at runtime.

The database also holds information needed by the policy wizard, specifically the currently registered users and their details.

```
  CREATE TABLE users (
    code INT AUTO_INCREMENT,     -- 1, 2, ...
    username VARCHAR(32),        -- e.g. marc
    password VARCHAR(32),        -- e.g. ab*123
    address VARCHAR(32),         -- e.g. marc@sa.fr
    locale TEXT,                 -- en-gb, fr-ca
    stage INT NOT NULL,          -- 0 beginner, 1 intermediate,
                                 -- 2 expert, 3 administrator
    PRIMARY KEY (code)
  );
```

This table must be populated by at least an entry for the policy system administrator, always called *admin*. To avoid confusion between the administrator account and any account that this individual might have, the administrator should have a distinct email address (even if it is an alias). Entries for other users can be pre-loaded into the database, but this is not required as the policy wizard allows users to be added, modified and deleted. The file *lib/user_setup.sql* contains the following code:

```
INSERT INTO users VALUES (
NULL, 'admin', 'some_password', 'admin@cs.stir.ac.uk', 'en-gb', 3
);
...
```

3.2 Running The Policy Store

The policy store is a tuple space, currently using IBM TSpaces (http://www.alphaworks.ibm.com/tech/tspaces/download/). In principle, any XML database could be used. It is assumed that a system administrator has already installed and configured TSpaces. This might be used only by the policy system, but may also be shared with unrelated applications. Everything needed by the policy system is in the *Policies* tuple space.

TSpaces requires a configuration file. Among other things, this defines how policies are checkpointed (i.e. stored). This allows policies to be preserved in the event of system failure or shutdown (though recently written data may be lost). In general, avoid shutting TSpaces down abruptly as data may be lost. An example configuration is provided in *tuples.properties*. The configuration file should be readable only by whichever account runs TSpaces, as it contains the administrator username and password. Similarly, the cache and checkpoint files should be readable/writable by only this account. The following is an extract of the configuration file; see the TSpaces documentation for the details. (This example is for a Windows system. On a Unix system, remove the "C:" prefixes.)

```
[Server]
CheckpointDir                = C:/usr/local/tspaces/ts-checkpoint
CheckpointInterval           = 10.0
checkpointWriteThreshold     = -1
DeadLockInterval             = 15
ExpireInterval               = 15
ResultOrderFIFO              = false
Port                         = 8200

[HTTPServer]
ClassesDirectory             = ./
HTTPAdminSupport             = false
HTTPServerSupport            = true
HttpPort                     = 8201

[FileStore]
CacheDir                     = C:/usr/local/tspaces/ts-cache

[AccessControl]
ACVerifierClass              = com.ibm.tspaces.security.AccessControlVerifier
AdminUser                    = some_user
AdminPassword                = some_password
AdminGroup                   = AdminGroup
CheckPermissions             = true
TopGroup                     = UserGroup

[Group-UserGroup]
accent
root
Group AdminGroup

[Group-AdminGroup]
root

[DefaultACL]
accent                       Read Write
root                         Read Write Admin
UserGroup                    Read
```

```
  [CreateACL]
  accent                      Create
  root                        Create
```

Typically a system should be configured to start the policy store automatically on boot-up. The policy store *must* be running before the policy server is started. Conversely, the policy server should be stopped before the policy store is stopped. Convenience scripts *ts-start*, *ts-stop* and *ts-admin* are provided with the policy system distribution. These allow TSpaces to be started, stopped and administered.

Start up TSpaces. The *ts-admin* script can then be run to bring up a GUI that allows tuple space access to be controlled. The GUI also allows TSpaces to be shut down cleanly. Log in the with administrative user and password specified in the configuration file.

In the 'Group and User Hierarchy' pane, right click on the 'Users' symbol to 'Add A New User' – in this case, *accent*. Then right click on *accent* to 'Set User Password'. This password will need to be entered into other policy system configuration files.

The policy system stores policies in the *Policies* tuple space. Once this exists (it may not until the policy server is first run), set the ACL (Access Control List) for this tuple space. Typically allow user *accent* permissions for Read and Write, and the system administrator permissions for Read, Write and Admin.

To set up user *accent*, in the 'T Spaces Names' pane, click on 'Policies'. Right click in the 'ACL' pane to 'Add ACL Entry'. Give user *accent* the permission 'Read Write'. Finally, click on 'Apply'. In the administrative application, use 'Shutdown' to cleanly close down TSpaces.

The tuple space server requires access to the Java classes of all objects that might be stored in the server. Furthermore all these objects must be serialisable. Access is provided by including the classes in the classpath when TSpaces is started. Since the tuple space is used to store XML objects, the *xml4j.jar* file must be on the class path.

The tuple space server can be started at the command line: *ts-start*. In the event of data corruption or starting TSpaces from scratch, use the '-b' option with this script to start with a blank tuple space. An even more severe choice is to use the '-B' option that also wipes out any stored access control information. Do not use either option lightly!

A web interface allows the contents of the tuple space to be inspected by a web browser at port 8201 on the system running TSpaces. Whether this is enabled depends on the configuration file. The '-S' option to *ts-start* will ensure that the web interface is provided. However this is a security risk (users could view any policy) and is discouraged in a production installation.

3.3 Running The Policy Server

The policy server is controlled by a configuration file that contains Java properties. The exact format is therefore not critical. A typical example is given below for the system *guilder.cs.stir.ac.uk* running the policy server and the policy database. The outgoing email server is *smtp.cs.stir.ac.uk*, and the policy store server is *dubloon.cs.stir.ac.uk*. The policy database and policy store can run on different systems or on the same system as the policy server.

```
policy.message.port      9998
policy.upload.port       9999
policy.log.directory     ../../log

mail.server              smtp.cs.stir.ac.uk
mail.port                some_port
mail.password            some_password
mail.subject             Policy System Message

database.server          guilder.cs.stir.ac.uk
database.port            3306
database.username        some_username
database.password        some_password
database.name            accent
database.table           terminology_mapping
```

```
tuples.server           dubloon.cs.stir.ac.uk
tuples.port             8200
tuples.username         some_username
tuples.password         some_password
tuples.name             Policies
```

policy.*: These entries concern the policy server: the port for messages from the communications layer, the port for policy/variable uploads from the policy wizard, and the location of call logs. If a policy invokes a *log_event* action, the log entry is written to this directory. Entries are appended to the file *user*.log for each individual user. A relative path for the log directory is relative to wherever the policy server was started (e.g. the *bin* directory containing *ps-start*). An absolute path may be given (e.g. *C:/Temp/calls*); on a Windows system ensure that backslashes are given twice in the configuration file (e.g. *C:\\Temp\\calls*).

mail.*: These entries concern outgoing email: the mail server name, the subject of messages sent by the policy system, the mail server port number (optional), and the password used with the mail server (optional).

database.*: These entries refer to the policy database: its server name and port, the username and password used to access it, the name of the policy system database, and the name of the protocol-policy terminology mapping.

tuples.*: These entries refer to the policy store: its server name and port, the username and password used to access it, and the name of the tuple space used by the policy system.

On startup, the policy server reads this file and creates a Java properties object. The configuration file normally resides in the parent *PolicyServer* directory. The *server.properties* file is the default one, but any file can be named on the command line when the policy server is started up.

The *PolicyServer* directory includes subdirectories *bin* (compiled code), *doc* (JavaDoc for the code), *lib* (required jar files) and *src* (Java). The code is in packages named *uk/ac/stir/cs/accent/**.

The convenience script *ps-start* is provided to start the policy server. Note that it includes a number of required jar files from the *lib* directory. The script can be edited to use anything other than the default configuration file. The main class *uk.ac.stir.cs.accent.server.PolicyServer* accepts the command-line options '-c' (console output, the default), '-g' (GUI output), '-h' or '-?' (for help). These may be followed by the name of the properties file to be used.

The policy server may have to run on a system without any GUI support (e.g. the Mitel 7000). An alternative main class *PolicyServer.java.nogui* is provided for this purpose. It accepts the same command-line options, but '-g' (GUI output) id disallowed. If the source code is available, copy *PolicyServer.java.nogui* in *src/uk/ac/stir/cs/accent/server* to *PolicyServer.java* and recompile. If the source code is not available, copy *PolicyServer.class.nogui* in *bin/uk/ac/stir/cs/accent/server* to *PolicyServer.class*.

When running via a plain console, log messages are sent to the terminal. When running via a GUI, log messages appear in various windows. Facilities are provided to add policies from XML files, to clear out all policies in the tuple space (not recommended!), to save the logs to a file, to empty the logs (which will otherwise build up over time), and to quit the policy server (not recommended in a live system!). In the event of system failure or shutdown, the policy server will recover on startup.

Chapter 4

Interface to the Call Control Layer

The policy server interacts with the call control layer when enforcing policies on ongoing communications. This interaction is handled by two interacting parts: a module in the communications architecture and the message handler class of the policy server. The policy server provides a TCP/IP socket for interaction with the call control layer. The information exchanged across this interface is defined by structured strings. An incoming (from the policy server viewpoint) string will be processed by a number of classes before the policy applicator considers the policies that are applicable. The same classes will construct another string as response which needs to be decomposed by the call control module. Figure 4.1 depicts the overview of the message flow in the implemented SIP solution, although other protocols will be handled in a similar fashion.

We will now consider the structure of the strings that are exchanged in some more detail.

4.1 From Call Control Layer to Policy Server

The call control layer will be at some suitable point be enhanced with a module that allows us to observe and control the underlying message flow. For example in a SIP context the module can be placed on a proxy server, in H.323 the Gatekeeper might be the appropriate place. The SIP module extracts the current SIP message and passes it to the policy server, suspending processing until a response has been received.

