4. Xweb Implementation and Experiments: Web Proxy-caches, Chat Server and Servlet Application Networks Deployment

To verify the hypotheses made at chapter 3, it is necessary to implement a prototype and conduct experiments on it. Since it is not possible to test how it deploys every kind of application network. It has been selected three application network types with very different characteristics. Those three kinds of application networks will be deployed by the prototype. This chapter describes how it has been implemented a basic programmable infrastructure that permits to compose application networks, and deployment manager and resource agents that implement building blocks to dynamically deploy application network. Nexts it is described experiments carried out with such prototype, and measurements made to evaluate its temporal response. Finally it is concluded which hypothesis have been verified and additional requirements found.
4.1. Xweb Programmable Internet Service Infrastructure Implementation

As explained in chapter 3 this infrastructure must provide shared resources; support virtualization so that multiple applications can coexist simultaneously; and must provide an execution environment so that new applications can be introduced. Some of the programmable infrastructures at chapter 2 were considered initially as Xweb programmable Internet service infrastructure. However they presented two problems that made us not to use any of them. In first place they were domain specific and fairly complex, with much functionality that we did not required. In second place, none provided all functionality that we need, specially support to create connections among application instances in different nodes and to coordinate those instances. Therefore we decided to build a basic infrastructure on which to compose application networks.

Resources required by applications networks

Xweb was built using PC boxes with off-the-shelf CPU, RAM memory, hard-drives and Ethernet cards. Those machines were connected by a local area network attached to the Internet. Each node provides network storage and processing resources. Respective properties of those resources are: for network connectivity its IP identifier, and adjacent network regions where service can be provided. Though all nodes are connected to the same subnet, they were manually configured as being adjacent to different AS. Resource constraints are maximum bandwidth available, and maximum storage, which again were manually configured.

<table>
<thead>
<tr>
<th>Resource Type</th>
<th>Resource Properties</th>
<th>Resource Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>Net Conectivity</td>
<td>IP, Nearby Regions (#AS)</td>
<td>Max BW (Mbits)</td>
</tr>
<tr>
<td>Storage</td>
<td></td>
<td>Max Storage (Mbytes)</td>
</tr>
</tbody>
</table>

*Table 4.1. Xweb resources*
Virtualization

Linux [Tor94] and TOMCAT [APA00] environments provide support for multiple simultaneous processes of several application networks. Linux is a multi-process operating system that permits multiple concurrent open network connections for several servers operation. TOMCAT is an environment that supports simultaneous execution of multiple servlet applications [SUN97], with multiple open network connections each. It takes advantage of Java Virtual Machine support for multithreading and network connections.

It has been configured manually a maximum number of simultaneous virtual applications. A value of 10 was chosen for simplicity. A maximum limit on the percentage of total resource a single application can use, sets the minimum number of applications at each node.

<table>
<thead>
<tr>
<th>Max Resources per App (for Min # applications)</th>
<th>Max # applications</th>
</tr>
</thead>
<tbody>
<tr>
<td>20 %</td>
<td>10</td>
</tr>
</tbody>
</table>

Table 4.2. Virtualization limits

Programmability

Linux and TOMCAT are the execution environments of the Xweb programmable infrastructure, therefore it is possible to deploy Linux servers and Servlet based application networks. Execution environment properties are its type and compilation capabilities.

<table>
<thead>
<tr>
<th>Execution Environments</th>
<th>Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux-i386</td>
<td>Libraries=RedHat7.2</td>
</tr>
<tr>
<td>TOMCAT</td>
<td>Libraries=Java-1.3</td>
</tr>
<tr>
<td></td>
<td>Compiler=Gcc</td>
</tr>
<tr>
<td></td>
<td>Compiler=Javac</td>
</tr>
</tbody>
</table>

Table 4.3. Execution environments properties
Fig. 4.4. Xweb programmable infrastructure with resources (Storage & Net BW) and execution environments (TOMCAT, LINUX), resource agents (RA) and deployment managers (DplyMgr)
4.2. Deployment Manager Implementation

Deployment managers provide an interface for service providers to input application network specifications. This interface is implemented by a number of Web forms provided by a web server. Web scripts providing those forms communicate with deployment managers. Deployment managers also discover and monitor resources agents using multicast expanded ring searches, and keep such information in a resource availability table.

To deploy an application network deployment managers perform a resource mapping function that creates a deployment plan. This plan involves resource allocation, code distribution and service composition. Deployment managers command resource agents to perform required actions.

![Deployment Manager Architecture Diagram]

The user interface and resource discovery functionality is built reusing Xbone overlay manager user interface and resource discovery implementation [Tou01]. Also the protocol between deployment managers and resource agents is based on the Xbone protocol.
4.2.1. Multicast Expanded Ring Search for Resource Discovery & Monitoring Implementation

Deployment managers require a resource discovery phase to find out resource nodes. Xweb uses multicast expanded ring search Xbone implementation. To maintain such information up to date Xweb permanently monitors every resource node. It has the advantage of faster local access to such information, however at the expenses of maintaining a table of resource availability information and creating monitoring traffic. Such operation is also implemented using multicast expanded ring search, with the intention that new resources are discovered at the same time.

