Project no. IST-033576

XtreemOS

Integrated Project
BUILDING AND PROMOTING A LINUX-BASED OPERATING SYSTEM TO SUPPORT VIRTUAL ORGANIZATIONS FOR NEXT GENERATION GRIDS

Specification of application firewall

D3.5.8

Due date of deliverable: 31/05/2008
Actual submission date: 02/06/2008

Start date of project: June 1^{st} 2006

Type: Deliverable
WP number: 3.5
Task number: 3.5.10

Responsible institution: XLAB d.o.o.
Editor & and editor's address: Jaka Močnik
XLAB d.o.o.
Teslova 30
SI-1000 Ljubljana
Slovenia

Version 1.1 / Last edited by Matej Artač / 02/06/2008

Project co-funded by the European Commission within the Sixth Framework Programme		
Dissemination Level		
PU	Public	√
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

Revision history:

Version	Date	Authors	Institution	Section affected, comments
0.1	23/04/08	Matej Artač	XLAB d.o.o.	Initial Version
1.0	15/05/08	Jaka Močnik	XLAB d.o.o.	Final text, ready for internal reviews
1.1	02/06/08	Matej Artač	XLAB d.o.o.	Text updated with reviwers' comments

Reviewers:
Julita Corbalan (BSC), Haiyan Yu (ICT)

Tasks related to this deliverable:

Task No.	Task description	Partners involved°
T3.5.8	Application Firewall for Highly Available Services	XLAB*, ULM

°This task list may not be equivalent to the list of partners contributing as authors to the deliverable
*Task leader

Contents

List of Figures

Executive Summary

In this document, we present the design and specification of the Application Firewall (AFW) module for Application Execution Manager (AEM) service [2].

The Application Firewall is used to implement controlled access to network resources (i.e. outgoing network connections) for processes of jobs run via AEM. It does so by exploiting the netfilter/iptables framework of the Linux kernel [5].

By applying appropriate declarations to the node and VO policies [3], the system administrator may limit the ability of the job processes to connect to networked services in order to exercise control over consumed network resources or prevent malevolent jobs to send out illicit traffic (i.e. spam email, sensitive data, etc.). The jobs themselves, on the other hand, may specify the desired connectivity parameters required for their execution in their respective JSDL [4] description in a manner similar to other required resources. The node selection process will select execution nodes whose policies permit such connections.

The document is structured as follows: the first section introduces the Application Firewall concept and explain the rationale for its introduction in the job execution framework. Then, a thorough description of its design and implementation follows. Integration with AEM and security infrastructure is described next. We conclude with an overview of the current status and required future work.

1 Introduction

One kind of the computing resources that need to be allocated, monitored and provided to jobs based on their requirements are the communication resources (i.e. the computer network).

Networking capabilities of the jobs submitted to execution nodes by users must be controlled very strictly, as malevolent or excessive traffic not only hinders the node itself (by reducing the bandwidth available to other applications) running on the same node, but usually has a significant impact on the networks the node participates in.

A node sending out excessive amounts of data can hamper the local area network the node is part of, can swamp routers at the edge of this LAN, and can in some cases even have an adverse impact on the node as a part of the public Internet as well. As an example of the latter, a job may use multiple nodes to orchestrate a distributed denial-of-service (DDoS) attack, or send a mass of spam e-mail. In the best case, this will result in the node or even the whole network being blacklisted and unable to send even legitimate e-mail, in the worst case, it can result in the organisation owning the node being held legally responsible.

Based on this reasoning, the ability of jobs submitted to execution nodes to use communication resources must be strictly controlled by the system administrators of the Grid, first and foremost by the local node administrator, but by the Virtual Organization administrator as well. As our work is exclusively concerned with outgoing network connections, in the context of this document, the term communication resource is used to describe an *outgoing network traffic* initiated by a submitted user job.

On the one hand, communication resources that are made available to jobs running in a given VO and a given node should therefore be described in the (VO and node) policies. On the other hand, a submitted job should clearly state the communication resources it requires for successful execution. The requirements related to the communication resources are defined in a JSDL document in order to have the resource matching process select appropriate nodes where required communication resources could be allocated.

