Markus Wulff, Gilbert Babin, Peter Kropf, and Qiaomei Zhong. Communication in the WOSTM. Research report no. DIUL-RR-9902. Université Laval. Ste-Foy, Québec, Canada. February 1999.

Communication in the WOS^{TM}

Markus Wulff*†, Gilbert Babin*, Peter Kropf*, Qiaomei Zhong*

February 18, 1999

Abstract

1 Introduction

The rapid development of networked and mobile computing, as demonstrated with the ever growing Internet (or Web) has lead to a global infrastructure, as well as to the introduction of new IT functionality. The presently available tools essentially allow users to download files, execute remote pre-defined scripts, fetch mobile code to be run locally or develop and run distributed applications within specialized metacomputing environments. The underlying model for those tools consists mostly in a client-server or master-slave setup with the network as the transport means. However, the role of the network is evolving towards the service delivering platform where services are offered through delivery contacts. A service may be understood in this context as a piece of software or hardware (computation or storage capacity, communication channels, specialized drivers, etc.). From a user's point of view, a service may be of any kind: the Web, conferencing, database search, Mail, Media, scientific applications or simply the translation of document from one format to another.

The use of widely distributed computing resources is motivated by various reasons such as load sharing, performance aggregation (including the exploitation of workstation idle time), reliability, availability and fault tolerance, function sharing and data sharing. To take advantage of the global infrastructure, which can be seen as a very powerful virtual global computer, mechanisms for efficient resource management are needed. However, the heterogeneous and dynamic nature of this infrastructure ensure that it is impossible to provide a complete catalogue of all the resources available. Therefore, new approaches are needed which take into account the inherently decentralized and dynamic properties of the Internet and distributed systems in general.

The WOS approach to global computing [1, 4] aims to provide service mechanisms which meet the requirements of the net-centric view of services and processes. This is achieved with the *eduction engine* or WOS-node, which integrates client, server, and broker/trader functionalities. The WOS envisages a series of versioned servers or nodes, each capable of providing a set of services, that can pass on to each other requests when appropriate. Each node uses warehouses to store and continuously update information about the node and available services and resources.

There are several approaches to integrate the computational resources available over the Internet into a global computing resource. The closest approach to the WOS is the Jini architecture proposed by SUN [11]. Jini allows one to build federations of nodes or distributed objects offering different services each relying on its own service protocol. Lookup services provide location and discovery functionality. The WOS approach is qualitatively different and more general in that federations, i.e. subsets of nodes of the WOSNet, defining a specific environment are dynamically and autonomously created. This is achieved with versioning and powerful lookup/discovery protocols and generalized service communication protocols.

Other efforts to exploit distributed resources for wide-area computing include Linda, PVM, MPI, CORBA, Netsolve [7], Globe [13], Legion [10], Globus [9] and WebOS[12]. In contrast to the WOS approach, most of the

*Laboratoire PARADIS, Université Laval, Québec, Canada, G1K 7P4
†Student trainee from Fachbereich Informatik, Universität Rostock, D-18051 Rostock, Germany

systems require login privileges on the participating machines, or require operating system or compiler modifications. They usually also require architecture specific binaries. The use of Java addresses the latter issue in a number of projects including Atlas [2], ParaWeb [5], Charlotte [3], Popcorn [6] and Javelin [8]. Those projects aim mostly to provide Java oriented programming models for Internet-based parallel computing. Our approach is orthogonal to these proposals in that Java oriented programming models could be integrated. But the WOS is different in that it does not require any global centralized catalogue of resources as it is for example necessary in Javelin, ParaWeb, Atlas or Globus.

While the WOS architecture relies on the decentralized eduction engines with their warehouses, it considers the communication protocols to be the centralized part. The communication protocols may thus be seen as the "glue" of the WOS architecture. Communication between nodes is realized through a simple discovery/location protocol (WOSRP) and a generic service protocol (WOSP). The WOSP protocol is in fact a protocol language with a corresponding parser and serves to easily configure service-specific protocol instantiations. For example, one WOSP instantiation could implement an interface to XML or CBL (Common Business Library). At the lower levels of the protocol stack, we assume the usage of the TCP/IP protocol family.

This paper presents the interfaces required to use both WOSRP and WOSP within a WOS node. It also illustrates the use of these interfaces in standard WOS communications. We start by describing briefly the structure of WOS nodes in the next section. Section 3 shows how WOSRP is used within a WOS node to discover the existence of other WOS nodes and/or particular or just any suitable WOSP versions. Section 4 describes the connectionless communication mode of WOSP. This mode is used for asynchronous communications between two WOS nodes. Section 5 shows how a WOSP connection may be established between two WOS nodes. In particular, Sections 3 through 5 provide a detailed description of each required interface specifications, which are written using the Java language. Utility classes need to support the communications are described in Section 6. We conclude in Section 7 with a brief discussion on the implementation of these interfaces and an outlook of future work.

2 Structure of a WOS node

The structure of a WOS node is shown in Fig. 1. In this paper, we focus on the WOSP/WOSRP interface, which isolate the network from the WOS node, and on the Eduction & Search Engine Layer, in particular on the WOSP Version Manager, which handles WOS version information.