Resource agents listen to multicast address 224.192.0.1 waiting for resource availability requests. Resource discovery messages exchanged between deployment managers and resource agents are sketched in fig.4.6. Deployment manager can send multicast packets with different TTL values. Higher TTL value packets reach farther apart nodes, however more network traffic is created. Therefore packets sending rate should be inversely proportional to its TTL value. Status of nodes nearer to a deployment manager will be more up to date than that of farther apart nodes.

Resource agents implement mechanisms to discover its local resources properties and status, and keep it up to date, which will be described at section 4.4.1. Resources agents return deployment managers a list of its resources, and its characteristics. Responses are only sent to trusted managers in a unicast secure SSL connection to the requesting node. Deployment managers create with this data a resource availability table to be used to effectuate resource mapping. An example of a resource availability table is at table 4.5.
Deployment Manager

Mcast

Resource Agent (pc1)

Resource Agent (pc2)

Resource Agent (avant)

Sec Unicast

Xweb ResourceDiscovery Request
ID = mosaic.ac.upc.es

Xweb ResourceDiscovery Response
ID = pc1.ac.upc.es
NetworkCapacity = 10 Mbits
ServiceRegions = 1300, 1200, 1100, 3110
StorageCapacity = 1000 Mbytes
ExecutionEnvironment = Linux, TOMCAT
MaxNumApps = 10

Sec Unicast

Xweb ResourceDiscovery Response
ID = pc2.ac.upc.es
NetworkCapacity = 10 Mbits
ServiceRegions = 300, 1200, 1100
StorageCapacity = 4000 Mbytes
ExecutionEnvironment = Linux
MaxNumApps = 10

Sec Unicast

Xweb ResourceDiscovery Response
ID = avant.ac.upc.es
NetworkCapacity = 10 Mbits
ServiceRegions = 1300, 200, 3110
StorageCapacity = 10000 Mbytes
ExecutionEnvironment = Linux, TOMCAT
MaxNumApps = 20

Fig. 4.6. Xweb resource discovery messages
### Table 4.5. Resource availability

<table>
<thead>
<tr>
<th>NodeID</th>
<th><strong>Net BW Max Capacity (Mbits), Nearby Regions(#AS):</strong></th>
<th><strong>Storage (Mbytes):</strong></th>
<th><strong>Max # Apps; Min Resources per App:</strong></th>
<th><strong>Execution Environment, Libraries:</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>pc1</td>
<td>10, 1300-1200-100-200;</td>
<td>1000;</td>
<td>10; 20%;</td>
<td>Linux-i386, RedHat7.2; TOMCAT, Java1.3</td>
</tr>
<tr>
<td>pc2</td>
<td>10, 1300-1200-100;</td>
<td>1000;</td>
<td>10; 20%;</td>
<td>Linux-i386, RedHat7.2; TOMCAT, Java1.3</td>
</tr>
<tr>
<td>mosaic</td>
<td>10, 1300-200-1200;</td>
<td>4000;</td>
<td>10; 20%;</td>
<td>Linux-i386, RedHat7.2; TOMCAT, Java1.3</td>
</tr>
</tbody>
</table>

4.2.2. **Deployment Manager Web-based Interface Implementation**

As commented deployment manager user interface is based on Xbone overlay manager interface. Application providers access deployment managers to request deployment of new application networks through a web-based interface. A web interface permits several service providers access to it from remote locations. This web interface is made up of a number of CGI perl scripts that create application network deployment request from data input in web forms. Scripts send those requests to deployment managers, which are executing on the same machines as the web server. Once the application network has been deployed, deployment manager will reply the POST request with an OK/ not OK deployment process finished.
Web form at figure 4.8. shows which specifications requirements are supported by deployment manager implementation. Resource requirements supported are: execution environment that is determined by type of service, minimum storage size, total network traffic generated, and maximum distance to server; demand is specified as a list of AS regions; service wide constraints are maximum number of edge servers. Besides service creators input other properties of the application network such as its name, code location, start time, and service duration. As well service providers are given the option to adjust some deployment parameters such as code distribution mechanism and redeployment interval.
Fig. 4.8. Service providers fill in deployment interface
4.2.3. Resource Mapping Algorithm Implementation

The resource mapping algorithm implementation is a simplified version of that proposed in section 3.2, since in our small-scale prototype the more complex algorithm could make little optimizations. The algorithm implemented by the deployment manager, shown in figure 4.9., selects for each demand region a resource node that has enough resources among those that are nearer to the region. This algorithm does not optimize number of resource locations since for each demand region it chooses one location, but it provides a good service since each demand region has one server that is nearby and has enough resources. The connected placement algorithm is also simplified; edge servers are connected in a tree whose root is the node not been selected for any region.