Finally, the service providing local job execution capabilities must take into account the communication resources required by a submitted job and — if these requirements are in accordance with the local node policy — allow use of the required resources to processes belonging to that job.

The next section presents the design of the Application Firewall.

2 Design of the Application Firewall

This section describes the design of the Application Firewall. It starts with a description of the basic concepts used and the requirements for the AFW, and continues with a detailed design.

2.1 Basic Concepts

A *job* is a collection of (Linux) processes, including the initial job process created by AEM executing a *submitJob* action on behalf of a user and all processes that this initial process forks.

A *communication resource* is an outgoing IP network connection, defined by

- *destination IP address*,
- *destination port number*, and
- *transport level protocol* (i.e. TCP, UDP)

2.2 Requirements

The Application Firewall must allow for the following functionality:

1. To prevent *any* outgoing network connections for all jobs by default, and
2. To allow outgoing network connections on a per-job basis to destinations specified by the communication resources listed in the job requirements.

It should be explicitly noted that:

- As the process of node selection is performed by the SRDS [1] prior to actual job submission, nodes whose combined local and VO policies do not allow connections specified by the required communication resources to be made will be excluded. Hence, the AFW does not need to perform any additional checking whether these rules are actually allowed by the node. If the job was executed on the given node, it is assumed that it may be granted all the communication resources it lists as required.
- Extracting information from a JSDL on which communication is requested by a job, and whether the requested communication is allowed (by a VO or a node policy) must be performed by job execution and resource discovery selection services (AEM, obtaining either VO level or local node level policies). These services in turn should relay the requested and permissible communication types to AFW in order to drive its actions,

- Keeping track of the processes belonging to a job is not in the scope of AFW functionality: instead this information should be acquired by job monitoring services and relayed to AFW as necessary.

2.3 Filtering Network Traffic

AFW will use the Linux netfilter/iptables framework [5] for filtering (allowing or rejecting) outgoing network traffic. This framework is commonly used for various packet inspection and transformation purposes such as implementing firewalls and NAT gateways.

IPTables operate on *chains of rules*. Each packet in the system is first sent to the appropriate chain, based on whether it is an incoming packet (destination is the local system), an outgoing packet (source is the local system), or a packet being forwarded (neither source nor destination addresses point to the local system). The *INPUT*, *OUTPUT* and *FORWARD* built-in rule chains are used for these packet types, respectively. The packet checking procedure traverses the rules in the chain until a rule matches the packet (see below), and then the action specified by the matching rule is executed. The actions are diverse, but most commonly the action is forwarding the packets to another chain and/or transforming some data in the packet header. As AFW filters the outgoing network traffic, we will insert rules in the *OUTPUT* chain in order to allow or reject the network traffic originating from the node executing the job.

2.3.1 Matching packets from job processes

Netfilter uses *match targets* to classify packets which should be rejected or allowed. The *owner* match extension of match targets are particularly suitable for our purpose of classifying network packets based on which job they belong to. The owner extension provides the following match targets:

- *--uid-owner userid*: matches if the packet was created by a process with the given effective (numerical) user id.

- *--gid-owner groupid*: matches if the packet was created by a process with the given effective (numerical) group id.

- *--pid-owner processid*: matches if the packet was created by a process with the given process id.

- *--sid-owner sessionid*: matches if the packet was created by a process in the given session group.

The first two match targets that allow matching against user or group ID of the process that sent the packet, are not suitable for our needs, since the same user (or even the same group) could have multiple jobs executed concurrently with different communication resource requirements.

The last one that matches against the session ID of the process sending the packet seems to be the ideal choice: with AEM forking a user process and giving it a dedicated session ID that are shared by all forked children processes, matching a packet against the job that sent it is easy. However, any job process (that is not the session leader, i.e. the initial process) can call the system call *setsid()*, which assigns the calling process (logically belonging to a job with the current session ID) a new session ID, thus allowing it to skip packet filtering by AFW.

The only solution (that does not require providing a dedicated match target for AFW) is therefore to use the *--pid-owner* target with one rule for each process belonging to a given job in order to match all processes in a job.

