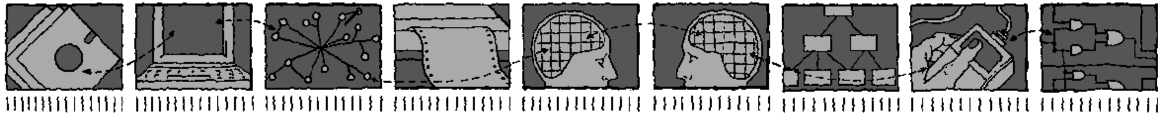
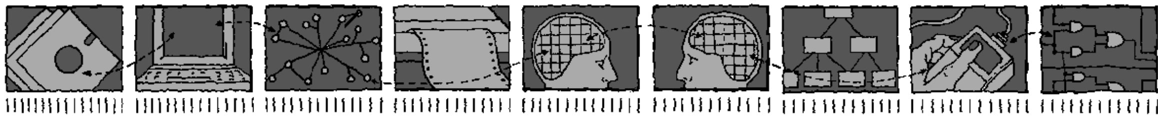


Department of Computing Science and Mathematics
University of Stirling



The ACCENT Policy Server

Stephan Reiff-Marganiec, Kenneth J. Turner, Lynne Blair and Feng Wang

Technical Report CSM-164

ISSN 1460-9673

August 2013

Department of Computing Science and Mathematics
University of Stirling

The ACCENT Policy Server

Stephan Reiff-Marganiec, Kenneth J. Turner, Lynne Blair and Feng Wang

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

Telephone +44 1786 467 421, Facsimile +44 1786 464 551
Email srm13@le.ac.uk, kjt@cs.stir.ac.uk, lb@comp.lancs.ac.uk

Technical Report CSM-164

ISSN 1460-9673

August 2013

Abstract

The ACCENT project (Advanced Call Control Enhancing Networked Systems, www.cs.stir.ac.uk/accent) was concerned with developing a practical and comprehensive policy language for call control. The project 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. The approach was subsequently extended for home care and telecare on the MATCH project (Mobilising Advanced Technologies for Care at Home, www.match-project.org.uk) and for sensor networks and wind farms on the PROSEN project (Proactive Control of Sensor Networks, www.cs.stir.ac.uk/~kjt/research/prosen).

This document focuses on the policy server layer of the architecture. The 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 policy server, 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, 6.8.2 and 6.8.3).

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 6.8.5). 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 6.8.3).

Relative to the version of May 2005, this Technical Report has been updated as follows:

- Minor editorial revisions have been made.
- Technical revisions to the Mitel 7000 Interface have been made in section 6.8.3.

Relative to the version of December 2005, this Technical Report has been updated as follows:

- The abstract and section 1 have been updated to mention new application domains.
- Section 6.8.3 has been slightly updated to reflect the current Mitel 7000 version.

Relative to the version of April 2009, this Technical Report has been updated as follows:

- The whole document has been substantially revised to bring it up to date, particularly with respect to new application domains (home care, sensor networks), embedding in OSGi, and support for goals.

Relative to the version of July 2011, this Technical Report has been updated as follows:

- The property *server.log.lines* has been added in section 3.3 to define how many lines should be preserved in a log.
- Section 4.2 now describes how system variables are set for OSGi on a *device_in* trigger.

Relative to the version of September 2012, this Technical Report has been updated as follows:

- Section 6.8.4 has been added to describe the query protocol used to interrogate the policy activation history.
- Section 6.10.3 describes the new policy activation records.

Contents

Abstract **i**

1 Introduction **1**

2 Policy System Architecture **2**
2.1 Communications Layer 2
2.2 Policy Server Layer 3
2.3 User Interface Layer 3
2.4 Language Support and System Limitations 4
2.4.1 Policy Server Support 4
2.4.2 SER Support 4
2.4.3 GNU GK Support 4
2.4.4 Mitel 7000 Support 5
2.4.5 MATCH Support 5
2.4.6 PROSEN Support 5

3 Running The Policy System **6**
3.1 Running The Policy Database 6
3.2 Running The Policy Store 8
3.3 Running The Policy Server 9

4 Interface to The Communications Layer **12**
4.1 Call Control Interface 12
4.2 OSGi Interface 14

5 Interface to The User Interface Layer **15**
5.1 Message Structure 15
5.1.1 User Interface to Policy Server 16
5.1.2 Policy Server to User Interface 17

6 Policy Server **19**
6.1 Architecture 19
6.2 Implementation 19
6.3 Package uk.ac.stir.cs.accent.environment 20
6.3.1 General Principles 20
6.3.2 Currently Defined Environment Variables 21
6.4 Package uk.ac.stir.cs.accent.event 21
6.5 Package uk.ac.stir.cs.accent.expression 22
6.6 Package uk.ac.stir.cs.accent.goal 22
6.7 Package uk.ac.stir.cs.accent.interaction 22
6.8 Package uk.ac.stir.cs.accent.protocol 23
6.8.1 Class SIPMessageHandler 23
6.8.2 Class H323MessageHandler 23
6.8.3 Class Mt7000MessageHandler 23
6.8.4 Class QueryMessageHandler 26

6.8.5 Class ContextHandler . . . 27
6.9 Package uk.ac.stir.cs.accent.repository . . . 27
6.10 Package uk.ac.stir.cs.accent.server . . . 27
6.10.1 Classes EventAdminPostEventService, MessageBrokerPostEventService . . . 27
6.10.2 Class MessageHandler . . . 28
6.10.3 Class PolicyApplicator . . . 28
6.10.4 Class PolicyEvaluator . . . 29
6.10.5 Class PolicyResponse . . . 29
6.10.6 Class UploadHandler . . . 30
6.10.7 Other Classes . . . 31
6.11 Package uk.ac.stir.cs.accent.service . . . 31
6.12 Package uk.ac.stir.cs.accent.store . . . 31
6.13 Package uk.ac.stir.cs.accent.utility . . . 31

7 Call Control Approach **32**
7.1 A Worked Example based on SIP . . . 32
7.2 Addition of Further Call Control Protocols . . . 34
7.3 Adding A New Handler Class . . . 36
7.3.1 Adding to The Policy Database . . . 36
7.3.2 Class Outline . . . 36
7.4 Adapting PolicyResponse . . . 37

8 Addition of Environment Variables **38**

9 Conclusion **40**

List of Figures

2.1 Overview of the Policy System Architecture (for Call Control) 3

4.1 Message Flow between Communications Layer and Policy Applicator (for Call Control) 13

6.1 Overview of the Policy Server 20

7.1 Hash Table Entries for SIP Call 33
7.2 Alternative Message Flow for Specific Protocol Handlers 35

Chapter 1

Introduction

The ACCENT project was concerned with developing a practical and comprehensive policy language targeted at the call control domain. The project studied a number of distinct tasks: the definition of the language (discussed in [11, 17]) 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 [12] and a comprehensive policy system has been implemented.

The approach was subsequently extended for home care and telecare on the MATCH project (Mobilising Advanced Technologies for Care at Home, www.match-project.org.uk) and for sensor networks and wind farms on the PROSEN project (Proactive Control of Sensor Networks, www.cs.stir.ac.uk/~kjt/research/prosen). This document focuses on the policy system layer of the architecture. The policy server layer is concerned with storing, deploying and enforcing policies. It represents the core functionality of the three-layer architecture. This document discusses the implementation of the policy server.

The following sections consider the context of the software discussed in this document; for a fuller account see [12]. Startup and shutdown of the policy server are discussed, along with the interfaces to the managed system layer and the policy definition (or user interface) layer. The implementation details describe the individual classes, their role and interworking. The architecture is open and as such can be extended. In fact two foreseen extensions are described: the addition of further protocols for call control, and support for further environment variables to be used in policies. Finally, further ideas and suggestions are presented for continuation of the work.