Fig. 4.9. "pseudo-code" for Xweb resource mapping placement algorithm

4.2.4. Deployment Protocol and Transactional Control Implementation

Deployment protocol contains several command transmitted from deployment managers to resource agents over a reliable connection. It has been implemented three main commands shown at table 4.5. Allocation commands are used to reserve an amount of required resources for an application. Installation commands are used so that resource agents download application code and install it. Activation commands are used to configure communication channels and start the service. Each command also needs some parameters to be sent to resource agents. Allocation commands need a storage size and network bandwidth parameters, which indicate quantity of each resources needs to be allocated. Installation
commands need a code location from which to download application code. Execution commands need a list of application instances to connect to, and a list of coordination rules.

<table>
<thead>
<tr>
<th>Type</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Allocation</td>
<td>ApplicationId, StorageRequirements, NetworkRequirements</td>
</tr>
<tr>
<td>Installation</td>
<td>ApplicationId, CodeLocation</td>
</tr>
<tr>
<td>Activation</td>
<td>ApplicationId, PeerServersList, CoordinationRules</td>
</tr>
</tbody>
</table>

*Table 4.6. Deployment commands*

So that deployment is executed as a transactional operation, a deployment control table is required which stores every command sent, its identifier and its status. Deployment is not completed until every command has been correctly executed. Deployment managers implement a timeout that triggers a deployment checking routine. It checks whether every command has been correctly executed. Else a deployment cancellation is started that releases resources and stops servers started. At table 4.7. it is show an example of a deployment control table.

<table>
<thead>
<tr>
<th>AgentID</th>
<th>Command Sent</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>pc1</td>
<td>122, “Allocate appl.com 200 Mbytes, 1Mbit”</td>
<td>OK</td>
</tr>
<tr>
<td></td>
<td>125, &quot;Install appl.com avant/squid.rpm&quot;</td>
<td>pending</td>
</tr>
<tr>
<td></td>
<td>128, “Activate appl.com”</td>
<td>pending</td>
</tr>
<tr>
<td>pc2</td>
<td>123, &quot;Allocate appl.com 200 Mbytes, 1Mbit”</td>
<td>OK</td>
</tr>
<tr>
<td></td>
<td>126, &quot;Install appl.com avant/squid.rpm&quot;</td>
<td>pending</td>
</tr>
<tr>
<td></td>
<td>129, “Activate appl.com Parent=pc1”</td>
<td>OK</td>
</tr>
<tr>
<td>Mosaic</td>
<td>124, “Allocate 200 Mbytes, 1Mbit”</td>
<td>OK</td>
</tr>
<tr>
<td></td>
<td>127, &quot;Install appl.com avant/squid.rpm&quot;</td>
<td>pending</td>
</tr>
<tr>
<td></td>
<td>130, “Activate appl.com Parent=pc1”</td>
<td>pending</td>
</tr>
</tbody>
</table>

*Table 4.7. Deployment process control table example*

4.2.5. Deployment Manager Strategy
Xweb deployment managers can be configured to provide different deployment strategies: which type of application they can deploy, how these applications are configured and how they are deployed. Deployment managers have a web interface where configurations are input fig. 4.10. Concerning deployment parameters it has to be configure which is the maximum multicast search radius to be used to discover resources. And it has to configure which timeout will have to be waited before sending a deployment cancellation request. Concerning application parameters, deployment managers are configured on which service types they are going to deploy, and the address of a server to behave as redirector server for server-based coordinated application networks, (a future improvement is to deploy this server too).

Plus each deployment request can adjust some deployment parameters: code distribution mechanism, and modification interval, as shown at previous figure 4.8.

![Local Deployment Config - Microsoft Internet Explorer](image)

**Fig. 4.10. Xweb deployment manager configuration interface**
4.3. Resource Agents Implementation

Resource agents inform deployment managers on resource availability status and carry out deployment managers commands for resource allocation, code distribution and service composition functionality. Resource agents implement a mechanism to obtain resource availability information to be sent to deployment managers. Resource agents implement mechanisms to allocate resources, to obtain application code and install it, to configure application communications and coordination rules, and to perform resource binding and starting the program.

Resource agents’ implementation of interface with deployment managers is based on Xbone resource daemons communication module.

Fig. 4.11. Resource agents internal architecture
4.3.1. Resource Availability Implementation

In the develop prototype status of most resources is being provided through manual static configuration, none of the propose mechanisms at section 3.4 was implemented though they were investigated. Resource agents are configured with local resources available for service activation: service storage capacity (in Mbytes), service traffic capacity (in Mbits), nearer regions where service can be provided (AS numbers), maximum number of simultaneous applications, execution environments types (Operation system, Virtual Machine).

Bandwidth probes [Iai99], load monitors and storage monitors should be used to obtain resources availability information. Network topology adjacent to resources could be obtained with network topology discovery tools such as traceroute [Jac89] or Mercator [Gov00] that discovers any routing node and its relation to other routers using a combination of hop limited probes, proactive probes and several heuristics.

![Fig. 4.12. Xweb Resource Availability configuration](image-url)
4.3.2. Local Resource Allocator Implementation

Resource agents keep a list of allocated and free local resources. At our implementation there are two allocation tables for storage resources and network bandwidth capacity. On receiving an allocation command, resource agents check if they have available resources to deploy a new service in its node. If so they proceed to decrease free resource table and increase allocated resources table.

