This is an unfinished draft. If you have comments or corrections, please mark this document up and send it to jorg@comm.utoronto.ca. If you send your comments in plain text, please include the date (see upper left corner), the page number and the paragraph number. If you find discrepancies between this document and the most recent version of the HyperCast, please give a detailed description of the problem.

Thank you,

Jörg Liebeherr

CHAPTER 3

Monitor and Control System

The ability to remotely control and coordinate a network is an important aspect of a network architecture. It includes a set of functions to monitor and control the configuration, state and traffic flow in a network and detect failure conditions. A system that performs these functions, referred to as a *network management system*, includes a component for describing and presenting the information that is being monitored, and a component for accessing monitored information. The latter consists of a system that collects the monitored information, systems that are being monitored, and a protocol for exchanging information between the two types of systems. They are referred to, respectively, as *managing entity*, *managed systems*, and *management protocol*.

Today, all operators of large-scale networks rely on a network management system to control and coordinate deployed networked hardware. Although application-layer overlay networks are software artifacts, they can benefit from a network management system in the same way as a system of networked hardware devices. Tasks such as monitoring and visualizing the topology of an application-layer overlay network that is running on hundreds of different computers become tedious unless facilities are in place that collect relevant information and make it accessible to a monitoring system.

Another way to motivate the need for a network management system in overlay networks is derived from the principles and practices of protocol design. Protocols for computer networks are built around three operational planes: the data plane, the control plane, and the management plane. The data plane is concerned with the exchange of application data. The control plane comprises the functions that do not themselves transport data, but set up and maintain the ability to transport data, e.g., a routing protocol. The management plane comprises the network management system that permits monitoring information related to the control plane and the forwarding plane. Most comprehensive protocol architectures draw a clear distinction between these operational planes. The design of a protocol architecture for overlay networks is no exception and should follow the same principles.

HyperCast has a complete network management system that gives remote access to internal state variables of overlay sockets. With the monitor and control system, a single monitor application can manage all aspects of a HyperCast overlay network. In the network management system, each overlay socket maintains a collection of variables with state information, called *statistics*. The statistics of an overlay socket are organized as an XML document. The format of the XML document is defined in *XML schema files* that are dynamically generated and that can be queried through the monitor and control system. In the context of HyperCast, a managing system is called a *monitor*, managed systems are called *portals*, and the management protocol is called the *monitor protocol*. Monitors and portals are both components of application programs. They communicate through the monitor protocol by exchanging XML formatted message.

The components of the monitor and control system are illustrated in Figure 1. The figure shows a monitor application that contains a monitor, and an application program with an overlay socket and a portal. When the monitor application wants to access a statistic from a remote overlay socket, it has its monitor send a request message to the portal of the remote socket. When the portal receives the message, it interprets the message and accesses the statistic of the overlay socket. The portal puts the results in a monitor message and sends the message to the monitor, which, in turn, passes the results of the query to the monitor application. As indicated in the figure, a portal can also access statistics from an application program.

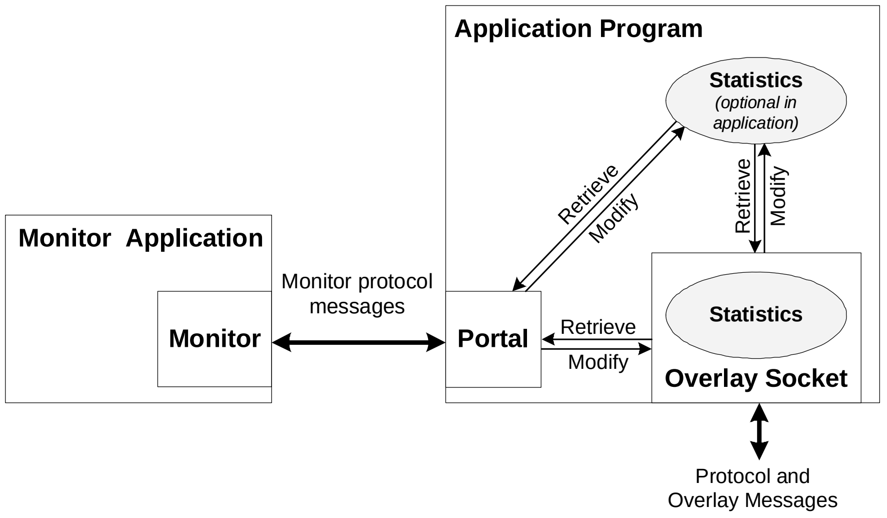


Figure 1. Components of the monitor and control system in HyperCast.

Monitor protocol messages between monitors and portals are transmitted over a separate HyperCast overlay network, which is referred to as *monitor overlay network*. Setting up an overlay network for monitor and control has several advantages. First, by taking advantage of the multicast operations available in a HyperCast overlay network, a single monitor can control a large number of remote applications. Second, it is possible to take advantage of the security features available in HyperCast. Third, since HyperCast overlay networks can deal with different underlay network technologies, using HyperCast overlays for sending monitor protocol messages extends the same capabilities to the monitor and control system.

Figure 2 illustrates the relationship between a monitor overlay network and an overlay network that is being monitored. The monitor overlay network, shown on the left of the figure, performs monitor and control functions for the overlay network shown on the right of the figure. Each application program has one portal that connects to the monitor overlay network. All portals are accessed by a single monitor. It is possible to have more than one monitor connected to the same monitor overlay network. Also, it is feasible to have multiple monitor overlay networks, each with one or more monitors, that manage the same overlay network.

The monitor and control system of HyperCast has several measures to protect against unauthorized access. First, application programs exercise control if they can be accessed by a remote network management system by deciding whether (or not) they instantiate a portal. An application can prevent remote access to its state variables by not creating a portal. In addition, authentication of monitors and portals, as well as integrity and confidentiality of the data transmitted on the monitor overlay network can be realized through the security architecture of HyperCast overlay sockets.

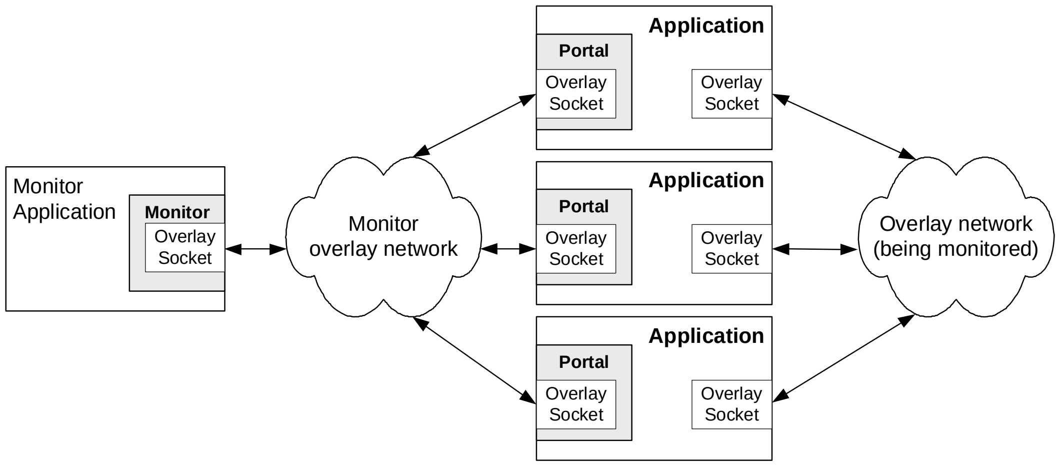


Figure 2. A monitor overlay network.

The monitor and control system of HyperCast bears some similarities to the Simple Network Management Protocol (SNMP) framework, which provides facilities for managing and monitoring network resources in the Internet. The SNMP framework consists of SNMP agents, Management Information Bases (MIBs), SNMP managers, and the SNMP protocol. An SNMP agent is a software component that runs on a host, router, or another piece of network equipment, and that maintains information about its configuration and current state in a database. The organization of a database is specified in a MIB, which uses the syntax of ASN.1 (Abstract Syntax Notation 1) to describe managed data and their properties. An SNMP manager is an application program that contacts SNMP agents to query or modify the database of an agent. The SNMP protocol is an application layer protocol for exchanging data between SNMP agents and managers.

We can relate SNMP to the monitor and control system of HyperCast. The set of statistics in HyperCast correspond to database of managed objects in SNMP, the XML schema description corresponds to a MIB, a portal corresponds to an SNMP agent, a monitor corresponds to a SNMP manager, and the monitor protocol corresponds to the SNMP protocol. A major difference of the monitor and control system of HyperCast to the SNMP framework is that the structure of managed data in HyperCast is dynamically created when a query is made to the XML schema. In HyperCast, each component of the overlay socket builds and maintains its own statistics. Differently, MIBs in SNMP are static files that describe the managed information of a specific system configuration. Each modification to the managed information in SNMP generally requires constructing and disseminating a new MIB file. HyperCast avoids this problem by generating XML schema description of managed information on the fly. In this way, HyperCast can deal with a large variety of different configurations and even permit dynamic changes to the managed information of a system. Another advantage over SNMP is that the application of XML technology makes it is easier to access complex objects, such as tables or complete sets statistics of a component of an overlay socket.

3.1. STATISTICS IN HYPERCAST

HyperCast statistics give access to state information of an overlay socket and, if so configured, of an application program. Statistics are available for many components of an overlay socket. Each statistic is stored and maintained by the component where this statistic plays a role. For example, the statistic that keeps track of the number of bytes transmitted by the node adapter is kept in the node adapter. The components of the overlay socket that maintain statistics are as follows:

- The **overlay socket** maintains statistics that track the forwarding of messages and the delivery of messages to applications. In addition, the statistics of the overly socket provide access to the statistics of other components of the overlay socket.
- The **overlay node** has statistics that relate to the logical address of the overlay socket, the neighbors of the overlay socket in the overlay network, and state information of the overlay protocol.
- The statistics of the **socket adapter** and the **node adapter** track the number of transmitted and received messages and bytes. The node adapter, also gives access to the physical address of the overlay socket.
- The **configuration object** has statistics for the configuration attributes of an overlay socket. All configuration parameters of a socket can be accessed (and possibly modified) through the monitor and control system.
- The **message buffer** has statistics that record the backlog of received messages that have not been retrieved by the application program.
- The **message store** supports statistics specific to the enhanced message services.
- HyperCast statistics can also be defined for **application programs**. In this fashion, state information about application programs can be remotely monitored. Support of application-defined statistics must be provided by the application programmer.
- A **portal** maintains statistics that keep track of messages exchanged by the monitor protocol.

The statistics of an overlay socket have a hierarchical organization as illustrated in Figure 3. The hierarchy reflects the software architecture of the overlay socket. Requests to access a statistic are processed from the top of the hierarchy and are forwarded to the component where the statistic is stored. Consider an access to the statistics to the number of overlay protocol messages that have been transmitted by the overlay socket. Since the node adapter is responsible for transmitting protocol messages, the corresponding statistic, with name *USentPackets*, is maintained by the node adapter. When this statistic is requested from an overlay socket, the request is forwarded to the node component, which, in turn, forwards the request to the node adapter.

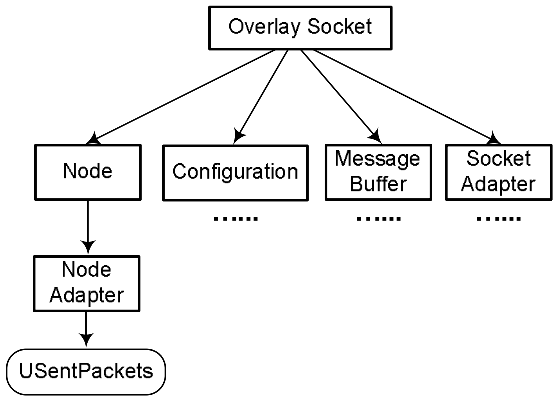


Figure 3. Hierarchy of overlay socket statistics. (*USentPackets* is a statistic in the node adapter. The location of other statistics is indicated by dots).

AN XML PRIMER

XML (eXtensible Markup Language) is a standardized method for representing text and data across different hardware and software platforms. XML documents look similar to HTML (Hypertext Markup Language) documents. HTML and XML have in common that they use tags and attributes to mark up documents. Unlike HTML, the attributes and tags in XML are not predefined. The following is an example of an XML document:

```
<?xml version="1.0"?>
<project  location="University of Virginia">
        <projectname>HyperCast</projectname>
        <version>3</version>
        <year />
</project>
```

The first line (<?xml version="1.0"?>) contains the XML declaration that specifies the XML version and possibly other information. The remainder of the document consists of elements. An element is a unit of data surrounded by an opening tag and a closing tag. For example, in <version>3</version>, 3 is the data, <version> is the opening tag and </version> is the closing tag. The tag <year />, represents an empty element, which is identical to <year></year>. Elements can be contained (nested) in other elements, resulting in a hierarchically structured XML document. The first element of an XML document is called the root element. It contains all other elements of the document. In the example, the element with tag <project> is the root element. The root element contains three elements. Elements can be further specified with attributes. In the example, the root element has one attribute (location="University of Virginia").

The XML syntax specifies rules for tagging and nesting elements and attributes. Documents with correct XML syntax are called well-formed. The permissible content of elements and their position in an XML document is specified in a schema description, which is kept in a separate file. If a well-formed XML document complies with its schema description it is said to be valid. *XML schema* is one of several available languages to specify a schema description. XML schema descriptions follow the XML syntax. The XML schema description of the XML document above is as follows:

```
<?xml version="1.0"?>
```

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <xsd:element name="Project">
 <xsd:complexType>
   <xsd:sequence>
            <xsd:element name="projectname"     type="xsd:string">
            <xsd:element name="version"  type="xsd: positiveInteger default="3">
            <xsd:element name="year"      type="xsd:gYear">
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
</xsd:schema>
```

The first line is the XML declaration. The second line is the opening tag for the root element, which specifies a schema element. The attribute xmlns:xsd=http://www.w3.org/2001/XMLSchema defines a namespace for the schema. Namespaces in XML can avoid conflicts for elements that have the same name. The assigned value xsd is the standard name space for XML schema descriptions. All tags in the schema file have a prefix that refers to the namespace. The elements in the root element define the structure of the XML document. A project is defined as a sequence of three elements: one element specifies the project name, another the version and the last element is a date. XML schema requires that elements with nested elements are of type complexType. The other elements are defined using predefined types in XML schema: a string, a positive integer, and a data type that specifies a year.

An XML document is linked to an XML schema by specifying the location of the schema file in an attribute of the root element. In our example, assuming that the XML schema is stored in file project.xsd, the opening tag of element Project should be changed as follows:

```
<project xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:NamespaceSchemaLocation="project.xsd">
```

XPath is an extension to the XML framework to identify locations in an XML document. The syntax for the most basic XPath expressions is similar to addresses in a file system. In an XPath expression, a forward slash refers to the root element. In the above XML document, the XPath expression /version refers to the element describing the version. XPath defines more complex expressions to select parts of a document based on the value of attributes or based on the value of data in an element.

DOM (Document Object Model) is a programming language and operating system independent API for representing XML documents. To make an XML file accessible via the DOM API, it is converted into a document object using a DOM parser. A DOM document has a tree structure consisting of a hierarchy of nodes, each representing an element of the XML. The tree structure of the document object makes it easy to navigate and manipulate the content of the XML document. The DOM API can convert XML documents into DOM documents and vice versa.

Statistics in XML Format

All statistics of an overlay socket are organized as a single XML document. The XML document for the statistics of an overlay socket generally has the following structure:

```
<Socket>
        <Node>
                …
                <NodeAdapter>
                        …
                        <UPacketsSent>120</UPacketsSent>
                        <UBytesSent>12384 </UBytesSent>
                </NodeAdapter>
        </Node>
        <Config> … </Config>
        <MsgBuf> … </MsgBuf>
        <SocketAdapter> …  </SocketAdapter>
</Socket>
```

The precise structure of the XML document and the names of the elements are defined during the configuration of the overlay socket. The XML document of the overlay socket is recursively built from XML documents that describe the statistics of the overlay socket components. Components of the overlay socket with statistics include the overlay node, the configuration object, the message buffer and the socket adapter. Each component of the overlay socket defines an XML document that specifies the statistics of this component. If a component has subcomponents, then it uses the XML documents of its subcomponents. For example, the XML document of the overlay node includes the XML document containing the statistics of the node adapter. The names of the tags are defined by the object that contains the statistics using values in the configuration file, but can be modified. For example, *NodeAdapter* is the default name for the element that contains the statistics of the node adapter of an overlay socket.

The permissible content of the XML document and its precise structure is specified by an XML schema description. When a monitor program wants to remotely access the statistics of an overlay socket, and does not know which statistics are available, it can request a XML schema description from the overlay socket. If so requested, an overlay socket can return two XML schema files: one schema describes the readable statistics, and one schema describes the writeable statistics. Generally, writeable statistics are also readable, and, therefore, appear in both schemas.

Since the statistics available in an overlay socket are dependent on the configuration of the socket, so is the XML schema describing the structure format of the statistics. To deal with the large variety of socket configurations, the XML schema description for the statistics of an overlay socket is dynamically generated when the schema is requested. Each component of the overlay socket supplies XML schemas for the statistics supported by this component. When a schema is requested from an overlay socket, the schema is recursively generated from the schemas of its components.

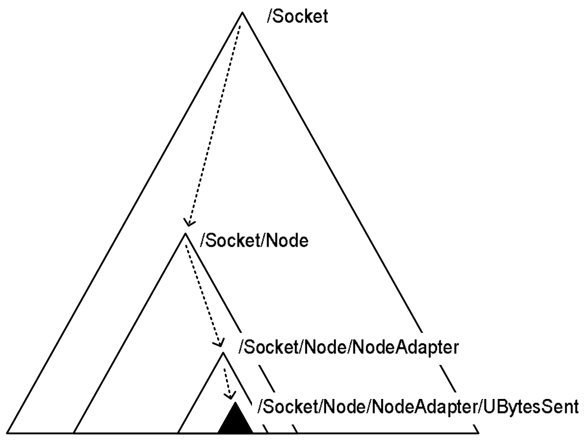


Figure 4. Accessing an overlay socket statistic with XPath expressions.

Requests for statistics in HyperCast are specified with XPath expressions. With XPath expressions, it is possible to access a specific statistic or a subset of the available statistics. For example, the XPath expression to locate the statistic that keeps track of the number of bytes transmitted by the node adapter socket is