Fig. 2 provides a detailed description of the layered node structure. Users interact with the WOS through a User Interface. That interface provides a unique gateway to the services available on the WOS. All service requests are made through that interface. The User Interface will display execution status and results, as they come. The Host Machine Manager handles all service requests received by this node. It is responsible for responding to resource search requests and for executing services, once approved. The User Manager is responsible for the coordination of any WOS-service require by/for any given user. It requests and allocates the resources required by a service, based on information stored in the local warehouses.

The next two layers shown in Fig. 2 implement WOSP. We mentioned earlier that WOSP is versioned. But how can we implement a single infrastructure to handle a multi-versioned protocol ? The answer is quite simple. First, we use WOSRP to lookup the different available WOSP versions, as described later (Sect. 3). Second, WOSP versions actually differ only on the semantics they convey. A single, open and complete syntax can therefore be defined to handle the transmission of different universe of discourse. The WOSP Parser handles the conversion to and from that syntax. The different versions of WOSP are implemented by specializing the WOSP Analyzer module. At run-time, a warehouse lookup is made to bind the node to the appropriate instance of the WOSP Analyzer, based on the WOSP version ID.

Finally, the WOSRP layer isolates the WOS node from the network. It provides two basic services : locating WOS nodes understanding specific WOSP versions and connecting to other WOS nodes using a specific WOSP version.

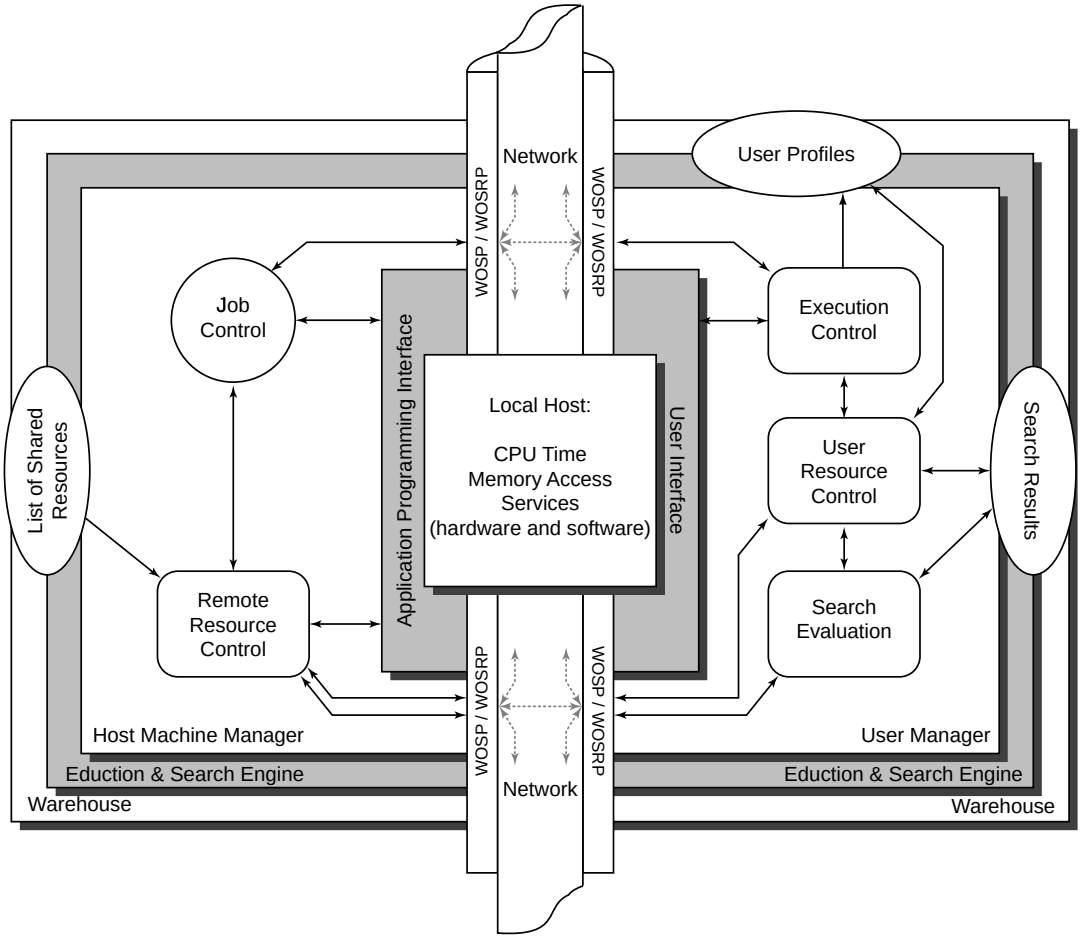

Figure 1: The structure of a WOS node.

3 Requests and Replies on WOSP Version Information

The WOSRP handles any request made by the WOSP Version Manager to locate WOS nodes or to locate WOS nodes using specific WOSP versions. In this section, we describe how this is accomplished.