With this implementation, resources will be provided only to a limited number of competing applications. However if some application executing in a node tries to use more resources that allocated, it will be able to do so. So that an application is only able to use only those resource allocated to it, it is required some mechanism to enforce strict partitioning so that different applications do not interfere with each other; mechanisms which provide this capability are resource containers [Ban99], cluster reserves [Aro00], Qlinux [Sun00] and QoS Web servers [Pan98] [Egg99].

4.3.3. Code Distribution Implementation

Xweb implements a code distribution mechanism based on HTTP. Deployment managers send resource agents a code server location and code file name as a URL. Resource agents download service software from such location through an HTTP Get. Application code is distributed as RPM packages or WAR archives.
4.3.4. Server Installation Implementation

Binding servers code to permanent storage is known as server installation. It comprises writing application code files to permanent storage. Squid and IRCd servers are installed using “RPM” utility [Ewi96]. Such program accepts Linux RPM package archives that contain all executables, libraries, configuration files and dependencies to execute a program. If WAR files are placed in a directory location, TOMCAT automatically, creates a hierarchy of directory and places Servlet class files.

4.3.5. Server Activation and other Resource Bindings Implementation

Resource binding for most program takes place inside program code, however some applications as squid proxy-cache perform a preliminary independent step to create local
directories for caching files. For example configuring squid as figure 4.15., it is created a
cache storage of size 100 Mbytes partitioning in 16 directories and 256 subdirectories.

<table>
<thead>
<tr>
<th>Squid Configuration file</th>
</tr>
</thead>
<tbody>
<tr>
<td>cache_dir /var/spool/squid 100 16 256</td>
</tr>
</tbody>
</table>

Fig. 4.14. Squid initial resource binding configuration

Resource agents check that configured parameters are within a correct range and with
adequate execution rights prior to spawn service process. Since most application service can
be executed in Unix operating system, we used Unix `exec()` function [Ker84] to implement
service activation. Because we do not modified service programs, Xweb implements dynamic
resource binding halting and restarting services. Service programs are stopped, read its new
bindings from a configuration file, and restart binding to them. Squid proxy caching server
can be configure to use a limited amount of RAM memory, and an specified network address
and port for client incoming requests, fig 4.15.

<table>
<thead>
<tr>
<th>Squid Configuration file</th>
</tr>
</thead>
<tbody>
<tr>
<td>cache_mem 8 MB</td>
</tr>
<tr>
<td>tcp_incoming_address pcl.ac.upc.es</td>
</tr>
<tr>
<td>http_port 3128</td>
</tr>
</tbody>
</table>

Fig 4.15. Squid main resource binding configuration

TOMCAT also loads and executes automatically any servlet at a configured file
location as soon as a request for such servlet is received. As well any servlet bytecode file that
has being modified since last loaded, are automatically reloaded. TOMCAT has to be initiated
at resource agents’ initialisation time.

4.3.6. Dynamic InterServer Communication Channels Implementation

Connectivity among application instances takes place through TCP connections established at
server start time from configuration read in a file, in fig. 4.16. squid proxy cache
squid1.net1 is configure to communicate with a proxy cache at squid2.net1 at port 3128 and 3120. In another node squid2.net1 is configured to listen at those ports.

Again we have not modified service programs. To modify communication channels among servers, configuration files need to be changed and applications need to be halted, read its configuration file, and restarted creating a new communication channel. Fortunately squid proxy cache can be made to re-read its configuration file without stopping it.

### 4.3.7. Per Servers Coordination Implementation

Coordination at the Xweb is application dependant; web proxy caches and chat servers are configured per node, whereas servlet application networks use a centralized coordination server. In figure 4.16, proxy cache squid1.net1 is configured so that squid2.net1 is its parent cache, and squid3.net1 is one of its sibling caches. Web request it cannot resolve will be forwarded first to its sibling cache peers then to its parent cache. 3128 and 3120 are TCP and UDP ports where sibling and parent caches listen for communication.

Again however most applications can only be configured at start time on which will be their parents or peer servers, therefore resource agents required to stop, reconfigure and restart servers to modify coordination rules

<table>
<thead>
<tr>
<th>Configuration file for squid1.net1</th>
<th>Configuration file for squid2.net1</th>
</tr>
</thead>
<tbody>
<tr>
<td>tcp_outgoing_address squid1.net1</td>
<td>tcp_incoming_address squid2.net1</td>
</tr>
<tr>
<td>cache_peer squid2.net1 parent 3128 3130</td>
<td>http_port 3128</td>
</tr>
<tr>
<td>cache_peer squid3.net1 sibling 3128 3130</td>
<td>icp_port 3130</td>
</tr>
</tbody>
</table>

Fig 4.16. Squid communication channels and coordination rules

### 4.3.8. Centralized Coordination Server Implementation

Servlet application network used an HTTP server with a script that redirected requests as its coordination server. Deployment managers configured redirection rules. They commanded a special agent in such a node as shown in figure 4.17.
Fig. 4.17. Resource agent and central coordinator server configuration
A servlet application network and its redirector are shown at figure 4.18. Redirection rules shown redirect clients from origin addresses 136.0.0.0 to servlet executing at mosaic, those coming from addresses 137.0.0.0 are redirected to servlet executing at pc1, and so on.