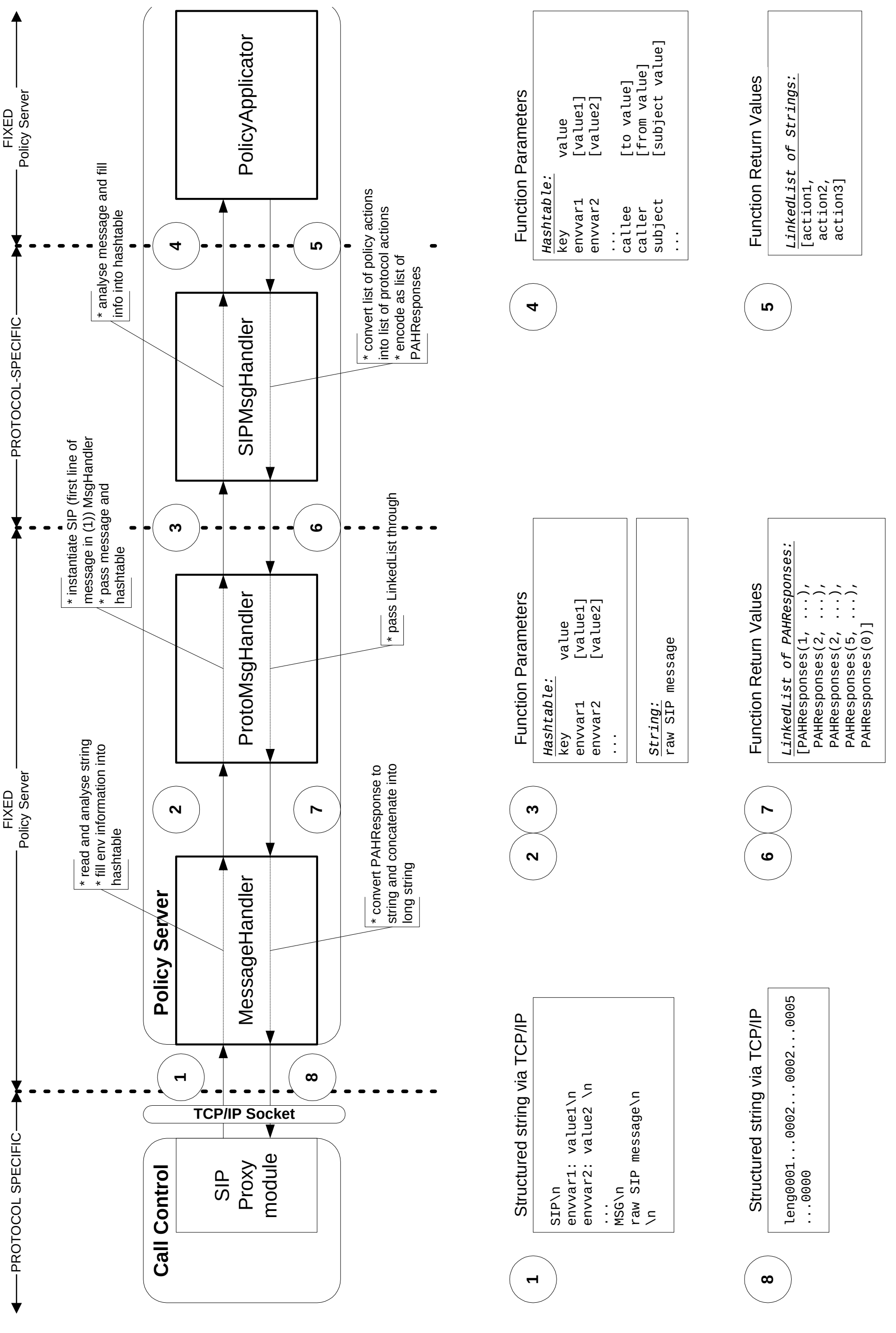
However, rather than just passing the SIP message to the policy server some additional information must be included. This is required to maintain the protocol independence of the policy server. The additional information required includes a protocol identifier and possibly some environment information.

A string passed from the call control to the policy server has the following grammar:

```
<msg>          ::= <protocol>\n<envvarlines>\nMSG\n<message>\n\n
<envvarlines> ::= <envvarline> | <envvarline><envvarlines>
<envvarline>  ::= <envvar>: <value>
<protocol>    ::= SIP | H323 | ...
<message>     ::= a raw sip message | an empty message string | ...
<envvar>      ::= a variable that is understood by the policy applicator
<value>       ::= a suitable value for the given variable
```

For details on *envvar*, please refer to section 6.3. What defines a suitable *value* for an *envvar* should be obvious from the variables. *protocol* can be any string that would be legal as a class name in Java – the reason being that this string will be prefixed to *MessageHandler* to determine the protocol specific handler class to handle communications from this protocol.

Finally, message will be the raw message as passed from the call control layer to the protocol specific message handler for further processing and data extraction. Should all information have been extracted in the call control side and be passed as <*envvarlines*>, the message can possibly be an empty string. If a raw message ends in a double newline, this must be stripped off, as <*msg*> concludes in a double newline.


PROTOCOL SPECIFIC
FIXED Policy Server
PROTOCOL-SPECIFIC
FIXED Policy Server
Call Control
SIP Proxy module
TCP/IP Socket
Policy Server
MessageHandler
ProtoMsgHandler
SIPMsgHandler
PolicyApplicator
1
2
3
4
5
6
7
8
* read and analyse string * fill env information into hashtable
* instantiate SIP (first line of message in (1)) MsgHandler * pass message and hashtable
* analyse message and fill info into hashtable
* convert list of policy actions into list of protocol actions * encode as list of PAHResponses
* pass LinkedList through
* convert PAHResponse to string and concatenate into long string
Structured string via TCP/IP
Function Parameters
Function Parameters
Function Return Values
Function Return Values
Structured string via TCP/IP

```
SIP\n
envvar1: value1\n
envvar2: value2 \n
...
MSG\n
raw SIP message\n
\n
```

```
Hashtable:
key        value
envvar1    [value1]
envvar2    [value2]
...
```

```
String:
raw SIP message
```

```
Hashtable:
key        value
envvar1    [value1]
envvar2    [value2]
...
callee     [to value]
caller     [from value]
subject    [subject value]
...
```

```
LinkedList of Strings:
[action1,
 action2,
 action3]
```

```
LinkedList of PAHResponses:
[PAHResponses(1, ...),
 PAHResponses(2, ...),
 PAHResponses(2, ...),
 PAHResponses(5, ...),
 PAHResponses(0)]
```

```
leng0001...0002...0002...0005
...0000
```

Figure 4.1: Message Flow between Call Control Layer and PolicyApplicator

4.2 From Policy Server to Call Control Layer

The response from the policy server is a single long string, which is obtained in a multi-step process. The policy applicator presents the protocol specific handler with a list of actions in policy terms. The protocol specific handler retrieves the meaning of these actions and creates a list of *PolicyResponse* that matches the list of actions but is specific to the underlying protocol.

The list of *PolicyResponse* is passed through the message handler, which will construct the long string to be sent to the call control layer. Each individual *PolicyResponse* is converted to a string using the *toString()* method of the *PolicyResponse* object. The strings are then concatenated. The exact structure of the strings obtained from a *PolicyResponse* object depends on particulars of the object and is discussed in detail in section 6.5.5.

The final string is prefixed with a four-byte representation of an integer (least significant byte is last) which represents the length of the response in bytes. The long string is finally decomposed in the call control layer module and the actions are committed to the call control.

Chapter 5

Interface to the User Layer

The policy server provides a TCP/IP port for policy management. In this section we discuss the interface that can be used by applications in the user interface layer.

The *UploadHandler* class provides the policy server side of the interface. Essentially communication between the user interface and the policy server is achieved by exchange of messages on the policy upload port (default: port 9998) on the policy server. The messages are structured strings, which are discussed next. For convenience, two implementations of classes (Java and PHP) have been provided to simplify the exchange, but are not discussed here.

Please also refer to section 6.5.1 for details on the *UploadHandler* class.

It has been decided to provide a direct interface between the policy server and the application layer, in contrast to using the underlying communications architecture to manage policies. This decision has been motivated by several key factors:

- independence of underlying protocols: if the underlying protocol were used to manage policies it would be required to support the functionality that is needed. This might be possible for SIP (e.g. by piggy-backing the information on register messages), but can not be seen to hold universally.
- possibility of richer feedback: by providing a custom interface, it is simpler to provide the required information to the policy server but also possible to receive any feedback that is required. This point is in particular important when a policy management operation fails: informative feedback can be provided to the user, and the user has the option to resolve the problem.
- several different interfaces are possible: it is envisioned that user interfaces must be provided depending on the technical skills of users (system administrators might wish for richer possibilities than lay end-users) as well as interfaces for different devices (mobile phone users might prefer voice interfaces, whereas from within an office environment a web interface would be the preferable option).

5.1 Message Structure

The messages between the policy server and the application layer are structured strings, although a future extension could be to use a more standardised protocol such as SOAP.

The current protocol allows for six operations:

- upload policy
- update policy
- delete policy
- enable policy
- disable policy
- query policies

Each of these corresponds to a message in the proprietary protocol. There are two types of messages: requests and responses. The former are sent to the policy server, the latter are returned by the policy server.

A message has the following general structure, where the lines are separated by newline. This implies that any XML included in the commands cannot contain newline characters. A message is terminated by a double newline (meaning a newline character on its own in a line). A subset of the following is used according to the specific request or response:

```
EVENT\n
UNIQUE POLICY ID\n
USER\n
POLICY\n
SUGGESTED SOLUTIONS\n
COMMENT\n
REASON\n
\n
```

To simplify parsing, requests are named in upper case: UPLOAD, UPDATE, ENABLE, DISABLE, DELETE, QUERY. Responses are also given in upper case: SUCCESS, FAIL. Request parameters typically include:

- *polid*: a unique policy identifier, formed by concatenating the address of the owner, a '*' symbol, and the policy identifier:

  ```
  bob@acme.com*reject incoming
  ```

  DELETE and QUERY accept '*' as a policy identifier to mean all tuples belonging to the owner.