3.1 WOSRP Headers for Requests and Replies

WOSRP sends requests to (remote) WOSRP servers using the header shown in Fig. 3. The first sequence in the WOSRP header is `00` for a request. The first option, "specific or any", is set to `0` when information about a specific WOSP version is requested, in which case the version ID must be provided. That option is set to `1` when information about any WOSP version can be supplied. The second option, "spoken or known", indicates the level of language knowledge of the WOS node being interrogated. The option is set to `0` when we require the server to speak that specific WOSP version. The option is set to `1` when we only ask for a server that knows another server (including itself) that speaks the WOSP version.

The "hopcount" is the maximum number of nodes to which a message will be forwarded. Each node decrements this counter by one before it sends the message to the next WOS node. If the counter reaches zero, the message must not be forwarded to another host. The sender IP address and port number are the location, where all replies are sent to. At the end of the header, the WOSP version ID is enclosed, if needed ("specific or any" is set to `0`).

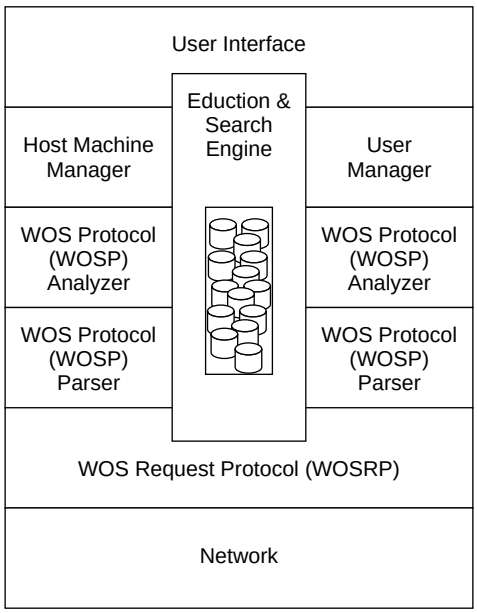

Figure 2: Functional layers of a WOS node.

The WOSRP header for replies is shown in Fig. 4. The first sequence for replies is `01`. The Server IP address and port number identify the WOS node that knows/speaks the version of WOSP indicated in the message (Version ID). The sender IP address and port number is the address of the sender of the message.

3.2 Interface Specifications for Requests and Replies

The communication for these information requests/replies works like shown in Fig. 5. In the first step (1,2)[1] the WOSP Version Manager checks the local WOSP Version Warehouse. Then, if no information is found, a WOSRP client is launched (3) with the following information :

```
Parameters : - recipient IP address and port number
             - hop count
             - version ID (required when specific_or_any is set to  1)
Options : - specific_or_any
          - spoken_or_known
```

After sending this message (4), the remote WOSRP Server launches a WOSP Version Manager (5) and sends it a message with the following information :

```
- sender IP address and port number
- specific_or_any
- spoken_or_known
- hop count
- version ID (if required by option specific_or_any)
```

The Version Manager checks its WOSP Version Warehouse (6,7) for information including :

```
- version ID
- spoken_or_known
- IP address (of the node that speaks the version above)
- port number (where the server is listening for incoming messages)
```

[1] The numbers indicated in the text refer to numbers in the figure.

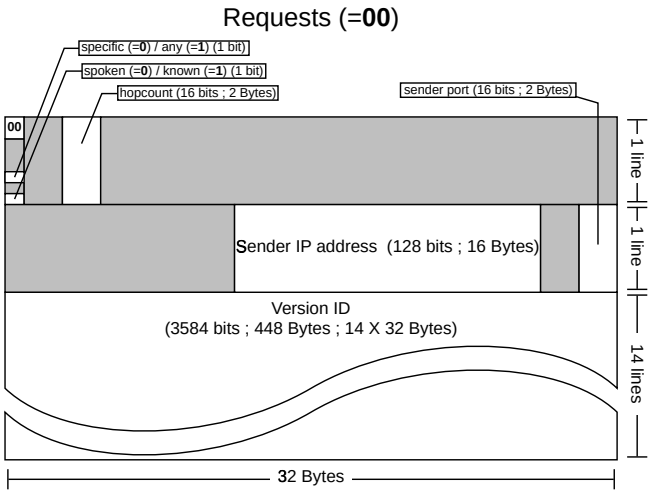

Figure 3: WOSRP header for requests.

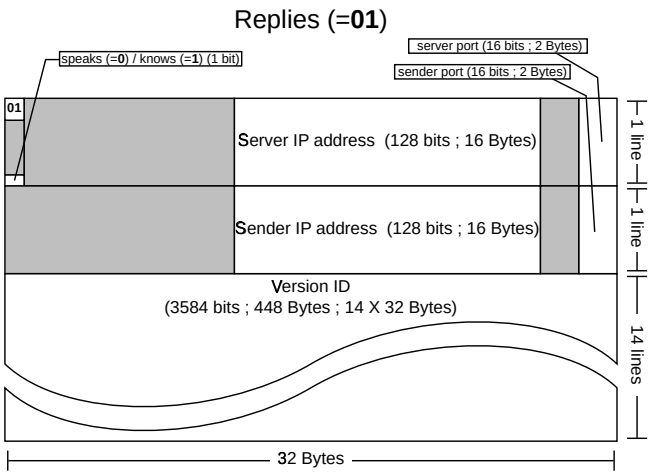

Figure 4: WOSRP header for replies.