2.3.2 Matching packets against allowed resources

Now that packets belonging to a job have been successfully matched, we need to match their destination addresses against allowed resources: if at least one of the allowed resources matches the packet, the packet may be sent to the network. Otherwise it is logged and dropped.

Standard packet field matches are used to this purpose:

- protocol match target (i.e. *-p tcp*) is used to match the desired transport protocol (currently TCP and UDP are supported by AFW),
- the destination address (i.e. *-d 201.123.123.012*) and the destination port (i.e. *-dport 8080*) match targets are used to match the destination socket address.

2.4 AFW Filtering Rules

Based on the above description and decisions, the AFW filtering rules will be organised as follows:

1. for each job with ID <*J*>, a new user-defined per-job chain, named *xos-job-*<*J*>, will be created.
2. for each communication resource specified by the job requirements, a new rule is added to the aforementioned per-job chain, allowing the packet matching the specified protocol, and destination address and port to be sent over the network. All other packets that are directed to the per-job chain are dropped.

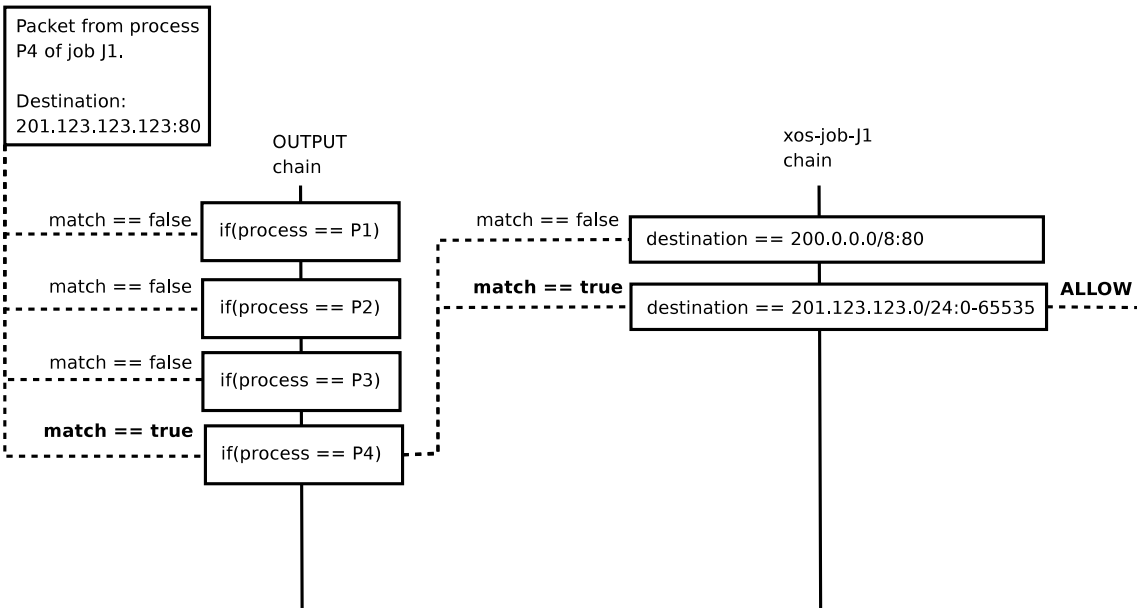

Figure 1: Path of a packet through the AFW filtering rules.

3. for each process with process ID <*P*> belonging to job with ID <*J*>, a new rule will be inserted in the *OUTPUT* chain, directing packets originating from process <*P*> to the per-job chain *xos-job-*<*J*>.

Figure 1 shows path of the packet through the iptables chains.

2.5 Description of Communication Resources

The allowed communication resources must be described in policy files stating capabilities of the node or VO. Communication resources required by a job on the other hand must be stated in JSDL files containing the job requirements. The complete schema and examples of appropriate JSDL extension as well as XACML policy describing communication resources are provided in appendix A. Due to complex matching requirements (IP matched against netmask, matching of two port ranges), common matching functions provided by XACML core are not sufficient for port and netmask matching and they have to be provided by the XACML library.

2.6 Integration with AEM

AFW does not initiate any actions on its own — it must be driven by AEM in order to respond to the following events, adjusting the filtering rules as required.