- *policy*: the policy information as *one line* of XML; examples for a policy and a variable could be:

  ```
  <?xml version="1.0" encoding="UTF-8"?>
  <policy id="reject incoming" enabled="true" owner="bob@acme.com"
    applies_to="bob@acme.com" changed="2005-02-13T16:40:00">
  ...
  </policy>

  <?xml version="1.0" encoding="UTF-8"?>
  <variable id="wife" value="alice@firm.com" owner="bob@acme.com"
    applies_to="bob@acme.com" changed="2005-02-13T16:40:00"/>
  ```

5.1.1 User Layer to Policy Server

UPLOAD

This command is used to upload a new policy to the server.

```
UPLOAD
polid
policy
```

UPLOAD will succeed if there are no conflicts and the polid has not been used. It will fail otherwise, reporting on the reason for failure and including the problematic policies in the list of policies of the response message.

UPDATE

This command is used to update an existing policy (i.e. for edit).

```
UPDATE
polid
policy
```

UPDATE will succeed if there are no conflicts and the policy in question exist. It will fail otherwise, reporting on the reason for failure and including the problematic policies in the list of policies of the response message.

ENABLE

This command is used to activate an existing policy.

```
ENABLE
polid
```

ENABLE will succeed if there are no conflicts and the policy exists. It will fail otherwise, reporting on the reason for failure and including the problematic policies in the list of policies of the response message.

DISABLE

This command is used to deactivate an existing policy.

```
DISABLE
polid
```

DISABLE will succeed if there are no conflicts arising from the deactivation and the policy exists. It will fail otherwise, reporting on the reason for failure and including the problematic policies in the list of policies of the response message.

DELETE

This command is used to delete an existing policy.

```
DELETE
polid
```

DELETE will succeed if there are no conflicts arising from the deletion and the policy exists. It will fail otherwise, reporting on the reason for failure and including the problematic policies in the list of policies of the response message. If the policy identifier part of *polid* is '*', the effect is to delete all tuples associated with the owner ('policy', 'resolution' or 'variable').

QUERY

This command is used to request an existing policy or variable.

```
QUERY
owner
type
id
```

This gives the owner's username, the type of information required ('policy', 'resolution' or 'variable') and the id of the tuple ('*' to query all tuples, or the id of a specific tuple). If all tuples are queried, only the top level policy/variable elements are returned. In addition, the value of a *long* variable is replaced by a string such as '>>>152' to mean it is 152 Kbytes long. (In the code, *long* is defined by the constant *largeLength*, currently 1024 bytes.) This is to allow a list of large audio clips to be queried efficiently.

QUERY will always succeed. However, there might be a number of responses. Usually it will return an empty policy document or one populated by a number of policies/variables. The former case occurs when no policies/variables exist for the user, the latter when they do.

5.1.2 Policy Server to User Layer

SUCCESS

This response is used to report success to the user. There are two versions: either a one-liner just containing SUCCESS for all requests but QUERY; or a longer version that contains a policy document as response to a QUERY request.

```
SUCCESS
[policydoc]
```

FAIL

This response is used to report failure to the user.

```
FAIL
policydoc
suggestionsdoc
reason
comment
```

- *policydoc* will contain a list of conflicting policies, or if there was a problem related to the id (existence or absence of a policy with this id) just the policy under consideration.
- *suggestionsdoc* will contain a list of possible solutions if there was a conflict. Such solutions could be automatically determined.
- *reason* will contain either the string CONFLICT, the string ID or the string GENERAL FAILURE to identify the nature of the problem. Again, the reason codes will be in upper case. General failure will occur when a command has not been recognised or is incomplete (e.g. an UPLOAD attempt not providing a policy or containing a policy in invalid XML).
- *comment* will contain additional information with respect to the problem. In the case of a conflict, a textual (human readable) description of the conflict is provided.

Chapter 6

Policy Server

6.1 Architecture

The policy server layer consists of policy servers and policy stores as discussed in [11]. Here we are only concerned with the architecture of a policy server.

A policy server is inherently multi-threaded and complex. If we consider the tasks a policy server needs to perform, we can identify key functionalities that are reflected in the design of packages and classes. The structure of the policy server is depicted in Figure 6.1.

Briefly, a policy server needs to handler policy deployment and policy enforcement, possibly a multitude of both requests simultaneously. Hence the core of a policy server provides a mechanism to handle multiple incoming requests of each type: deployment and enforcement. For each of these two fundamental actions we provide a class to handle them: *UploadHandler* and *MessageHandler*.

An upload handler only deals with policies and hence is relatively straightforward: it accepts management requests and performs the associated actions. Part of the upload process is detecting policy conflict. As conflict is a central (and, in enforcement, recurring) problem a special package is provided to handle conflict.

A message handler is more complex. Here we had to consider extensibility in the design. Two obvious extensions are support for further protocols and further policy environment variables. As these two extensions are foreseen, the functionality required for each is again extracted into separate packages. A message handler will instantiate an appropriate message-specific handler to translate message information into policy information. The policy handling, as performed by a policy applicator, is then independent of the underlying protocol.

6.2 Implementation

The policy server is written entirely in Java and consists of six packages. The following sections discuss the individual packages.

The existing classes have dependencies on some standard libraries, all of which are publicly available. The policy store makes use of a tuple space implementation; the chosen one is available from IBM (`http://www.alphaworks.ibm.com/tech/tspaces/download`). Several parts require database access, and the chosen database is MySQL (`http://www.mysql.com/`) together with the MySQL connector library (`http://dev.mysql.com/downloads/connector/j/3.0.html`). SIP messages may be processed with the NIST-SIP parser (`http://dns.antd.nist.gov/proj/iptel/`). TSpaces, NIST-SIP and the developed code itself require access to several other libraries (antlrall, Xerces). The NIST-SIP distribution includes *antlrall.jar*, while *xerces.jar* is available from Apache (`http://www.apache.org`).

The main class is *uk.ac.stir.cs.accent.server.PolicyServer*, which initialises the system and starts all required threads. The other classes in the *server* package are concerned with multi-threading and core functionality of the policy server. Details are discussed in section 6.5. The *utility* package (section 6.8) provides some functions that are independent of the policy server project. The *store* package provides the implementation of the data storage used by all other parts of the system (section 6.6). As policy interactions need to be dealt with, and as this has been foreseen as a major part of the policy server, interaction handling has been separated into the *interaction* package (section 6.4). The remaining two packages, *protocol* (section 6.7) and *environment* (section 6.3), provide parts of

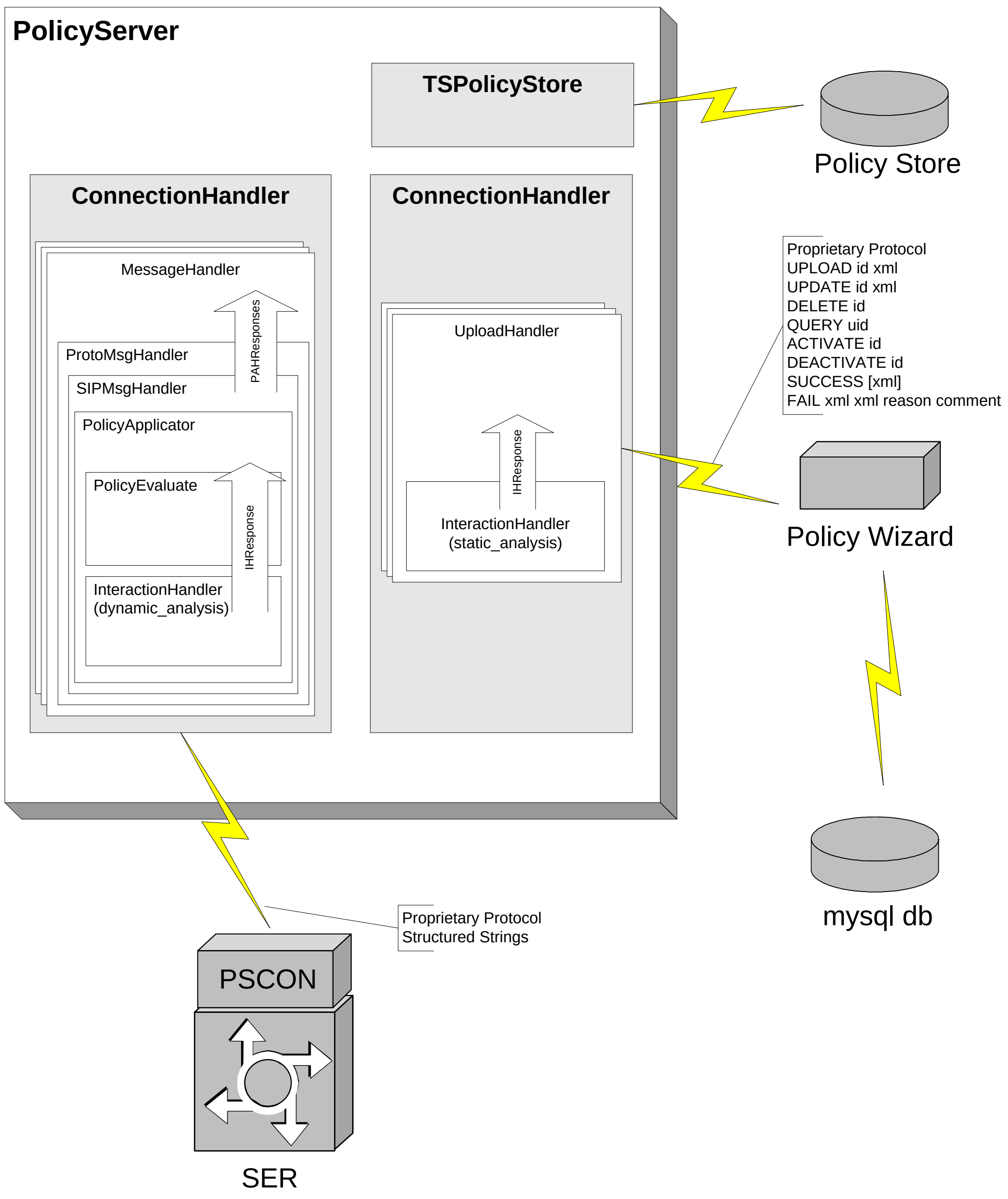


Figure 6.1: Overview of the Policy Server

the system that are core, but dependent on the underlying protocol and the environment variables used in policies respectively. It is these two packages that are most likely to require extensions.