[11, 12] describe the initial design work, while [18] reviews the implementation for call control. [1, 13, 15] discuss policy conflict as a form of feature interaction. Technical reports describe the ACCENT policy language [17], the web-based policy wizard [16] and the policy environment [14]. Further information is available from www.cs.stir.ac.uk/accent.

Chapter 2

Policy System Architecture

Figure 2.1 shows the three-layer architecture consisting of the Communications Layer, the Policy Layer and the User Interface Layer. A three-tier architecture is used in completely different ways in other application areas. However, a three-tier policy architecture emerges naturally for this work.

Existing call control architectures have used similar architectures for some time. For example in the IN (Intelligent Network), 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 environment [5], the Communications Layer is provided by SIP proxies, while additional functionality can be added to the proxies in form of CPL (Call Processing Language [8]) or CGI scripts (Common Gateway Interface, essentially providing the control functionality of the policy server layer).

However, the ACCENT architecture differs in several key aspects. With reference to figure 2.1:

- 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. The IN does not provide any such mechanism, although this was partially a goal of Parlay and JAIN [9]. The User Interface Layer and the Policy Server Layer make use of context information to provide new capabilities.
- The Policy Server Layer is independent of the managed system. This makes it possible to work with several underlying systems if required. Advances in the Communications Layer do not adversely impact on higher layers.
- No complex intra-layer signalling is required, as messages are re-routed through policy servers.

The architecture makes the underlying managed system invisible to the higher layers. It enables end-user configurability and provides communication between policy servers which can be used for interaction handling.

2.1 Communications Layer

The Communications Layer represents the managed system. For call control, this assumes end-devices such as traditional phones or softphones 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. For home care, the communications layer contains a home network with sensors and actuators. For sensor networks, the communications layer contains a sensor network and various controllers.

Two crucial assumptions are made about the communications layer. The policy servers must be provided with key messages (e.g. outgoing call initiated, medication alert, over-temperature alert); further processing of a message is suspended until a policy server has dealt with the message. A mapping from low-level messages of the communications layer to more abstract policy events is assumed.

This report is not concerned with any further details of the Communications Layer and simply assumes that both requirements are fulfilled. Practical implementation work has shown that this is a realistic assumption.

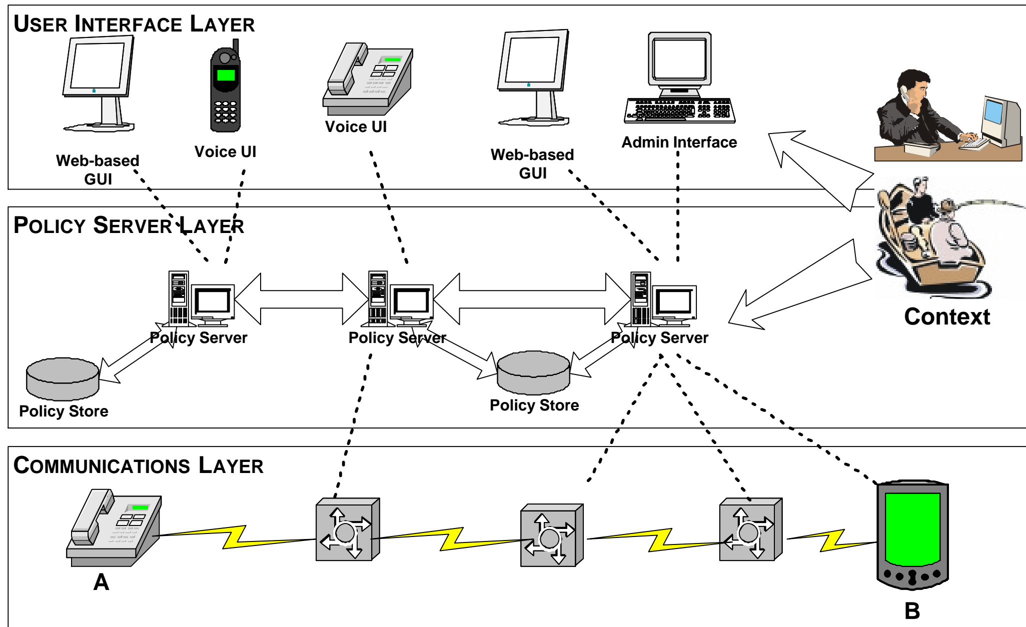


Figure 2.1: Overview of the Policy System Architecture (for Call Control)

2.2 Policy Server Layer

The Policy Server Layer is the heart of the approach. This layer contains a number of policy servers that interact with the underlying managed system(s). It also contains a number of policy stores – database or tuple space servers where policies can be stored and retrieved by the policy servers as required. Several policy servers might share a policy store, and each policy server might control more than one managed system.

The policy servers interact with the user interfaces in the policy creation process as discussed in section 5. They also interact with the communications layer where policies are enforced; this is discussed in detail in section 7. A further role of the policy server is to detect and resolve (or suggest resolutions of) conflicts among policies. To enable richer resolution mechanisms, communication between policy servers for information exchange is permitted. Further, the policy servers can use contextual information to influence policy-based control.

2.3 User Interface Layer

The User Interface Layer allows users to create and deploy new policies. Interface styles can include graphical, web-based and voice-controlled applications, each providing functionality targeted at certain types of users and devices.

A web-based user interface is convenient for most functions. However mobile users might not have the capability to use such technology, so voice-controlled interfaces may be more appropriate. These also suit users who simply want to activate or deactivate policies. A voice interface would be essential for disabled or partially sighted users. Both web and voice interfaces aim to guide the user in an intuitive way, preferably in natural language. Libraries of policies can be simply adapted and combined.

Administrative interfaces are system-oriented and exist mainly for system administrators to manage more complex functionality such as definition of goals, template policies and conflict resolution policies.

The policy definition environment provides access to contextual variables which are instantiated at run-time by the policy servers with live information. For example ‘secretary’ in a forwarding policy could be filled in from a company organisation chart, while ‘holiday’ could be completed using holiday and absence information. Also, the location of a user could be determined from an active badge or RFID (Radio Frequency Identification) system.

2.4 Language Support and System Limitations

The policy server supports nearly all of the APPEL language described in [17]; the managed system that controlled by the policy server may be more limited.

2.4.1 Policy Server Support

Policy server support of APPEL is currently as follows:

- Policies must be explicitly triggered (e.g. by an external call or sensor event). Although a policy can be defined without an explicit trigger (e.g. a policy that depends on time only), it will be activated only if an external event causes it to be considered (e.g. timer expiry).
- 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 email and SMS (Short Message Service). Variable substitution is supported in the associated text message.
- Static detection of policy conflicts is not implemented in the policy server, though it is supported by the separate RECAP tool (Rigorously Evaluated Conflicts Among Policies [2]). Dynamic detection of policy conflicts is limited to a single server. A single level of resolution checking is supported, so conflicts among resolutions are not detected.
- The policy server does not verify the source of any external or policy information. Thus any Internet system may maliciously provide incorrect information or 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.
- If it is planned to run the policy database on a Mitel 7000, note that the firewall settings may need to be changed if MySQL is to be accessed from outside the Mitel 7000 (e.g. by the policy wizard).

2.4.2 SER Support

The interface to SER (SIP Express Router, iptel.org/ser) makes use of the call control interface described in section 4.1. Support of APPEL is currently as follows:

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

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

2.4.3 GNU GK Support

The interface to the GNU GK (H.323 GateKeeper, www.gnugk.org) makes use of the call control interface described in section 4.1. Support of APPEL is currently as follows:

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

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

2.4.4 Mitel 7000 Support

The interface to the Mitel 7000 ICS (Internet Communications Server, a softswitch or PBX) makes use of the call control interface described in section 4.1. Support of APPEL is currently as follows:

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.