The following events detected by AEM should result in invoking appropriate actions by means of the AFW public API.

- *start of a new job*:
  - create the per-job filtering chain, and
  - add matching rules to the per-job chain as specified by the required communication resources listed in the job requirements document (refer to section 2.4).
- *creation of a new process in the scope of a job* (either the initial job process or due to fork of one of the existing job processes): add a rule to the *OUTPUT* chain, directing packets originating from the newly created process (matching is performed on PID basis as explained in section 2.3).
- *death of a process belonging to a job*: remove the rule for this process' PID from the *OUTPUT* chain. Furthermore, if this is the last process of the job (i.e. the job ends), remove the per-job chain.

Considering that a job can be executed VO-wide, while the AFW operates on each resource (node), the above job lifecycle can be interpreted more narrowly. The start of a new job, from the node's point of view, occurs whenever a portion of a job is about to be executed on a local node (e.g., a resumption of a checkpointed job). Similarly, if the job stops executing on the node, but is still being executed elsewhere, or a checkpoint request arrives, the Execution Manager should perform the clean-up, removing the local per-job chain.

The public API of the AFW is presented in appendix B

Interactions in the Grid when a user submits a job are given in figure 2: Job Manager, VOPS and Resource Manager are global entities, implemented in a distributed manner, whereas Execution Manager and Application Firewall are local to the selected resource (i.e. the node where the job executes).

3 Implementation of the Application Firewall

3.1 Implementation

Application Firewall is implemented as a Java library that will be used by AEM invoking appropriate methods, as described in section 2.6. Its public API is presented in appendix B.

The iptables kernel subsystem is used to provide the actual packet filtering. The AFW Java library uses the iptables executable provided on GNU/Linux systems to change the filtering rules on the system it runs as. As changing filtering

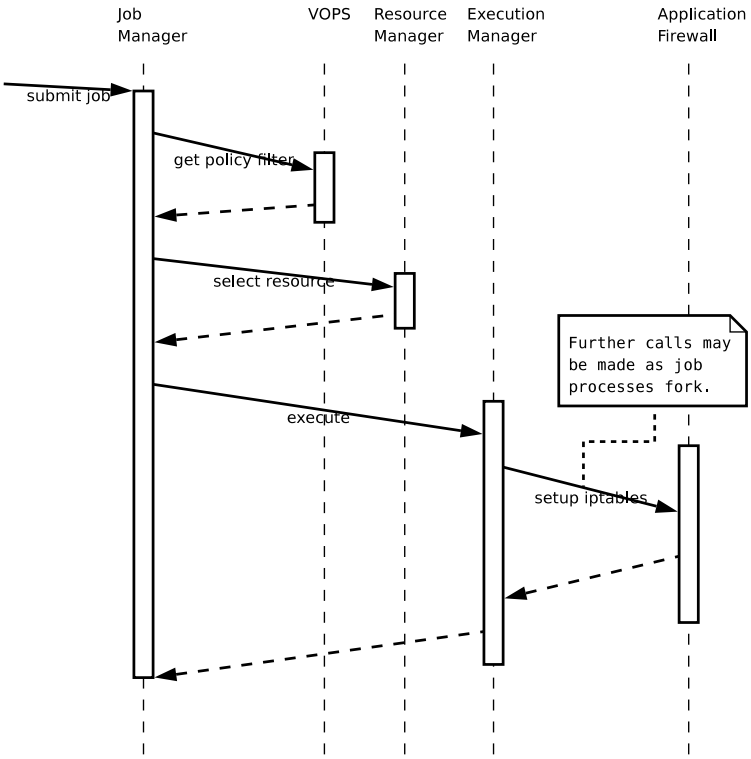

Figure 2: Component interactions during a job submission using AFW.

rules requires superuser privileges, the user with whose privileges the Java Virtual Machine containing the AEM runs, should have sudo configured to allow execution of iptables executable with superuser privileges, without requiring a password. Therefore, it is strongly advised to run AEM as a dedicated user with disabled shell login.

Figure 3 shows the implementation of firewalling used by AEM on the node where a job executes.