The following sections are aimed at providing a more conceptual overview of the packages and classes. Additionally, the code is documented using JavaDoc.

6.3 The uk.ac.stir.cs.accent.environment Package

6.3.1 General Principles

Each policy refers to a number of concepts such as time, caller or subject. A more complete list of these concerns has been specified in [10]. Each of these concepts will be compared to actual values in a policy body: this forms the policy conditions. A typical example is *time lt 12:00*, stating (informally) that it currently should be before midday.

In this package there is a class for each concept; the classes are derived from the abstract class *Evaluate*. *Evaluate* is implemented by *EvaluateDefault* which provides standard implementations for all comparison operators (*eq*, *ne*, *gt*, *lt*, *ge*, *le*, *in*, *out*) that can be used in policies. The default implementation will simply return false for all operations, ignoring the argument.

An implementation should exist for all operators for each environment variable. For example for time an implementation is provided by *EvaluateTime* which checks the relation of the current time to the value provided by the policy. As an example, *in* tests whether the current time is in a given list. Using the *in* operator, *caller* could be checked against a list of addresses. Not all operators make sense for all environment variables, in which case irrelevant operations may return false. We return to this in Chapter 9.

6.3.2 Currently Defined Environment Variables

The current implementation provides environment variables for the concepts identified as important in [10]. The specific list of environment variables is defined in [13]. Evaluation classes have been provided for most of these. It is important to note that for some classes it is expected that the values are provided by the communications layer or are extracted from the message, while other classes will get the information from databases with context information. For those classes that require information from the message/communications architecture it is essential that values are provided by the protocol-specific message handler in the environment variable hash table indexed by the names provided here. Values in this hash table are generally key–LLinkedListpairs. As the values need to be compared to the values provided in the policies, their types are important. This is also relevant, as the types must be independent of the underlying protocol: a caller identifier in H.323 is simply a number, in SIP it is a SIP URI – the policy will just talk about a general domain entity. Some types are fixed collections.

Some variables must have meaningful values, due to a fundamental role; others can have meaningful values. However, we assume that all variables are initialised. The more variables are instantiated with values, the better the match from the policies will be: in general if a policy expresses a condition based on a specific variable and that variable has no associated value the condition will evaluate to false (and hence the policy might not be applied).

The term 'domain identifier' needs some explanation. Currently we use Internet/email like style for domains: e.g. a specific user can be *xyz@here.com*, but we might wish to talk about the domain, e.g. *here.com*. However, this requires some further investigation, as it might not be suitable to express all the values that we require and the use of an LDAP directory name could prove more fruitful.

There are only two mandatory variables. Note that they are special in that there is no key–LinkedList but rather a key–value pair, and that there is no Evaluate class for them. The mandatory variables are *SERVER_NAME* and *user* with example instantiations *SERVER_NAME: cs.stir.ac.uk* and *user: kjt@cs.stir.ac.uk*.

SERVER_NAME: Domain Identifier, used to determine whether this is an outgoing or incoming communication.

user: Domain Identifier, used to identify the user for whom policies should be retrieved.

'Optional' variables are filled from the message/communications layer; see [13] for the details. (Note again that *triggers* has no *Evaluate* class.) Variables filled from databases or the context information are defined by key–value pairs. Again, see [13] for the details.

Note that the split in the two above classes might not prove appropriate in the future; it merely represents the current implementation. Also, the types stated might not prove to be the best possible choice. However, we would hope that as many concepts as possible can be expressed in a small number of types. This motivated the use of domain identifiers for locations (*4B59@cs.stir.ac.uk* could be a room), roles (e.g. *secretary@cs.stir.ac.uk*) and capabilities. (While one could argue that a Java expert will always be a Java expert, the person might choose not to act as such outside a particular domain.)

Further variables are instantiated in the protocol-specific message handlers for purposes that are specific to the handler, hence they are not discussed here.

6.4 The uk.ac.stir.cs.accent.interaction Package

The interaction handler package provides the classes *InteractionHandler*, and *InteractionResponse* and *ActionType*. *InteractionHandler* is instantiated whenever interactions need to be dealt with. Interaction handlers can be instantiated for the static context (i.e. policy deployment) and dynamic context (i.e. policy enforcement). An interaction handler object provides methods for interaction detection and handling: *staticAnalysis* for static interaction detection and *dynamicAnalysis* for dynamic detection and resolution.

Static analysis is required during policy management to determine whether policies conflict. In particular the purpose of static analysis is to determine that:

1. a new/newly enabled/updated policy of a user is consistent with respect to other policies by that user.
2. a new/newly enabled/updated policy of a user is consistent with respect to other policies imposed on that user by the organisation (as far as they are known).
3. a disabled/deleted policy does not lead to any conflicts between other policies by that user
4. a disabled/deleted policy does not lead to any conflicts between other policies imposed on that user by the organisation (as far as they are known).

In the case that a conflict is detected, it is essential that a description of the conflict and preferably some possible solutions are reported. Static detection is less time-critical than dynamic analysis.

In the current version of the policy server, static detection has not been implemented. Static detection methods from the feature interaction context could be applied, and several such methods have been suggested in [11]. In addition, the algorithm described in [15] could be useful.

Dynamic analysis is more complex: algorithms here are applied at runtime, i.e. while communications messages are actually being routed and call control is performed. Hence, any algorithm must be fast, work on minimal information, detect all possible conflicts, and resolve the detected conflicts. The worst-case resolution is that none of the policies is applied (i.e. all the user's wishes are ignored), but we should be able to do much better than that. Ideas here involve run-time methods from the feature interaction community (see [10, 11]). The policy server architecture allows communication between policy servers which can be used for negotiation in the case of conflicts. The important concept of user preferences (as expressed in the policy language) is also useful to determine the outcome that violates the user's wishes the least.

In the current version of the policy server, dynamic detection has been implemented only for the case of a single server. Language constructs for handling dynamic conflict resolution are defined in [13]. The approach implemented for dynamic analysis is described in [1]. Resolution policies resemble call policies but differ in the following respects:

- The triggers of a resolution policy are the actions of a call policy. The arguments of these triggers are named *variable1* to *variable9* for use in resolution conditions.
- The conditions of a resolution policy resemble those of call policies, but are extended to include *preference1* to *preference9* and *variable1* to *variable9*.
- The actions of a resolution policy resemble those of call policies, but are extended to include various *apply* operators that allow generic resolutions to be defined.

Initially, all the actions resulting from call policies are determined. The policy applicator is then instantiated recursively to check these against the available resolutions. Conflict detection and resolution are effectively defined by the same resolution policy. If there is a conflict, the resolution policy determines the outcome. If resolution does not lead to a single action, an effectively arbitrary choice is enforced. Note that there is just one level of conflict handling: conflicts among resolution policies are not handled. This could be important if resolutions are defined at multiple levels in an organisation.

6.5 The uk.ac.stir.cs.accent.server Package

The *server* package provides the core functionality of the prototype. This package contains the following classes:

- ConnectionHandler
- H323Response
- LogFilter
- MessageHandler
- PolicyApplicator
- PolicyEvaluator
- PolicyResponse
- PolicyServer and PolicyServerNoGUI
- UploadHandler

PolicyServer is the main class. (*PolicyServerNoGUI* is for use on systems that have no GUI, e.g. the Mitel 7000.) On startup, the policy server notes the system environment from the properties file, a policy store object is created, and a connection to the policy store is established. Failing to connect to the policy store will lead to termination. The properties are then saved to the policy store. Information on the supported protocols and the relation between policy and protocol terminology is obtained from a database and stored as hash tables in the policy store. Finally two *ConnectionHandler* objects are created and the corresponding threads started: one will handle the communication with the user interface layer, the other with the communications layer. Each thread will respond to messages received on one of the two ports specified in the properties (default: 9999 for User Interface and 9998 for Communications messages).

A connection attempt to the *ConnectionHandler* will create a further thread. The thread for communication with the user interface simply starts an *UploadHandler* thread for each communication attempt. The *Connection-Handler* for communications events starts a *MessageHandler* thread for each communication. Next we consider the workings and purposes of the *UploadHandler* and the *MessageHandler* in more detail.

We will return to discuss the *PolicyApplicator* and *PolicyEvaluator* classes as well as the class *PolicyResponse* in sections 6.5.3 to 6.5.5.

6.5.1 The UploadHandler Class

The upload handler provides the point of connection between the user interface layer and the policy server. In particular it provides the interface for policy management by the users.

An upload handler reads a message from the input stream, attempts to perform the action requested by the message, and then provides feedback on the output stream before terminating.

Each thread communicates directly with the remote end. To simplify the task of the user interface developers, two classes have been implemented that can be used on the remote end of the communication: one for PHP (*lib_polserver.php*), one for Java (*Connector.java*). These classes are not discussed further here, but see section 5 for more details.

A request from the user interface consists of two or three lines separated by newlines followed by a final double newline character. The first line determines the operation required (either UPLOAD, UPDATE, ENABLE, DISABLE, DELETE or QUERY). The remaining lines contain arguments (policy identifier, user identifier and policy text (XML) as required). Details of the request and response structure has been discussed in section 5.

The upload handler processes each of these requests, and then commits a response to the user interface. The response can be in one of three forms, with lines again separated by newlines.

SUCCESS is sent when the requested operation concluded successfully. This does not apply to the QUERY operation.

SUCCESS XMLDOC is sent when a query operation concludes successfully. In this case *XMLDOC* is an XML document containing the policies applicable to the user.

FAIL XMLDOC1 XMLDOC2 STRING1 STRING2 is sent when the operation failed. In this case more information is provided to allow the user to resolve the issue. *STRING1* contains a reason code (*ID*, *GENERAL_FAILURE* or *CONFLICT*), *STRING2* contains further information documenting the reason. An *ID* reason will occur when the id of the policy does not exist (in case of update en/disable and delete) or exists (in the case of upload). *CONFLICT* occurs if a policy conflict has been detected. *GENERAL_FAILURE* points to any system problems. *XMLDOC1* contains an XML document with the problematic policies, *XMLDOC2* contains an XML document with suggestions for possible solutions. Both XML documents might be empty documents (<*policy_document/*>).