If no appropriate version is found, the request is sent to another WOS node (not shown in Fig. 5), unless the hop count value is 0. If the search is successful, the remote WOSP Version Manager launches a new WOSRP client and sends the reply to the local node (8,9,10).

The local WOSP Version Manager which receives the reply, updates its local WOSP Version Warehouse with information received (11).

3.2.1 WOSRP Interfaces

Every interface function returns `0` in case of success and a value $\neq$ `0` if an error occurred, except as indicated.

```
Step (3)
   int WOSRP_Request(InetAddress RecipientIP, short RecipientPort,
                     boolean SpokenOrKnown, boolean SpecificOrAny,
                     short HopCount, String VersionID)
Step (8)
   int WOSRP_Reply(InetAddress RecipientIP, short RecipientPort,
                   InetAddress ServerIP, short ServerPort, boolean SpokenOrKnown,
                   String VersionID)
```

3.2.2 WOSP Version Manager Interfaces

```
Step (5)
   int WOSPVM_Request(InetAddress SenderIP, short SenderPort,
                      boolean SpokenOrKnown, boolean SpecificOrAny,
                      short HopCount, String VersionID)
Step (10)
   int WOSPVM_Reply(InetAddress ServerIP, short ServerPort, boolean SpokenOrKnown,
                    String VersionID)
```

3.3 The "Bootstrap" Problem : Installing a New WOS node

When a new WOS node is added to the WOSNet, all it knows is the initial list of WOSP versions it understands[2]. It knows nothing about other nodes in its neighborhood. The first order of business for the node is therefore to locate

[2]This initial list still needs to be defined

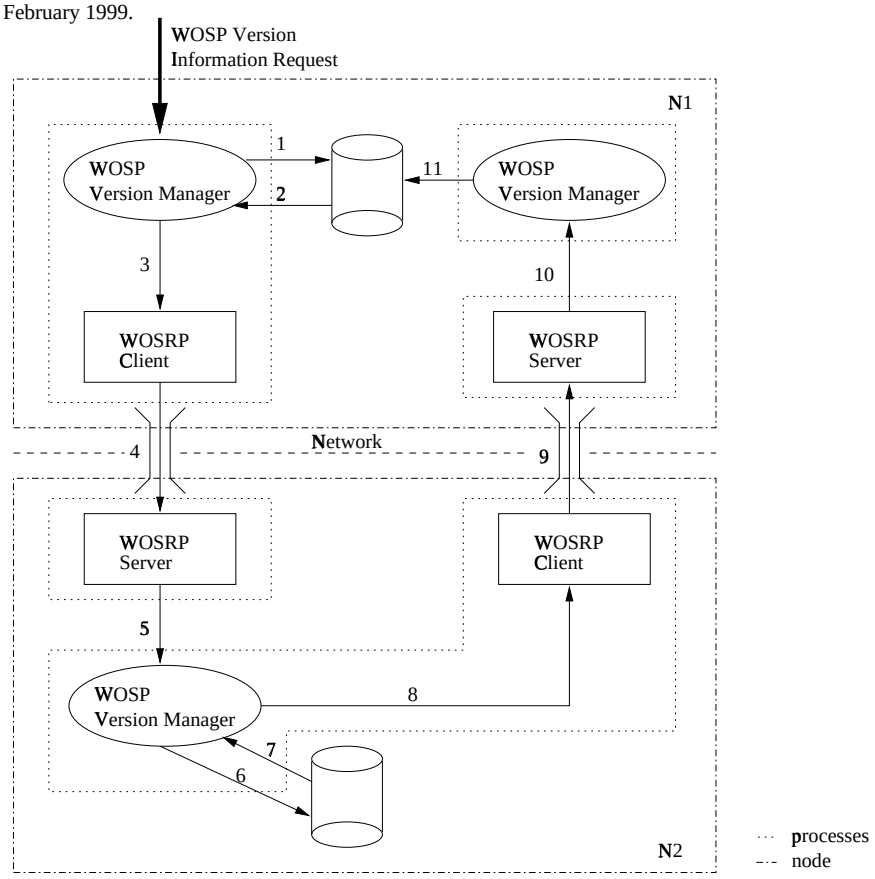

Figure 5: The WOSP version information request/reply communication.

other WOS nodes in its neighborhood. To do this, the WOSP Version Manager will broadcast to its local network a WOSRP message requesting information about any WOSP version. If no answer is received (i.e., no WOS nodes are located in the local network), the WOSP Version Manager in the worst case broadcasts to the next network level. This process continues until at least one WOS node is found or until every machine on the Internet is visited. This approach will not flood the Internet, since we proceed by recursive waves, instead of broadcasting everywhere at one time. If at least one WOS node is found, requests for specific WOSP versions may be sent as seen above.