Fig.4.18. Deployment of a central coordination server based application network
4.4. Security Measures Implementation

4.4.1. X509 certificates

Resource, application code and deployment managers are authenticated by a certification authority whose code is included in OpenSSL libraries [Ope99]. They are provided with a certificate to be used for trusted third party authentication when communicating among them. Resources authenticate to deployment managers that they are providing resource availability information and to be allowed permission to execute an application; also they authenticate to code server to download application code. Application code is authenticated when downloaded from a server code that has been certified. Deployment managers need to authenticate to resources to request resource availability, and to perform deployment operation: allocate resources, install and activate applications.

4.4.2. SSL communication

SSL communication channels are implemented with OpenSSL crypto libraries [Ope99]. OpenSSL provides integrity and secrecy for communication of service specifications messages, deployment operations messages and application code transfers.

Resource availability integrity and secrecy has only been implemented at resource agents to deployment managers’ communications. Communications from deployment managers to resource agents takes place through unsecured multicast messages, therefore any node can know who is requesting resource availability by wiretapping.

Deployment manager communicate with resource agents through a secure SSL connection. In this way only authenticated deployment manager can perform operations over resources. As well deployment manager only deploy application networks over authenticated
nodes. Both parties are certified by a certification authority and accept connection only from nodes certified by it.

4.4.3. Java Sandbox

Malicious programs or badly behaving programs (because of programmer errors or programs exceptions not caught correctly) can make an uncontrolled usage of resources. It denies resource to other well-behaving programs. Only an execution environment that controls how much resources are being consumed by each program, and kills or delay execution of them can avoid this. One execution environment that can provide such measures is Java Virtual Machine SandBox [Gon97]. It checks every instruction in Java compiled bytecode programs before execution. The sandbox determines which operations a program can perform: reading local files, connected to network sites. Xweb incorporates Java Virtual Machine as one of its execution environments.

4.4.4. Other measures not implemented

Secure multicast communications has not been implemented since it solves only a minor security thread as commented. There exist several implementations with various level of performance and functionality [Mit97].

Non-disclosure of resource properties (resource secrecy) is not possible with Xweb framework. However multicast injection implementation explained in chpt. 5 does not required resource availability messages between deployment managers and resource agents, and not between resource agents either. Therefore resources need not to publish its properties, so there is no disclosure of resource properties.

Malicious resources nodes (even authenticated) can read and modify program execution unless using techniques such as encrypted programs execution [San97], however this are far from being a reality.
As well resources agents can report at resource discovery time maliciously or by error certain capabilities that are not certain. This can only be noticed if a program is executed at them. A resource a testing period could identify such erroneous nodes.

<table>
<thead>
<tr>
<th>Vulnerable Element</th>
<th>Measures Implemented</th>
</tr>
</thead>
</table>
| Resources          | • Install and execute application code downloaded from secure server.  
                    | • Java Virtual Machine, only for Java App. |
| Communication      | • X509 based parties’ authentication.  
                    | • SSL TCP connections.  
                    | • (Certified UDP multicast packets.)  
                    | • (Resource secrecy using multicast injection.) |
| Applications       | • Certified resources are allowed to download and, execute applications.  
                    | • (Resource testing period.)  
                    | • (Encrypted Code.) |

*Table 4.8. Security Measures Implemented*
4.5. Experiments Setup

4.5.1. Infrastructure Resources and Execution Environments

Xweb prototype uses four Linux machines from the ac.upc.es domain: avant, mosaic, pc1 and pc2. Computers mosaic, pc1 and pc2 constitute the programmable Internet service infrastructure. Each node had a Pentium IV processor with 10Mbits LAN Internet connectivity and 1 or 4 Gbyte storage capacity. They run Linux RedHat 7.2 and a TOMCAT container with JDK1.3. A resource agent was installed on them. Resource agents were configured network bandwidth capacity, storage capacity, nearby regions, maximum applications per node as shown in table 2. In avant it was installed a deployment manager. It was not used a code server, applications code was downloaded from Internet web servers to measure code distribution in a real setting.

<table>
<thead>
<tr>
<th>NodeID</th>
<th>Net BW (Mbits), Nearby Regions(#AS):</th>
<th>Storage (Mbytes):</th>
<th>Max # Apps:</th>
<th>Execution Environment, Libraries:</th>
</tr>
</thead>
<tbody>
<tr>
<td>pc1</td>
<td>10, 1300-1200-100-200;</td>
<td>1000;</td>
<td>10;</td>
<td>Linux-i386, RedHat7.2; TOMCAT, Java1.3</td>
</tr>
<tr>
<td>pc2</td>
<td>10, 1300-1200-100;</td>
<td>1000;</td>
<td>10;</td>
<td>Linux-i386, RedHat7.2; TOMCAT, Java1.3</td>
</tr>
<tr>
<td>Mosaic</td>
<td>10, 1300-200-1200;</td>
<td>4000;</td>
<td>10;</td>
<td>Linux-i386, RedHat7.2; TOMCAT, Java1.3</td>
</tr>
</tbody>
</table>

*Table 4.9. Prototype Resources*
4.5.2. Specifications of Application Networks to be deployed

There were deployed three types of application networks: squid proxy caching hierarchies, chat servers networks and servlet-based distributed web application. Application networks specifications are described in table 4.10. Web cache hierarchy (Cache ALN) used squid proxy caching server [Wes98] and chat server networks (Chat ALN) used Internet Relay Chat server [Kal00], HelloWorld follows servlet specifications [SUN97].