Each operation is implemented by a method. The general operation of the methods is similar as discussed next: the policy store is queried for the existence of a policy with the provided id. This might raise an *ID* failure as discussed before, which will lead to a termination of the operation. If this succeeds, a static interaction handler will be created and the policy will be checked for conflicts. If a conflict occurs, a *CONFLICT* error will be reported. Assuming that no conflict occurs, the operation proceeds. If no system problem occurs, success will be reported, otherwise *GENERAL_FAILURE* is issued to the user interface.

XML Documents are serialised from their DOM tree representation into Strings using the XML serialiser provided by *org.apache.xml.serialize.XMLSerializer*.

6.5.2 The MessageHandler Class

Each message handler thread is responsible for processing a single message passed from the underlying call architecture. A message handler reads a structured text message as discussed in Chapter 4.

The first line of the string is used to identify the corresponding protocol specific message handler (for SIP this would be *SIPMessageHandler*). The following lines contain data that is not part of the raw message, each structured in key–value pairs. The message handler will add values from this to a hash table, hashed by the respective key. This is ended by the text string *MSG* on its own on a line. Everything following is the raw message and will be passed to the specific message handler together with the environment variable hash table.

Message handler then awaits a response from the protocol-specific handler, which is assumed to be a linked list of *PolicyResponse*. This linked list is serialised into a long text string maintaining the order in the list and making use of *PolicyResponse.toString()*. The string is finally sent to the output stream and the thread terminates.

Note: It might be required to make message handlers semi-persistent. Currently a transaction as seen from the message handler is receiving an incoming message, processing it, generating and sending the respective response. However, it might be necessary to extend the scope of a transaction to include the successful processing at the remote end – especially if we wish pre and post negotiation.

6.5.3 The PolicyApplicator Class

The policy applicator applies policies to a message. This applicator is protocol-independent as it purely works on policies. The applicator makes use of a (dynamic) interaction handler to determine any conflicts. The applicator receives a hash table containing environment information as instantiation data. The data in the hash table is extracted by the protocol-specific handler from the message and further information is provided directly by the call architecture. A policy applicator returns an ordered list of actions that need to be performed (the order is first in list, first to be done). Again, this response is protocol-independent.

The hash table containing the environment information passed to the policy applicator requires some further explanation. The table consists of key–value pairs, where the keys are the environment variables as defined in section 6.3, and the values are linked lists (using Java's LinkedList ADT) containing none, one or many values of the appropriate type as was identified in section 6.3. For example, the *triggers* environment variable will provide a list of the triggers such as *[all calls, incoming calls]*; *caller* might just be mapped to a list with a single domain identifier.

This class makes some assumptions about the structure of the XML policy document: the document is assumed to be a policy document as specified by the policy language schema. We assume here that the top-level nodes within a policy document are *policy* nodes that have as one descendant a *policy_rules* node. Anything below the level of a *policy_rules* node is processed by a *PolicyEvaluator* object.

A policy applicator follows a four step algorithm to determine what actions are taken:

1. retrieve applicable policies: This step retrieves policies from the policy store that are enabled and are applicable to the given user or domain. The attributes *enabled* and *applies_to* from a policy element are evaluated to determine this.

2. filter policies based on conditions: however, not all policies retrieved in step 1 will be applicable at all times. This step refines the list of found policies. The given policies are checked to determine whether they are triggered (if they have a trigger) and whether their conditions are satisfied. If a condition is satisfied the *eval* attribute of that condition will be set to true, otherwise it will be set to false. If a policy is triggered (or in fact does not specify a trigger, i.e. is a goal) the *triggered* attribute of each *policy_rules* node will be set to true (or false otherwise).

3. call interaction handling functionality: once we have determined which policies should be applied, they are tested for consistency. Interaction handling is discussed in more detail in section 6.4.

4. produce list of actions to be taken: after the applicable and non-conflicting policies have been identified a response needs to be created. This response is an ordered (first in list means first to be done) list of all the actions that should be performed to satisfy the policies. Note that this list is expressed as actions in policy terminology.

6.5.4 The PolicyEvaluator Class

A policy applicator requires evaluation of policies. The questions here are which policies are triggered, and which conditions are satisfied. We also need to know which actions need to be performed. All these aspects are handled by traversing the DOM (Document Object Model) of the policy XML document.

A *PolicyEvaluator* object provides methods for exactly this traversal. Clearly this means that the code in this class is linked very closely to the underlying XML document structure, with the result that changes to the policy definition (the XML schema) need to be reflected in this class.

The main methods in this class provide answers to the questions above: *hasRulesTriggered* determines for a given *policy_rule* or *policy_rules* node whether the policy rule is triggered. *evalConditions* evaluates the conditions of a *conditions* node. Both return a boolean result, but also set the respective attributes (as described above). Finally *getRulesActions* produce a list of actions to be performed.

Determining the triggering or satisfaction of a condition, as well as the appropriate actions of a policy, is reliant on an understanding of the semantics of the policy language. While we currently do not have a formal semantics (and have not found a reason to have such) an informal discussion is provided in [12].

6.5.5 The PolicyResponse Class

PolicyResponse represents a data type that encapsulates the communication between the protocol specific message handlers and the *MessageHandler*. Any message passed from the message handler to the protocol specific handler will be analysed and policies will be applied to it. The response from the specific handler is a list of actions that should be performed. It is these actions that are encoded using *PolicyResponse*.

The protocol specific handler constructs the list of responses. The message handler only converts the individual responses into strings using the *toString()* method of the *PolicyResponse* object and simply joins the strings maintaining the order of the list. The final string is then sent to the underlying call architecture, where an interface module needs to interpret it and enforce the actions.

This use allows for *PolicyResponse* to be subclassed for use in protocol-specific handlers where *PolicyResponse* is not suitable. For more details see section 8.

The current PolicyResponse type implements 8 cases: they are identified by a type number ranging from 0 to 7 where the type number identifies the action to be taken. The meaning of the types is as follows:

- **type 0**: End of Responses
- **type 1**: Generate Reply
- **type 2**: Generate Request
- **type 3**: Process Original
- **type 4**: Add Header

- **type 5**: Replace Header
- **type 6**: Add Body
- **type 7**: Replace Body

End of responses will always be the last action to be taken. Generate reply and generate request produce new (currently stateless) messages. Adding headers and body simply appends further information to the respective message part, whereas replacing removes existing information and places new information at the same place (if empty strings are provided as new information, this simply removes data). Each of the operations requires a number of arguments, as determined by the appropriate constructors for PolicyResponse.

It is worthwhile considering the *toString()* method, as the result is dependent on the type of the PolicyResponse. Let us consider each type in turn:

- **type 0**: the four byte string 0x0000 is constructed.
- **type 1**: a reply consists of a reply code, and reply text, the response string will be as follows: the type (0x0001) followed by a 4 byte representation of the reply code, followed by a four byte representation of the length of the response string followed by the actual response string.
- **type 2**: a request consists of 6 text fields; the string representation will have the type encoded (0x0002) followed by six 4-byte integer and text pairs, where each integer encodes the length of the following string.
- **type 3**: the four byte string 0x0003 is constructed.
- **type 4**: this type specifies the header as text, hence the response string is 0x0004 followed by a 4-byte integer representing the text length and the actual text.
- **type 5**: this type specifies the new header and the old header as text, hence the response string is 0x0004 followed by two 4-byte integer and text pairs representing the text length and the actual text of the two headers.
- **type 6**: same as add header, apart from the type code which is 0x0006.
- **type 7**: same as replace header, apart from the type code which is 0x0007.

Note: Reply and request might be required to be stateful. How this can be handled and whether it is required must be explored. It then also will be an issue to determine whether state must be maintained in the call processing architecture or in the policy server.

6.6 The uk.ac.stir.cs.accent.store Package

The policy server requires access to data storage for several purposes. Foremost, a place to store user policies is required. However, parts of the system require access to configuration data as well as quick access to values that are stored in a database. This package provides an interface to all the storage functionality that is required. The interface *StoreInterface* has been implemented by *StoreTuples*, providing the functionality required using the IBM Tuple Space. The *PolicyStore* class extends the interface and implementation with methods of general utility.

The methods provided by the interface can be grouped into several categories: methods to place into and remove environment information from the store (*getHash*, *getProperties*, *putHash*, *putProtocols*, *putProperties*), methods for more global functionality (*emptyStore*, *serverShutdown*), as well as methods operating on policies in the store (*addTuple*, *addTuples*, *changeTuple*, *deleteTuple*, *enableTuple*, *existingTuple*, *readTuples*).

The first group allows to place system-wide environment information into the policy store and retrieve the same. The second group is concerned with completely clearing the store (this removes all stored policies and environment information) or removing information that should only be maintained while the server is running (i.e. removing environment information, but not stored policies). The last group is the most important, as methods in this group are used when policies are uploaded, modified or deleted from the store, as well as when they must be retrieved to be applied.

The reference implementation for the policy store has been based on TSpaces, a tuple space implementation from IBM. It is justified to ask the question why TSpaces has been chosen. We will attempt to answer this in the next few paragraphs, and present alternatives and thoughts.

Early on in the project we had a suggestion that a tuple space might be as suitable storage for policies, partly based on the fact that it is possible to have a fast hardware implementation of a tuple space. Tuple spaces are based on Linda ([2, 3]) and essentially provide a distributed communications space with in and out operations. This model is suitable for policies; they can be stored into the tuple space and they can be read from the tuple space.

Also, tuple spaces store tuples, which would serve for a somewhat flexible structure of the stored elements: that is some tuples could have three elements while others could have six. This would provide a certain flexibility. However, for retrieval the structure needs to be known, as tuples are retrieved by providing a template tuple that has the same structure and matches some or all elements directly or by placeholders.

As we are using Java, two prominent implementations stood out in 2002: TSpaces and JavaSpaces. For our purpose it was important that a template returns all matching tuples, not just one. This is not supported by JavaSpaces and hence it did not seem a suitable choice. Hence we decided on TSpaces, which also supports storage and retrieval of XML in a straightforward fashion: A special XML Tuple can be created from XML documents and stored easily; retrieval is via a subset of XQL (XML Query Language).