4 The Connectionless Mode

Connectionless communication means that there will be no WOSP connection established to send a message. In other words, WOS nodes communicate asynchronously.

4.1 WOSRP Headers for Connectionless Communications

The WOSRP header for connectionless communications is shown in Fig. 6. In connectionless mode, the WOSRP Header is used to identify the appropriate WOSP version to process a message. The WOSP message itself is actually encapsulated within the WOSRP message. It follows the header and ends with a `DLE`[3]/`EOT`[4] sequence, marking the end of the WOSRP message. Any `DLE` or `EOT` character enclosed in the WOSRP message itself is escaped with a `DLE` character, as illustrated in Fig. 6 (left).

[3]Data Line Escape
[4]End Of Transmission

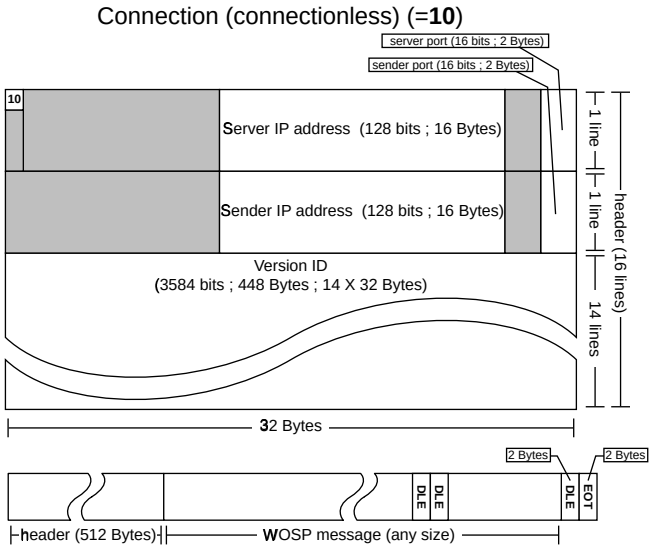

Figure 6: WOSRP header for connectionless mode.

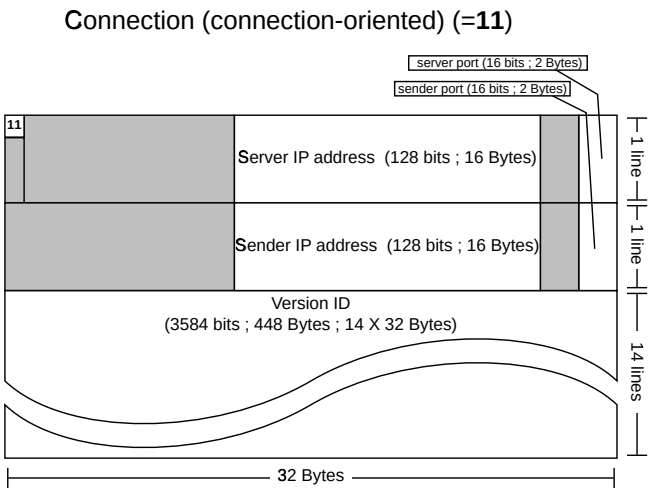

Figure 7: WOSRP header for connection oriented mode.

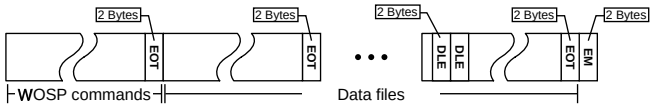

Figure 8: WOSP message format.

Note that the WOSP message is encoded in Unicode and has its own structure (see Fig. 8 and [1]). The message starts with a sequence of commands/replies, qualified by data and metadata. That sequence ends with an `EOT` character. Optionally, data files may be appended to the WOS message. Each data file ends with an `EOT` character. Any `EOT`, `DLE` or `EM`[5] character within a data file is escaped with a `DLE` character. The complete WOSP message ends with an `EM` character.

The sequence for connectionless mode messages is `10`. The Server IP address and port number identify the WOS node to which the message is addressed. The sender IP address and port number is the address of the sender of the message. The Version ID indicates the version of WOSP used in the enclosed message.

4.2 Interface Specifications for Connectionless Communications

The schematic flow for the connectionless mode is shown in Fig. 9. When WOSP Analyzer 1 at node N1 wants to send a message, it has to call the WOSP Parser with the message to be sent as a parameter (1). The message must be in triplet format (see Sect. 6.2). The message ID is generated by the WOSP Parser and sent back to the WOSP Analyzer on successful transmission of the message. Successful transmission means that the WOSP Parser has generated a message from the triplets and sent that message via WOSRP (2,3) to the remote node.

In the next step, WOSRP adds a header with IP address, port number, version ID to the message. The message is filtered as explained above. The WOS PID (see Sect. 6.4) is included in the message ID generated by the WOSP Parser.