2.4.5 MATCH Support

The interface to MATCH (Mobilising Advanced Technologies for Home Care, www.match-project.org.uk) makes use of the OSGi interface described in section 4.2. Support of APPEL is currently as follows:

triggers: *device_in*, *receive_message*, *timer_expiry*

actions: *device_out*, *send_message*.

2.4.6 PROSEN Support

The interface to PROSEN (Proactive Control of Sensor Networks, www.cs.stir.ac.uk/~kjt/research/prosen) makes use of the OSGi interface described in section 4.2. Support of APPEL is currently as follows:

triggers: *device_in*, *receive_message*, *timer_expiry*

actions: *device_out*, *send_message*.

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 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 to hold dynamic policy information, policies and policy variables.

3.1 Running The Policy Database

The policy database is a conventional relational database, currently MySQL (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. *mysql* 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* is provided as follows. 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 'a_password';

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

FLUSH PRIVILEGES;
```

The database provides information to the policy server as to which protocols are currently supported and also the mappings between policy terminology and its meaning in protocol terms. For call control, 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* is provided as follows:

```
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 identifier. 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, see the APPEL policy language definition [17].

Protocol terms can be more complex, e.g. adding a video channel might be more than a simple operation. It is the role of a 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, while non-actions are translated from protocol to policy terms (e.g. to identify triggers). The file *lib/terminology_setup.sql* is provided as follows:

```
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 run-time of the policy server no further mappings can be added.

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

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* is provided as follows:

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

3.2 Running The Policy Store

The policy store is implemented as a tuple space server that supports XML. Currently this is IBM TSpaces (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 *tuples.properties* is as follows. 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                 = a_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 *tspaces-start*, *tspaces-stop* and *tspaces-admin* are provided with the policy system distribution. These allow TSpaces to be started, stopped and administered.

Start up TSpaces. The *tspaces-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: *tspaces-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 *tspaces-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. A typical example is given below. The policy database and policy store can run on different systems or on the same system as the policy server.

```
# Policy Server properties when running on the local machine

# Database server name (e.g. "localhost"), remote access port number (e.g. 3306)

database.server         localhost
database.port           3306

# Accent database name (e.g. "accent") , username for accessing this
# (e.g. "accent"), password for accessing this, terminology mapping table name
# (e.g. "terminology_mapping")

database.name           accent
database.username       accent
database.password       a_password
database.table          terminology_mapping

# Policy event handler (e.g. "EventAdmin" for plain OSGi, "MessageBroker" for
# MATCH" or "ContextServer")
```

```
event.provider          EventAdmin
# event.provider        MessageBroker
# event.provider        ContextServer

# Application domain and administrator email for this

application.domain      home_care
application.owner       admin@cs.stir.ac.uk

# Email server (e.g. "smarthost.cs.stir.ac.uk"), SMTP port number (e.g. "25"),
# email user address and password, subject used for email messages from the
# system

mail.server             smarthost.cs.stir.ac.uk
mail.port               25
mail.user               kjt@cs.stir.ac.uk
mail.password           a_password
mail.subject            Policy System Message

# Prefix of system policies (instantiated prototypes) and system variables
# (e.g. "!"), prefix of prototype parameters (e.g. "$")

parameter.prefix        $
system.prefix           !

# Port used for sending events to the policy server (e.g. 9998) and port used
# for uploading/querying policies (e.g. 9999)

policy.message.port     9998
policy.upload.port      9999

# Directory (relative to policy server root) for logging

policy.log.directory    PolicyServer

# Name of the ontology server host (e.g. "localhost")

poppet.host             localhost

# Hex flags to turn on debugging (0204 typically, 03FF for everything):
#  0000 (nothing), 0001 (prototype contribution), 0002 (diagnostics)
#  0004 (dynamic analysis summary, 0008 (filtered policies), 0010 (goal formula)
#  0020 (identical scores), 0040 (policy indexes), 0080 (optimised policies)
#  0100 (dynamic results), 0200 (static analysis summary)

server.debug            0204

# Number of lines to preserve in a server log

server.log.lines        2048

# Name of the tuple server host (e.g. "localhost"), port number (e.g. "8200")

tuples.server           localhost
tuples.port             8200

# Name of the policy database (e.g. "Policies"), policy username (e.g.
# "accent"), and password
```

```
tuples.name             Policies
tuples.username         accent
tuples.password         a_password
```

On startup, the policy server reads this file and creates a Java properties object. The *PolicyServer.properties* file is the default one, though any file name or location can be used.

The policy server can be run from the command line using the *polserv-start* script (and stopped with *polserv-stop*). The *polserv-start* script 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 interface, the default), '-g' (GUI interface), '-h' or '-?' (for help). These may be followed by the name of the properties file to be used.

The policy server can also be run as a bundle within OSGi (Open Services Gateway initiative, www.osgi.org). The current OSGi implementation of the policy server is designed for Knopflerfish (www.knopflerfish.org). More information can be found in [14].

The *PolicyServer* directory includes subdirectories *bin* (compiled code), *doc* (JavaDoc for the code), *lib* (required jar files), *log* (log file), *resource* (icons), *src* (Java), and *test* (test files). Apart from the GNU *Getopt* package, the code is in packages named *uk/ac/stir/cs/accent/**.

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) is disallowed. Copy *PolicyServer.java.nogui* in *src/uk/ac/stir/cs/accent/server* to *PolicyServer.java* and rebuild.

When running via a plain console, log messages are sent to the standard output/error. When running via a GUI, log messages appear in various windows. When running in OSGi, log messages appear in the main platform window. 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 Communications Layer

The policy server interacts with the Communications Layer when enforcing policies. A message handler class of the policy server deals with incoming messages, whether from a module in the underlying system (call control) or from an event (OSGi).

4.1 Call Control Interface

For call control, the policy server provides a TCP/IP socket for interaction. 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 are handled in a similar fashion.

Rather than just passing a call control 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:

```
<message>     ::= <protocol>\n<varlines>\nMSG\n<message>\n\n
<varlines>    ::= <varline> | <varline><varlines>
<varline>     ::= <var>: <value>
<protocol>    ::= SIP | H323 | ...
<message>     ::= a raw SIP message | an empty message string | ...
<var>         ::= an environment variable understood by the policy applicator
<value>       ::= a suitable value for the given variable
```

For details of *var*, refer to section 6.3. What defines a suitable *value* for an *var* should be obvious from the variable. *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 with this protocol.

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

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.

A list of *PolicyResponse* values is passed through the message handler, which constructs the string to be sent to the Communications 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 6.10.5.

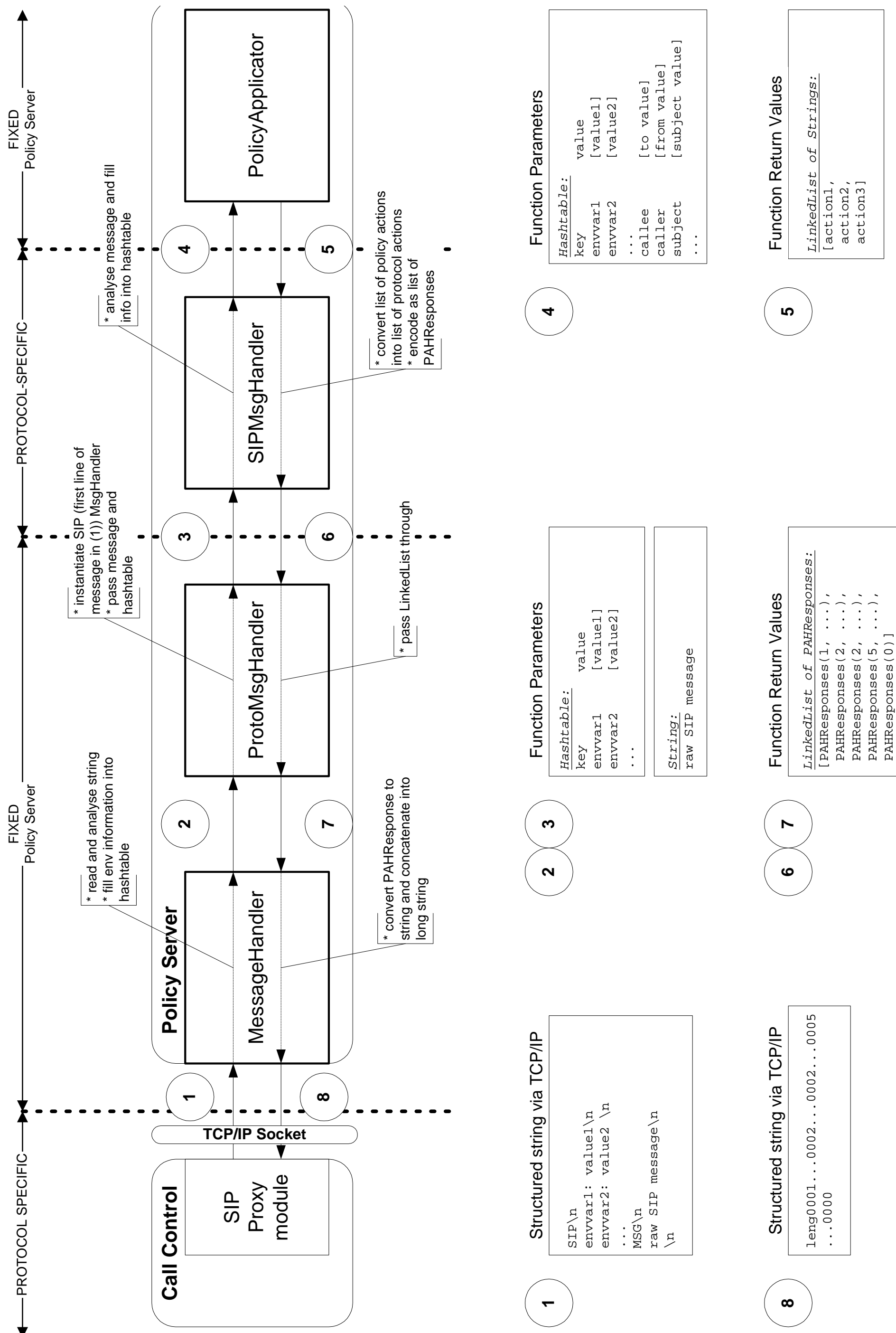


Figure 4.1: Message Flow between Communications Layer and Policy Applicator (for Call Control)

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 decomposed in the Communications Layer module and the actions are performed call control.

4.2 OSGi Interface

The interface to OSGi makes use of the Event Admin service that allows two-way communication with the policy server using events for both inputs (triggers) and outputs (actions). This decouples sensors and actuators, and also lends itself to event filtering for sensor fusion and actuator fusion. As wellas the OSGi Event Admin service, an alternative interface has been implemented for the MATCH event broker that uses REDS (Reconfigurable Event Dispatching System). The policy server design allows for selection of event system by setting the *event.provider* property to *EventAdmin* or *MessageBroker* (though code for the latter is currently not in use).

For plain OSGi events, the event *topic* identifies the kind of message, the *user* indicates the user whose policies should be checked (input) or who is causing the action (output), and various arguments establish the message parameters. The format of the latter is specified in [17]. The general structure of events is as follows:

```
<topic>  ::= uk/ac/stir/cs/accent/<message>
<user>   ::= <email address>
<arg1>   ::= <message type>
<arg2>   ::= <entity name>
<arg3>   ::= <entity instance>
<arg4>   ::= <message period>
<arg5>   ::= <parameter values>
```

Input events are *device_in* (sensor input), *receive_message* (email or SMS) and *timer_expiry*. Output events are *device_out* (actuator output) and *send_message* (email or SMS).

As a concrete example, suppose a hall light reading of 60% is measured at 1PM. This should be checked against the policies for admin@stir.ac.uk. The event parameters would be:

```
<topic>  ::= uk/ac/stir/cs/accent/device_in
<user>   ::= admin@stir.ac.uk
<arg1>   ::= reading
<arg2>   ::= light
<arg3>   ::= hall
<arg4>   ::= 13:00:00
<arg5>   ::= 60
```

For a *device_in* event, a system variable is set with name *arg3_arg2* (or just *arg2* if *arg3* is empty). Its value is set to *arg5* (or to *arg1* if *arg5* is empty). For example:

device_in(open,door,front): will set *front_door* to *open*

device_in(movement,hall): will set *hall* to *movement*

device_in(reading,temperature,lounge,,20): will set *lounge_temperature* to *20*

device_in(reading,humidity,,60): will set *humidity* to *60*.

Chapter 5

Interface to The User Interface Layer

The policy server provides a TCP/IP port for policy management. This section discusses 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 9998) of the policy server. Messages are the structured strings discussed below. For convenience, two implementations of classes (Java and PHP) have been provided to simplify the exchange, but are not discussed here. See section 6.10.6 for details of the *UploadHandler* class.

A direct interface is provided between the policy server and the user applications, 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 would not work 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 particularly important when a policy management operation fails: informative feedback can be provided to the user, providing guidance to resolve the problem.
- Several different interfaces are possible: it is envisaged that user interfaces will reflect the technical skills of users, e.g. system administrators might wish richer possibilities than lay end-users. Different interfaces might be preferred for different devices, e.g. mobile phone users might prefer voice interfaces, whereas a web interface might be preferable for a desktop computer.

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 interfaces supports upload, update, delete, enable, disable and query. There are two types of messages: requests and responses. The former are sent to the policy server, the latter are returned by it. Each message line ends with a newline ('\n'), and a whole message ends with a double newline.

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:

- *identifier*: a unique document identifier, formed by concatenating the address of the owning user, a '*' symbol and the policy identifier, e.g.:

  ```
  bob@acme.com*Reject incoming calls
  ```

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

- *policy*: the document 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"/>
```

- *type*: the document type as *goal*, *policy*, *prototype*, *resolution* or *variable*

5.1.1 User Interface to Policy Server

UPLOAD

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

```
UPLOAD
identifier
policy
```

UPLOAD will succeed if there are no conflicts and the identifier is not in use. It will fail otherwise, reporting on the reason for failure and including the problematic cases in the the response message.

UPDATE

This command is used to update an existing policy:

```
UPDATE
identifier
policy
```

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

ENABLE

This command is used to activate an existing policy:

```
ENABLE
owner
identifier
type
```

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 cases in the response message.

DISABLE

This command is used to deactivate an existing policy:

```
DISABLE
owner
identifier
type
```

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 cases in response message.

DELETE

This command is used to delete an existing policy:

```
DELETE
owner
identifier
type
```

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 cases in 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
identifier
type
```

This gives the owner's email address, the type of information required ('goal', 'policy', 'prototype', 'resolution' or 'variable') and the identifier of the tuple ('*' to query all tuples, or the identifier of a specific document). If all documents 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 *LARGE_LENGTH*, currently 1024 bytes.) This is to allow a list of large audio clips to be queried efficiently.

QUERY will always succeed, but there might be a number of responses. Usually QUERY 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 Interface

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 a response to a QUERY request:

```
SUCCESS
policy
```

FAIL

This response is used to report failure to the user:

```
FAIL
policy
suggestions
reason
comment
```

- *policy* may contain a list of conflicting policies, or if there was a problem related to the identifier (existence or absence of a policy with this identifier) just the policy under consideration.
- *suggestions* may contain a list of possible solutions if there was a conflict. Such solutions could be automatically determined.

- *reason* may contain the string CONFLICT, the string ID or the string GENERAL FAILURE to identify the nature of the problem. 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, stores and databases as discussed in [12]. The policy server is inherently multi-threaded and complex. The structure of the policy server is depicted in figure 6.1.

Briefly, the policy server needs to handle policy deployment and policy enforcement, with possibly several requests simultaneously. Hence the core of a policy server provides a mechanism to handle multiple incoming requests of each type: deployment and enforcement. Each of these two fundamental actions has a class to handle them: *UploadHandler* and *MessageHandler*.

An upload handler deals only 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 and allows for extensibility in the design. Future extensions could include support for further protocols and further policy environment variables, handled by separate packages. A message handler instantiates an appropriate handler to translate message information into policy information. Policy handling, as performed by the policy applicator, is then independent of the underlying protocol.

6.2 Implementation

The policy server is written entirely in Java and consists of multiple packages. The existing classes have dependencies on some standard libraries that are publically available. The policy store is a tuple space server, currently IBM TSpaces (www.alphaworks.ibm.com/tech/tspaces/download). Several parts require database access, and the chosen database is MySQL (www.mysql.com) together with the MySQL connector library (dev.mysql.com/downloads/connector/j). SIP messages may be processed with the NIST-SIP parser (www-x.antd.nist.gov/proj/iptel). TSpaces, NIST-SIP and the developed code itself require access to several other libraries (Antlr, Xerces). The NIST-SIP distribution includes *antlrall.jar*, while *xerces.jar* is available from Apache (xerces.apache.org/xerces-j). The policy server packages are as follows:

environment: This supports environment variables (section 6.3).

event: This deals with OSGi events (section 6.4).

expression: This supports expression evaluation(section 6.5).

goal: This supports goal realisation by synthesis of prototype policies (section 6.6).

interaction: This handles policy interactions (section 6.7).

protocol: This deals with protocol dependencies (section 6.8).

repository: This provides OSGi services to use the policy store (section 6.9).

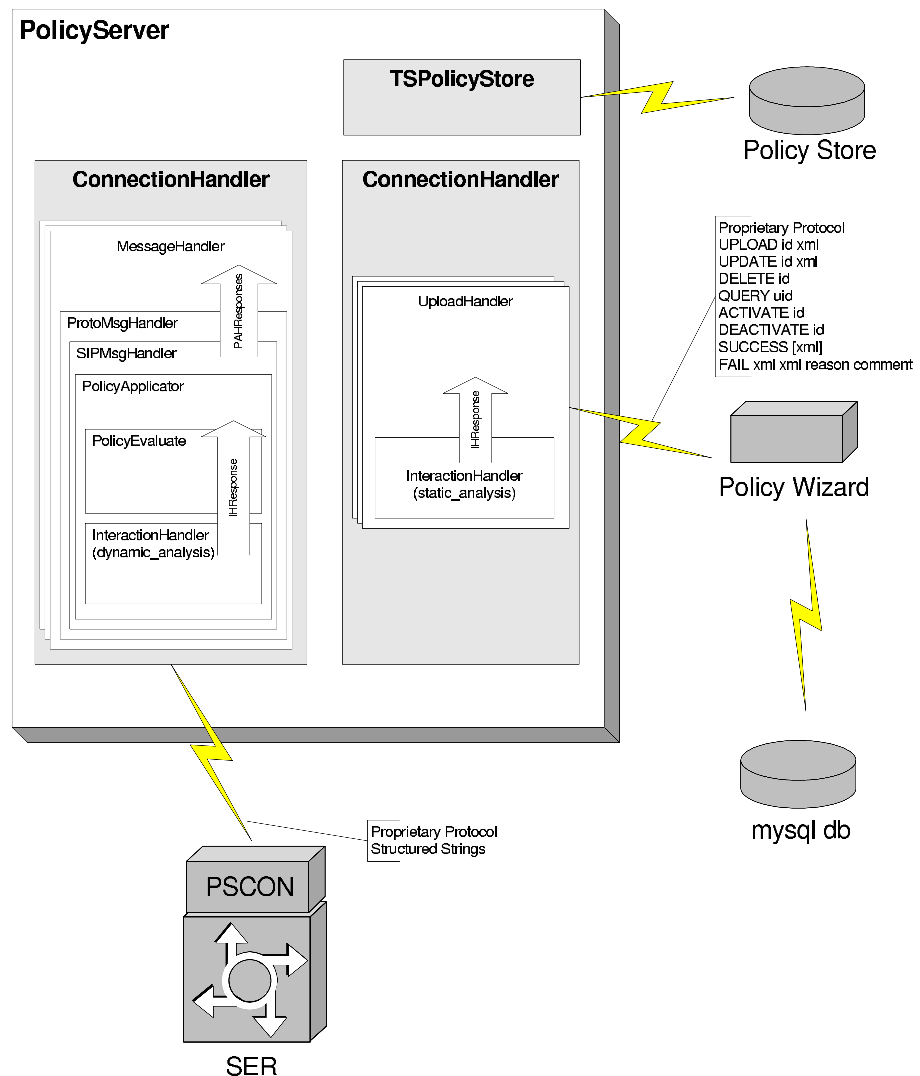


Figure 6.1: Overview of the Policy Server

server: The main class is *uk.ac.stir.cs.accent.server.PolicyServer*, which initialises the system and starts all required threads. The other classes are concerned with multi-threading and core functionality of the policy server (section 6.10).

service: This provides OSGi services to use the policy server (section 6.11).

store: This provides the implementation of the data storage used by all other parts of the system (section 6.12).

utility: This provides some common functions (section 6.13).

The following sections provide a conceptual overview of the packages and classes. Additionally, the code is documented using JavaDoc.

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

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 [11]. Each of these concepts is compared to actual values in a policy body: these form the policy conditions. A typical example is *time lt 12:00*, stating that it currently should be before midday.

The policy server has a class for each concept, 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 simply returns 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 another 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.

6.3.2 Currently Defined Environment Variables

The current implementation provides environment variables for the concepts identified as important in [11]. The specific list of environment variables is defined in [17]. 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 underlying system, it is essential that values are provided by the specific message handler in the environment variable hash table indexed by the names provided here. Values in this hash table are generally linked lists. 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, but in SIP it is a URI. A policy will just talk about a general domain entity. Some types are fixed collections.

All environment variables are assumed to be initialised. The more variables are instantiated with useful 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 then the condition will evaluate to false (and hence the policy might not be applied).

The term 'domain identifier' needs some explanation. Currently Internet/email addresses are used for domains. A specific user might be *xyz@here.com*, considered to be part of the domain *here.com*. The use of a name from LDAP (Lighweight Directory Access Protocol) might be considered in future.

There are only two mandatory variables. Note that they are special in that they have only a plain value, and there is no *Evaluate* class for them:

SERVER_NAME: Domain identifier (e.g. *cs.stir.ac.uk*), used to determine whether this is an outgoing or incoming message.

user: Domain identifier (e.g. *mike@project.org*), used to identify the user for whom policies should be retrieved.

'Optional' variables are supplied by the Communications Layer; see [17] 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 [17] for the details.

Note that the split in the two above classes might not prove appropriate in the future. A simple approach was also desired. 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 Package uk.ac.stir.cs.accent.event

AbstractEventHandler deals with events in general, which may be OSGi events or MATCH broker events. The *OSGiEventHandler* class extends this to deal with device in/out events and their parameters. *OSGiEventListener* is the actual class that receives OSGi events and passes them to *OSGiEventHandler*. *BrokerEventHandler* is for the MATCH event broker (though the code is currently not in use).

6.5 Package uk.ac.stir.cs.accent.expression

Expression evaluation (used by *set_variable*) is defined using JavaCC (javacc.java.net). The key file here is *Expression.jj* which is compiled to all other files.

6.6 Package uk.ac.stir.cs.accent.goal

StaticAnalyser is responsible for checking prototypes against goals whenever there is a change in either. This first deletes existing instantiations of prototypes, then re-instantiates prototypes that contribute to goals. The *DynamicAnalyser* class is called when prototype-derived policies are selected in response to a trigger. It calls *OptimiserManager* to set things up for goal selection. This in turns calls *GoalOptimiser* to choose the set of prototypes that maximise the contribution to goals. *EvaluationFunction* is a dummy class whose *evaluate* function is dynamically defined based on the overall evaluation function. This makes use of Java reflection and the JavAssist package (www.javassist.org).

6.7 Package uk.ac.stir.cs.accent.interaction

The interaction handler package provides the classes *InteractionHandler*, *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 may be used during policy definition to determine whether policies conflict. In particular the purpose of static analysis is to determine if:

- a new/newly enabled/updated policy of a user is consistent with respect to other policies by that user.
- 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).
- a disabled/deleted policy does not lead to any conflicts between other policies by that user
- 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).

If a conflict is detected, a description of the conflict and preferably some possible solutions should be provided. 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 [12]. In addition, the algorithm described in [19] could be useful.

Dynamic analysis is more complex. Algorithms here are applied at run-time, i.e. while a message from the managed system is being processed. 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). The approach could involve run-time methods from the feature interaction community (see [11, 12]). 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 preference (as expressed in the policy language) is used to influence the best outcome.

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 [17]. The approach implemented for dynamic analysis is described in [1, 15]. Resolution policies resemble call policies but differ in the following respects:

- The triggers of a resolution policy are the actions of a regular policy. The arguments of these triggers are named for use in resolution conditions.

- The conditions of a resolution policy resemble those of call policies, but are extended to include *preference0* to *preference9* (preferences of conflicting policies) and *variable0* to *variable9* (arguments of conflicting triggers).

- The actions of a resolution policy resemble those of regular policies, but are extended to include various *apply* operators that allow generic resolutions to be defined.

Initially, all the actions resulting from triggered policies are determined. The policy applicator is then instantiated recursively to check these against the available resolutions. Both conflict detection and resolution are 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.

6.8 Package uk.ac.stir.cs.accent.protocol

MessageHandler 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 policy server. *DefaultMessageHandler* will be chosen by the system if no specific handler can be found.

6.8.1 Class SIPMessageHandler

The *SIPMessageHandler* class is designed to interface with a module specially written for the Fraunhofer Institute SIP Express Router (SER). Limitations of the support for APPEL are described in section 2.4.2. This 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 event and condition mapping. At this point the hash table contains information in policy terminology ready for processing by the policy applicator. The 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. This is discussed in more detail in section 7.2, which is a 'must read' for developers intending to add support for other protocols.

6.8.2 Class H323MessageHandler

The *H323MessageHandler* class deals with messages exchanged with an H.323 gatekeeper (GNU GK). Limitations of the support for APPEL are described in section 2.4.3. 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 [6, 7].

6.8.3 Class Mt7000MessageHandler

Mt7000MessageHandler handles a Mitel 7000 ICS that supports SIP telephones. The *Mt7000Command* class sends commands, while *Mt7000Response* deals with response to these.

The Mitel 7000 runs the completely separate *uk.ac.stir.cs.accent.mt7000* package that interfaces with these classes in the policy server. Limitations of the support for APPEL are described in section 2.4.4. This external package contains 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 adapts open source code by Christian d'Heureuse (www.source-code.biz). It converts audio clips to/from Base64 format encoding.

Originally the *ServiceListener* class needed to be installed in */usr/share/telephony/com/mitelknowledge/b2bua* on a Mitel 7000. However, this is no longer necessary for later Mitel 7000 versions.

The script *polint-start* is provided to start the policy interface on the Mitel 7000 (while *polint-stop* stops it). 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 uses JAR and class files in the */usr/share/telephony* directory.

The *polint-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 version currently supported. 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 *POLSERVICE* 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 Mitel 7000 versions that have been tested, 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'
;
```