3.2 Installation Requirements

Each node where AEM using AFW runs, should have the following software installed:

- *iptables* kernel subsystem either built in the kernel statically or in the form of a loadable module for the running kernel.
- *iptables userspace software*: the *iptables* executable is required.
- *sudo* command.

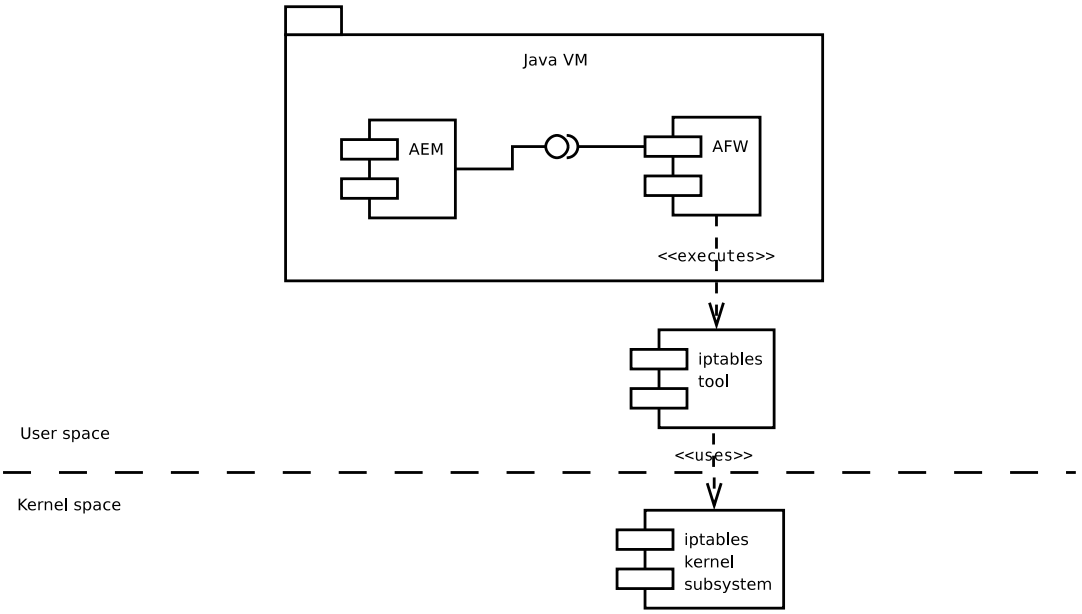

Figure 3: AEM and AFW on the job execution node.

Furthermore, *sudo* should be configured in such a manner that the user running the Java Virtual Machine containing the AEM can run *iptables* executable with superuser privileges *without having to enter the password*. The latter can be accomplished by the sample */etc/sudoers* sudo configuration file (assuming that the user aem is the one running the Java VM) presented in figure 4:

4 Conclusion

In the distributed job execution of the user-provided jobs and processes, the processes may need to communicate with remote processes. To do that, they implement their own protocols that use TCP or UDP for initiating transfers, sending or receiving data. Considering these channels are vital for the operation of the Grid infrastructure and is often required by the owners of the hosting nodes, the communication can easily be seen as a resource very much akin to the processing time and the data storage. Further, the unchecked network traffic can provide malicious software an easy way to hinder or prevent normal operation of the system.

In this document we have provided the design and specification of the Application Firewall (AFW). The AFW is a module that provides a means to enforce the node-level or VO-level policies related to the network traffic. It uses /it iptables kernel framework to set up the chains of rules, which filter the network traffic. The filtering occurs on the process level and can thus provide each process its custom

```
# /etc/sudoers

Defaults      env_reset

# Host alias specification

# User alias specification

# Cmnd alias specification
Cmnd_Alias    AEM=/sbin/iptables

# User privilege specification
aem           ALL=NOPASSWD: AEM
```

Figure 4: Sample contents of */etc/sudoers* file.

set of rules for denying or granting traffic. The parameters of the traffic include the source/target address, the port number and the protocol.

When integrated with the AEM, the AFW can provide the network filtering for each job separately. AEM will use the AFW to set up the rule chain specific for a job once this job starts on the node. It will notify AFW with the information on the processes spawned by the job's processes in order to have AFW include the new processes into the rule chain. By default the network traffic is disabled for the job's processes. They are then enabled or disabled according to the requirements specified by the job's JSDL document. It is up to AEM or other services to ensure the rules in JSDL conform with those of the local or VO-level policies.