```
/Socket/Node/NodeAdapter/UBytesSent
```

Figure 4 illustrates the relation of this XPath expression to the organization of statistics in the overlay socket. Each triangle in the figure represents the set of all statistics of a particular component or a single statistic. The outermost triangle represents the XML element with the statistics of the overlay socket, which is specified as */Socket*. The triangle labeled */Socket/Node* denotes the statistics of the overlay node, and the triangle */Socket/Node/NodeAdapter* denotes the statistics of the node adapter. The innermost triangle identifies the statistic */Socket/Node/NodeAdapter/UBytesSent*. To request the statistic *UBytesSent* of the node adapter, a request for */Socket/Node/NodeAdapter/UBytesSent* is sent to the overlay socket. When the overlay socket receives the request, it requests the statistic */Node/NodeAdapter/UBytesSent* from the overlay node, which in turn forwards a request for expression */NodeAdapter/UBytesSent* to the node adapter. Finally, the node adapter looks up the value of the statistic *UBytesSent*.

With XPath expressions it is possible to identify statistics that are not scalar values. For example, the neighborhood table in the node component of an overlay socket, which stores information about the neighbors of an overlay socket in the overlay topology, is identified with the XPath

```
/Socket/Node/NeighborTable
```

If one is interested in the second row of the neighborhood table, the XPath expression is

```
/Socket/Node/NeighborTable[2]
```

It is feasible to access all statistics of an entire subtree of the XML document. As an example, to access all statistics available in the overlay node and its subcomponents, the XPath expression is

```
/Socket/Node
```

To access all statistics, the XPath expression is

/Socket or /*

XPath expression may refer to collections of elements in a document, so a query for a statistic may return an array of XML elements. The HyperCast software only implements a small subset of the available XPath syntax. Future versions of the software may provide a more comprehensive implementation.

To modify a statistic of an overlay socket, in addition to locating the statistic with an XPath expression, one needs to specify the new value of the statistic. To set the Heartbeat timer (defined in the Hypercube overlay protocol) to 1000, the expression is

(/Socket/Node/HeartBeat, <HeartBeat>*1000*</HeartBeat>)

The value of the statistic is expressed in terms of an XML element. The element must be valid with respect to the XML schema for writeable statistics. It is possible to modify an entire subtree of the writeable statistics. For example, the following tuple can be used to modify all writeable statistics of the overlay node

(/Socket/Node, <Node> ... </Node>)

Application Programming Interface for Statistics

Statistics are accessed through a statistics API, which is supported by all overlay socket components that maintain statistics. The statistics API specifies methods for querying and modifying statistics, as well as a method to query the XML schemas that describe statistics. The implementation of HyperCast stores XML documents with statistics as DOM documents.

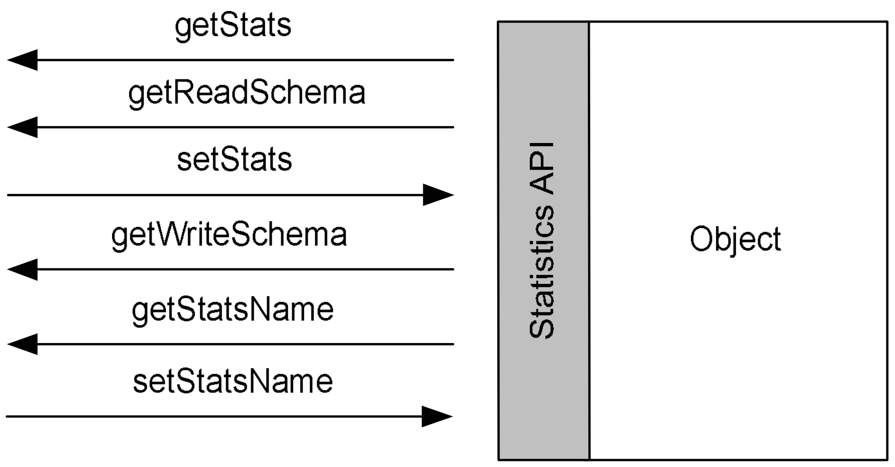


Figure 6. Statistics API.

The statistics API defines methods to read and write statistics stored in an object, as well as the name of component (see Figure 6). The method *getStats* is responsible for retrieving statistics, *setStats* can modify statistics, *getReadSchema* retrieves the XML schema description of readable statistics, and *getWriteSchema* retrieves the XML schema description of writeable statistics. Requests for statistics of subcomponents are forwarded to the referenced component. The methods *getStatsName* and *setStatsName* retrieve and modify, respectively, the name of the element associated with the object.

The forwarding of requests is illustrated in Figure 7 for the XPath expression */Socket/Node/NodeAdapter/UBytesSent.* Referring to Figure 7, when the overlay socket analyzes the prefix of the XPath expression */Socket/Node/NodeAdapter/UBytesSent*, it determines that the referenced statistic is located in the overlay node. Then, it requests the expression with XPath */Node/NodeAdapter/UBytesSent* from the overlay node. The overlay node, in turn, requests the statistics with XPath expression */NodeAdapter/UBytesSent.* At the node adapter, processing the statistic results in the lookup of the state variable that is associated with the statistic *UBytesSent*. The result is then recursively passed up in the hierarchy as a DOM element.

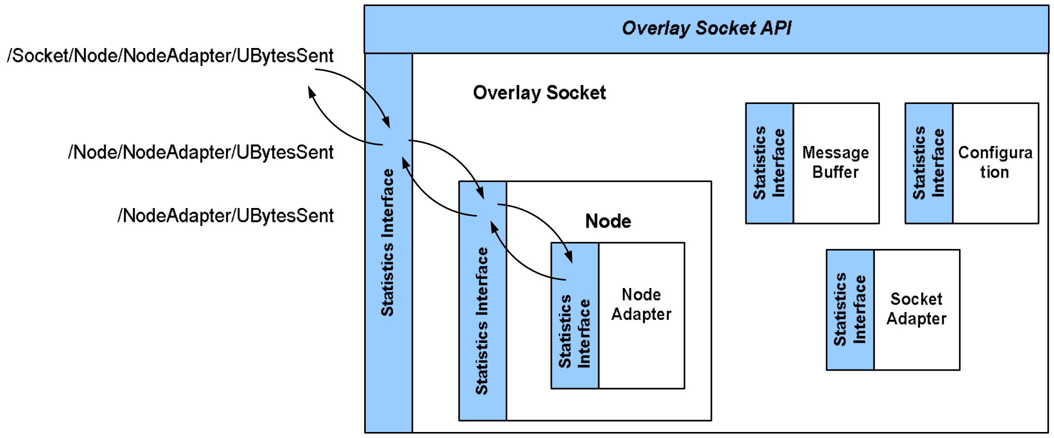


Figure 7. Accessing a statistic of an overlay socket.

In the Java implementation of HyperCast, the statistics API is specified in the *I_Stats* interface. The following describes the methods the *I_Stats* interface:

Element[] getStats(Document doc, XPath XPathExp)

Retrieves statistics from an object. The first argument is a document object that gives a handle to a DOM document where the XML data is stored. The second argument is an XPath expression that specifies the requested statistics. Since an XPath expression matches one or more statistics, the *getStats* method returns the result of the query as an array of DOM elements. Each element in the array may contain a single statistic or a subtree of statistics.

Element[] setStats(Document doc, XPath XPathExp, Element newValue)

Modifies the statistics of an object. The first argument plays the same role as in the *getStats* method, that is, it is a handle to a DOM document. The second argument is an XPath expression that locates the statistics to be modified, and *newValue* contains the new value of the statistics that is modified in the form of a DOM element. When the given XPath expression matches multiple statistics, each of them is modified with *newValue* and the array of the new values, which are actually set to the matched statistics, are returned in the form of an array of elements. The *newValue* element can be as simple as single integer or as complex as an entire subtree that contains all statistics of a component of the overlay socket. For each matched element with respect to the given XPath expression, one can think of the *setStats* operation as substituting a complete subtree of the XML document, where the XPath expression references the position of the subtree in the document, and *newValue* is the new value of the subtree.

Element[] getReadSchema (XPath XPathExp)
Element[] getWriteSchema(XPath XPathExp)

> These methods return descriptions of the available readable or writeable statistics of the component identified by an XPath expression. The XML schema of an overlay socket component contains a description of the statistics supported by the component and all of its sub-components. Since an XPath expression matches one or more statistics, the above methods return the array of schemas with each as a DOM element. The *getReadSchema* method returns an array of schemas. Each schema in the array is for a readable statistic that can be retrieved with *getStats*. The *getWriteSchema* method returns an array of schemas for the writeable statistics that can be modified with *setStats.*

String getStatsName ()
Retrieves the element name of the object. The element name of an object is originally assigned from the attribute *StatName* in the configuration file. The XML schema for configuration files specifies the default names for the components.

void setStatsName (String name)
Modifies the element name of the statistic as given by the argument.

When an error occurs during the access of statistics, e.g. a statistic is not found or does not have a name, an attempt is made to modify a read-only statistic, or a format violation occurs, a *HyperCastStatsException* is thrown by the overlay socket component where the error occurred.

Next we discuss an application program that creates an overlay socket and then accesses a statistic of the overlay socket. The program queries the number of transmitted bytes by the node adapter, which are kept in the statistic "/Socket/Node/NodeAdapter/UBytesSent". The complete program is as follows:

```
import hypercast.*;
import hypercast.util.XmlUtil;
import java.io.*;
import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.apache.xpath.*;

public class AccessStatistics {

  public static void main(String args[]) {
      HyperCastConfig ConfObj = HyperCastConfig.createConfig("hypercast.xml");
      I_OverlaySocket MySocket=ConfObj.createOverlaySocket(null);
      MySocket.joinOverlay();

      Document doc = XmlUtil.createDocument();
      XPath xpath =  XmlUtil.createXPath("/Socket/Node");

      Element[] resultElements = null;
      try {
```

```
                resultElements = MySocket.getStats(doc, xpath);
        } catch (HyperCastStatsException e) {
                System.err.println("Query fails:" + e.getMessage());
        }

        if (resultElements != null) {

            for (int i=0; i<resultElements.length; i++) {
                Document resultDoc = XmlUtil.createDocument();
                resultDoc.appendChild(resultDoc.importNode(resultElements[i], true));
                try {
                        XmlUtil.writeXml(resultDoc, System.out);
                 }  catch (IOException e) {
                        System.err.println("Can't write XML file:" + e.getMessage());
                }
        }
   }
   /* Add code segment "Modify a statistic" here */

   MySocket.closeSocket();
}
```

The program imports several Java packages that define classes related to processing DOM objects **[citeMcLaughlin][citeGriffith].** The program first creates an overlay socket that joins an overlay network. Then, the program instantiates an empty DOM document, which will be serve as a handle, and an XPath expression for the statistic */Socket/Node/NodeAdapter/UBytesSent.*

```
Document querydoc = XmlUtil.createDocument();
XPath xpath = XmlUtil.createXPath("/Socket/Node/NodeAdapter/UBytesSent");
```

The statistic is retrieved with the getStats method of the Statistics API. The parameters are the DOM document and the XPath expression:

```
resultElements = MySocket.getStats(querydoc, xpath);
```

The result of the query is returned as an array of elements. For each returned element, an empty DOM document is created and the element is added to the created document:

```
Document resultDoc = XmlUtil.createDocument();
resultDoc.appendChild(resultDoc.importNode(resultElements[i], true));
```

Then, the DOM document containing the element is displayed by invoking

```
XmlUtil.writeXml(resultDoc, System.out);
```

This results in the following output:

```
<?xml version="1.0" encoding="UTF-8"?>
<UBytesSent>12384</UBytesSent>
```

If the requested statistics is */Socket/Node/NodeAdapter*, then the output of the program is an XML document that contains all statistics of the node adapter:

```
<?xml version="1.0" encoding="UTF-8"?>
<NodeAdapter>
<RecvBuf>
<MaxMessages>100</MaxMessages>
<NumOfMsgsInQueue>0</NumOfMsgsInQueue>
<NumOfReadersWaiting>1</NumOfReadersWaiting>
<NumOfWritersWaiting>1</NumOfWritersWaiting>
</RecvBuf>
<UPacketsReceived>3501</UPacketsReceived>
<UBytesReceived>2345</UBytesReceived>
<UPacketsSent>325760</UPacketsSent>
<UBytesSent>13011</UBytesSent>
</NodeAdapter>
```

The following program segment modifies a statistic of the overlay socket (The segment can be added in the AccessStatistics class at the indicated position). The segment modifies the boolean statistic Running in the overlay socket. If this statistic is set to false, the overlay socket leaves the overlay network.

```
// Code segment: Modify a statistic
xpath = XmlUtil.createXPath("/Socket/Running");
Element element = XmlUtil.getXmlValue( doc, "/Running", "false");
try {
        resultElements = MySocket.setStats(doc,xpath,element);
} catch (HyperCastStatsException e) {
        System.out.println("Set request fails:" + e.getMessage());
}
```

3.2. HYPERCAST STATISTICS IN APPLICATION PROGRAMS

Application programs may specify their own statistics that provide access to and control of application-specific information. An application with an overlay socket can link the statistics of the overlay socket to those of the application. When an application program defines statistics, it puts itself at the top of the statistics hierarchy, as illustrated in Figure 8.

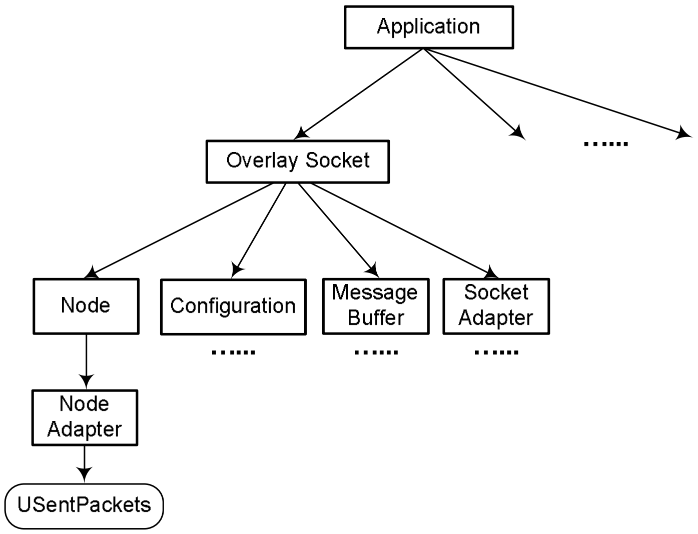


Figure 8. Hierarchy of overlay socket statistics, when the application supports statistics.

Here, the statistics of the overlay socket are nested in the element for the application statistics, e.g.,

```
<Application>
        <Socket>  ... </Socket>
</Application>
```

If an application program contains multiple overlay sockets, then the *Application* element can have one element for the statistics of each socket, e.g.,

```
<Application>
        <Socket>  ... </Socket>
        <Socket>  ... </Socket>
        <Socket>  ... </Socket>

</Application>
```

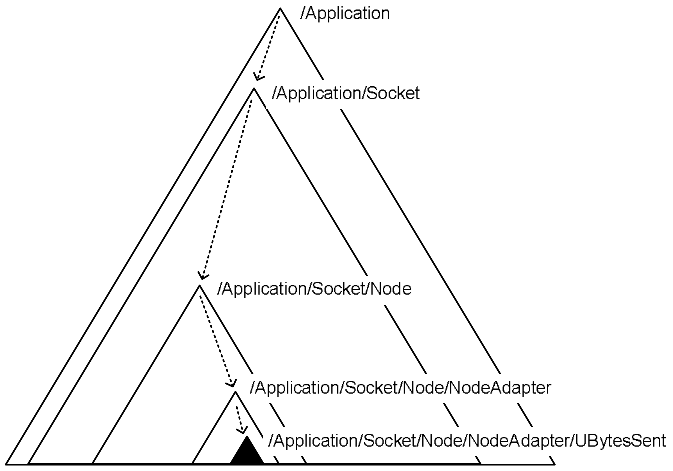

/Application
/Application/Socket
/Application/Socket/Node
/Application/Socket/Node/NodeAdapter
/Application/Socket/Node/NodeAdapter/UBytesSent
Figure 9. Accessing an overlay socket statistic with XPath expressions.

Figure 9 depicts a situation where the application supports statistics and the application has one overlay socket. In this case, the statistics of the overlay socket are an element in the statistics of the application. The root element, denoted by */Application*, of the XML document are the statistics of the application. The statistics of the overlay socket are denoted by */Application/Socket*. Here, the XPath expression to locate the statistic *UBytesSent* in the node adapter is */Application/Socket/Node/NodeAdapter/UBytesSent*.

The StatsProcessor and SimpleStats Classes

An application program with application-defined statistics must implement the statistics interface, and must be able to process queries made to the statistics API, and statistics accessed through the application must be linked to objects that are accessed by the application. Since all statistics are made available as DOM elements and statistics are accessed using XPath expressions, an implementation of the statistics API may involve considerable effort. To reduce the burden on application programmers (as well as HyperCast developers) HyperCast offers an implementation of two classes, *StatsProcessor* and *SimpleStats*, that hide XML-related programming tasks and offer a uniform method for defining statistics in an application program. The cost of this convenience is that each statistic must be explicitly defined as an object.

Consider an application that defines the following statistics:

```
<AA>
        <BB>
                </CC>
        </BB>
        <DD>
                <EE/>
                <EE/>
        </DD>
        </FF>
 </AA>
```

Each element is defined within an object that implements the I_Stats class. Let us refer to the object that defines statistics *X* as *ObjX* and its class as *ClassX*. Objects that define statistics with nested elements (</AA>, </BB>, </DD>) define a *StatsProcessor*. Objects that define elements with text, i.e., the leafs in the XML tree, extend the *SimpleStats* class.

When an object instantiates a *StatsProcessor* it determines whether the statistics that are accessed through it are readable or writeable. For example, when object ObjAA creates its *StatsProcessor* by

```
MyStatsprocessor = new StatsProcessor(ObjAA,  true, false );
```

This statement defines that the statistics accessed through the *StatsProcessor* are readable (first flag) but not writeable (second flag). Once a *StatsProcessor* is created, the object adds the nested elements to it. This is done with the method addStatsElement which has the following signature:

```
public void addStatsElement(String Name, I_Stats statsObj, int minoccur, int maxoccur)
```

The first parameter is the name of the element, the second parameter is the I_Stats object, and the third and fourth define the number of occurrences of this element. The minimum and maximum occurrence are used to define the XML schema for this element. In our example, object ObjAA adds nested elements as follows:

```
ObjAA.addStatsElement ("BB",  ObjBB, 1, 1);
ObjAA.addStatsElement ("DD",  ObjDD, 1, 1);
ObjAA.addStatsElement ("FF",  ObjFF, 1, 1);
```

Object ObjAA also needs to specify a name for its own statistic. This is done with the call

```
        ObjAA.setStatsName("AA");
```

Each object with a StatsProcessor redirect calls to its statistics to the *StatsProcessor* object. For each method of the statistics API, the *StatsProcessor* has a corresponding method that handles the call to the statistic. For the *getStats* method, the *StatsProcessor* has a method with name *getStatsResults*. The redirection of getStats method to the corresponding method of the StatsProcessor is done as follows:

```
public Element[] getStats (Document doc, XPath xpath)
        throws HyperCastStatsException {
        return statsProcessor.getStatsResult (doc, xpath);
}
```

The redirection for the other methods of the statistics API is done accordingly.

Lastly, the application must specify statistics that are the leaf elements in the XML tree, and which contain a scalar value of a statistic. Each of these elements (i.e., </CC>, </EE>, </FF> is defined by an object that extends the simple *SimpleStats* class. The *SimpleStats* class provides methods that deal with the manipulation and presentation of a well-formed XML document for the statistic. The application programmer must provide methods for the construction of an XML schema for the element, and must specify the relationship of the getStats and setStats methods to the application program. As an example, we provide the definition of ClassCC for element </CC>. Assuming that the statistic is a read-only statistic, the application program must specify the methods getStats and getReadSchema.

```
class ClassCC extends SimpleStats {
        protected String getStats() { return VariableCC; }

        public Element[] getReadSchema (Document doc, XPath xpath)
                throws HyperCastStatsException {
                return XmlUtil.createSchemaElement (doc/* document instance */,
                        statisticsName/* element name */, "xsd:String"/* type */,
                        null/* restriction base */, null/* pattern value */);
        }
}
```

Here, the </CC> element is bound to a text variable with identifier VariableCC. Whenever, the statistic is accessed, the value of VariableCC is returned. The XML schema for the statistic is defined with the help of the createSchemaElement method of the XMLUtil class.

Each *I_Stats* object defines the schema element to describe its characteristics, such as type and data pattern. If an *I_Stats* object defines a *StatsProcessor* instance, which implies that it contains nested statistics, it is an intermediate node on the schema tree and which is defined as an XML element of type *complexType*. The corresponding schema element is defined as below:

```
<xsd:element name="xxxx">
    <xsd:complexType>
         ……
    </xsd:complexType>
</xsd:element>
```

name in the above definition refers to the name of the statistic.

For simple statistics, they have *type* attribute and can optionally have *pattern* attribute to define the constraints. When a simple statistic does not have *pattern* attribute, the corresponding schema element is defined as following:

```
   <xsd:element name="xxxx" type="xsd:xxxx" />
```

For example, the schema elements for the statistics CurrentTime and KillApp are:

```
   <xsd:element name="CurrentTime" type="xsd:Long" />
   <xsd:element name="KillApp" type="xsd:Boolean" />
```

When a simple statistic defines *pattern* attribute, the corresponding schema element is:

```
   <xsd:element name="xxxx"/>
      <xsd:simpleType>
         <xsd:restriction base="xsd:xxxx">
            <xsd:pattern value="xsd:xxxx" />
         </xsd:restriction>
```

```
    </xsd:simpleType>
  </xsd:element>
```

For example, the schema element for the logical address of a SPT overlay node is defined as:

```
  <xsd:element name="LogicalAddress"/>
    <xsd:simpleType>
      <xsd:restriction base="xsd:String">
        <xsd:pattern value="\d+" />
      </xsd:restriction>
    </xsd:simpleType>
  </xsd:element>
```

Above element defines the logical address to be a string consisting of one or more digits.

A static method *createSchemaElement* is defined in the class *StatsUtil* to simply the creation of schema elements for simple statistics:

```
public static Element[] createSchemaElement(Document doc,
        String statsName, String statsType, String restrictionBase, String
patternValue)
```

The parameter *doc* provides a container to help creating Element and Node instances. Parameters *Stats*, *statsType* are passed as the values for the attributes *name* and *type*, *restrictionBase* and *patternValue* are passed as the values in the elements *restriction base* and *pattern value*. Once this method is called, the different form Element is created depending on the value of parameter *statsType*. If *statsType* is "*xsd:simpleType*", the *createSchemaElement* method creates the schema element for a statistic having *pattern* attribute; otherwise it creates the schema element for a statistic without *pattern* attribute defined.

If an application program calls the method *createSchemaElement*, it needs to add the following statement at the beginning of the program:

```
import hypercast.util.XmlUtil;
```

An Extended Example

We now present an extended example that applies the discussed concepts. The following program fragment presents an application program that defines a few statistics taking advantage of the *StatsProcessor* class.

```
import java.io.IOException;
import org.apache.xpath.XPath;
import org.w3c.dom.Document;
import org.w3c.dom.Element;
import hypercast.HyperCastConfig;
import hypercast.HyperCastStatsException;
import hypercast.I_OverlaySocket;
import hypercast.I_Stats;
import hypercast.SimpleStats;
import hypercast.StatsProcessor;
import hypercast.util.XmlUtil;

public class HelloWorld_Statistics implements I_Stats
{
        private String Location = "Location unknown";
        private I_OverlaySocket Mysocket;
        private String statisticsName = null;
        private StatsProcessor statsProcessor;

        public HelloWorld_Statistics () {
                HyperCastConfig sConf =
                        HyperCastConfig.createConfig("hypercast.xml");
                Mysocket = sConf.createOverlaySocket(null);

                InitStatisticsStructure();
        }
        ...
}
```

By implementing the I_Stats interface and by defining a StatsProcessor, the class indicates that it supports the statistics API with support of the StatsProcessor class. The method *InitStatisticsStructure* defines the structure of the statistics of the application.

```
private void InitStatisticsStructure() {
        statsProcessor = new StatsProcessor(this /* I_Stats object */,
                        true /* readable flag */,
                        true /* writable flag */);
        this.setStatsName("HelloWorld");

        statsProcessor.addStatsElement ("CurrentTime" /* name */,
                                new CurrentTime() /* I_Stats object */,
                                1 /* minoccurs */,
                                1 /* maxoccurs */);
        statsProcessor.addStatsElement ("StopApp", new KillApp(), 1, 1);
        statsProcessor.addStatsElement ("Location", new Location(), 1, 1);
        statsProcessor.addStatsElement("MySocket", this.Mysocket, 1, 1);
}
```

This defines the following statistics. , the statistics have the following structure:

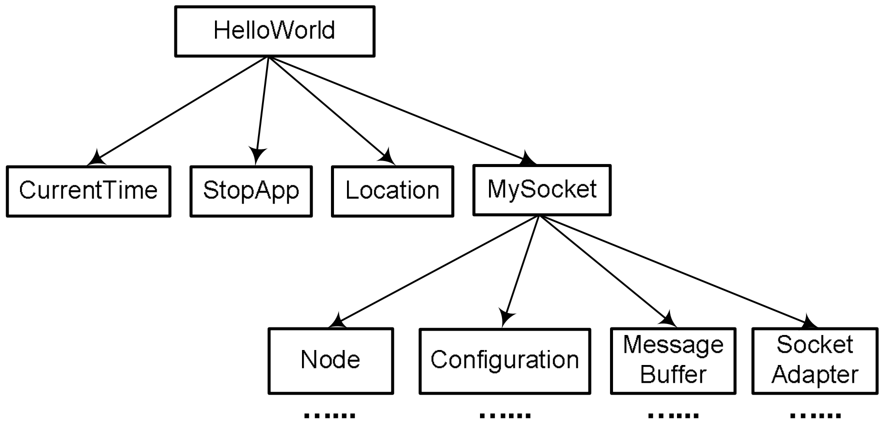


Figure 10. Statistics of example.

The statistics are defined as readable and writeable, meaning that they are defined in both XML schemas for readable and writeable statistics. The statistics CurrentTime, StopApp, and Location are scalar values which are defined by objects derived from the SimpleStats class. The statistic with name MySocket relates to the overlay socket that was created previously in the constructor.

The *HelloWorld_Statistics* class defined as above requires several specifications, which are described next. First, the *HelloWorld_Statistics* class must redirect the calls to the *I_Stats* interface to methods of the *SecurityProcessor* class. As noted earlier, the names of the methods in the *StatsProcessor* have a suffix `Results'. The implementation of the *I_Stats* interface therefore is:

```
public Element[] getStats (Document doc, XPath xpath)
        throws HyperCastStatsException {
        return statsProcessor.getStatsResult (doc, xpath);
}

public Element[] setStats (Document doc, XPath xpath, Element newValue)
        throws HyperCastStatsException {
        return statsProcessor.setStatsResult (doc, xpath, newValue);
}

public Element[] getReadSchema (Document doc, XPath xpath)
        throws HyperCastStatsException {
        return statsProcessor.getReadSchemaResult (doc, xpath);
}

public Element[] getWriteSchema (Document doc, XPath xpath)
        throws HyperCastStatsException {
        return statsProcessor.getWriteSchemaResult (doc, xpath);
}

public String getStatsName () {
```

```
           return statisticsName;
}

public void setStatsName (String name) {
          statisticsName = name;
}
```

Next, for each of the scalar statistics, we need to provide a class that extends the SimpleStats class. The class specifies the actions performed when accessing a statistic and the format of the schema element.

The CurrentTime statistic reads the system clock whenever the statistic is read. It is defined as a read-only only statistic, thus, only the method getStats and getReadSchema need to be specified:

```
class CurrentTime extends SimpleStats {                // read-only statistic

          protected String getStats() {
                    return Long.toString(System.currentTimeMillis());
          }

          public Element[] getReadSchema (Document doc, XPath xpath)
                    throws HyperCastStatsException {
                    return XmlUtil.createSchemaElement (doc/* document instance */,
                              statisticsName/* element name */, "xsd:Long"/* type */,
                              null/* restriction base */, null/* pattern value */);
                    }
          }
```

The KillApp statistic is a Boolean write-only statistic that cannot be read, but only be modified. W

the system clock whenever the statistic is read. It is defined as a read-only only statistic, thus, only the method *getStats* and *getReadSchema* need to be specified:

```
class KillApp extends SimpleStats {

          protected String setStats (String newValue) {
                    if (newValue.equals ("true"))   System.exit (0);
                    return newValue;
          }

          public Element[] getWriteSchema(Document doc, XPath xpath)
                    throws HyperCastStatsException {
                    return XmlUtil.createSchemaElement(doc/* document instance */,
                              statisticsName /* element name */, "xsd:Boolean"/* type */,
                              null/* restriction base */, null/* pattern value */);
          }
}
```

```
class Location extends SimpleStats {
        protected String getStats() { return Location; }

        protected String setStats(String newvalue) {
                Location = newvalue;
                return Location;
        }

        public Element[] getReadSchema (Document doc, XPath xpath)
                throws HyperCastStatsException {
                return XmlUtil.createSchemaElement (doc/* document instance */,
                        statisticsName/* element name */, "xsd:String"/* type */,
                        null/* restriction base */, null/* pattern value */);
        }

        public Element[] getWriteSchema(Document doc, XPath xpath)
                throws HyperCastStatsException {
                return XmlUtil.createSchemaElement(doc/* document instance */,
                        statisticsName/* element name */, "xsd:String"/* type */,
                        null/* restriction base */, null/* pattern value */);
        }
}
```

```
public void ModifyStatistics ( String XPathPosition,  String XPathElement,
                               String ValueElement)
```

```
public void AccessStatistics (String XPathexpr)

public void AccessReadSchema (String XPathexpr)

public void AccessWriteSchema (String XPathexpr)

void setStatsName (String name)
```

```
public synchronized static void main (String[] args) {
        HelloWorld_Statistics app = new HelloWorld_Statistics ();
        app.AccessStatistics("/*");
        app.AccessStatistics("/HelloWorld/CurrentTime");
        app.AccessReadSchema("/*");
        app.AccessWriteSchema("/HelloWorld");
        app.ModifyStatistics("/HelloWorld/Location", "/Location", "Toronto");
        app.ModifyStatistics("/HelloWorld/StopApp", "/StopApp", "true");
}
```

This *HelloWorld* program creates an overlay socket, as seen in the chapters describing the application programming interface, and it defines a *StatsProcessor*. An application program that supports statistics must implement the *I_Stats* interface. When the statistics are implemented by the *StatsProcessor* class, the *I_Stats* methods simply invokes methods of the *StatsProcessor* as follows:

```
}
```

The application defines the supported statistics in the method InitStatisticsStructure(), which is specified as follows:

```
  private void InitStatisticsStructure() {
        statsProcessor = new StatsProcessor(this/* I_Stats object */,
            true/* readable flag */, true/* writable flag */);

        statsPro.addStatsElement("Socket"/* name */,
            MySocket/* I_Stats object */, 1/* minoccurs */, 1/* maxoccurs */);
        statsProcessor.addStatsElement ("CurrentTime", new CurrentTime(), 1, 1);
        statsProcessor.addStatsElement ("StopApp", new KillApp(), 1, 1);
  }
```

```
class CurrentTime extends SimpleStats { // read-only statistic
      protected String getStats() {
          return Long.toString(System.currentTimeMillis());
      }

// Check: setStats on read-only statistics should cause an exception in the base class)

      public Element[] getReadSchema (Document doc, XPath xpath)
```

```
        throws HyperCastStatsException {
         return StatsUtil.createSchemaElement (doc/* document instance */,
                statisticsName/* element name */, "xsd:Long"/* type */,
                null/* restriction base */, null/* pattern value */);
      }
}
```

```
class KillApp extends SimpleStats {
    String setstats (String newValue) {
          if (newValue.equals ("true"))   System.exit (0);
          return newValue;
    }

    public Element[] getWriteSchema(Document doc, XPath xpath)
         throws HyperCastStatsException
    {
         return StatsUtil.createSchemaElement(doc/* document instance */,
               statisticsName/* element name */, "xsd:Boolean"/* type */,
              null/* restriction base */, null/* pattern value */)
    }
}
```

First, the method creates a *StatsProcessor* object:

```
statsProcessor = new StatsProcessor(this, true, true);
```

Above call passes the reference to the *I_Stats* object to the *StatsProcessor* and sets the readable and writable flags of the *I_Stats* object.

The *StatsProcessor* maintains all statistics of an *I_Stats* object. Adding the statistic Socket to the StatsProcessor is done by the invocation:

```
statsPro.addStatsElement("Socket"/* name */,

        MySocket/* I_Stats object */, 1/* minoccurs */, 1/* maxoccurs */);
```

which defines the name of the statistic as *Socket* and binds it to the *I_Stats* object *MySocket*. It also defines the minimal and maximal occurrence times of the statistic.

The above program also adds statistics *CurrentTime* and *StopAppl* to the StatsProcessor. Different from the *Socket* statistic, these statistics, refered to as simple statistics, are bound to an object of a class that is defined specifically for the defined statistics. The classes, respectively, extend the type *SimpleStats*, which, in turn, implements the *I_Stats* interface. Classes that extend *SimpleStats* have a simplified interface which requires the application programmer to specify only the methods *getstats* and getReadSchema if they define readable statistics, and specify *setStats* and *getWriteSchema* if they are bound to writable statistics. The following class definitions define the implementation of statistics for *CurrentTime* and *KillApp*.

The implementation of *CurrentTime* reads the time of the local system. Since *CurrentTime* is a readable statistic, the class *CurrentTime* only implements *getStats* and *getReadSchema* methods. The *KillApp* statistic is a write-only statistic that terminates the application, if the supplied value is set to *true*. It only overrides *setStats* and *getWriteSchema* methods.

3.3. THE MONITOR PROTOCOL

In the monitor protocol, monitors and portals exchange XML formatted messages over a HyperCast overlay network, the monitor overlay network. Portals are part of an application program that is being monitored, and monitors are generally part of an application program that issues queries and collects responses. We will refer to application programs that contain a monitor as a *monitor application*. In most cases, we will assume that the monitor overlay network has one monitor that communicates with one or more portals, but there can be more than one monitor in the same monitor overlay network.

The monitor protocol performs two tasks. The first task is the establishment of connectivity between monitors and portals. The second task is the exchange of monitor and control information. Each portal stores the logical address of at most one monitor. This address can be initialized by an attribute from the configuration file, or it is obtained from a monitor protocol message that advertises a monitor to the portal. If there are multiple monitors the portal stores the address of the most recently advertised monitor. Each monitor maintains a list, called the *portal list*, with the logical addresses of all known portals. The portal list may be empty when the monitor is started, and is updated from monitor messages sent by portals to the monitor.

Messages of the Monitor Protocol

Monitor protocol messages are XML documents. The root element of the message specifies the message type. The attributes of the root element contain meta information about the message, including the address of the sender (*Src*), the address of the receiver (*Dest*), a unique message identifier (*MsgID*), and a Timestamp (*TimeStamp*). The other elements are message type dependent.

We first describe the messages that establish and maintain connectivity between the monitor and portals. Since both monitor and portals are already connected to the same overlay network, the information to be exchanged is minimal.

- **AdvertisePortal:** Portals advertise themselves to a monitor by sending *AdvertisePortal* messages every $t_{AdvertisePortal}$ milliseconds. No message is sent when the portal does not have an address for a monitor. When a monitor receives an *AdvertisePortal* message, it adds the portal to its portal list. If the portal is already included, it updates the timestamp in the list. If no *AdvertisePortal* message has been received from a portal in the portal list for $t_{PortalTimeotl}$ milliseconds, the corresponding entry is marked as inactive. However, the entry is kept in the portal list, until it is explicitly removed by the application that uses the monitor. The format of the message is:

```
    <AdvertisePortal Sender="..." Dest="..." MsgID="..." TimeStamp="...">
<Stats index="1"  xpath="/Portal/Portal/Node/NodeAdapter/UBytesSent" />
```

- **AdvertiseMonitor:** A monitor sends one *AdvertiseMonitor* message every $t_{AdvertiseMonitor}$ millisecond to all portals in the portal list. When a portal receives an *AdvertiseMonitor* message, it updates the logical address information of its monitor. Each portal only maintains information about one monitor. When a portal does not hear from a monitor for $t_{MonitorTimeout}$ milliseconds it assumes that the monitor does not exist and does no longer send *AdvertisePortal* messages. The format of the message is as follows:

```
    <AdvertiseMonitor  Sender="..." Dest="..." MsgID="..." TimeStamp="..." />
```

Both portals and monitors can be set to silent mode. In this mode, portals and monitors do not send advertisements. A monitor set to silent mode does not set entries in the portal list as inactive.

The second task of the monitor protocol is the processing of queries that access statistics. There are three types of queries: queries to obtain a schema description of a statistic, queries to retrieve a statistic, and queries to modify a statistic. A query consists of a request message that is sent by a monitor, and a reply message that is sent back to the monitor. A monitor that has sent a request message waits for reply message from portal. When the portal receives a request, it translates the request into a call to the statistics API of the overlay socket, and builds a reply message. Once completed, the reply message is sent to the sender address of the request message. (This address can be different from the address of the monitor that is kept at the portal.) Request and reply messages have a structure as shown in Figures 9 and 10. The figures depict the messages as XML documents and as DOM document trees. Each message contains one or more elements with name *Stats*, which will be referred to as *statistic element*. Each statistic element has an index and an XPath expression that locates a statistic. The statistic elements of a request to read a statistic does not have nested elements. In a reply message and in a request to change a statistic, the statistic element includes the value of the statistic as a well-formed XML element.

The MsgID attribute acts a unique identifier for a query. The sender of a request message creates an identifier, and the reply message uses the same identifier. The monitor uses the *MsgID* attribute of a message to match an incoming reply message with a previously sent request message. Reply messages are always sent as unicast message to the sender of the corresponding request message. All other monitor messages can be sent as unicast or broadcast messages. For broadcast messages, the destination attribute is ignored.

```
<GetRequest  Src= “100011” Dest=“101010” MsgID=“13” TimeStamp="100516">
      <Stats index=“0”  xpath=“/Portal/Socket/Node/NodeAdapter/UPacketsSent” />
      <Stats index=“1”  xpath=“/Portal/Socket/Node/NodeAdapter/UBytesSent” />
</GetRequest>
```

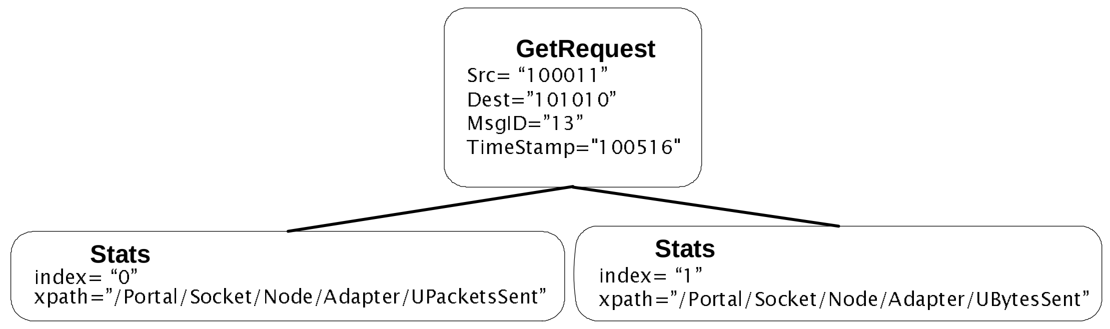

GetRequest
Src= “100011”
Dest=”101010”
MsgID=”13”
TimeStamp="100516"
Stats
index= “0”
xpath=”/Portal/Socket/Node/Adapter/UPacketsSent”
Stats
index= “1”
xpath=”/Portal/Socket/Node/Adapter/UBytesSent”

Figure 9. *GetRequest* message as XML document and as DOM document tree.

```
<GetReply  Src= “101010”  Dest=”100011”  MsgID=”13”  TimeStamp="106340">
        <Stats index=“0”    xpath=”/Portal/Socket/Node/NodeAdapter/UPacketsSent”>
                <UPacketsSent>120</UPacketsSent>
        </Stats>
        <Stats index=”1”    xpath=”/Portal/Socket/Node/NodeAdapter/UBytesSent”>
                <UBytesSent>12384</UBytesSent>
        </Stats>
</GetReply>
```

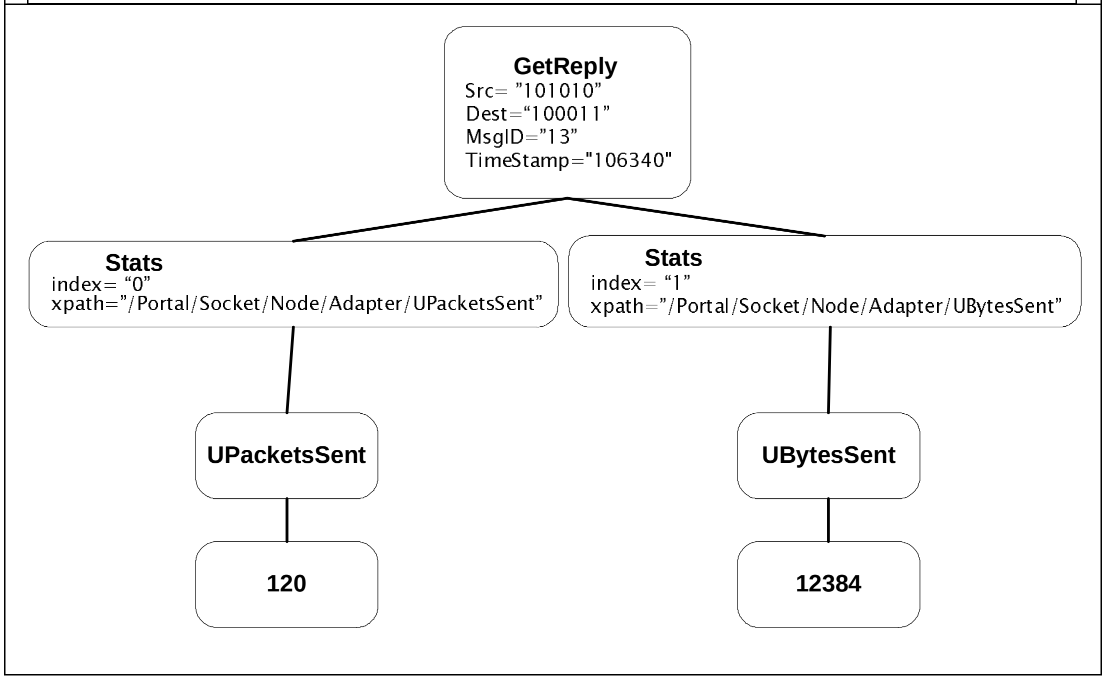


Figure 10. *GetReply* message as XML document and as DOM document tree.

The steps performed by a portal when it receives a request are illustrated in Figure 10. A request to access a statistic or a schema arrives to the portal at the overlay socket of the portal which is connected to the monitor overlay network. The payload of the overlay message is an XML formatted monitor message which contains one or more statistic elements each containing a request to read or modify a statistic. An XML parser in the portal transforms the monitor message into a tree-structured DOM document. Each statistic element in the document results in an access to the statistics API of the object that is identified in the XPath expression of the query. Each access returns a DOM element with the result of the request, which is added to a DOM document. When all statistic element of a request message are processed, and after some manipulation, the DOM document with the results consists of a reply message. The DOM document is translated into an XML document with text and tags by an XML serializer. The result is a monitor message, which is transmitted to a monitor using the overlay socket.

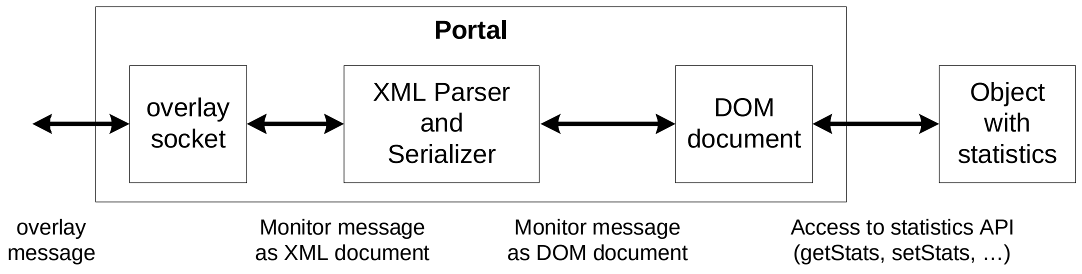

Portal
overlay socket
XML Parser and Serializer
DOM document
Object with statistics
overlay message
Monitor message as XML document
Monitor message as DOM document
Access to statistics API (getStats, setStats, …)

Figure 11. Functions performed by portal when accessing a statistic.

The request and reply messages for queries in the monitor protocol are as follows:

- **ReadSchemaRequest, ReadSchemaReply, WriteSchemaRequest, WriteSchema Reply:** A request for an XML schema of readable statistics (*ReadSchemaRequest*) or writeable statistics (*WriteSchemaRequest*) is sent from the monitor to a portal. Each query can sent multiple requests for schemas. Each request is specified as a query element that identifies an XPath. The portal translates a request into a call to either *GetReadSchema()* or *GetWriteSchema()* of the statistics API. The portal transforms the requested XML schemas into well-formatted XML documents and sends the result in a reply message (*ReadSchemaReply* or *WriteSchemaReply*). When the schema cannot be retrieved or an exception is raised at the portal, the reply message returns an error tag that explains the error, e.g., *<error>* content of error message *</error>*. The message format for requesting and sending a schema with *ReadSchemaRequest* and *ReadSchemaReply* are shown below. Requesting the schema for writeable statistics is done in the same fashion.

```
<ReadSchemaRequest    Sender="…"  Dest="…"  MsgID="…"  TimeStamp="…">

    <Stats index="0" xpath= "…" />
    . . .
</ReadSchemaRequest>

<ReadSchemaReply  Sender="…" Dest="…" MsgID="…" TimeStamp="…">
    <Stats index="0" xpath= "…">
        <xsd:schema xmlns:xsd="http://wwww/w3/org/2000/08/XMLSchema">
        . . .
        </xsd:schema>
    </Request>
    … . .
</ReadSchemaReply>
```

- **GetRequest, GetReply:** A *GetRequest* message is a request from the monitor to the portal to retrieve statistics information from a portal. A request may contain multiple statistic element, each specifying an XPath expression of a. When a portal receives a *GetRequest* message, it calls *getStats* for each statistics element. The results of *getStats* are sent to the sender of the request message in a *GetReply* message. When the query is invalid or when an exception is raised at the portal, the *GetReply* message contains an error message. A reply message contains the same message identifier and timestamp as the request message that started the query.

```
<GetRequest  Sender="..." Dest="..." MsgID="..." TimeStamp="...">
    <Stats index="0" xpath= "..." />
    . . .
</GetRequest>

<GetReply  Sender="..." Dest="..." MsgID="..." TimeStamp="...">
    <Stats index="0" xpath= "...">
        Well-formed XML message that contains the requested statistic or an
        error message
    </Stats>
    . . .
</GetReply>
```

- **SetRequest, SetReply:** A *SetRequest* message, sent from a monitor to a portal, contains requests to modify statistics. Each statistic element in the request specifies an XPath expression and the new content of the element. For each statistic element, the portal issues a *setStat()* call to the statistics API, which sets the new value and returns the result. When all requests are processed, the portal sends a SetReply message which contains the query as well as the new value of the statistic. If the query could not be completed, for example, the XPath expression was not invalid or an exception is raised, then an error message is returned in the *SetReply* message. The *SetReply* message has the same message identifier and timestamp as the corresponding *SetRequest* message.

```
<SetRequest  Sender="..." Dest="..." MsgID="..." TimeStamp="...">
    <Stats index="0"  xpath= "...">
        Well-formed XML message that contains the value of the statistic
    </Stats>
    . . .
</SetRequest>

<SetReply  Sender="..." Dest="..." MsgID="..." TimeStamp="...">
    <Stats index="0" xpath= "..." />
     . . .
</SetReply>
```

- **SetTriggerRequest, SetTriggerReply:** A *SetTriggerRequest* message, sent from a monitor to a portal, contains requests to set up a notification trigger. Each *SetTriggerRequest* contains information about one notification trigger, including the conditionXPath of the trigger, logical operator used by the trigger, the value for comparison, dataXPath of the trigger, and polling period for checking the trigger condition. When the *SetTriggerRequest* is received by the portal, the portal checks the validity of the conditionXPath and the availability of the logical operator definition, if either of the two gives a negative result, the *SetTriggerReply* would contain an error message, otherwise a trigger will be set up. The trigger polls the application constantly, the time interval between two polling is specified by the pollingPeriod value in the *setTriggerRequest* message, and the XPath of the statistic element the trigger polls is specified by the condtitionXPath value in the *SetTriggerRequest*. The trigger, then does a comparison between the result of the polling and a reference value (compareValue in the *SetTriggerRequest*), the logical operation used in this comparison is specified by the ConditionOperator value in the *SetTriggerRequest*. When this logical comparison produces a positive result, Portal will send a notification message to the monitor, the notification will contain the value of the statistic specified by dataXPath.

```
<SetTriggerRequest  Sender="..." Dest="..." MsgID="..." TimeStamp="...">
    <Stats index="0"  xpath= "/TriggerList/Trigger">
        Well-formed XML message that contains the information about the trigger.
    </Stats>
    . . .
</SetTriggerRequest>

<SetTriggerReply  Sender="..." Dest="..." MsgID="..." TimeStamp="...">
    <Stats index="0" xpath= "..." />
        error message or a  well-formed XML structure representing the trigger
</SetTriggerReply>
```

- **NotificationRequest, NotificationReply:** A *NotificationRequest* message is created by a trigger in the portal and sent from the portal to monitor when the pre-defined condition described by a *SetTriggerRequest* is satisfied. The *NotificationRequest* contains statistic elements whose XPath is given in the *SetTriggerRequest*(dataXPath). When the *NotificationRequest* is sent, the polling action of the trigger is suspended for a short period of time to allow the monitor to react to the event. For every *NotificationRequest* received by a monitor, a *NotificationReply* is sent back to the portal as an acknowledgement. If no *NotificationReply* is received, the portal would retransmit the *NotificationRequest*, the number of retransmission is defined in the configuration file for the portal. The message ID for both the *NotificationRequest* and the *NotificationReply* is the same as the message ID of the *SetTriggerRequest* that creates the notification trigger.Remark: when a notificationRequest is sent out, the polling of statistics in the portal will stop for pollingPeriod*retransmission, to allow the monitor to respond to the notification.

```
<NotificationRequest  Sender="..." Dest="..." MsgID="..." TimeStamp="...">
    <Stats index="0"  xpath= "...">
        Well-formed XML message that contains the value of the statistic
    </Stats>
    . . .
</NotificationRequest>

<NotificationReply  Sender="..." Dest="..." MsgID="..." TimeStamp="...">
            <Stats index="0" xpath= "..." />
             . . .
    </NotificationReply>
```

- **RemoveTriggerRequest, RemoveTriggerReply:** A *RemoveTriggerRequest*, sent from the monitor to portal, stops a trigger's polling action and remove it from the portal's trigger list. The trigger to be removed can be identified by its alias, or its index in the trigger list. The monitor can also send out a *RemoveTriggerRequest* that removes all the triggers set up by itself. If the Trigger is successfully removed, the *RemoveTriggerReply* would contain the message ID used by the trigger when sending *NotificationRequest*, otherwise, the reply would contain an error.