<table>
<thead>
<tr>
<th>Specifications</th>
<th>Proxy-cache Hierarchy</th>
<th>Chat servers Network</th>
<th>Servlet CDN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application Code</td>
<td>squid</td>
<td>IRCd</td>
<td>HelloWorld.java</td>
</tr>
<tr>
<td>Execution Environment</td>
<td>Linux-i386</td>
<td>Linux-i386</td>
<td>TOMCAT</td>
</tr>
<tr>
<td>Per server storage</td>
<td>200 Mbytes</td>
<td>200 Kbytes</td>
<td>20 Kbytes</td>
</tr>
<tr>
<td>Per client Net BW</td>
<td>10 Kbit</td>
<td>1 Kbit</td>
<td>1 Kbit</td>
</tr>
<tr>
<td>Client session duration</td>
<td>10 sec</td>
<td>100 sec</td>
<td>1 sec</td>
</tr>
<tr>
<td>Demand Regions</td>
<td>All</td>
<td>All</td>
<td>All</td>
</tr>
<tr>
<td># clients per Region</td>
<td>100</td>
<td>100</td>
<td>1000</td>
</tr>
<tr>
<td>Constraints</td>
<td>None</td>
<td>None</td>
<td>None</td>
</tr>
</tbody>
</table>

Table 4.10. Application Networks Specifications input at experiments
4.5.3. Temporal Response Measurements

We measured temporal response to a deployment request event because we are interested in finding how quickly application networks can be deployed, and how fast application networks adapt to demand and resource availability variations. Deployment managers and resource agents were instrumented to log start and end time of every operation. Time was measured at each node using OS functions with microseconds resolution. Clocks were not synchronized since we are only interested in measuring intervals of time [Pax97]. Fig. 4.23. shows Xweb temporal response.

![Time plot of deployment action](image-url)
Xweb temporal response main phases are resource discovery, deployment plan creation and deployment. Resource discovery time $t_{\text{Resource\_Discovery}}$ initiates when a multicast resource discovery messages is sent, it ends when resource availability information from a number of nodes is saved. It is equal to the sum of discovery request transmission, availability check and response times, which tooks the longest, plus time to update resource availability table. $t_{\text{Mcast\_Request}}$ is proportional to network diameter and message size; $t_{\text{Resource\_Availability\_Check}}$ depends on agent resource availability implementation: a probe, a query or last notified value. $t_{\text{Secure\_Unicast\_Request}}$ shall be fairly long since secure SSL connections require several messages exchanges. $t_{\text{Update\_Resource\_Table}}$ is proportional to file write-time.

\[
\begin{align*}
t_{\text{Resource\_Discovery}} &= \text{MAX} \left[ t_{\text{Mcast\_Request}} + t_{\text{Resource\_Availability\_Check}} + t_{\text{Secure\_Unicast\_Response}} \right](i) + t_{\text{Update\_Resource\_Table}} \\
\end{align*}
\]

(Eq. 4.1)

Deployment plan creation time $t_{\text{Deployment\_Plan\_Creation}}$ starts when deployment managers receive specifications, it ends when connections to resource agents start. It is the sum of $t_{\text{Resource\_Mapping}}$, which is the computation time required to calculate a resource mapping and a deployment plan it will be proportional to number of resource nodes, demands areas and algorithm complexity. And $t_{\text{Update\_Application\_Table}}$, time to save deployment plan information; it is proportional to file write-time.

\[
\begin{align*}
t_{\text{Deployment\_Plan\_Creation}} &= t_{\text{Resource\_Mapping}} + t_{\text{Update\_Application\_Table}} \\
\end{align*}
\]

(Eq. 4.2.)

Total deployment time $t_{\text{Total\_Deployment}}$ starts when connections to resource agents start, and ends when latest resource agent confirms it has performed all indicated commands without errors. Therefore it is equal to the sum of command transmission, execution and response times which tooks the longest, plus time to update application control table.
\[ t_{\text{TotalDeployment}} = \text{MAX} \ [t_{\text{SecureUnicastRequest}} + t_{\text{NodeDeploymentTime}} + t_{\text{SecureUnicastResponse}} (i) + t_{\text{UpdateApplicationTable}} \] \] (Eq. 4.3.)

\[ t_{\text{NodeDeployment}} \text{ is the sum of all deployment actions duration to be performed at a resource node.} \]

\[ t_{\text{ResourceAllocation}} \text{ depends on the resource allocator implementation.} \]

\[ t_{\text{CodeDistribution}} \text{ is proportional to code size, and code server distance.} \]

\[ t_{\text{ServerInstallation}} \text{ is proportional to size and number of application code files and file write time.} \]

\[ t_{\text{ResourceBinding}} \text{ is proportional to number of resources required.} \]

\[ t_{\text{ServerActivation}} \text{ is time to spawn process,} \]

\[ t_{\text{ServerCoordination}} \text{ shall be proportional to number of peer servers.} \] (Eq. 4.4.)