Note: It might be worthwhile to investigate XML databases and migrate the policy store to an XML database. Such a migration should be straightforward, as the functionality is encapsulated by an interface. Hence the changes to the policy server code would be minimal (essentially the PolicyServer class requires the policy store variable to be of a different type).

6.7 The uk.ac.stir.cs.accent.protocol Package

The *MessageHandler* class in the *server* package instantiates a *ProtocolMessageHandler* whenever a message needs to be processed, and then calls the *handleMessage* method on the newly created instance. An instance of *ProtocolMessageHandler* essentially deals with the protocol of the underlying message, and dynamically instantiates the appropriate protocol-specific handler.

Every protocol specific handler object is derived from the abstract class *GenericProtocolHandler* and should be included in the *protocol* package. Several implementations of the abstract class are included in the standard distribution. *DefaultMessageHandler* will be chosen by the system if no specific handler can be found.

6.7.1 The SIPMessageHandler Class

The *SIPMessageHandler* class is designed to interface with a module specially written for the Fraunhofer Institute SIP Express Router (SER). The SIP message handler performs operations that are specific to SIP messages. In particular, it parses SIP messages and prepares data for the policy applicator. The response from the policy applicator is converted back into SIP terminology by this class. This class should be seen as a reference implementation for developers who are extending the policy server to include further protocols.

The initialisation of this class sets the local reference to the store and then loads the SIP-related terminology mappings: the mapping from policy to SIP actions, and the mapping from SIP events and conditions into their respective policy counterparts.

The class then parses the SIP message. The parsed structure is analysed and the hash table (as received from the message handler) is completed with the information that was extracted from the message – making use of the events and condition mapping. At this point the hash table contains only information in policy terminology ready for processing by a policy applicator – hence a policy applicator is instantiated, and its *applyPolicies* method returns a list of actions to be taken. The final step is to convert the list of policy actions into actions at SIP level (making use of the action mapping).

Note: While this is suitable for SIP, developers for other protocols might decide to parse the message differently. We will discuss this in more detail in section 8, which is a 'must read' for developers intending to add support for other protocols.

6.7.2 The H323MessageHandler Class

The *H323MessageHandler* class deals with messages received from an H.323 gatekeeper (GNU GK). Responses to these messages are handled by the *H323MessageResponse* class. The details of H.323 message handling are not discussed further here, since they are described in a Technical Report [5] and in a journal article [6].

6.7.3 The Mt7000MessageHandler Class

The policy system interface module is located in package *uk.ac.stir.cs.accent.mt7000* containing the following classes:

- *Mt7000Interface* is the main class that sends call messages to the policy server and applies the responses it receives in return.
- The *Policy7000* class handles communication with the policy server. It maps between policy server user addresses and Mitel 7000 extension numbers using the *interface.properties* configuration file.
- The *Mt7000Command* class encapsulates commands to the Mitel 7000; it corresponds to the same class in the policy server.
- The *ThirdPartyCC* class supports third-party call control of the Mitel 7000. It has been adapted and extended with the permission of MKC Networks as the copyright holder.
- The *Constants* class defines key Mitel 7000 values. It is used with permission of MKC Networks as the copyright holder.
- The *Base64Coder* class is an adaptation of open source code by Christian d'Heureuse (`http://www.source-code.biz`). It converts audio clips to/from Base64 format encoding.

The *ServiceListener* class needs to be installed in */usr/share/telephony/com/mitelknowledge/b2bua*. This class has been adapted with the permission of MKC Networks as the copyright holder.

The script *pi-start* is provided to start the policy interface on the Mitel 7000. So it can be compiled (though not run) outside the Mitel 7000, a number of required jar files are provided in the *lib* directory. On a Mitel 7000, the code accesses jar and classes files in the */usr/share/telephony* directory.

The *pi-start* script can be edited to use anything other than the default configuration file. The main class *uk.ac.stir.cs.accent.mt7000.Mt7000Interface* takes no command-line parameters.

The Mitel 7000 package is configured by a Java properties file *interface.properties* in the root directory of the code. The exact format is not critical. The following example is for the interfacing code and the policy server running on a Mitel 7000 as the local host.

```
# General properties

attendant.first         8800
attendant.last          8809

policy.host             localhost
policy.port             9998

# Phone numbers and user addresses

2000                    anne
2001                    bert
2002                    cath
2003                    dave
```

attendant.*: These entries define the first and last autoattendant numbers used by the policy code for playing audio clips. The Mitel 7000 does not directly support playing audio clips in the current system version (3.0 SP2). *The play_clip action cannot be followed by any other in the current implementation.*

Playing an audio clip is achieved by redirecting a call to an autoattendant. On the Mitel 7000, autoattendant files are stored in directories like */opt/mksip/prompts/SAM/8800*. Separate within-hours and out-of-hours prompts may be stored in the files *welcomeDay.wav* and *welcomeNight.wav*.

The autoattendant pool is used in cyclic order, storing the audio clip to be played in the next autoattendant to be used. In the above example, autoattendants 8800 to 8809 are used in sequence and then cycle back to 8800. In principle, several autoattendants may be used concurrently if there are concurrent *play_clip* actions. However, it has not been yet checked if this might cause interference problems.

Messages may be recorded for other autoattendants for use with the *reject_call* action. These autoattendant numbers may then be cited when this action is defined. For example, *reject_call(8810)* may used for a general 'you could not be connected' audio response. Other, more specific, responses could also be defined (e.g. to report policy conflicts).

policy.*: These entries define the policy server host and port number. In the above example, the policy server is also running on the Mitel 7000 but could in principle be located elsewhere.

extensions: The Mitel 7000 uses extension numbers internally to identify users. These are mapped by the configuration file to usernames known to the policy system. It is assumed that the domain name of the Mitel 7000 is the same as that of the users. For example if the Mitel 7000 has hostname *pbx.cs.stir.ac.uk*, then extension 2000 corresponds to user *anne@cs.stir.ac.uk*. The interface code also performs the inverse mapping, e.g. user *anne@cs.stir.ac.uk* to extension 2000.

Database entries used by the Mitel 7000 in database *mksip* need to be set up manually as follows. The file *lib/mt7000_setup.sql* contains the following code. (B2BUA means Back-to-Back User Agent.) *Check first whether these entries are already present in the tables otherwise duplicates will be created.*

```
INSERT INTO PubSubConfig SET
  id='B2BUA',
  comm_id='POLSERVICE',
  host='localhost',
  port=16682,
  is_server=0,
  transport_strategy='com.mitelknowledge.messaging.MessageComm',
  serialization_strategy='java',
  description='Policy Server'
;

INSERT INTO PubSubConfig SET
  id='POLSERVICE',
  comm_id='B2BUA',
  host='localhost',
  port=16682,
  is_server=1,
  transport_strategy='com.mitelknowledge.messaging.MessageComm',
  serialization_strategy='java',
  description='Policy Server'
;

INSERT INTO ServiceTriggers SET
  id='3PCC',
  orig_call_delivery=2,
  orig_failed=2,
  orig_disconnected=2,
  term_call_delivery=2,
  term_failed=2,
  term_disconnected=2,
;

INSERT INTO TriggerSub SET
  id='3PCC',
  topic='mkcnetworks.com/trigger',
  priority=1,
  enabled=1,
  description='Third Party Call Control'
;
```

This creates publish/subscribe entries in the *PubSubConfig* table for the policy interface. Then subscriptions for particular triggers are created in the *ServiceTriggers* table. (In a trigger subscription, '0' means no action, '1' means notification only, and '2' means that call processing is blocked while the trigger is analysed.) The service identifier *POLSERV* is matched by the policy interface code. After any changes in trigger settings, it is necessary to restart the Mitel 7000 B2BUA with:

```
cd /usr/share/telephony/com/mitelknowledge/watchdog

WatchdogManager.sh restart:B2BUA
```

Certain Mitel 7000 services may conflict with the policy system interface if they block (value '2') on triggers used by the policy system. For example, bandwidth capping is known to interfere. In the current Mitel 7000 version, such a service must be disabled. For example, the relevant trigger subscription can be disabled with the following (also given in *lib/mt7000_setup.sql*):

```
  UPDATE TriggerSub SET
    enabled=0
    WHERE id='BANDWIDTH'
  ;
```

6.7.4 The ContextHandler Class

The *ContextHandler* class is a pseudo protocol handler; it does not implement *GenericProtocolHandler* because of the significantly different design. The *PolicyStore* class registers with the policy store for writes to *presence* and *availability* variables. When one of these changes, it causes *ContextHandler* to be instantiated and run in a separate thread. *ContextHandler* retrieves and applies matching policies for the given user, variable and value.

6.8 The uk.ac.stir.cs.accent.utility Package

This package contains two classes, *Now* and *Utilities*. The former provides a convenient way to obtain the current date/time information. The latter provides a function that converts integers into 4-byte bit patterns, where the last bit contains the LSB of the original integer. This is used when integers are exported as part of structured strings when communicating with the underlying call control architecture. The *Utilities* class also contains a *replaceAll* function to replace occurrences of patterns in a String with new values. Note that such a function exists in the Java library but does not work correctly in version 1.4.1 (in particular $ is not escaped properly); the method provided here fixes the problem.

Chapter 7

A Worked Example based on SIP

Consider again Figure 4.1 where we have shown a sample of the message flow between the call control layer and the policy applicator. In this section we enhance on this by providing a concrete example, based on a SIP message. Numbers in the following correspond to the numbers in Figure 4.1

1. Data passed from SIP Proxy Module to MessageHandler The message composed and sent by the SIP Proxy Module (pscon for SER) is as follows:

```
SIP\n
SERVER_NAME: d254196.cs.stir.ac.uk\n
MSG\n
INVITE sip:cs.stir.ac.uk SIP/2.0\n
Via: SIP/2.0/UDP 139.153.254.196:5062\n
CSeq: 3732 REGISTER\n
To: "Ken" <sip:kjt@cs.stir.ac.uk>\n
Expires: 900\n
From: "Jianxiong" <sip:jxp@cs.stir.ac.uk>\n
Call-ID: 1815056773@139.153.254.196\n
Content-Length: 0\n
User-Agent: KPhone/2.11\n
Event: registration\n
Allow-Events: presence\n
Contact:"root"<sip:root\@139.153.254.196:5062;transport=udp>;
methods="INVITE, MESSAGE, INFO, SUBSCRIBE, OPTIONS, BYE, CANCEL, NOTIFY, ACK"\n
\n
```

2. Data passed from MessageHandler to ProtocolMessageHandler The message handler processes this message by:

- extracting SIP as the protocol identifier
- inserting (SERVER_NAME:d254196.cs.stir.ac.uk) into the hash table
- collating the rest of the input (i.e. the SIP message) into a message string.