```
<RemoveTriggerRequest  Sender="..." Dest="..." MsgID="..." TimeStamp="...">
    <Stats index="0"  xpath= "...">
        xml message containing the alias or index of the trigger to be removed
    </Stats>
    . . .
</RemoveTriggerRequest>

<RemoveTriggerReply  Sender="..." Dest="..." MsgID="..." TimeStamp="...">
        <Stats index="0" xpath= "..." />
        . . .
    </RemoveTriggerReply>
```

Programming with Portals

A portal is a component of an application program that is being monitored. Portals are configured from a HyperCast configuration file, which specifies the attributes of the portal as well as the attributes of the overlay socket running in the portal. Portals and monitors that join the same monitor overlay network must have compatible configuration files. An application with an overlay socket and a portal requires two configuration files, one file for the configuration of the overlay socket created by the application and another file for the portal and the overlay socket running inside by the portal. A configuration file for a portal or monitor contains an attribute *MonitorAndControl* that specifies the properties of a portal or a monitor. The configuration of a portal is done as follows:

```
<MonitorAndControl>
    <Portal>
        <MonitorAddress /> 
        <TimeAdvertise>5000</TimeAdvertise>
        <MonitorTimeout>60000</MonitorTimeout>
        <Transcript>
            <RecordIncoming>true</RecordIncoming>
            <RecordOutgoing>true</RecordOutgoing>
            <Append>true</Append>
        </Transcript>
    </Portal>
</MonitorAndControl>
```

The element *MonitorAddress* initializes the address of the monitor. The address is the logical address of the monitor in the monitor overlay network. In the above example the address is left empty. The remaining attributes set parameters at the portal. *TimeAdvertise* specifies the time inteval in milliseconds between transmissions of *AdvertisePortal* messages by this portal. *MonitorTimeout* is timeout value. If a portal has not received an *AdvertiseMonitor* message for *MonitorTimeout* milliseconds, the portal stops sending *AdvertisePortal* messages to this monitor. A portal can be set to record all transmitted or received monitor protocol messages into a file. The element *Transcript* contains configuration information for this feature. The flags *RecordIncoming* and *RecordOutgoing*, respectively, determine if incoming and outgoing messages are recorded. The flag Append specifies the recorded data is appended to the existing file, or if the content of the file is overwritten.

A portal is created by an application program in a similar fashion as an overlay socket. The application program first creates a configuration object and then uses the configuration object to create a portal. When creating a portal, the application provides an object that will be managed through the portal. This can be the application, an overlay socket run by the application, or any other object that implements the statistics API.

All statistics that are managed through the portal are referenced have element as root element, and all XPath expressions of statistics managed through a portal have the prefix */Portal*. If an overlay socket (with statistic name *Socket*) is managed through the portal, then all accesses to the statistics of the socket have the prefix */Portal/Socket*. The portal may also have statistics of its own. The top of the hierarchy of statistics accessed through a portal are illustrated in Figure 12. The managed object is the object that is provided to the portal when the portal was created, i.e., the application program, an overlay socket, or some other object.

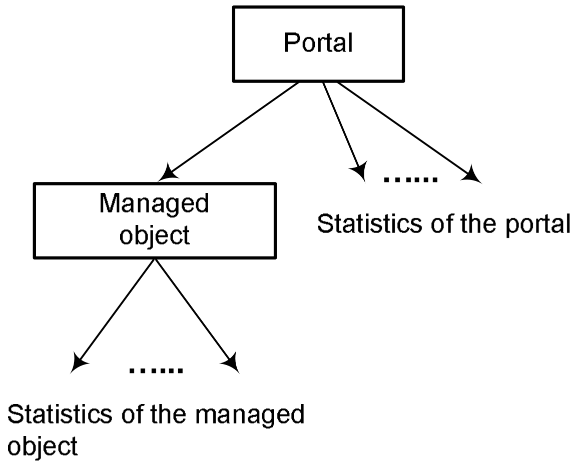


Figure 12. Hierarchy of statistics accessed through a portal.

The following code fragment is a HelloWorld program that creates an overlay socket and a portal.

```
import org.apache.xpath.XPath;
import org.w3c.dom.Document;
import org.w3c.dom.Element;
import hypercast.HyperCastConfig;
import hypercast.HyperCastStatsException;
import hypercast.I_OverlaySocket;
import hypercast.I_Stats;
import hypercast.SimpleStats;
import hypercast.StatsProcessor;
import hypercast.MonitorAndControl.I_Portal;
import hypercast.util.StatsUtil;

public class ApplicationWithPortal {
   private I_OverlaySocket socket;
   private I_Portal portal;
   private String statisticsName = null;

   public synchronized static void main (String[] args) {
      ApplicationWithPortal app = new ApplicationWithPortal();
      app.startSockets();
   }

   public ApplicationWithPortal() {
      HyperCastConfig sConf = HyperCastConfig.createConfig("hypercast.xml");
      socket = sConf.createOverlaySocket(null);
      HyperCastConfig pConf = HyperCastConfig.createConfig("Portal.xml");
      portal = pConf.createPortal(socket, "Socket");
   }

   public void startSockets() {
      portal.activatePortal();
      socket.joinOverlay();
   }
}
```

The program illustrates the similarity between the creation of a portal to the creation of an overlay socket. A portal is created as follows:

```
HyperCastConfig pConf = HyperCastConfig.createConfig("Portal.xml");
portal = pConf.createPortal(socket, "MySocket");
```

The first statement creates a configuration object from the file *Portal.xml*. The configuration file must contain attributes needed to configure the portal and the overlay socket in the portal. The portal is created with the *createPortal* method. The first argument of the method call specifies the object that is remotely accessed by the monitor protocol. This object must be of a type that implements the *I_Stats* interface. The second argument specifies the name by which the object is referenced. In the above example, the overlay socket *MySocket* is assigned to the portal, and is associated with the element Socket. This results in a hierarchy of statistics as shown in Figure 3 and 4. Recall that all statistics accessed through a portal are contained in the root element *Stats*. Therefore, the statistic that provides the number of bytes transmitted by the node adapter are referenced through the XPath expression */Portal/MySocket/Node/NodeAdapter/UBytesSent.*

Alternatively, the portal could be created with the application program as the managed object. In this case, the invocation to create the portal would be as follows:

```
portal = pConf.createPortal(this, "MyAppl");

private StatsProcessor statsProcessor;
```

initStatisticsStructure();

Here, the application must provide an implementation of the I_Stats interface. This can be done as shown in the subsection “Statistics for application programs”, with a StatsProcessor object.

If the implementation results in a hierarchy of statistics as shown in Figure 8 and 9, the statistics accessed through the portal are accessed with XPath expressions with prefix */Portal//MyApplication/Socket.* (Note that *Socket* is the default name of the statistic representing the overlay socket. By invoking socket.setStatsName(“MySocket”), the prefix is changed to */Portal/MyApplication/MySocket.*)

Once a portal is created, the portal can be opened and closed with the method invocations

MyPortal.activatePortal();

MyPortal.deactivatePortal();

When a portal is opened, the overlay socket in the portal joins the monitor overlay network, and the portal sends and receives advertisement messages. A newly created socket is not opened. As soon as the portal is opened it can be accessed by a monitor. When a portal is closed, the portal leaves the monitor overlay network, and is no longer accessible by a monitor. The *openPortal* and *closePortal* methods give an application program explicit control whether it can be remotely monitored.

The example program illustrates that the portal and the overlay socket in the application operate independently. An application program can start a portal even if the program does not have an overlay socket. In fact, portals can be deployed for remote monitoring of any distributed application that implements the *I_Stats* interface.

Next, we given an overview of the API for creating and operating portals. With exception of the method to create a portal, the methods are defined in the *I_Portal* interface.

I_Portal createPortal (I_Stats StatsObject, String StatsName)

I_Portal createPortal (I_Stats StatsObject)

This method of the class *HyperCastConfig* creates a new portal. The configuration object must specify attributes needed to create a portal. The first argument is an object that implements the *I_Stats* interface. This object contains the statistics that are made available for remote monitor and control through the portal. The second argument specifies the name of the element by which the object will be referenced. If the object already has a name assigned, then the second argument can be dropped.

void activatePortal ()

Activating a portal means that the overlay socket of the portal joins the monitor overlay network, and the portal sends and receives advertisement messages. An open portal processes queries that arrive from a monitor. When a portal is created, it must be explicitly opened by the application program.

vod deactivatePortal ()

When a portal is deactivated, the overlay socket of the portal leaves the monitor overlay, and the portal no longer sends or receives monitor protocol messages. A closed portal can be re-opened with the *openPortal* method.

vod closePortal ()

When a portal is closed, the overlay socket of the portal is closed.

void startTranscript(String filename)

Starts a transcript of all transmitted and received monitor protocol messages. The command writes an XML declaration and an opening tag <*transcript*> to the file, and then appends each transmitted or received monitor protocol message. The attributes specified in the *Transcript* element determine which messages are recorded and if recorded messages are appended to the existing file.

void stopTranscript()

Stops the transcript of monitor protocol messages, and writes the closing tag *</transcript>*.

void setSilentOn()

void setSilentOff()

These methods turn the silent mode of the portal on and off. In silent mode, the portal does not send advertisement messages. By default, the silent mode is turned off. When the set of portals and monitors does not change over longer time periods, setting the monitor to silent mode reduces the traffic due to advertisement messages.

Programming with Monitors

A monitor in the HyperCast software is part of a monitor application that exchanges messages with portals over a monitor overlay network. In a later section, we will discuss the *RunController*, a monitor application that is part of the HyperCast software. Here, we discuss how to write a monitor application.

Each monitor maintains a table, called the portal list, which contains the portals known to the monitor. A monitor learns about portals through the configuration file or through advertisement messages of the monitor protocol. An entry in the portal list contains the logical address of a portal in the monitor overlay network, the time since the last advertisement message is contained, and whether the portal is active or inactive. A portal is marked as inactive if the monitor has not received an advertisement from the portal for a long period of time. A monitor application addresses portals through the index in the portal list. This makes the monitor application independent of the addressing scheme used in the monitor overlay network. When a monitor sends a request message to portal 0, the request message is sent to the logical address of the portal that is located in position 0 of the portal list. Entries for inactive portals are not deleted. It is possible to delete inactive entries and reset the index of the portal list. This, however, may change the addresses of active portals.

A monitor is configured in the same way as a portal, i.e., from a HyperCast configuration object. The configuration file for the monitor must contain an attribute MonitorAndControl that specifies the properties of the monitor. The following is an example of a monitor configuration.

```
<MonitorAndControl>
  <Monitor>
        <PortalList />
        <TimeAdvertise>5000</TimeAdvertise>
```

```
                        <PortalTimeout>60000</PortalTimeout>
                        <QueryTimeout>5000</QueryTimeout>
                        <Transcript>
                            <RecordReceives>true</RecordReceives>
                            <RecordSends>true</RecordSends>
                            <Truncate>true</Truncate>
                        </Transcript>
        </Monitor>
    </MonitorAndControl>
```

The attributes are similar to those of a portal. The attribute *PortalList* initializes the portal list of the monitor. In the above example the attribute is empty. If the portal list is initialized to the logical addresses 00110, 10110, and 10011, the attribute is replaced by:

```
    <PortalList>
        <LogicalAddress>00110</LogicalAddress>
        <LogicalAddress>10110</LogicalAddress>
        <LogicalAddress>10011</LogicalAddress>
    </PortalList>
```

Portals that are entered in this fashion are static, in the sense that they are never set to an inactive status because of missing *AdvertisePortal* messages. The attribute *TimeAdvertise* specifies how frequently *AdvertiseMonitor* messages are sent by the monitor. *PortalTimeout* specifies the maximum time that a monitor waits for an *AdvertisePortal* message before setting the portal list entry to be inactive. The time unit is a millisecond. The Transcript element has the same interpretation as in the *Portal*.

Below is a complete application program that uses a monitor.

```
import hypercast.HyperCastConfig;
import hypercast.HyperCastStatsException;
import hypercast.util.*;
import hypercast.MonitorAndControl.*;
import org.w3c.dom.Element;

public class SimpleMonitor implements I_ReceiveCallback, I_TimeoutCallback {
    private I_Monitor monitor;
    private String stat = "/Portal/Appl/Socket/Node/LogicalAddress";

    public static void main (String[] args) {
        SimpleMonitor simpleMonitor = new SimpleMonitor();
        int activePortalIndex = simpleMonitor.listenForPortal();
        simpleMonitor.sendRequest (activePortalIndex);
    }

    public SimpleMonitor() {
        HyperCastConfig config = HyperCastConfig.createConfig("Monitor.xml");
        monitor = config.createMonitor();
        monitor.activateMonitor();
    }
    public int listenForPortal() {
        while (true) {
            System.out.println ("Checking for active Portal...");
            int [] activePortals = monitor.getActivePortalIndices();
            if (activePortals.length > 0) {
                System.out.println ("Found with index: " + activePortals[0]);
                return activePortals[0];
            }
            try { Thread.sleep (3000); } catch (InterruptedException e) {}
        }

    }
    public void sendRequest (int portalIndex) {
        MonMessage request = monitor.createGetRequest();
        request.addStat (StatsUtil.createXPath (stat));
        try {
            monitor.sendTo (portalIndex, request, this, this);
        } catch (NoSuchPortalException nspe) {
            System.err.println ("Bad Portal: " + portalIndex + "\n" + nspe);
        }
        System.out.println ("Sent request with ID: " + request.getMessageID());
    }
    public void close() {
        System.out.println ("Closing...");
        monitor.closeMonitor();
    }

    public synchronized void receiveMessage (MonMessage reply) {
        System.out.println ("Received reply with ID: " + reply.getMessageID());
        try {
            Element[] value = reply.getStatValue (StatsUtil.createXPath(stat));
            String scalar = value[0].getChildNodes().item (0).getNodeValue();
            System.out.println ("Statistic value: " + scalar);
        } catch (HyperCastStatsException hcse) {
            System.err.println ("Stats error reading reply message:\n" + hcse);
        }
        close();
    }
    public synchronized void timeoutMessage (MonMessage message) {
        System.out.println ("Timeout, message ID: " + message.getMessageID());
        close();
    }
}
```

The monitor application *SimpleMonitor* creates a configuration object from the configuration file *Monitor.xml*, and then instantiates the monitor from the configuration object. The instantiation of the monitor is done with:

```
I_Monitor MyMonitor = config.createMonitor();
```

Once the monitor is created, the monitor is opened by calling

```
MyMonitor.activateMonitor();
```

Activating the monitor is analogous to opening a portal. It means that the overlay socket in the monitor will join the monitor overlay network, and transmit *AdvertiseMonitor* messages. A monitor can disconnect from the monitor overlay network by deactivating the monitor, using the method call

```
MyMonitor.deactivateMonitor();
```

After the monitor is created the application program creates and configures a query for a statistic:

```
MonMessage MyQuery = MyMonitor.createGetRequest();
```

This creates an "empty" request message without a statistic element. Request messages to modify a statistic are created with the method *createSetRequest*, and request messages to retrieve schemas are done with the methods *createReadSchemaRequest* and *createWriteSchemaRequest*. Next, the program adds a statistic element to the query:

```
MyQuery.addStat(StatsUtil.createXPath("/Portal/Socket/Node/NodeAdapter/UBytesSent"
));
```

This adds a statistic element that requests retrieval of the statistic */Portal/Socket/Node/NodeAdapter/UBytesSent*. The request message is transmitted with:

```
MyMonitor.sendTo( 0, MyQuery, this, this);
```

The first argument is the address of the portal. The value 0 sends the message to the first portal (with index 0) in the portal list. The second argument contains the message. The third and fourth arguments supply, respectively, objects that implement the callback functions for processing the response to the message and for a timeout. In the example, the *SimpleMonitor* supplies both callbacks for processing a reply message and a timeout on a response. The callbacks are the methods *receiveMessage* and *timeoutMessage*. The callback for response messages is called when the reply to the request message arrives. A monitor uses the *MsgID* element in the message to match a request with the response. In the example, the *receiveMessage* method matches the XPath expression from the request message to identify the statistic element. When the statistic element is obtained, the method manipulates the statistic element to extract and display the value of the statistic. The timeout callback is invoked when the monitor waits for more than *QueryTimeout* milliseconds for a reply message. Generally, the timeout callback can implement a policy for retransmitting a request message. In the example, if a timeout occurs, the request message is transmitted, up to a maximum of five retransmissions.

Since queries may be broadcast, multiple replies may be received for the same query. For broadcast request messages, the callback method is called for each response that arrives to the query. The fourth argument specifies the timeout value, in milliseconds, for waiting on the response to the request message. A timeout occurs if the monitor waits for more than *QueryTimeout* milliseconds for a reply message.

The following is an overview of the programming interface of the monitor. Most methods are available through the *I_Monitor* interface. The methods are grouped into methods to manage the monitor, the callback functions, methods to create and send queries, methods to access and manipulate the portal list, and methods that perform miscellaneous functions.

Managing a monitor

I_Monitor createMonitor()

A monitor is created with this method of the HyperCastConfig class. If the configuration does not contain attributes needed to configure a monitor, the creation of the monitor fails.

void activateMonitor()

When a monitor is opened the overlay socket of the monitor joins the monitor overlay network, and the monitor sends *AdvertiseMonitor* messages to announce its presence in the monitor overlay network. A monitor application can send requests and receive responses as soon as a monitor has been opened.

void deactivateMonitor()

When this method is called, a monitor stops sending and receiving overlay protocol messages, and the overlay socket in the monitor leaves the monitor overlay network. A monitor application can re-open a monitor by calling the *activateMonitor* method.

void closeMonitor()

When this method is called, the monitor is firstly deactivated, and the overlay socket in the monitor is closed.

void setSilentOn()

void setSilentOff()

Turns the silent mode of the monitor on and off. In silent mode, the monitor does not send advertisement messages and does not set the status of the portals in the portal list to inactive. If the set of portals is stable and does not change, setting the monitor to silent mode may reduce the traffic generated by the monitor protocol. In a newly created portal, the silent mode is turned off.

Creating and sending queries

MonMessage createGetRequest()

MonMessage createReadSchemaRequest()

MonMessage createSetRequest()

MonMessage createWriteSchemaRequest()

A monitor application program can create four types of request message. When created, a message does not contain statistic elements. Statistics elements with requests are added using the *addStat* method of the *MonMessage* class.

void sendTo (int portalIndex, MonMessage message, I_ReceiveCallback receiveCallback,
    I_TimeoutCallback timeoutCallback)

Sends a request message to a portal. The first argument is the index of the portal in the portal list, the send argument is the message to be transmitted, the third argument is the callback method for the reply message, and the last argument is the callback when the no reply arrives after waiting for a time given by the *QueryTimeout* attribute.

void sendToAll (MonMessage message, I_ReceiveCallback receiveCallback,
  I_TimeoutCallback timeoutCallback)

Sends a monitor protocol message via broadcast which reaches all portals and nodes in the monitor overlay network. A request message that is broadcast may result in more than one reply message. When this happens, the receive callback is called for each returning reply message. When no reply message arrives after *QueryTimeout* milliseconds, the timeout callback is invoked.

Callbacks

When sending a query, a monitor application supplies two callback methods. One callback method is invoked when the monitor receives a response to the query. Another callback method is called when a timeout occurs for waiting on the response to a query. The methods belong to the interfaces *I_TimeoutCallback* and *I_ReceiveCallback*.

void receiveMessage (MonMessage message)

This is the callback method that handles an incoming response to a request message. The callback contains the received reply message as an argument.

void timeoutMessage (MonMessage message)

This callback method handles a timeout for waiting on a response to a request message. The argument contains the request message for which a timeout occurred as an argument. The timeout callback may implement a retransmission of a message.

Interacting with the portal list

The monitor maintains a database of portals, called the portal list, which contains the addresses and the status of portals. The portal list is updated from advertisement messages of the monitor protocol. A monitor application addresses portals through the index in the portal list. The monitor translates the indexes into logical addresses of the portal overlay network. The API of the monitor has a number of methods to query and manage the content of the portal list.

int[] getAllPortalIndices()

Returns the set of all portals in the monitor as an integer array. The values of the array are the indices in the portal list (with 0 as the first entry). Each entry in the portal list is either active or inactive.

int[] getActivePortalIndices()

Returns an integer array containing the active portals.

int[] getInactivePortalIndices()

Returns an integer array containing the inactive portals.

boolean isPortalActive(int portalIndex)

Tests if a portal is currently active.

void compactPortalList()

This command removes all portals that are inactive and re-computes the indices of the portal. Since the monitor application uses the index in the portal list as the address of a portal, calling this method may change the address of a portal.

void clearPortalList()

Removes all portal entries and resets the portal list. For the portal list, this method has the same effect as restarting the monitor.

void getPortalAddress(int portalIndex)

Returns the logical address of a portal.

int indexOfPortal (final I_LogicalAddress portalLogicalAddress)

Find the index of a portal by its logical address.

Other methods

void startTranscript(String filename)

Starts a transcript of transmitted and received monitor protocol messages. The command writes an XML declaration and an opening tag <*transcript*> to the file, and then appends each transmitted or received monitor protocol message. The attributes specified in the *Transcript* element determine which messages are recorded and if recorded messages are appended to the existing file.

void stopTranscript()

Stops the transcript of monitor protocol messages, and writes the closing tag <*/transcript*>.

Programming with Monitor Messages

Next we discuss how monitor applications manipulate monitor protocol messages. Since a monitor application is responsible for sending requests and processing replying queries, the application programmer of a monitor application must be familiar with building request messages and processing replies to these messages. Some messages of the monitor protocol, e.g., the *AdvertisePortal* messages, are transmitted automatically without requiring action by the monitor application. All messages discussed here are providd by the *MonMessage* class.

Adding and removing statistic elements

Monitor protocol messages are created through the *I_Monitor* API, as discussed previously. The available methods are createGetRequest, createReadSchemaRequest, createSetRequest, and createWriteSchemaRequest. The messages created in this fashion are empty in the sense that they do not include a statistic element. Statistic elements are added with the following methods.

addStat(XPath statistic)

Adds a statistic element to a request message. The location of the statistic is passed as an XPath expression and is added as an attribute to the statistic element. The method is used by monitor applications to add a request for a statistic or a schema to a request message.

addStat(XPath statistic, Element value)

Adds a statistic element to a reply message. The method is used by monitor application to add a query to modify a statistic to a *SetRequest* message. The first argument is the location of the statistic and the second argument is the value of the statistic. The value, specified in the second argument, is added as an element nested in the statistic element.

void removeStat(int index)

Removes a statistic element from a query or reply message. The statistic element is accessed by its index in the message.

void removeStat(XPath xpath)

Removes a statistic element from a query or reply message. The statistic element is accessed by matching the argument with the XPath attributed in the Stats element.

Processing a message

document getDocument()

Returns the entire monitor protocol message as a DOM document structure. This method can be invoked by a monitor application when the message should be entirely parsed and processed by the monitor application.

I_LogicalAddress getSender()

Returns the message of the sender attribute of a monitor protocol message.

I_LogicalAddress getDestination()

Returns the message of the destination attribute of a monitor protocol message.

long getTimeStamp()

Get the timestamp of a monitor protocol message.

long getMessageID()

Returns the message identifier attribute from a monitor protocol message.

int getStatsCount()

Returns the number of statistic elements in a query or reply message.

Element [] getStatValue(int index)

Returns a statistic element from a query or a reply message. If the argument is 0, the method returns the first statistic element, a 1 returns the second statistic element, and so forth.

Element [] getStatValue(XPath statistics)

Returns a statistic element from a query or a reply message. The method tries to find a match of the provided XPath expression with an XPath attribute in a statistic element of the message.

XPath getStatXPath(int index)

Returns the XPath attribute in a statistic element of a query or reply message.

Testing a message

boolean isStatError(int index)

Returns *true* if a statistic element contains an error message. The statistic element is accessed by its index in the message.

boolean isStatError(XPath xpath)

Returns *true* if a statistic element contains an error message. The statistic element is accessed by matching the argument with the XPath attribute in the Stats element.

Programming With Notification Triggers

Next we discuss how monitor applications can set up a notification trigger in the portal. This functionality allows the monitor to receive a asynchronous notification message when certain event takes place in the portal.

Setting up a notification trigger

This is part of the *I_Monitor* API, the two main methods here are: *createSetTriggerRequest* and *sendSetTriggerRequest*, the message created contains complete information about the trigger, do not use *addStats* or *removeStats* methods on the *SetTriggerRequest* message. The *sendSetTriggerRequest* would check the format and contents of the message. Exception will be thrown if there are deviations from the standard format.

MonMessage createSetTriggerRequest(XPath conditionXPath, XPath dataXPath, String operator, String compareValue, long pollingInt, String alias)

Returns a monitor message that contains all the information about a trigger. For a trigger to send out a notification message, the comparison of an application statistic with a reference value must give a positive result. The conditionXPath is the XPath representing the statistics to be polled by the trigger when making the comparision. The third argument “operator” is the String representation of the operator used in the comparison. The fourth argument “compareValue” is the reference value to be compared with; pollingInt is the period for the polling action of the trigger. The last argument is the alias for the trigger. The “dataXPath” is the XPath for the statistic element that is to be contained in the notificationRequest sent out when the condition is satisfied.

MonMessage sendSetTriggerRequest(int portalInd, MonMessage setTriggerReq, I_ReceiveCallback callbackForReq, I_TimeoutCallback toCallbackForReq, I_ReceiveCallback callbackForNoti)

Sends a monitor message that contains all the information about a trigger. The portalInd specifies which portal this SetTriggerRequest is sent to. There are three callbacks in the arguments, the first two are ReceiveCallback and TimeoutCallback for the setTriggerRequest, the third callback is for the NotificationRequest. The callback for NotificationRequest cannot be null here.

Removing a notification trigger

MonMessage createRemoveTriggerRequest( )

Creates a Monitor Message that removes all the triggers set up by this monitor.

MonMessage createRemoveTriggerRequest(int index)

Creates a Monitor Message that removes the trigger in the portal identified by index. If the index is pointing to a trigger that is not set up by this monitor, then the trigger will not be removed and the reply will carry a error message.

MonMessage createRemoveTriggerRequest(String alias)

Creates a Monitor Message that removes the trigger in the portal identified by alias.If the trigger identified by the trigger alias is not set up by this monitor, then the trigger will not be removed and the reply will carry a error message.

3.4. THE RUNCONTROL AND RUNSERVER APPLICATIONS

This section describes two applications, *RunControl* and *RunServer*, that exploit the features of the monitor and control system to execute large scale measurements of HyperCast overlay networks. *RunControl* and *RunServer* have been used to monitor overlay networks that involve more than 100 computers, where each computer runs an application with up to one hundred overlay sockets.

Figure 13 shows the interaction of the *RunControl* and *RunServer* applications. *RunControl* is a monitor application that acts as console for measurement experiments for an overlay network. *RunControl* contains a monitor that is attached to a monitor overlay network. *RunControl* provides to users a command line interface that permits users to start and monitor a measurement experiments with a HyperCast overlay network. The commands of RunControl are translated into queries of the monitor protocol.

RunServer is an application that maintains an array of overlay sockets and that has a portal. *RunControl* controls the experiment by sending monitor messages to portals of the *RunServers*. *RunServer* starts all sockets with identical configuration file. The overlay sockets running in the *RunServer* join the overlay network only if so instructed by a command of *RunControl*.

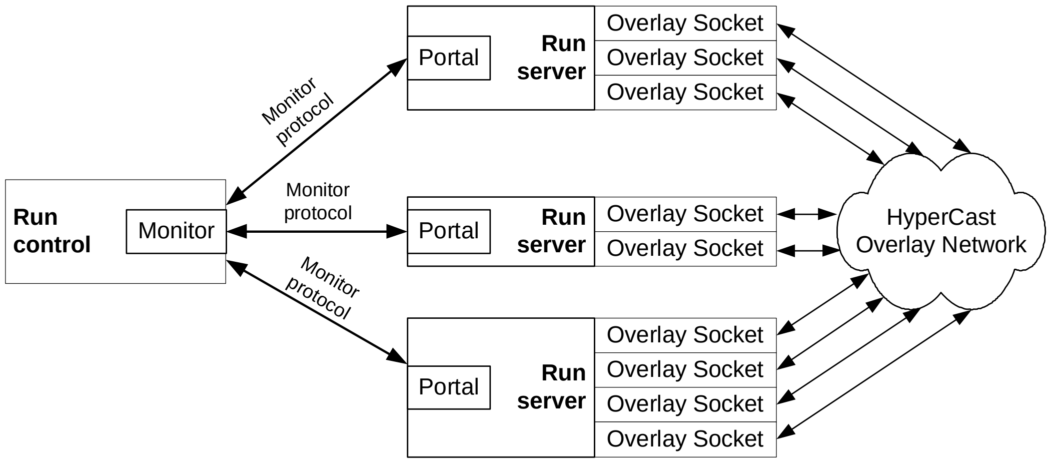


Figure 13. Communication between *RunControl* and *RunServers*.

RunControl is started with the following command:**1**

```
java -cp “hypercast3.0.jar;runcontrol.jar;xalan.jar”
                 commandline.RunControlCLI –mc Monitor.xml
```

The command assumes that the class path is set so that the libraries can be found. When downloaded, the JAR files are found in the lib subdirectory. The file Monitor.xml**2** is the configuration files for the monitor and overlay socket in the monitor. When started, *RunControl* displays a message which includes the (address) of RunControl in the monitor overlay network and a command prompt:

```
NONE_Node started: 128.143.69.53:7550:7551
Opening Monitor at Logical Address: 128.143.136.78:1500:1501
Monitor successfully created.
Started RunControl Command-line interpreter at Tue Nov 30 17:44:37 EST 2004
>
```

At the prompt, the user types one of the available commands.

A *RunServer* application can be started with the following command:

```
java -cp
“lib/hypercast3.0.jar;lib/runcontrol.jar;lib/xalan.jar;lib/bcprov-
jdk14-122.jar”
   runserver.RunServer –ns 10 –pc Portal.xml –sc hypercast.xml
```

1 In Unix command shells semicolons (“;”) need to be replaced by colons (“:”).

2 We assume that the Monitor.xml file employs the NONE protocol as overlay protocol to connect monitor and portals. In the NONE protocol logical addresses are identical to physical addresses.

With the option *–ns 10, RunServer* starts 10 overlay sockets, which, however, do not initially join the overlay network. *Portal.xml* is the configuration file for the portal, and *hypercast.xml* is the configuration file used by the overlay sockets of the *RunServer* application. The available options for the commands are given in the appendix to this chapter. In an experiment that involves multiple computers, one *RunServer* must be run on each systems involved in the experiment.

A Simple Experiment with RunControl and Runserver

Once *RunControl* and a set of *RunServers* are started, the user of the *RunControl* application can issue commands at the command prompt. A simple experiment may use the following set of commands.

```
NONE_Node started: 128.143.69.53:7550:7551
Opening Monitor at Logical Address: 128.143.69.53:7550:7551
Monitor successfully created.
Started RunControl Command-line interpreter at Tue May 31 06:21:28 EDT 2005

> list_portals
<Portal 0: Address=128.143.69.53:7552:7553 Status=Active>
1 Portals listed

> list_sockets
Sockets created   = 10
Sockets started   = 0
Sockets available = 10

> start_sockets
starting all sockets

> wait_until_stable
Stable at 3.7970002 seconds since experiment started.   (Command  took 0.016
seconds to run.)

> kill_remote_servers
Killing 1 servers.
```

Here, we assume that there is one *RunControl* and one *RunServer* command started as shown previously. The RunControl commands cannot be interrupted. Thus, when a a command gets stuck, the entire application must be restarted. In most cases, a restarted RunControl application resynchronizes with currently active *RunServer* applications.

We next discuss each of the above commands.

```
> list_portals
```

The command displays the status of portals at remote RunServer applications. The output

```
<Portal 0: Address=128.143.69.53:7552:7553 Status=Active>
```

Indicates that there is one portal, with index 0, that is available at logical address 128.143.69.53:7552:7553. The Status=active states that the remote *RunServer* and the *RunControl* application are communicating via the monitor protocol.

```
> list_sockets
```

This commands lists the number of available overlay sockets at the remote *RunServer* application. The command lists the number of sockets that have been created by the remote RunServer application, the number of sockets that have joined *the* overlay network, and the number of overlay sockets that have not yet started. The command results in the following output

```
Sockets created   = 10
Sockets started   = 0
Sockets available = 10
```

The output indicates that 10 overlay *sockets* have been created, but no overlay socket has joined the overlay network.

```
> wait_until_stable
```

With this command, *RunControl* waits until the overlay network has stabilized. The command requires that overlay sockets have a statistic */Socket/Node/Stable* which takes value *false* and *true*. The DT protocol supports this statistic, but other overlay protocols may not support this statistic. The command issues queries that request the value of the *Stable* statistic from all overlay sockets. As long as one RunServer returns a *false* for one of its overlay socket, the *RunControl* repeats the query every two seconds. When all *RunServers* have reported that all their overlay sockets have reached a stable state, the command is completed. Then, the command displays how long the command was running. The command displays the elapsed time since the last execution of the command *start_sockets*.