WOSRP sends the message through the network (3) to the WOSRP Server at the remote node N2 with the following parameters :

```
- Server IP address and port number
- Sender IP address and port number
- version ID
```

The remote WOSRP searches its local WOSP Version Warehouse (4,5) for the version ID and server name. If compatible, WOSRP removes the header, launches the corresponding WOSP Analyzer 2 server with the parameters below and sends it the message (6).

```
- Sender IP address
- Sender port number
- version ID
```

At the same time, the local WOSRP returns (4') and the WOSP Parser sends the generated message ID to the calling WOSP Analyzer (5').

The WOSP Analyzer 2 (at node N2) interprets its protocol commands and converts the message into triplet format using the WOSP Parser (7,8). Then, the WOSP Analyzer 2 executes the commands and generates an answer. This answer, if any, is sent back to the calling node in the same manner as the original message (9-16,12",13").

When WOSP Analyzer 1' receives the answer, it identifies the original sender by its WOS PID, which is stored in a local WOS Process Warehouse, and delivers the answer to it (17).

4.2.1 WOSRP Interface

```
Steps (2), (10)
   int WOSRP_Connectionless(InetAddress ServerIP, short ServerPort,
                            String VersionID, InputStream Message)
```

[5] End of Message

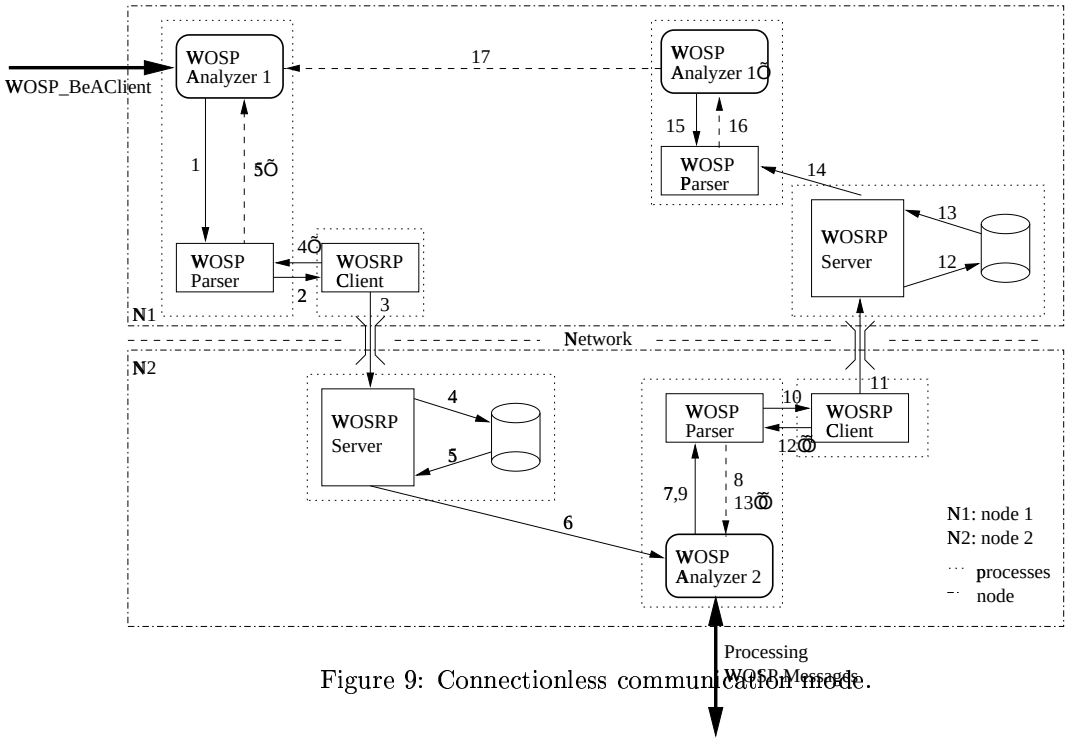

Figure 9: Connectionless communication mode.

4.2.2 WOSP Parser Interfaces

```
Steps (1), (9)
   String WOSP_SendConnectionless(InetAddress ServerIP, short ServerPort,
                                  String WOSPID,String VersionID, WOSP_ListOfTriplets Triplets)
   returns : MessageID if ok, null otherwise
   String WOSP_SendConnectionless(InetAddress ServerIP, String WOSPID,
                                  String VersionID, WOSP_ListOfTriplets Triplets)
   returns : MessageID if ok, null otherwise
Steps (7), (15)
   WOSP_ListOfTriplets WOSP_ReceiveConnectionless(InetAddress SenderIP, short SenderPort,
                                  BufferedReader Message)
```

4.2.3 WOSP Analyzer Interface

```
Starting the WOSP Analyzer Client
   int WOSP_BeAClient()
Step (6), (14)
   int WOSP_ServeConnectionless(InetAddress SenderIP, short SenderPort, BufferedReader in)
Steps (8), (16)
   void WOSP_ProcessBatchMessage(InetAddress SenderIP, short SenderPort,
                                 WOSP_ListOfTriplets Triplets)
```

5 The Connection Oriented Mode

Connection oriented communication occurs when two WOS nodes establish a bi-directional (half-duplex) communication link. In other words, WOS nodes communicate synchronously.