4.5.4. Experiments

Experiments were carried out entering manually specification through the deployment manager web interface. For each experiment it was specified a unique name, location of server code, start time, and duration of deployment. Execution environment is fixed when specifying type of service: Linux for Squid proxy-cache and Ircd, and TOMCAT for Servlets. Maximum distance between server and service regions was set to 2 hops; it was not a decisive value since nodes are at distance one from all possible regions. Traffic capacity required is total traffic per server per region. Finally it was specified that they use a high number of servers so that application networks make use of resources in every available node. Each experiment was repeated 5 times. Average values of measurements are reported.

In figure 4.22. it is sketched how Web proxy caching application networks were deployed in the experiments. As shown they were deployed forming a hierarchy with one parent proxy cache and two children proxy caches. Chat server networks were deployed
forming a mesh topology; every chat server was peer of the others. Servlet-based distributed application network were deployed forming a content distribution network CDN with a web server behaving as redirector server, as in previous figure 4.18.

*Fig. 4.22. Squid Web proxy-caching application network deployment experiments*
4.6. Results

Experiments show whether implementation validates framework, its components and architecture. Also experiment measurements indicate how fast application network can be deployed and which components can be streamlined.

4.6.1. Infrastructure shared resources, virtualization and programmability

From one experiment to another different application network are allocated same resources at each node. Therefore infrastructure supports resource sharing among different applications in time.

Experiments to show that several application networks can coexist over the same infrastructure were not design, though experiments show that infrastructure support creation and destruction of several application networks over time.

New application networks were activated at the infrastructure because execution environments were installed at each infrastructure node. Linux permits deployment of applications already compiled for that platform. Whereas TOMCAT containers require applications to be developed following Servlets specifications, however they benefit from Java Virtual Machine security. Source code application were not deployed.

4.6.2. Multicast expanded ring search for Resource discovery and Monitoring

Since we only have a small number of machines connected by a local area network, evaluation of multicast expanded ring search is very limited. In fact experiments performed in such a local area network can only discover nodes at a distance of TTL=1. Measurements made are reported in table 4.11. $t_{\text{Resource Availability Check}}$ is a low value since at the implementation it only reads a list of program variables which provides an out-of-date value,
however it will be a larger value in a complete system since it should probe resource containers through an out-of-program mechanism. The largest overhead of $t_{\text{Resource\_Discovery}}$ is due to secure discovery responses. Therefore scalability to large number of resource nodes will not be possible, since each discovered resource will require a secure connection to be established at deployment managers, which is resource and bandwidth expensive.

| $t_{\text{Resource\_Availability\_Check}}$ | 0.5 ms |
| $t_{\text{Secure\_Unicast\_Request}}$ | 53 ms |
| $t_{\text{Update\_Resource\_Table}}$ | 2.4 ms |
| $t_{\text{Resource\_Discovery}}$ | 58.4 ms |

*Table 4.11. Resource discovery measurements values*

### 4.6.3. Specifications Input and Mapping

Specifications have been choosen so that every resource nodes was selected for deployment. Again small scale of experiments did not permit to evaluated performance of mapping algorithm in complex scenarios where computational requirements are expected to be considerable. Therefore measurements made, reported in table 4.12. have little significance, and such low values were expected.

<table>
<thead>
<tr>
<th></th>
<th>Cache ALN</th>
<th>Chat ALN</th>
<th>Servlet CDN</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_{\text{Resource_Mapping}}$</td>
<td>1.4 ms</td>
<td>1.4 ms</td>
<td>N/A</td>
</tr>
<tr>
<td>$t_{\text{Update_Application_Table}}$</td>
<td>3.4 ms</td>
<td>3.4 ms</td>
<td>N/A</td>
</tr>
<tr>
<td>$t_{\text{Deployment_Plan_Creation}}$</td>
<td>4.8 ms</td>
<td>4.8 ms</td>
<td>N/A</td>
</tr>
</tbody>
</table>

*Table 4.11. Resource Mapping Measurements*

It was found that resource requirements should not be straightforwardly extrapolated from others work. Resource mapping of such requirements into actual resources might not provide service level desired, because each resource has its own performance model that determines actual service performance for a particular resource allocation. Therefore a service characterization phase is required to find out actual resource requirements of applications on available resources.
4.6.4. Deployment Atomicity

Deployment control table main goal was to perform deployment as an atomic operation. However there were not made experiments to evaluate its robustness. Experiments where network, resource allocation or service composition failed were not designed nor performed. It is subject of future work.