```
> kill_remote_servers
```

This command terminates all remote *RunServer* applications. This command requests each RunServer application to change the statistic */Portal/Appl/KillServer* to *true.* When the statistic is set to true, the *RunServer* application terminates.

A list of all available commands can be found in Appendix I.

RunServer Statistics

The *RunServer* application implements the I_Stats interface and supports its own statistics. When a *Runserver* creates a portal, it specifies the application as the managed object. The overlay sockets running in the application are accessed by specifying an index in the XPath expression. For example, to access the number of bytes that were transmitted by the node adapter of the first overlay socket at *Runserver* the monitor requests the following statistic:

/Portal/Appl/Socket[1]/Node/NodeAdapter/UBytesSent

The complete set of statistics defined in *Runserver* are illustrated in Figure 14 and explained in Table 1.

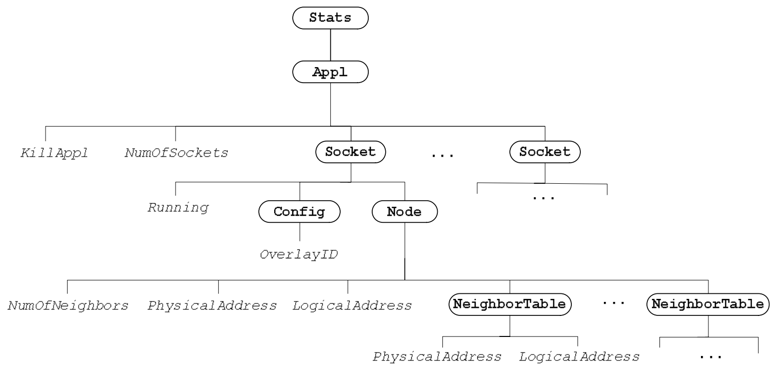


Figure 14. Statistics required by *RunServer*.

Table 1. Explanation of the *RunServer* statistics.

NumOfSockets	The number of overlay sockets that are started by *RunServer*.
Socket	Gives access to the statistics of an overlay sockets in *RunServer*. There are *NumOfSockets* overlay sockets. To access a specific overlay socket, the XPath expression specifies an index, e.g., *Socket[i]* with *i = 1, 2, …, NumOfSockets*.
Running	Indicates whether an overlay socket is currently joining the overlay network, i.e., is in state *running*. If *Running[i]=true* then the *i*-th overlay socket (*i = 1, 2, …, NumOfSockets*) joins the overlay network. If *Running[i]=false* the corresponding overlay sockets leaves the overlay network.
KillAppl	If the statistic is set to *true*, the *RunServer* application terminates.
OverlayID	Overlay identifier of the overlay network that is joined by the overlay socket.
LogicalAddress	Logical address of the overlay socket.
PhysicalAddress	Physical address of the overlay socket.
NumOfNeighbors	Number of neighbors of the overlay node in the overlay topology
NeighborTable	The statistics of a neighborhood table entry in the overlay node of an overlay socket. There are *NumOfNeighbors* entries in the table. To access a specific entry, the XPath expression specifies

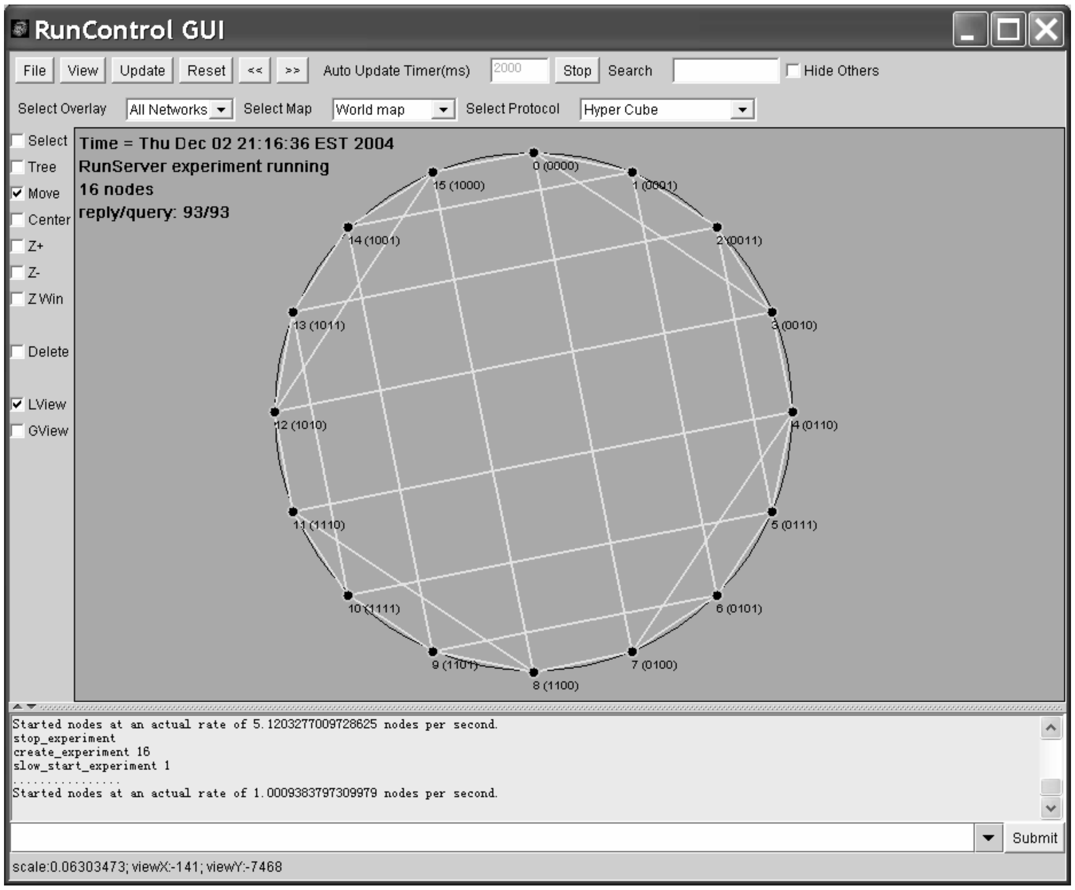
	an index, e.g., *NeighborTable[i]* with *i = 1, 2, …, NumOfNeighbors*. Each entry has two elements with names *LogicalAddress* and *PhysicalAddress.*

RunControl GUI

The *RunControl* GUI is a graphical front end to the *RunControl* application that displays the logical topology of an overlay network. *RunControlGUI* is a standalone application that interacts with remote *RunServers* and displays the overlay network as a graph. *RunControlGUI* can interact with remote application programs other than *Runservers*, as long as the applications support the statistics of *RunServer* (as shown in Figure 14). The *RunControlGUI* application is started in a similar fashion as RunControl witht eh following command:

```
java –cp "hypercast3.0.jar;runcontrol.jar;xalan.jar"
                 gui.RunControlGUI –mc Monitor.xml
```

Figure 15 depicts the user interface of *RunControlGUI* for a Pastry overlay network with 16 overlay sockets. The user selects the overlay protocol of the monitored overlay network, and *RunControlGUI* depicts an appropriate graph. *RunControlGUI* includes the complete *RunControl* command line interface. Users can type commands in the window at the bottom of the user interface. Results are displayed in the gray shaded window.



Auto Update Timer(ms) 2000

Select Overlay: All Networks · Select Map: World map · Select Protocol: Hyper Cube

Time = Thu Dec 02 21:16:36 EST 2004
RunServer experiment running
16 nodes
reply/query: 93/93

```
Started nodes at an actual rate of 5.1203277009728625 nodes per second.
stop_experiment
create_experiment 16
slow_start_experiment 1
................
Started nodes at an actual rate of 1.0009383797309979 nodes per second.
```

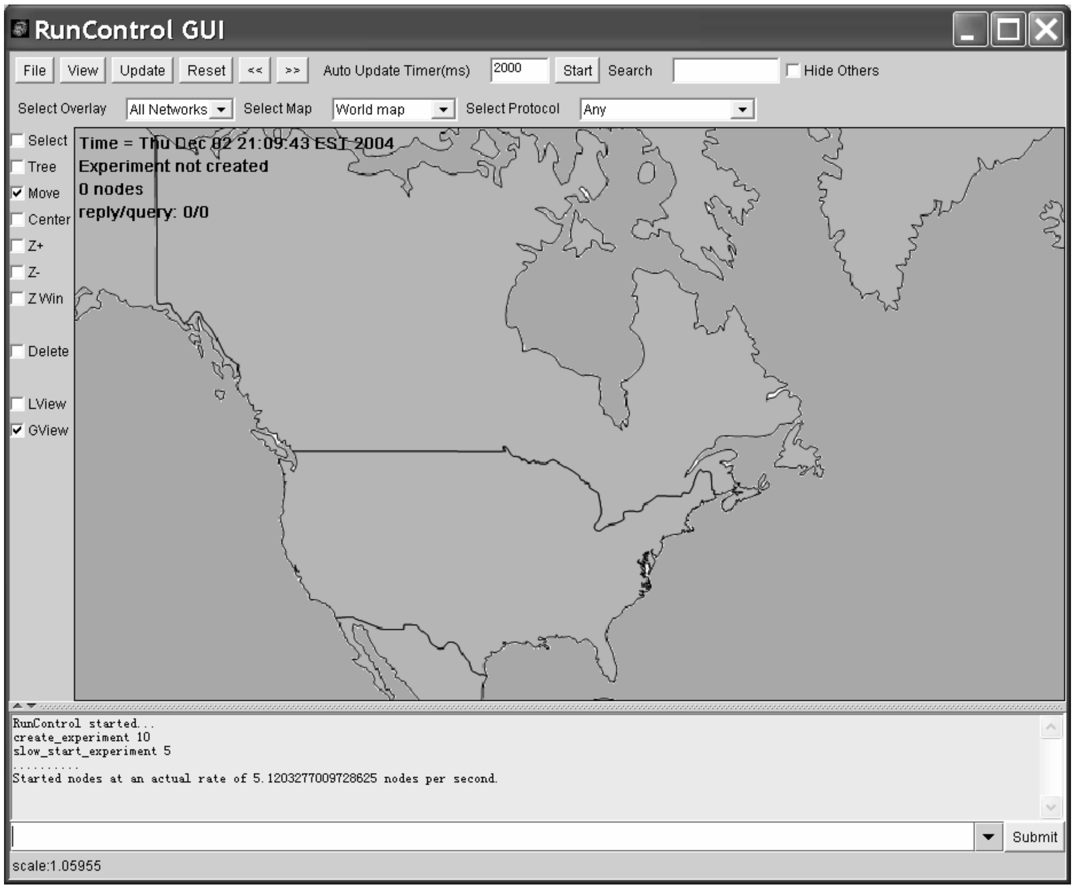
scale:0.06303473; viewX:-141; viewY:-7468

Figure 15. RunControl GUI (Logical View of a Pastry overlay).

RunControlGUI can display a *logical view* and a *geographical view* of an overlay network. The user switches between these views by checking either the *LView* (logical view) or *GView* (logical view) box. The logical view presents the overlay network as a logical graph selecting a representation that is suitable for the overlay topology. The geographical view displays the actual location of the overlay sockets in the overlay network. To display the geographical location of applications, *RunControlGUI* requires information on the location of remote applications so that the physical address of an overlay socket can be related to its geographical location. *RunControlGUI* reads geographical information from the file *location.xml* which has the following content:

```
<AddressLocator>
  <Host>
        <Address>128.143.71.21</Address>
        <Location>
                <LocationName> Charlottesville </LocationName>
                <Latitude>38.05</Latitude>
                <Longitude>-101.52</Longitude>
        </Location>
  </Host>
  <Host>
        ...
  </Host>
<AddressLocator>
```

The root element, *AddressLocator*, contains a sequence of *Host* elements that each specifies the address and the location of a host. In general, the address should be an initial substring (prefix) of the physical address of an overlay socket of an application. On the Internet, the address is generally the IP address of the host running the *RunServer* application. The location is specified in terms of latitude and longitude. The additional element *LocationName* has informational value. When *RunControlGUI* receives monitor protocol messages, it tries to match one of the *Host* elements with a prefix of the physical address of the overlay socket in the message, and displays the information in the window. If no match is found, a default geographical location is chosen for the overlay socket. The geographical view of an overlay network is shown in Figure 16. The user selects one of the available maps from a pull down menu. The available maps are image files, and meta information about the image files is found in the file *map.xml*. In Appendix III, we explain how the geographical locations are computed. Users can edit the file *map.xml* and add new image files to the application.



```
Time = Thu Dec 02 21:09:43 EST 2004
Experiment not created
0 nodes
reply/query: 0/0
```

```
RunControl started...
create_experiment 10
slow_start_experiment 5
..........
Started nodes at an actual rate of 5.1203277009728625 nodes per second.
```

```
scale:1.05955
```

Figure 16. RunControl GUI (Geographical View).

The *RunControlGUI* sends periodic queries to *RunServers* to request information about the neighbors of each overlay socket in the overlay network. The interval between queries is specified in the user interface. By pressing the *Update* button, a user can force an immediate transmission of queries. When requesting information from the *Runservers*, *RunControlGUI* constructs a monitor protocol request message for each active portal in the portal list, and queries the state of all overlay sockets running on that portal. For some overlay protocols, the query may request additional statistics. A query message may look as follows:

```
<GetRequest     Sender="128.143.136.78:1500:1501" Dest="128.143.136.78:3563:3562"
                MsgID="4" TimeStamp="1104775368977" >
   <Stats index="0" xpath="/Portal/Appl/NumOfSockets"/>
   <Stats index="1" xpath="/Portal/Appl/SocketTable/SocketTableEntry[0]/Running"/>
   <Stats index="2" xpath="/Portal/Appl/SocketTable/SocketTableEntry[0]/Node/LogicalAddress"/>
   <Stats index="3" xpath="/Portal/Appl/SocketTable/SocketTableEntry[0]/Node/PhysicalAddress"/>
   <Stats index="4" xpath="/Portal/Appl/SocketTable/SocketTableEntry[0]/Node/NumOfNeighbors"/>
   <Stats index="5" xpath="/Portal/Appl/SocketTable/SocketTableEntry[0]/Config/Public/OverlayID"/>
   <Stats index="6" xpath="/Portal/Appl/SocketTable/SocketTableEntry[0]/Node/NeighborTable"/>
   <Stats index="7" xpath="/Portal/Appl/SocketTable/SocketTableEntry[1]/Running"/>
   <Stats index="8" xpath="/Portal/Appl/SocketTable/SocketTableEntry[1]/Node/LogicalAddress"/>
   <Stats index="9" xpath="/Portal/Appl/SocketTable/SocketTableEntry[1]/Node/PhysicalAddress"/>
   <Stats index="10" xpath="/Portal/Appl/SocketTable/SocketTableEntry[1]/Node/NumOfNeighbors"/>
   <Stats index="11" xpath="/Portal/Appl/SocketTable/SocketTableEntry[1]/Config/Public/OverlayID"/>
   <Stats index="12" xpath="/Portal/Appl/SocketTable/SocketTableEntry[1]/Node/NeighborTable"/>

</GetRequest >
```

The first statistic element queries information about the application. All other statistic elements request statistics from a total of two overlay sockets. When the remote applications return reply message, the *RunControlGUI* analyzes the messages and updates the display of the overlay network in the user interface.

3.5. REFERENCES

[citeMcLaughlin] Brett McLaughlin, Java & XML, 2nd Edition: Solutions to Real-World Problems, O'Reilly, 2nd edition, 2001.

[citeGriffith] Arthur Griffith, Java, XML, and the JAXP, Wiley, 2002.

- F. Strauss and T. Klie. *Towards XML oriented internet management*. Integrated Network Management, 2003 IFIP/IEEE Eighth International Symposium, pp. 505 – 518. March 2003.
- Mi-Jung Choi, James W. Hong, and Hong-Taek Ju. *XML-Based Network Management for IP Networks*. ETRI Journal, Volume 25, Number 6, December 2003.
- Read: http://www.juniper.net/solutions/literature/white_papers/200017.pdf
- *XML Network Management Interface*. By Weijing Chen and Keith Allen (SBC Labs). IETF Netconf Working Group, Internet Draft. Reference: 'draft-weijing-netconf-interface-00'. June 2003, expires December 2003.
- http://xml.coverpages.org/Enns-XMLCONF-Vienna.pdf

APPENDIX I: METHODS TO ACCESS STATISTICS

```
public void AccessStatistics (String XPathexpr)  {
        Document doc = XmlUtil.createDocument();
        XPath xpath =  XmlUtil.createXPath(XPathexpr);

        Element[] resultElements = null;
        try {
                resultElements = this.getStats(doc, xpath);
        } catch (HyperCastStatsException e) {
                System.err.println("Query fails:" + e.getMessage());
        }

        if (resultElements != null) {
                for (int i=0; i<resultElements.length; i++) {
                   Document resultDoc = XmlUtil.createDocument();
                   resultDoc.appendChild(resultDoc.importNode(resultElements[i], true));
                   try {
                        XmlUtil.writeXml(resultDoc, System.out);
                   }  catch (IOException e) {
                        System.err.println("Can't write XML file:" + e.getMessage());
                   }
                }
        }
}
```

```
public void ModifyStatistics (  String XPathPosition,  String XPathElement,
                                String ValueElement) {
        Document doc = XmlUtil.createDocument();
        XPath xpath =  XmlUtil.createXPath(XPathPosition);
        Element element =  XmlUtil.getXmlValue( doc,  XPathElement, ValueElement);

        // Modify the statistic
        Element[] resultElements = null;
        try     {
                resultElements = this.setStats(doc,xpath,element);
        } catch (HyperCastStatsException e) {
                System.out.println("Query fails:" + e.getMessage());
        }

        // Display the result of the new statistic
        if (resultElements != null) {
                for (int i=0; i<resultElements.length; i++) {
                   Document resultDoc = XmlUtil.createDocument();
                resultDoc.appendChild(resultDoc.importNode(resultElements[i], true));
                try {
                    XmlUtil.writeXml(resultDoc, System.out);
                }  catch (IOException e) {
                    System.err.println("Can't write XML file:" + e.getMessage());
                }
            }
        }
}
```

```
public void AccessReadSchema (String XPathexpr)  {
        Document doc = XmlUtil.createDocument();
        XPath xpath =  XmlUtil.createXPath(XPathexpr);

        Element[] resultElements = null;
        try       {
                resultElements = this.getReadSchema(doc, xpath);
        } catch (HyperCastStatsException e) {
                System.err.println("Query fails:" + e.getMessage());
        }
        if (resultElements != null) {
                for (int i=0; i<resultElements.length; i++) {
                   Document resultDoc = XmlUtil.createDocument();
                   resultDoc.appendChild(resultDoc.importNode(resultElements[i], true));
                   try {
                        XmlUtil.writeXml(resultDoc, System.out);
                   }  catch (IOException e) {
                        System.err.println("Can't write XML file:" + e.getMessage());
                   }
                }
        }
}
```