5.1 WOSRP Headers for Connection Oriented Communications

The WOSRP header for connection oriented communications is shown in Fig. 7. In connection oriented mode, the WOSRP Header is used to identify the appropriate WOSP version to process a message, and to establish the communication channel between two WOSP Analyzers for that version. The WOSP Analyzers exchange WOSP messages (as described in Sect. 4) in turn, starting with a message from the WOS node that initiated the connection. Once a disconnect message is received (by either node), the communication channel is broken.

The sequence for connection oriented mode messages is `11`. The Server IP address and port number identify the WOS node to which the message is addressed. The sender IP address and port number is the address of the sender of the message. The Version ID indicates the version of WOSP used in the interaction that will follow.

5.2 Interface Specifications for Connection Oriented Communications

To establish a connection between two WOS nodes N1 and N2 (see Fig. 10), a *connection setup message* must be sent to the remote node (1). After sending the connection request, the sending node performs an active wait on a *connection ok* signal. The local WOSP Parser then sends the request via WOSRP to the remote node (2,3). WOSRP will only confirm the establishment of the connection to the local WOSP Parser if it receives a message from the remote WOSRP (normally the request is sent back). The remote WOSRP checks its warehouse (4,5) (as described in Sect. 4) and launches the WOSP Analyzer 2 (6). If successful, the WOSRP at the calling node will be informed about success (6') and can now return (7',8'). If not, the remote WOSRP shuts down the connection[6].

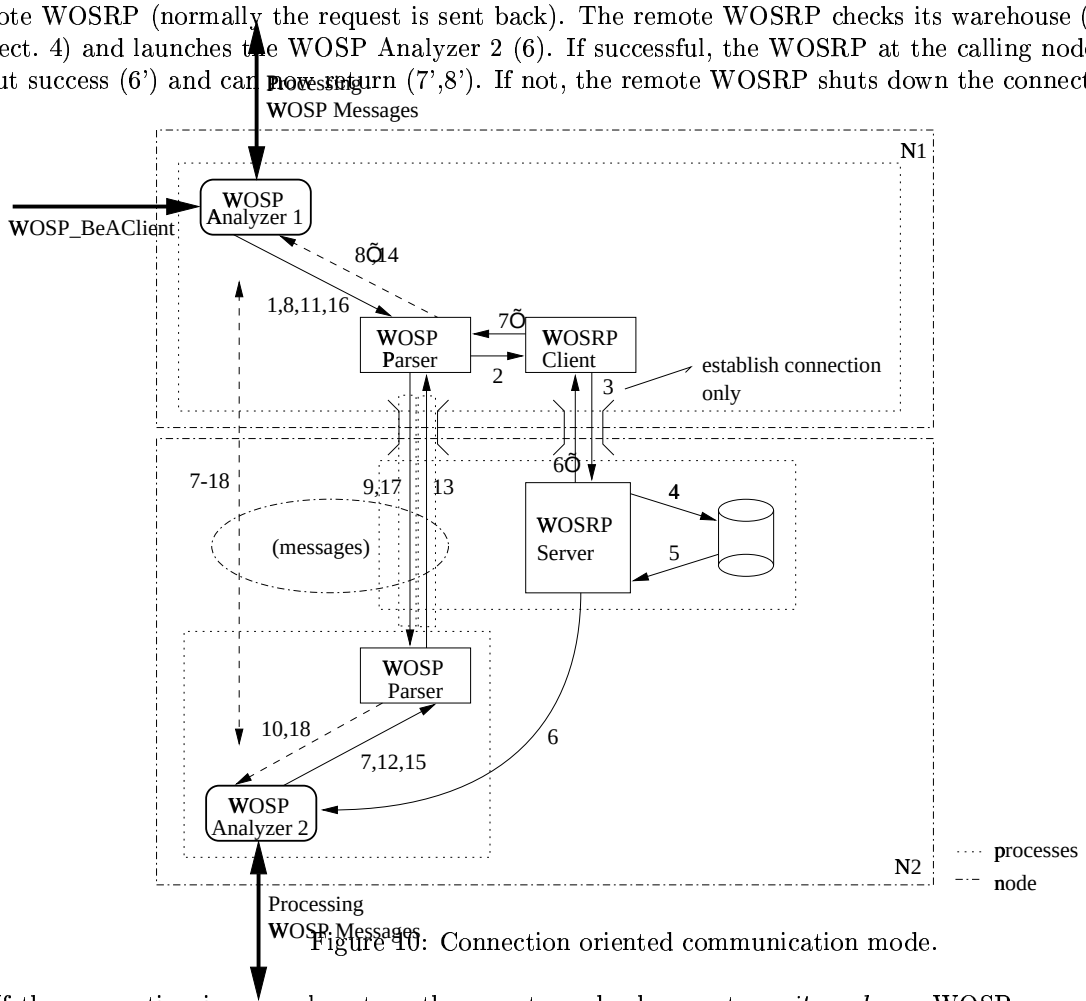

Figure 10: Connection oriented communication mode.