6.8.4 Class QueryMessageHandler

The *QueryMessageHandler* class is a pseudo protocol handler. This is designed for querying the policy server's log of policy activations. Messages sent to the handler have the form:

```
Query
user: ken@stirling.org.uk
_analysis: trigger
_parameter: device_in(reading,temperature,interior,,14)
MSG
```

Three environment variables are mandatory. The *user* variable specifies which user's policies are to be checked. The *_analysis* variable specifies which kind of analysis is required. The *_parameter* variable defines a parameter for the analysis. The message may also contain other environment variables (e.g. *time: 22:30:00*) that are needed for execution (i.e. when considering a trigger). The message body is not used for queries and should be empty.

The response has the following form. This is all on one compact line, although it is spaced out here. Parts of the response are separated by '&&', while arguments within one part are separated by '|'. (These conventions make it easy to parse the response.)

```
2013-07-25 18:23:49 Response:
action(2013-07-25 19:14:59|device_out(close,window,all)|!Ensure warmth) &&
action(2013-07-25 19:14:59|device_out(set,heating,,,15.5)|!Ensure warmth)
```

The *_analysis* options are as follows:

action: Check when an action was performed. The parameter is the policy action, e.g. *device_out(set,heating,,,25)*. For home care and sensor networks, the message type or the parameter values may be omitted; they then act as wild cards that match anything. The response has the following form. It gives the specified action if it has been executed. This may be followed by a different action for the same device if it has been more recently executed. For home care and sensor networks, the entity name and entity instance must match but the message type or parameter values may differ. If the specified action was never executed, the response will be empty.

```
execution(YYYY-MM-DD HH:MM:SS|action|policy) &&
execution(YYYY-MM-DD HH:MM:SS|action|policy)
```

inaction: Check when an action was not performed, i.e. when a similar action with different parameters occurred such as *device_out(off,heating,,,)* instead of *device_out(set,heating,,,25)*. The parameter is the policy action. For home care and sensor networks, the message type or the parameter values may be omitted; they then act as wild cards that match anything. The response has the following form. It gives an action different

from the specified one if it has been executed. For home care and sensor networks, the entity name and entity instance must match but the message type or parameter values may differ. The different action may be followed by the specified action for the same device if it has been more recently executed. If an action different from the specified action was never executed, the result will be empty.

```
execution(YYYY-MM-DD HH:MM:SS|action|policy) &&
execution(YYYY-MM-DD HH:MM:SS|action|policy)
```

policy: Check when a policy was performed. The parameter is the policy identifier, e.g. *Avoid cold house*. The response has the following form. It gives zero or more instances of the policy execution. The status will be 'applicable', 'inapplicable', 'optimal'" or 'non-optimal'. If the specified policy was never executed, the response will be empty.

```
execution(YYYY-MM-DD HH:MM:SS|status) &&
execution(YYYY-MM-DD HH:MM:SS|status)
```

trigger: Check what a trigger would cause. The parameter is the policy trigger, e.g. *device_in(active,smoke,,,)*. The response has the following form. It gives zero or more instances of actions that the trigger causes. If the specified trigger yields no actions (e.g. nothing was triggered, policy conditions did not hold, or the policy was non-optimal), the response will be empty.

```
action(YYYY-MM-DD HH:MM:SS|action|policy) &&
action(YYYY-MM-DD HH:MM:SS|action|policy)
```

6.8.5 Class ContextHandler

The *ContextHandler* class is a pseudo protocol handler. It does not implement *GenericProtocolHandler* because of its 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.

6.9 Package uk.ac.stir.cs.accent.repository

RepositoryService provides OSGi services to access the policy store (e.g. to add, change or delete tuples).

6.10 Package uk.ac.stir.cs.accent.server

This package provides the core functionality of the prototype. *PolicyServer* is the main class. (The *NoGUI* version 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. A *ConnectionHandler* object is created to handle communication with the User Interface Layer. If the policy server is running as an OSGI bundle, an OSGi event handler is created. If the policy server is not running as an OSGI bundle, a second *ConnectionHandler* object is created to interface with the Communications Layer. Each handler thread responds to messages received on one of the two ports specified in the properties (default 9998 for Communications Layer messages and 9999 for User Interface Layer 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 *ConnectionHandler* for communications events starts a *MessageHandler* thread for each communication.