With this document, the design of the AFW is complete and ready for integration with the rest of the XtreemOS services. We will test the module extensively and provide the reports of its performance or any modifications in the future work-package deliverables.

References

[1] XtreemOS Consortium. *D3.2.4 - Design and specification of a prototype service discovery system*, December 2007.

[2] XtreemOS Consortium. *D3.3.3 - Basic services for application submission, control and checkpointing, D3.3.4 - Basic service for resource selection, allocation and monitoring*, December 2007.

[3] XtreemOS Consortium. *D3.5.4 - Second draft specification of XtreemOS security services*, December 2007.

[4] JSDL specification, 2005.

[5] The Linux netfilter/iptables project homepage, 2008.

A Communication Resource Definition

In this appendix the schema for JSDL extension for communication resources, and an example of a JSDL declaration of job requirements and XACML declaration of node policy are presented.

A.1 Schema

Line breaks are sometimes added in order to improve readability.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:jsdl="http://schemas.ggf.org/jsdl/2005/11/jsdl"
  xmlns:tns="http://xtreemos.org/schemas/jsdl/net/200805"
  targetNamespace="http://xtreemos.org/schemas/jsdl/net/200805"
  elementFormDefault="qualified"
  attributeFormDefault="qualified">

  <xsd:complexType name="Network">

    <xsd:sequence>

      <xsd:element name="Netmask">
        <xsd:simpleType>
          <xsd:restriction base="xsd:string">
            <xsd:pattern
              value="[0-9]{1,3}.[0-9]{1,3}.[0-9]{1,3}.
                     [0-9]{1,3}(/[0-9]{1,3})?"/>
          </xsd:restriction>
        </xsd:simpleType>
      </xsd:element>

      <xsd:element name="Ports">
        <xsd:simpleType>
          <xsd:restriction base="xsd:string">
            <xsd:pattern
              value="[0-9]{1,5}(-[0-9]{1,5})?
                     (,[0-9]{1,5}(-[0-9]{1,5})?)*"/>
          </xsd:restriction>
        </xsd:simpleType>
      </xsd:element>
```

```
      <xsd:element name="Proto">
        <xsd:simpleType>
          <xsd:restriction base="xsd:string">
            <xsd:enumeration value="UDP"/>
            <xsd:enumeration value="TCP"/>
          </xsd:restriction>
        </xsd:simpleType>
      </xsd:element>

    </xsd:sequence>

  </xsd:complexType>

  <xsd:element name="Network" type="tns:Network"/>

</xsd:schema>
```

A.2 JSDL Example

An example of JSDL document fragment stating requirements for connection to any host in the address range 201.123.123.1 – 201.123.123.255, to any port in the range 1000 – 2000, or 60000 or 65000, with TCP protocol.

```
<jsdl:Resources>
  <network:Network
    xmlns:net="http://xtreemos.org/schemas/jsdl/net/200805">
    <net:Netmask>201.123.123.0/24</net:Netmask>
    <net:Ports>1000-2000,60000,65000</net:Ports>
    <net:Proto>TCP</net:Proto>
  </net:Network>
</jsdl:Resources>
```

A.3 XACML Example

An example of XACML document fragment stating that the policy allows jobs to connect to any host in the address range 201.123.123.1 – 201.123.123.255, to any port in the range 1000 – 2000, or 60000 or 65000 with TCP or UDP protocol. Some attribute values have been abbreviated (using "...") in order to improve readability.