4.6.5. Local Resource Allocation

Resource allocation at nodes is simply a write operation in a synchronized table. Consequently $\text{Resource Allocation}$ is very low. If resources provided strict resource partitioning with mechanisms such as resource containers [Ban99,] cluster reserves [Aro00], Qlinux [Sun00] and QoS Web servers [Pan98] [Egg99] it might be a longer operation.

4.6.6. Code Distribution

Code distribution in experiments used public Internet web servers that hosts code binaries of applications to be deployed. Therefore communications were not secured or authenticated. However $\text{Code Distribution}$ measurements were more realistic. Values obtained, table 4.12., are 1.5 seconds for chat server download, upto 10 seconds for squid proxy cache. It is proportional to application code size: squid code was 948KBytes, and Ircd was 230 Kbytes. Average transfers rate are 80Kbytes. It would decrease if a content distribution network were used to replicate code to locations near to resource node. Or if code packages were cached at each node for use by following services. Or if a basic package contained only basic code, and other add-ins could be incrementally loaded.
4.6.7. Server Installation, Execution and other Resource Bindings

RPM utility automates installation of Linux-based applications and provides several interesting options. Servlets are automatically installed and activated by TOMCAT. As expected $t_{\text{Code\_Installation}}$ is proportional to code size; at experiments it is a fairly long time, probably due to slow file write.

Binding to processor and main memory resources is performed at server execution. It involves reading several configuration files, checking various system parameters and variables and performing several operating system calls to be bound to resources; none of these operations can easily be abandoned for correct service working. Most operations are hard-coded into applications, and cannot be independently executed. $t_{\text{Server\_Activation}}$ is fairly high for applications deployed because many initialization operations are performed.

Squid proxy caching server performs binding to some resources in an isolated operation, which could be measured. It involves partitioning storage resources in a directory hierarchy where files will be cached afterwards. $t_{\text{Resource\_Binding}}$ has a high value for proxy-caching application networks due to large typical cache sizes, and slow file systems.

4.6.8. Servers Coordination

$t_{\text{Server\_Coordination}}$ is included in $t_{\text{Server\_Activation}}$ because coordination rules are read, and connections are created at service start time.

<table>
<thead>
<tr>
<th></th>
<th>Cache ALN</th>
<th>Chat ALN</th>
<th>Servlet CDN</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_{\text{Resource_Allocation}}$</td>
<td>2.1 ms</td>
<td>2.1 ms</td>
<td>N/A</td>
</tr>
<tr>
<td>$t_{\text{Code_Distribution}}$</td>
<td>10284 ms</td>
<td>1530 ms</td>
<td>N/A</td>
</tr>
<tr>
<td>$t_{\text{Code_Installation}}$</td>
<td>1980 ms</td>
<td>508 ms</td>
<td>75 ms</td>
</tr>
<tr>
<td>$t_{\text{Resource_Binding}}$</td>
<td>4440 ms</td>
<td>0 ms</td>
<td>0 ms</td>
</tr>
<tr>
<td>$t_{\text{Server_Activation}}$</td>
<td>460 ms</td>
<td>253 ms</td>
<td>200 ms</td>
</tr>
<tr>
<td>$t_{\text{Server_Coordination}}$</td>
<td></td>
<td></td>
<td>N/A</td>
</tr>
</tbody>
</table>

*Table 4.12. Node Operations Measurements*
4.6.9. Total Deployment Response Time

Deployment phase took from 2.5 to 18 seconds for chat servers and proxy-caches application networks respectively. As previously shown in equation 4.4. such time should correspond to the longest time of command transmission, execution and response. Experimental results corroborate this: total deployment time is always larger that node deployment time.

Servlet-based distributed application networks measurements were only taken for operations executed by resource agents. An estimated 500 ms can be expected to deploy such servlet based application network. It was a very simple “HelloWorld” application; therefore it is the lowest possible value we could obtain.

<table>
<thead>
<tr>
<th>Action</th>
<th>Cache ALN</th>
<th>Chat ALN</th>
<th>Servlet CDN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node_Deployment</td>
<td>17164 ms</td>
<td>2391 ms</td>
<td>400 ms</td>
</tr>
<tr>
<td>Update_Application_Table</td>
<td>3.5 ms</td>
<td>3.5 ms</td>
<td>N/A</td>
</tr>
<tr>
<td>Total_Deployment</td>
<td>18275 ms</td>
<td>2502 ms</td>
<td>N/A</td>
</tr>
</tbody>
</table>

*Table 4.11. Deployment Actions Duration*

The larger overhead was due node deployment operations. As previously shown such time has as its larger overhead code distribution and installation. Therefore to decrease total deployment time, decreasing code distribution is the best option.
4.7. Summary

This prototype is an instantiation of the model and framework proposed at chapter 3. It has been tested in experiments to show that implemented mechanisms provide expected functionality and to measure its temporal response. Results show it deploys application networks in the stated manner, and that it provides an adequate response time, lower than what could be obtained previously through manual operations. Experiments carry out with such implementation show which components might be streamlined to decrease deployment time even further.

However this prototype does not allowed us to show how the framework scales to a system over a wide area network, with a larger number of resources and deployment nodes. We evaluate this by simulation in following chapters.