```
public void AccessWriteSchema (String XPathexpr)  {
        Document doc = XmlUtil.createDocument();
        XPath xpath =  XmlUtil.createXPath(XPathexpr);
        Element[] resultElements = null;
        try {
                resultElements = this.getWriteSchema(doc, xpath);
        } catch (HyperCastStatsException e) {
                System.err.println("Query fails:" + e.getMessage());
        }
        if (resultElements != null) {
            for (int i=0; i<resultElements.length; i++) {
                    Document resultDoc = XmlUtil.createDocument();
                    resultDoc.appendChild(resultDoc.importNode(resultElements[i], true));
                    try {
                            XmlUtil.writeXml(resultDoc, System.out);
                    }  catch (IOException e) {
                            System.err.println("Can't write XML file:" + e.getMessage());
                    }
               }
          }
}
```

APPENDIX I: OPTIONS FOR RUNCONTROL AND RUNSERVER

This appendix lists all options available for the applications *RunControl*, *RunControlGUI*, and *RunServer.*

java –cp *jarfiles* commandline.RunControlCLI –mc *file1* [–qr *retries*]
java –cp *jarfiles* gui.RunControlGUI –mc *file1* [–qr *retries*]

–cp *jarfiles*	A list of the JAR archives that contain the Java programs needed to run the RunControl application. *RunControl* requires the executables of HyperCast (*hypercast3.0.jar* or similar), the archive that contains the *RunControl* application (*runcontrol.jar* or similar), and the Apache Xalan-Java XSLT processor (*xalan.jar*), e.g., *"hypercast3.0.jar; runcontrol3.0.jar;xalan.jar"*
–mc *file1*	The configuration file for the monitor.
–qt *timeout*	The timeout values in milliseconds for transmitting request messages by the monitor protocol.
–qr *retries*	Maximum number of retransmissions for a query transmitted by RunControl.

java –cp *jarfiles* runserver.RunServer [–ns *NumSock*] [–start] –pc *file1* –sc *file2*

–cp *jarfiles*	A list of the JAR archives that contain the Java programs needed to run the RunServer application. *RunServer* requires the executables of HyperCast (*hypercast3.0.jar* or similar), the archive that contains the *RunServer* application (*runcontrol.jar* or similar), and the Apache Xalan-Java XSLT processor (*xalan.jar*).), e.g., *"hypercast3.0.jar; runcontrol3.0.jar;xalan.jar"*
–ns *NumSock*	Number of overlay sockets that will be created by *RunServer.*
–start	Determines if the overlay sockets immediately join the overlay network after they are created. By default, the overlay sockets created by RunServer do not start the overlay network.
–pc *file1*	Specifies the configuration file for the portal.
–sc *socket*	Specifies the configuration file for the overlay sockets of *RunServer.*
–help	Displays the options available for starting *RunServer*.

APPENDIX II: COMMANDS OF RUNCONTROL

Here we discuss the commands that can be typed at the command prompt of RunControl. The commands start a measurement experiment, collect data, and modify parameters of an experiment in execution.

date

> Prints the current date and time.

echo *text*

> Displays the string *text* that is typed as an argument.

clear_portal_list

> Removes all portal entries and resets the portal list. For the portal list, this method has the same effect as restarting the monitor.

compact_portal_list

> Removes all portals that are inactive and re-computes the indices of the portals. Since the monitor application uses the index in the portal list as the address of a portal, calling this method may change the address of a portal.

exit

> Terminates the *RunControl* application.

get_value [monitor | *portalindex*] *XPathExpression*

> Accesses a statistic at a RunServer application or the monitor. The command has two mandatory arguments. The first argument specifies a *RunServer* application by the index in the portal list of the monitor (Run *list_portals* to get the list.). If the argument is set to monitor, the command accesses a statistics of the monitor in the *RunControl* application. The second argument specifies the requested statistic in terms of a XPath expression.

get_readschema [monitor | *portalindex*] [*Xpathexpression*] [*stdout* | *fname*]
get_writeschema [monitor | *portalindex*] [*Xpathexpression*] [*stdout* | *fname*]

> Retrieves a schema of readable or writable statistics from a RunServer *application*. The first argument specifies a remote *RunServer* application through the index of the corresponding portal. If the argument is set to monitor, the command accesses the monitor in the *RunControl* application. The second argument is an XPath expression that identifies the requested schema. The third argument specifies if the output should be written to the monitor (*stdout*) or to a file with name *fname*.

help *[command]*

> Without an argument, the command displays a complete list of available commands. When *help* is called with a command name as argument, a description of the command is displayed.

kill_remote_servers

Terminates all remote *RunServers.*

list_portals [all]

Displays the contents of the portal list in the monitor of *RunControl*. By default, only active portals are displayed. If this command is run with the argument *all*, both active and inactive portals are shown.

list_sockets [verbose | row *i*]

Lists information about the number and the state of overlay sockets in the remote *RunServer* application. The command lists the number of sockets that have been created at the *RunServers* (*created sockets*), the number of sockets that have joined the overlay network (*started sockets*), and the number of overlay sockets that have not yet started (*available sockets*). Without an argument, the commands lists aggregate values for all remote *RunServers*. With the argument *verbose*, the command additionally displays the status at each remote *RunServer*. With the argument *row i*, the command displays the state only at the *i*th *RunServer*.

mtime

Displays the number of milliseconds since January 1, 1970.

pause *text*

Displays the typed *text* and waits until the user presses the return key.

quit

Terminates the *RunControl* application.

resync_sockets

Requests that all currently running overlay sockets are stopped. The command has, in most cases, the same effect as the command *stop_sockets all*, but there are differences. The sync*_sockets* command contacts all *RunServers* in the portal list and requests that all overlay sockets leave the overlay network. (Differently, *stop_sockets all* contacts only *RunServers* that have started overlay sockets according to the state information at the *RunControl* program. Therefore, when the state information at *RunControl* is not consistent with the actual state of the remote overlay sockets, the *sync_sockets* command can help to recover to a consistent state.)

run *script*

Reads the file *script* and executes the lines in the file as *RunControl* commands.

silent_mode [on | off]

Turns the silent mode of the monitor and the portals on and off. The silent mode on portals is set by sending a *SetRequest* message that sets the corresponding statistic. If the silent mode is turned on, a monitor or portal does not send advertisements. Also, a monitor in silent mode does not set the status of any portal in its portal list to inactive. By default, the silent mode is set to off.

set_value [monitor | *portalindex*] [*Xpathexpression*] [*newValue* | -f *fname*]

Modifies a statistic at a RunServer application or at the monitor. The command has three mandatory arguments. The first argument specifies a *RunServer* by the index in the portal list or lists the word monitor access a statistic in the monitor. If the argument is set to monitor, the command accesses the monitor in the *RunControl* application. The second argument specifies the requested statistic in terms of a XPath expression. The third argument is the new value of the statistic. For statistics that use a built-in type of XML schema, e.g., integers, strings, etc., the new value of the statistic can be typed in as a text string. Otherwise, the new value of the statistic is provided as a well-formed XML document stored in file *fname*.

sleep *[n]*

Does not show a command prompt for *n* seconds.

start_sockets [*n* | all] [uniform] [rate *R*]

Requests that new overlay sockets at the remote *RunServers* join the overlay network. The request is issued by setting the statistic *Running* of an overlay socket to *true*. The total number of started overlay sockets cannot exceed the total number of sockets that are created at remote *RunServers.* If a request would exceed the maximum permitted number of overlay sockets, the maximum number overlay sockets is started and a message is displayed indicating that the request exceeds the total number.

The command can take several arguments. All arguments are optional. A number indicates the number of overlay sockets. The argument *all* requests to start the maximum number of overlay sockets. By default, the command starts the maximum number of overlay sockets. An argument *uniform* indicates that overlay sockets are started at remote *RunServers* in a round-robin fashion. If the argument is not provided, *RunControl* contacts the first *RunServer* and allocates the maximum number of overlay sockets at this *RunServer*. Then *RunControl* contacts the second *RunServer* and allocates the maximum number of overlay sockets, and so forth, until the desired number of sockets has been started. The order in which *RunServers* is contacted is determined by the order of portals in the portal list of the monitor. The third argument (*rate R*) specifies the maximum rate at which overlay sockets are started. Here, *R* is a number that indicates the maximum number of overlay sockets that will be requested to start per second. If a rate is not specified then overlay sockets will be started as quickly as possible.

start_transcript *fname*

Starts writing monitor protocol messages to the specified file.

stop_sockets [*n* | all] [uniform] [rate *R*]

Requests that overlay sockets at the remote *RunServers* leave the overlay network. Sockets are stopped by setting the statistic Running in the overlay socket to false. If the request is larger than the total number of overlay sockets that are currently started, then all overlay sockets are stopped and a message is displayed. The arguments *n, all, uniform* and *rate R* have the same interpretation as in the *start_sockets* command.

stop_transcript

Stops writing monitor protocol messages to a file and closes the file.

wait_until_stable

This command is only available in overlay sockets that support a boolean statistics /*Socket/Node/Stable* (e.g., sockets that run the DT protocol) **Currently, the command assumes that the DT protocol (DTBuddyList, DTServer, or DTBroadcast) is running.** This statistic should be supported in overlay protocols where a socket can determine if the overlay protocol has reached a local definition of stability. The command queries all overlay sockets at RunServer applications and waits until all overlay sockets report that the value of the statistic is *true*. ~~At that time, the command displays how long the command has been running.~~ Should be: At that time, the command displays the time lag since the last time the command *start_socket* has been issued.

check_clock_synchronization

This command checks if the clocks of remote *RunServers* are synchronized with the local clock. The command sends a query message to the RunServers to query the CurrentTime statistic, storing the time the query is sent (*SendTime*) and the time the reply is received (*ReceiveTime*). The clock of a remote RunServer is said to be synchronized if SendTime ≤ CurrentTime at RunServer ≤ ReceiveTime. A warning message is printed if the clocks of the servers are no synchronized.

get_~~exact_~~stable_time (for DT)

Nodes are either in state “STABLE” OR “LEADER STABLE” This command prints the exact time when the overlay network became stable. The stable time is defined as the last time when there is any neighborhood change on any node. The result of this command is in milliseconds. A warning message is printed if the network is not in stable state.

(This is not clear. Stability and neighborhood change are different concepts?)

get_~~exact_~~start_time (for DT)

This command prints the starting time of the overlay network. The starting time of a network is defined as the time when the last started socket joined the network. This command queries every running sockets on RunServer for their starting time, and select the maximum one out of all results as the starting time of the overlay network.

(This is not clear. The time of the “last started socket” is always defined._What is missing is a test that all sockets have been started.)

poll_stability (for DT)

This command displays the number of stable sockets at each RunServer.

timeout_values [maximum number of retransmission attempts]

This command displays or sets the parameter for the retransmission of query messages. There is only one parameter, which is the maximum retransmission attempts of all messages. That means if a query message has been timeouted and retransmitted more than this "maximum retransmission attempts", no more retransmission would happen. With no parameters, it displays the current value of maximum retransmission attempts. With parameters, it sets the value.

(This is not clear. Is the max. retransmission of any message since the RunControl started, or is it the number of retransmissions of the last message?)

set_trigger [portalIndex] [condition Xpath] [operator] [compareValue] [pollingPeriod] [data Xpath] [Alias]

This command sets up a new trigger. It has seven arguments. The first argument specify the index of the portal in which the trigger is to be set up. The second argument specifies the XPath expression for the statistic element that is to be checked to for the trigger condition. The third argument specifies the operator used for deciding whether the condition has been satisfied, it can be ==, >=, <= etc. The fourth argument is the value used for comparison in the condition checking. The fifth argument specify the polling period for the portal querying the application statistics The sixth argument is the XPath expression for the data content that is to be contained in the notification message sent back when the condition is met. The last argument specifies the alias used to identify this trigger.

remove_trigger [portalIndex] [condition Xpath] [operator] [compareValue] [pollingPeriod] [data Xpath] [Alias]

This command removes trigger in the portal. The trigger to be removed can be identified by its alias or its index in the portal's trigger list. Whether the user uses alias or index, it can only remove trigger set up by this monitor. When using alias, if this monitor has set up multiple triggers with the same alias, the first trigger will be removed. The first argument indicates the index of the portal to which the removal request is sent to. The second argument indicates whether the trigger is to be identified by alias or index, and if all triggers are to be removed The third argument specifies the actual alias/index of the trigger, if the second argument is "-all" , then the third argument is ignored.

APPENDIX III: DISPLAYING MAPS IN RUNCONTROLGUI

The geographical view of *RunControlGUI* can display the geographical location of overlay sockets on a map, given that the longitude and latitude information of computer systems where the overlay socket is running are available. Here we discuss how the positions of an overlay socket is computed.

All geographical maps use some kind of projection that map all or part of a sphere (the earth) into grid values on a map. *RunControlGUI* displays maps that perform a Mercator projection, a commonly used projection method. The map shown in Figure A.1 depicts a Mercator projection of a world map. Note that the distance between parallels increases with the distance from the equator. For maps with a Mercator projection, a coordinate with longitude $\phi \in [-180,180]$ and latitude $\lambda \in [-90,90]$ is mapped to an x and y coordinate in a coordinate system by the following equations:

$$x = a_1(\lambda - \lambda_0) + b_1$$

$$y = a_2(\ln(\tan\phi + \sec\phi)) + b_2$$

For $a_1=a_2=1$ and $b_1=b_2=0$, a position is mapped to a coordinate system where the x-axis is at the equator and the y-axis is at longitude λ_0. The parameters a_1, a_2, b_1, and b_2 scale the x and y coordinates to the size of the image file. The parameters are determined from two reference locations.

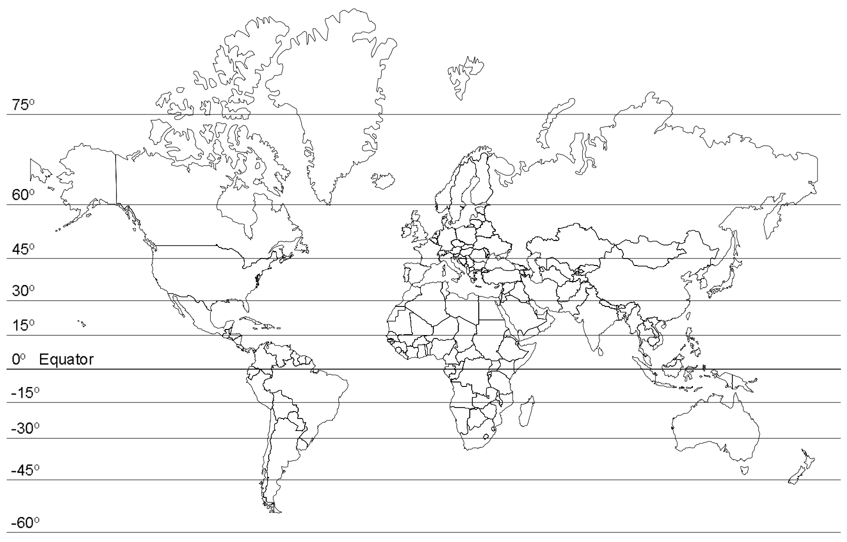


Figure A.1. World map as Mercator projection.

The file *map.xml* specifies the maps that can be displayed by the *RunControlGUI*. The map to be displayed is selected from the user interface. Each map is specified as follows:

```
<Map type="mercator"          name="World map"
     width="2245"             height="1443"
     upleft="-180, 90"        downright="180, -90"
     ref1_position="2060,1257" ref1_location="151.0,-34.0"    ref1="Sydney"
     ref2_position="349,672"   ref2_location="-123.06, 49.13" ref2="Vancouver">
   visio_worldmap.png
</Map>
```

The value of the element is the image file that contains the map, here *visio_worldmap.png.* The image file can be a PNG or JPEG file. The attribute *type* defines the type of projection. Currently, only maps with a Mercator projection are supported. The second attribute is the name of the map as it is displayed in the user interface. The width and height attributes give the pixel size of the map image. The attributes *upleft* and *downright* specify the longitude and latitude of the upper left corner and the lower right corner of the map. The remaining attributes define two reference points which determine the scaling parameters a_1, a_2, b_1, and b_2. A reference point consists of a place (e.g., *ref1*), its pixel position (*ref1_position*) and its geographical location (e.g., *ref1_location*). In the example, the reference points are Sydney and Vancouver. The geographical location is the longitude and latitude of the reference. The pixel position is the (x, y) position in the file *visio_worldmap.png,* where position (0,0) is the upper left corner of the image (Most image editing programs can display the pixel position of a location of an image file). To obtain useful parameter values, the two reference points should be locations that are geographically far apart.

APPENDIX IV: XMLUTIL: HELPER METHODS TO FOR XML PROCESSING

The XMLUtil class in the hypercast.util package provides a few helper methods that simplify the handling of XML documents and XPath expressions. The following list specifies some of the static methods of this class. We use the following abbreviations:

Document:	org.w3c.dom.Document
Element:	org.w3c.dom.Element
Node:	org.w3c.dom.Node
XPAth:	org.apache.xpath.XPath

static Document createDocument()
static Document createDocument(byte[] array)
static Document createDocument(File xmlFile)
static Document createDocument(InputStream xmlFileStream, String fileName)
static Document createDocument(String xmlFilename)
static Document createDocumentFromString(String xmlDocString)

All these methods create a new Document object. Without an argument an empty document is created. from given XML File instance. There is one method to create a document from a byte array, three methods to create a Document from a file and two methods to create a document from a string.

static Element[] createSchemaElement(Document doc, String name, String type, string restrictionBase, String patternValue)

Returns the element which represents a statistic in the schema document.

static XPath createXPath(String xpathStr)

Reformats a string containing an XPath as an object of type org.apache.xpath.XPath

static Node findChildNode(Node node, String name)

Finds a child node by name.

static void writeXml(Document saveDoc, OutputStream output)

Write an XML document to an output stream.