```
<Rule RuleId="PermitTraffic" Effect="Permit">
  <Description>
    Rule for permitting TCP and UDP traffic to
    port range 1000-2000, 60000, 65000 for
    201.123.123.0/24.
  </Description>
  <Target>
    <Subjects>
      <AnySubject/>
    </Subjects>
    <Resources>
      <AnyResource/>
    </Resources>
    <Actions>
      <Action>
        <ActionMatch MatchId="...:string-equal">
          <AttributeValue DataType="...#string">
            action:AEM:SubmitJob
          </AttributeValue>
          <ActionAttributeDesignator
            AttributeId="...:action-id"
            DataType="...#string"/>
        </ActionMatch>
      </Action>
    </Actions>
  </Target>
  <Condition FunctionId="...:and">
    <Apply FunctionId="...:and">
      <Apply FunctionId="...:netmask-match">
        <AttributeValue DataType="...#string">
          123.456.789.0/24
        </AttributeValue>
```

```
        <Apply FunctionId="...:string-one-and-only">
          <ResourceAttributeDesignator
            AttributeId="resource:jsdl:network:netmask"
            DataType="...#string"/>
        </Apply>
      </Apply>
      <Apply FunctionId="...:protocols-match">
        <AttributeValue DataType="...#string">
          TCP,UDP
        </AttributeValue>
        <Apply FunctionId="...:string-one-and-only">
          <ResourceAttributeDesignator
            AttributeId="resource:jsdl:network:proto"
            DataType="...#string"/>
        </Apply>
      </Apply>
    </Apply>
    <Apply FunctionId="...:ports-match">
      <AttributeValue DataType="...#string">
        1000-2000,60000,65000
      </AttributeValue>
      <Apply FunctionId="...:string-one-and-only">
        <ResourceAttributeDesignator
          AttributeId="resource:jsdl:network:ports"
          DataType="...#string"/>
      </Apply>
    </Apply>
  </Condition>
</Rule>
```

B AFW Public API

This appendix presents the public API of the Application Firewall Java library.

B.1 The *Rule* Class

```
package eu.xtreemos.xosd.afw;

public class Rule {

  enum Protocol {
    TCP,
    UDP
  };

  enum Policy {
    ACCEPT,
    REJECT,
    DROP
  };

  public Rule(Protocol proto, String dstAddress,
    String dstPortRange, Policy policy);

  public Protocol getProto();

  public String getDstAddress();

  public String getDstPortRange();

  public Policy getPolicy();

}
```

B.2 The *IFirewall* Interface

```
package eu.xtreemos.xosd.afw;

/**
 * IFirewall is an interface for application firewall used to
 * prevent outgoing network traffic to jobs run via AEM.
 *
 * Various implementations of this interface provide different
 * firewalling backends to actually perform firewalling.
 * Netfilter/iptables implementation is currently the only
 * planned one.
 *
 * @author jaka@xlab.si
 */
public interface IFirewall {

  /**
   * Sets up firewall rules for the job processes. Called
   * when a job is started.
   *
   * @param jobId id of the new job.
   * @param rules a set of rules to be applied for the new
   * job. Can be an array of size 0, if only the default
   * policy should be used.
   * @param default policy, applied to packets, if no rules
   * match. Use DROP if no outgoing traffic should be allowed.
   *
   * @throws FirewallException if firewall rules for the
   * given jobId already exist, or an error occured.
   */
  public void setUpJobFirewall(String jobId, Rule[] rules,
    Rule.Policy defaultPolicy) throws FirewallException;

  /**
   * Removes firewall rules for the job processes. Called
   * when a job finishes.
   *
   * @param jobId the id of the job that finished.
   *
```

```
     * @throws FirewallException if no firewall rules for
     * the given jobId exist, or an error occured.
     */
    public void discardJobFirewall(String jobId)
      throws FirewallException;

    /**
     * Adds a rule that matches packets from the process with
     * PID pid against the job firewall rules. Called when a
     * new process of a job is created.
     *
     * @param jobId the id of the job.
     * @param pid PID of the new process belonging to that
     * job.
     *
     * @throws FirewallException if this PID was already a part
     * of the job, or an error occured.
     */
    public void addJobProcess(String jobId, String pid)
      throws FirewallException;

    /**
     * Removes a rule that matches packets from the process
     * with PID pid against the job firewall rules. Called
     * when a process of a job ends.
     *
     * @param jobId the id of the job.
     * @param pid PID of the job process that ended.
     *
     * @throws IllegalStateException if this PID was not a part
     * of the job, or an error occured.
     */
    public void removeJobProcess(String jobId, String pid)
      throws FirewallException;

}
```