6.10.1 Classes EventAdminPostEventService, MessageBrokerPostEventService

The classes, supported by *PostEventService*, handle OSGi events and MATCH events.

6.10.2 Class MessageHandler

Each message handler thread is responsible for processing a single message passed from the underlying managed system. A message handler reads a structured text message as discussed in section 7.

The first line of the string is used to identify the corresponding protocol-specific message handler (e.g. for SIP this would be *SIPMessageHandler*). The following lines contain data that is not part of the raw message, each structured as key-value pairs. The message handler will add values from this to a hash table, indexed by the given 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.

The message handler then awaits a protocol-specific response which is assumed to be a linked list of *PolicyResponse* values. 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 pre- and post- negotiation were required (prior to and following policy execution).

6.10.3 Class PolicyApplicator

The policy applicator applies policies to a message. The applicator is protocol-independent as it works only 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 a protocol-specific handler from the message, while further information may be provided directly by the underlying system. The policy applicator returns an ordered list of actions that need to be performed (first in the list is 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 containing no, one or many values of the appropriate type as identified in section 6.3. For example, the *triggers* environment variable provides a list of 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, which is assumed to be a policy document as specified by the APPEL language schema. The top-level nodes within a policy document are *policy* nodes that have single descendant *policy_rules* nodes. Anything below the level of a *policy_rules* node is processed by a *PolicyEvaluator* object.

A policy applicator goes through the following procedure to determine what actions should be taken:

- 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*, *applies_to*, *valid_from* and *valid_ to* from a policy element are evaluated to determine this.

- Filter policies based on conditions: however, not all policies retrieved in the first step will be applicable at all times. The list of applicable policies is therefore refined. 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).

- Select policies to optimise goals: the list of eligible policies is passed to the dynamic analyser. This extracts policies that derive from prototypes (and therefore support goals). This set is optimised with respect to the overall evaluation function, and re- combined with non-goal policies.

- Deal with interactions: applicable policies are tested for consistency. Interaction handling is discussed in more detail in section 6.7.

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

In cooperation with the *PolicyEvaluator* and *DynamicAnalyser* classes, the policy applicator tracks the evaluation that follows a trigger. This leads to activation records being stored in the *PolicyServer* class. These records define the following. Policies are listed by their identifier.

- the policy activation time (YYYY-MM-DD HH:MM:SS)
- the list of triggers
- the list of triggered policies
- the list of applicable policies (a subset of triggered policies that have valid conditions)
- the list of regular policies that are applicable (a subset of applicable policies)
- the list of goal-related policies that are triggered and have valid conditions (a subset of applicable policies)
- the list of goal-related policies that are determined to be optimal (a subset of goal-related policies)
- the list of generated actions (including internal ones)
- the list of deconflicted actions (including internal ones)

6.10.4 Class PolicyEvaluator

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

A *PolicyEvaluator* object supports the method *hasRulesTriggered* to determine 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 attributes described in section 6.10.3. 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. Although APPEL is not formally defined, its semantics has been investigated [10].

6.10.5 Class PolicyResponse

PolicyResponse is used in call control for a data type that encapsulates the communication between a protocol-specific message handler and *MessageHandler*. Any message passed from the message handler to the protocol-specific handler are analysed and policies are 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*.

A protocol-specific handler constructs a list of responses. The message handler merely converts the individual responses into strings using the *toString* method of the *PolicyResponse* object and joins the strings, maintaining the order of the list. The final string is then sent to the underlying managed system, where an interface module needs to interpret it and enforce the actions.

This use allows for *PolicyResponse* to be subclassed for protocol-specific handlers where *PolicyResponse* is not suitable. For more details see section 7.2.

The current PolicyResponse type implements various cases identified by a type:

type 0: end of responses. The four byte string 0x0000 is constructed.

type 1: generate reply. 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: generate request. 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: process original. The four byte string 0x0003 is constructed.

type 4: add header. 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: replace header. 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: add body. Same as add header, apart from the type code which is 0x0006.

type 7: replace body. Same as replace header, apart from the type code which is 0x0007.

End of responses will always be the last element. 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*.

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

6.10.6 Class UploadHandler

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 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 discussed in section 5.

A request from the user interface consists of few lines of parameters separated by newlines, followed by a final double newline. The first line determines the operation required (either UPLOAD, UPDATE, ENABLE, DISABLE, DELETE or QUERY). The remaining lines contain arguments as required. Details of the request and response structure are given in section 5.

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

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

SUCCESS XMLDOC This 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 This 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 identifier of the policy does not exist (in case of update, enable, 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 that typically begins by querying the policy store for the existence of a policy with the provided identifier. This might raise an *ID* failure and lead to a termination of the operation. If the query 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* will be 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*.

6.10.7 Other Classes

LogFilter is used for managing log files. *PolicyFilter* is used for selecting policy files. *TimerHandler* deals with explicit timers as well as implicit ones that derive from policies with only time-related conditions.

6.11 Package uk.ac.stir.cs.accent.service

PolicyService defines an interface for OSGi services that allow policies to be added, changed and deleted. The *PolicyServiceImplementation* class is the implementation of this. *PolicyAction* defines an interface for getting various parameters of a policy (i.e. a device message).

6.12 Package uk.ac.stir.cs.accent.store

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* is 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 system-wide environment information to be placed into the policy store and retrieved. The second group is concerned with completely clearing the store (this removes all stored policies and environment information) or removing information that should be maintained only 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 for application.

The reference implementation for the policy store is based on TSpaces, a tuple space implementation from IBM. Tuple spaces are based on Linda [3, 4] 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 can be read from the tuple space.

Tuple spaces store tuples, which would serve for a somewhat flexible structure of the stored elements (e.g. 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: tuples are retrieved by providing a template tuple that has the same structure and matches some or all elements directly or by placeholders.

TSpaces and JavaSpaces were considered. A template returns all matching tuples, not just one. This is not supported by JavaSpaces so this was not a suitable choice. However, TSpaces does this and 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 would use a policy store variable of a different type.)

6.13 Package uk.ac.stir.cs.accent.utility

This package contains utility classes to support goals (*GoalUtilities*), protocol handlers (*HandlerUtilities*), time (*TimeUtilities*), timers (*TimerUtilities*) and common functions (*Utilities*).

Chapter 7

Call Control Approach

This section describes specific aspects of how call control and its associated protocols are handled.

7.1 A Worked Example based on SIP

Consider again figure 4.1 which shows a sample of the message flow between the Communications Layer and the policy applicator. This section elaborates 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 data in figure 7.1 is inserted. Values in square brackets represent lists The numbered notes are as follows:

Variable	Value	Note
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*

Figure 7.1: Hash Table Entries for SIP Call

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

2: As an important point, this 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 [13].