The message handler then instantiates a *ProtocolMessageHandler* and initialises it with the fact that we have a SIP message. It finally calls the *handleMessage()* method of the *ProtocolMessageHandler* and passes the message string and the hash table as arguments.

3. Data passed from ProtocolMessageHandler to SIPMessageHandler The *ProtocolMessageHandler* simply instantiates a SIPMessageHandler and calls the *handleMessage()* method of the SIP message handler with the message string and the hash table as parameters.

4. Data passed from SIPMessageHandler to PolicyApplicator The SIPMessageHandler has the capability of parsing and interpreting SIP messages. On receipt of the data it will parse the message string using the NIST SIP parser, and then extract information from the parsed message structure and insert the respective values into the hash table. In particular the following data is inserted (values in square brackets represent lists; the third column is a guide to comments in this document and the data therein is not inserted into the hash table):

caller	[*"kjt@cs.stir.ac.uk"*]	**3*
callee	[*"kjt@cs.stir.ac.uk"*]	**4*
call_content	[*""*]	**6*
call_type	[*""*]	**6*
event	[*""*]	**6*
forward_target	[*""*]	**6*
topic	[*""*]	**5*
device	[*""*]	**6*
media	[*""*]	**6*
quality	[*""*]	**6*
triggers	[*"registration attempt", "outgoing registration attempt"*]	**7*
user	*"kjt@cs.stir.ac.uk"*	**2*
from	*"Ken" <sip:kjt@cs.stir.ac.uk>*	**1*
to	*"Ken" <sip:kjt@cs.stir.ac.uk>*	**1*
call_id		**1*

Comments to above table:

**1*: not required by PolicyApplicator; these values are used to complete responses later on.

**2*: CRUCIAL: is used for retrieval of policies; depending on the direction of the message flow this is either caller (outgoing message) or callee (incoming message).

**3*: extracted from TO field.

**4*: extracted from FROM field.

**5*: extracted from SUBJECT field if one exists.

**6*: data for these fields is currently not extracted from the message.

**7*: the trigger is extracted from the method (first line of the SIP message). Trigger will always contain the method as well as a directional version of the method. The actual value that is inserted into the table is extracted from the terminology mapping: for example here 'REGISTER' and 'REGISTERout' are queried in the database and the terms 'registration' and 'outgoing registration attempt' are found to be the respective policy terms. For INVITE this could be 'outgoing call attempt' and 'call attempt'. The full list of terms available for policies is documented in [12].

Finally the SIPMessageHandler instantiates a PolicyApplicator, initialises the same with the hash-table, and calls the applyPolicies() method.

5. Data passed from PolicyApplicator to SIPMessageHandler Assume that a policy exists which states that 'kjt' cannot register here, with the respective policy action 'register not allowed'. This is in policy terms and again the policy language description in [12] describes possible values. There are no further actions to be taken.

Hence the SIP message handler receives a one element list: *["register not allowed"]* for processing.

6. Data passed from SIPMessageHandler to ProtocolMessageHandler The SIP message handler considers all elements in the response list from the policy applicator (here only one) and determines what the actions mean in SIP terms. The specified SIP action here, as extracted from the terminology mapping is: *reply:408:you are not allowed to register here;end of responses* which is encapsulated in a list containing two *PolicyResponse* objects which are passed to the protocol message handler:

```
PolicyResponse r1 =
  new PolicyResponse(1, 408, "you are not allowed to register here");
PolicyResponse r2 = new PolicyResponse(0);
LinkedList l = new LinkedList();
l.add(r1);
l.add(r2);
return l;
```

7. Data passed from ProtocolMessageHandler to MessageHandler The protocol message handler simply passes the response to the message handler; it does not perform any processing on the response.

8. Data passed from MessageHandler to SIP Proxy Module The message handler cycles through the list of PolicyResponse that it has received from the ProtocolMessageHandler; in this case the list contains two elements. It uses the *toString()* method of the PAH response object to serialise the data. The result will be a long string, that is passed to the Proxy Server module. The string for the given example is as follows (hex dump, as integers are encoded):

```
(type 1)      (408)        (36: length of text)
 00 00 00 01  00 00 01 98  00 00 00 24
(y  o  u     a  r  e     n  o  t     a  l  l  o  w  e  d )
 79 6F 75 32 61 72 65 32 6E 6F 74 32 61 6C 6C 6F 77 65 64
(t  o     r  e  g  i  s  t  e  r     h  e  r  e)
 74 6F 32 72 65 67 69 73 74 65 72 32 68 65 72 65
(type 0)
 00 00 00 00
```

Chapter 8

Addition of Further Protocols

A protocol-specific handler is required for every protocol type that should be supported. This handler creates the link between the protocol and the policy architecture. Essentially the task of the handler is to interpret the message as received from the communications architecture and provide information extracted from the same in terms of policy terminology. The handler then makes use of a *PolicyApplicator* to determine which requirements are placed by policies. The response from the policy applicator is a list of policy actions. The final task of the message-specific handler is to convert this list into a list of *PolicyResponse*, ready to be communicated to the underlying call architecture.

As example, consider the SIP protocol. A SIP message is parsed and the gathered information such as to, from, subject, call_id, method, etc. is stored in the hash table together with the corresponding values: e.g. to will be stored as 'callee' with the content of the 'to' field. Method will be stored as 'trigger' with the corresponding value. After the PolicyApplicator has finished, we receive a list of actions, such as *add_medium(medium)* or *forward_to(address)*. These are converted back into SIP-specific *PolicyResponse*, e.g. *add_medium(medium)* will result in a 'create new request' that will be an invite requesting the extra media; *forward_to(address)* will result in a 'create response' that will be a 4xx response with the new location. The actual mapping between policy and protocol terminology is obtained from the Accent database (the values from the database are stored into hash-tables at server startup and placed in the tuple space).

Figure 8.1 depicts an alternative message flow (compared to that in Figure 4.1). While the figure on a superficial glance might look virtually the same there are subtle differences. The main difference lies in the details of the messages, and here actually in that the message in (1) already contains all detail (as it is parsed in the back-end (GK module)) and that the H323MessageHandler is only passing the information to the policy applicator, rather than parsing the message and completing the data (as is the case with SER).

8.1 Adding a new handler class

To add a new handler class all that is required is a new class following the template in section 8.1.2. This class will interface between the core of the policy server and the policy applicator, as discussed in the overview. The class is placed into the *uk.ac.stir.cs.accent.protocol* package. The class might make use of terminology mappings, which need to be defined in the database. In general this class will also interface to a module in the call control layer (e.g. the *pscon.c* handler for SER or the H323 GK handler in Figure 8.1) which clearly needs to be provided, albeit outside the policy server and hence not discussed here (but Chapter 4 should give some insight).

Important Naming Convention: The new specific protocol handler class must be named according to the following convention: *Protocol*MessageHandler where *Protocol* is a unique protocol identifier. Note that this same protocol identifier must be used when messages are sent to the message handler as discussed in 4.

8.1.1 Adding to The Policy Database

The database has a table called 'terminology_mapping'. This table provides the mapping between policy terminology and protocol terminology and has been discussed before (section 3.1).

A typical entry in this table specifies a protocol, the policy term, the protocol term and information whether the term is an action or event/condition. Below are two examples for SIP:

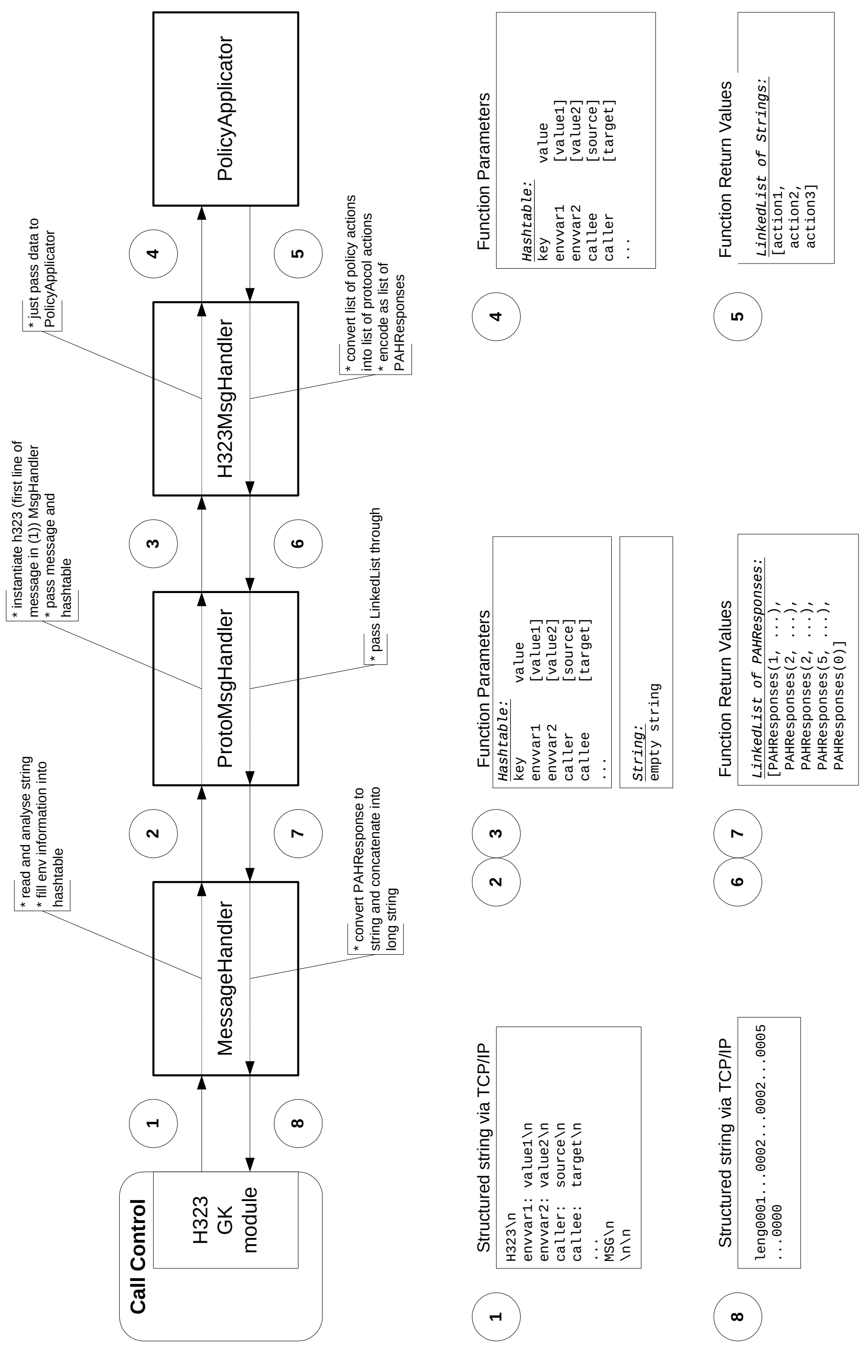


Figure 8.1: Alternative Message Flow for Specific Protocol Handlers

```
  INSERT INTO terminology_mapping VALUES
    (NULL, 'SIP','disconnect','BYE',3,'no');
  INSERT INTO terminology_mapping VALUES
    (NULL,'SIP','reject_call','1_:_408_:_I am not available','yes');
```

8.1.2 Class Outline

```
package uk.ac.stir.cs.accent.protocol;

// PolicyApplicator provides the objects that apply policies
import uk.ac.stir.cs.accent.server.PolicyApplicator;

// we need to know the used policy store
import uk.ac.stir.cs.accent.store.*;

// PolicyResponse is used to report back to the MessageHandler
import uk.ac.stir.cs.accent.server.PolicyResponse;

import java.util.*;

// import any other packages: e.g. message parser

public class MyMessageHandler extends GenericProtocolHandler {
  private PolicyStore policyStore = null;
  // any other variables

  // implement initialise() abstract method
  public void initialise(PolicyStore policyStore) {
    ps = polstore;
    // any other initialisation
  }

  // implement handleMessage() abstract method
  public LinkedList handleMessage(String message, Hashtable environmentHash) {
    // parse message

    // fill hash table with values from message (e.g. caller, callee, triggers)

    // get a PolicyApplicator Object
    PolicyApplicator policyApplicator =
      new PolicyApplicator(policyStore, environmentHash);

    // find out what actions the policy want to achieve
    LinkedList responses = policyApplicator.applyPolicies();

    // convert responses (LinkedList of actions) to protocol-specific messages
    // encapsulate result in a LinkedList of PolicyResponse values
    LinkedList result = convert(responses);
    return result;
  }

  // any other functions, e.g. convert()
}
```

8.2 Adapting PolicyResponse

If the provided *PolicyResponse* class 6.5.5 is unsuitable for the protocol to be added, it can be subclassed, and the subclass can be used in the specific protocol handler.

However, there are a few things to watch out for: the message handler might create responses of the original *PolicyResponse* class (only subtypes 0 and 3) which hence must be handled by the call control layer interface in addition to the new response types in the user-defined subclass.

The subclass must define a method *toString()* which encodes *PolicyResponse* objects in a way understood by the interface to the call control layer. The message handler uses this method to serialise *PolicyResponse* (and all subclasses thereof).

However, these restrictions are minimal, as this extensibility allows developers to produce any (text) response they wish without the core of the system having to be changed.

Chapter 9

Addition of Further Environment Variables

We have discussed the currently defined environment variables in section 6.3. However, it is envisioned that new policies might make use of further environment variables. If additional variables are required, the comparison operations on these variables need to be defined. *Evaluate* classes serve the purpose to implement the comparison operators in a meaningful way for a given environment variable.

We have discussed some of the implemented classes earlier in section 6.3. The next section provides an outline for a typical *Evaluate* sub-class.

Important Naming Convention: The new environment evaluate class must be named according to the following convention: Evaluate*Var* where *var* is a unique environment variable identifier as used in the policy language. The identifier is used when values are retrieved during a policy evaluation process. In determining the class name, the variable is split at '_' and each part is given an initial capital. For example, *caller_id* corresponds to *EvaluateCallerId*.

9.1 Class Outline

The following example assumes an environment variable called *my_var*.

```
package uk.ac.stir.cs.accent.environment;
import java.util.*;

public class EvaluateMyVar extends Evaluate {
  // hash table for current environment
  Hashtable environmentHash = null;

  // specific variables
  LinkedList myVar = null;

  // iterator over values
  ListIterator iterator = null;

  // local variables as required

  // initialisation method called dynamically in place of a constructor
  public void initialise(Hashtable hashTable){
    environmentHash = hashTable;
    myVar = (LinkedList) env.get("my_var");
    // get an iterator for this linked list
    iterator = myVar.listIterator;
  }

  // implementation of all abstract methods of Evaluate
  public boolean eq(String value) {
     // implementation of eq operation, assuming myvar is a list of strings
     boolean result = false;
     while (iterator.hasNext()) {
       String element = (String) iterator.next();
       if (element.equals(value))
          result = true;
     }
     return result;
  }

  public boolean ne(String v) {
    ...
  }

  public boolean lt(String v) {
    ...
  }

  public boolean le(String v) {
    ...
  }

  public boolean gt(String v) {
    ...
  }

  public boolean ge(String v) {
    ...
  }

  public boolean in(String v) {
    ...
  }

  public boolean out(String v) {
    ...
  }
}
```

Chapter 10

Further Ideas

In the context of a widening acceptance of XML-based data exchange protocols, it would seem sensible to replace the current proprietary interface between the policy server and the call control layer with a more standard interface. SOAP would be one protocol that might be useful in this context. Should such a change be required there are essentially two classes that must be adapted: *PolicyResponse* and *MessageHandler*. In particular, the *toString()* method in *PolicyResponse* must return XML messages and the *MessageHandler.run()* method must be adapted to interpret SOAP messages that it receives (i.e. the receiving loop must be adapted). Also, the *MessageHandler.createRespString()* method must combine the *PolicyResponse* XML into a correct SOAP message.

The same issue applies to the interface between the policy server and the user interface layer. Again SOAP would be a suitable protocol. Here the changes required are restricted to the *UploadHandler* class, where again the *run()* method needs to be adapted to handle incoming and produce outgoing SOAP messages.

The conflict detection and resolution techniques require further development. For details please refer to the suggestions in section 6.4.

In section 6.3 we have discussed a number of environment variables and stated that implementations have been provided for most of the classes. However, several implementations are simplistic and need to be refined. The limitations and design decisions for the individual classes goes beyond the scope of this document, but has been indicated in the respective classes.

References

[1] Lynne Blair and Kenneth J. Turner. Handling policy conflicts in call control. In Stephan Reiff-Marganiec and Mark D. Ryan, editors, *Proc. 8th. Feature Interactions in Telecommunications and Software Systems*, pages 39–57. IOS Press, Amsterdam, Netherlands, June 2005.

[2] D. Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, pages 80–112, January 1985.

[3] D. Gelernter and N. Carriero. Coordination languages and their significance. *Communications of the ACM*, pages 97–107, February 1992.

[4] M. Handley, H. Schulzrinne, E. Schooler, and J. Rosenberg. SIP: Session initiation protocol. *Request for Comments 3261*, June 2002.

[5] Tingxue Huang. Policies for H.323 internet telephony. Technical Report CSM-165, Department of Computing Science and Mathematics, University of Stirling, UK, May 2005.

[6] Tingxue Huang and Kenneth J. Turner. Policy support for H.323 call handling. *Computer Standards and Interfaces*, 28(2):204–217, November 2005.

[7] JAIN. Java API's for Integrated Networks homepage
`http://java.sun.com/products/jain`. Visited on 10-12-2001.

[8] J. Lennox and H. Schulzrinne. CPL: A language for user control of internet telephony services. *IETF Internet Draft 06*, January 2002.

[9] Parlay. The Parlay API,
`http://www.parlay.org`. Visited on 10-12-2001.

[10] Stephan Reiff-Marganiec and Kenneth J. Turner. Use of logic to describe enhanced communications services. In Doron A. Peled and Moshe Y. Vardi, editors, *Proc. Formal Techniques for Networked and Distributed Systems (FORTE XV)*, number 2529 in Lecture Notes in Computer Science, pages 130–145. Springer, Berlin, Germany, November 2002.

[11] Stephan Reiff-Marganiec and Kenneth J. Turner. A policy architecture for enhancing and controlling features. In Daniel Amyot and Luigi Logrippo, editors, *Proc. 7th. Feature Interactions in Telecommunications and Software Systems*, pages 239–246. IOS Press, Amsterdam, Netherlands, June 2003.

[12] Stephan Reiff-Marganiec and Kenneth J. Turner. Feature interaction in policies. *Computer Networks*, 45(5):569–584, August 2004.

[13] Stephan Reiff-Marganiec, Kenneth J. Turner, and Lynne Blair. APPEL: The ACCENT project policy environment/language. Technical Report CSM-161, Department of Computing Science and Mathematics, University of Stirling, UK, December 2005.

[14] Kenneth J. Turner. The ACCENT policy wizard. Technical Report CSM-166, Department of Computing Science and Mathematics, University of Stirling, UK, December 2005.

[15] D. Wang, R. Hao, and D. Lee. Fault detection in rule-based software systems. In *Concordia Prestigious Workshop on Communication Software Engineering*, September 2001.