Finally, *SIPMessageHandler* instantiates a *PolicyApplicator*, initialises it with this 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 [13] 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 *SIPMessageHandler* 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 that 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, and 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* results that it has received from *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
```

7.2 Addition of Further Call Control 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 into a list of *PolicyResponse* values, 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 and METHOD are stored in the hash table together with the corresponding values. For example, 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, the result is a list of actions such as *add_medium(medium)* or *forward_to(address)*. These are converted back into SIP-specific *PolicyResponse* values, e.g. *add_medium(medium)* will result in a 'create new request' that will be an INVITE requesting the extra media. Similarly, *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 7.2 depicts an alternative message flow (compared to that of figure 4.1). While the figures at 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).

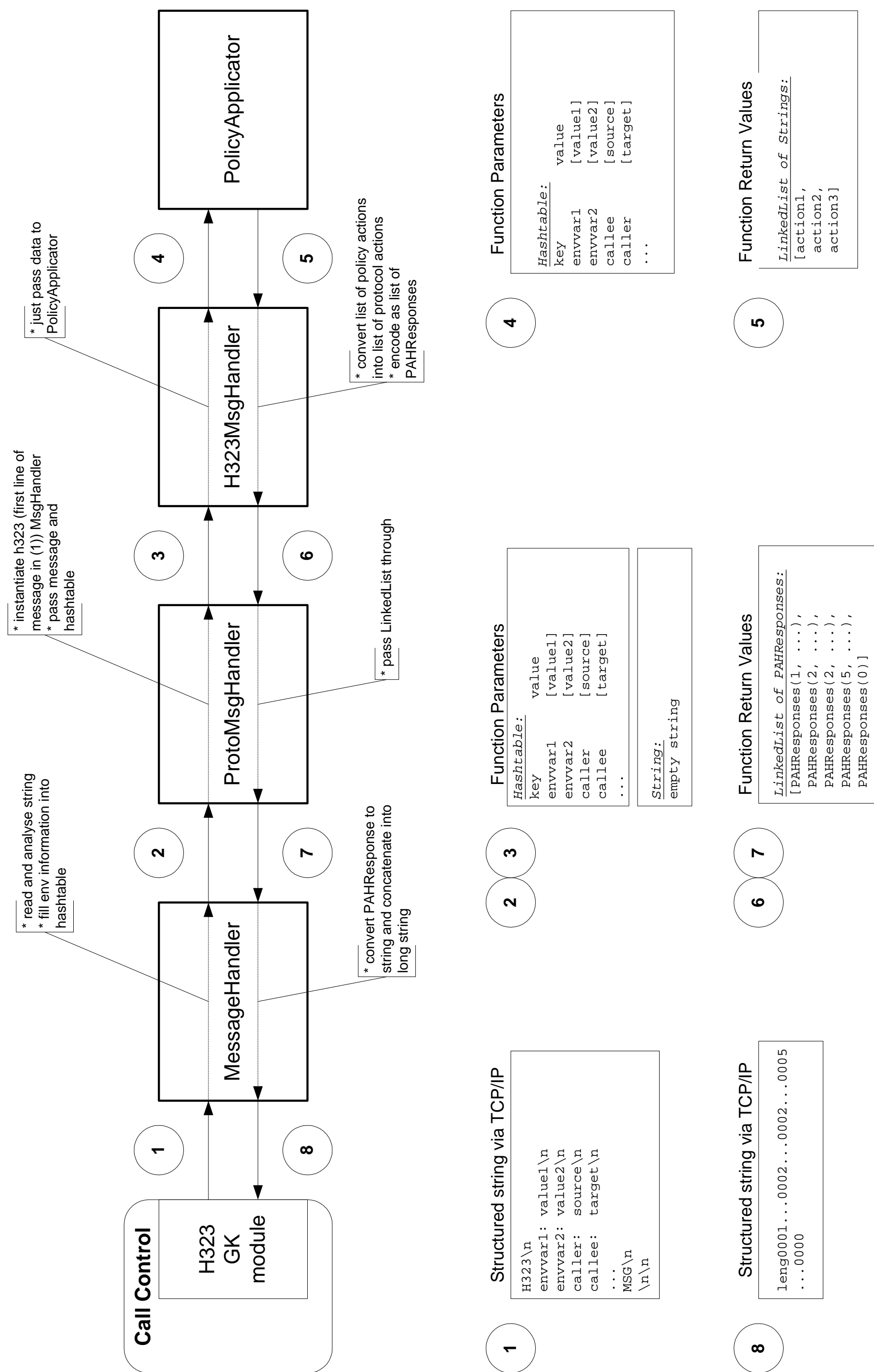


Figure 7.2: Alternative Message Flow for Specific Protocol Handlers

7.3 Adding A New Handler Class

To add a new handlerall that is required is a new class following the template in section 7.3.2. 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 Communications Layer (e.g. the *pscon.c* handler for SER or the H323 GK handler in figure 7.2). This needs to be provided outside the policy server and hence is not discussed here (but section 7 should give some insights).

The new protocol handler class **must** be named *<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 section 7.

7.3.1 Adding to The Policy Database

The database has a table called 'terminology_ mapping'. This table provides the mapping between policy terminology and protocol terminology as discussed in section 3.1.

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

```
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');
```

7.3.2 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()
}
```

7.4 Adapting PolicyResponse

If the *PolicyResponse* class provided (section 6.10.5) is unsuitable for the protocol to be added, it can be subclassed and the subclass 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 must then be handled by the call control functionality 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 and this extensibility allows developers to produce any (text) response they wish without the core of the system having to be changed.

Chapter 8

Addition of Environment Variables

The currently defined environment variables are discussed in section 6.3. However, it is envisaged 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 implement the comparison operators in a meaningful way for a given environment variable.

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*.

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 9

Conclusion

The design of the policy server has been presented. In particular, an explanation has been given of how it supports call control and the various protocols used for this.

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 Communications Layer with a more standard interface. SOAP is one protocol that might be useful in this context. Should such a change be made there are essentially two classes that must be adapted: *PolicyResponse* and *MessageHandler*. In particular, the *toString* method in *PolicyResponse* would have to return XML messages, and the *MessageHandler run* method would have to interpret SOAP messages that it received (i.e. the receiving loop must be adapted). Also, the *MessageHandler createRespString* method would have to convert the *PolicyResponse* XML into a correct SOAP message.

The same issue would apply to the interface between the policy server and the User Interface Layer. Again SOAP would be a suitable protocol. Here the changes required would be restricted to the *UploadHandler* class, where again the *run* method would need to deal with SOAP messages.

Section 6.3 discusses a number of environment variables and states that implementations have been provided for most of the classes. However, several of the implementations are simplistic and need to be refined.

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 Int. Conf. on Feature Interactions in Telecommunications and Software Systems*, pages 39–57. IOS Press, Amsterdam, Netherlands, June 2005.

[2] Gavin A. Campbell and Kenneth J. Turner. Policy conflict filtering for call control. In Lydie du Bousquet and Jean-Luc Richier, editors, *Proc. 9th Int. Conf. on Feature Interactions in Software and Communications Systems*, pages 83–98. IOS Press, Amsterdam, Netherlands, May 2008.

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

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

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

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

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

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

[9] Sun Microsystems. Java APs for Integrated Networks. `http://java.sun.com/products/jain`. November 2004.

[10] Carlo Montangero, Stephan Reiff-Marganiec, and Laura Semini. Logic based detection of conflicts in APPEL policies. In Ali Movaghar and Jan Rutten, editors, *Proc. Int. Symposium on Fundamentals of Software Engineering*, volume 4767 of *Lecture Notes in Computer Science*, pages 257–271. Springer, Berlin, Germany, October 2007.

[11] 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.

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

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

[14] Kenneth J. Turner. The ACCENT policy system. Technical Report CSM-188, Computing Science and Mathematics, University of Stirling, UK, August 2013.

[15] Kenneth J. Turner and Lynne Blair. Policies and conflicts in call control. *Computer Networks*, 51(2):496–514, February 2007.

[16] Kenneth J. Turner and Gavin A. Campbell. Goals and conflicts in telephony. In Masahide Nakamura and Stephan Reiff-Marganiec, editors, *Proc. 10th Int. Conf. on Feature Interactions in Software and Communications Systems*, pages 3–18. IOS Press, Amsterdam, Netherlands, June 2009.

[17] Kenneth J. Turner, Stephan Reiff-Marganiec, Lynne Blair, Gavin A. Campbell, and Feng Wang. APPEL: Adaptable and programmable policy environment and language. Technical Report CSM-161, Computing Science and Mathematics, University of Stirling, UK, June 2013.

[18] Kenneth J. Turner, Stephan Reiff-Marganiec, Lynne Blair, Jianxiong Pang, Tom Gray, Peter Perry, and Joe Ireland. Policy support for call control. *Computer Standards and Interfaces*, 28(6):635–649, June 2006.

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