If the connection is properly setup, the remote node changes to *wait mode* on WOSP messages (7). Now, all messages between node N1 and node N2 are sent over that connection using WOSP only (8,9). Once a message is

[6]The connection is realized as a TCP connection

received by the remote WOSP Parser, node N2 exits its *wait mode* and returns the message received to the WOSP Analyzer 2 (10). The dialog uses a send-wait-receive sequence. Therefore, after sending its first message, node N1 waits for a message (11) sent by node N2 (12-14), and so on (15-18), until either N1 or N2 stops sending replies and close the communication channel.

5.2.1 WOSRP Interface

```
Step (2)
   WOS_Connection WOSRP_SetupConnection(InetAddress ServerIP, short ServerPort,
                                String VersionID)
   returns : a connection object if ok, null otherwise
```

5.2.2 WOSP Parser Interfaces

```
Step (1)
   WOS_Connection WOSP_SetupConnection(InetAddress ServerIP, short ServerPort,
                               String VersionID)
   returns : a connection object if ok, null otherwise
Steps (8), (12), (16)
   int WOSP_SendConnection(BufferedWriter out, WOSP_ListOfTriplets Triplets,
                           String WOSPID)
Steps (7), (11), (15)
   WOSP_ListOfTriplets WOSP_ReceiveConnection(BufferedReader in)
   returns : List of triplets if ok, null otherwise
```

5.2.3 WOSP Analyzer Interfaces

```
Starting the WOSP Analyzer Client
   int WOSP_BeAClient()
Establishing the WOSP connection
   WOS_Connection WOSP_EstablishConnection(InetAddress ServerIP, short ServerPort,
                                String VersionID, WOSP_ListOfTriplets Triplets,
                                String WOSPID)
   returns : a connection object if ok, null otherwise
Step (6)
   int WOSP_ServeConnection(WOS_Connection connection)
Step (10), (14), (18)
   WOSP_ListOfTriplets WOSP_ProcessInteractiveMessage(WOSP_ListOfTriplets Triplets)
```

6 Other Implementation Considerations

This section describes the different data classes not part of the Java standard packages.

6.1 Pipes in WOS Communication

Pipes are used in two cases. In the connectionless communication mode (described in Sect. 4), pipes are used to send the message from the WOSRP Server to the WOSP Parser on the server side (steps 6 and 14 in Fig. 9). In this case, the data transmitted must be filtered to remove extraneous `DLE` characters and the terminating `EOT` character.

The second case for using pipes occurs in the connection oriented communication mode (see Sect. 5). In this mode, the local and the remote WOSP Parser are connected by two pipes, managed by the WOSRP server. These pipes do not perform any filtering, since the messages are sent directly from and to the WOSP Parser level.

The `WOSRP_Pipe` class is defined to answer both needs. It runs as a thread launched by the WOSRP server. Two constructors are available :

```
WOSRP_Pipe(BufferedReader in, PrintWriter out, boolean filtering);
                          Standard constructor.
WOSRP_Pipe(BufferedReader in, PrintWriter out,
           boolean filtering, int trace_level, String label);
                          Constructor to set debugging trace.
```

6.2 The Triplet Format of Messages

The WOSP Analyzer passes a message to be sent by the WOSP Parser as a list of triplets. The class which contains these triplets and the triplet count is class `WOSP_ListOfTriplets`. The type of each triplet is defined by the class `WOSP_Triplet`. A triplet is composed of the following elements :

```
- String name   (the identifier of the triplet)
- Integer type  (data, metadata, setup_command, ...)
- String value  (the value associated to the name)
```

To generate a triplet, the class `WOSP_Triplet` makes the following constructor methods available to the user :

```
WOSP_Triplet();                  Constructor for a empty triplet.
WOSP_Triplet(name, type, value); Constructor for a complet triplet.
WOSP_Triplet(name, type);        Constructor without value.
WOSP_Triplet(type, value);       Constructor without name.
```

and also the methods

```
void setName(String)      Set the triplet name.
void setType(int)         Set the triplet type.
void setValue(String)     Set the triplet value.
String getName()          Get the triplet name.
Integer getType()         Get the triplet type.
String getValue()         Get the triplet value.
```

to set the values of a triplet. The list of triplets must respect the following syntax, based on the type of each triplet :

```
+[ command *[ metadata ]* *[ data *[ metadata ]* ]* ]+ *[ file ]*
```

where

```
command  are triplets of type setup, execution or query
metadata are triplets of type metadata
data     are triplets of type data
file     are triplets of type data file
```

The semantics behind this is as follows : metadata triplets always follow the command or data triplet they describe, data triplets always follow the command triplet they apply to. This, in fact, is the same syntax and semantics associated with WOSP messages.

When the WOSP Analyzer sends the triplets, each command triplet has as value the number of triplets applying to that command (command, data and metadata triplets).

When the WOSP Analyzer receives triplets, the value of command triplets is the message ID and the sequence ID of the command (see Sect. 6.4).

6.3 The Class `WOSP_ListOfTriplets`

As described above, all triplets (of type `WOSP_Triplet`) for a message are stored in a `WOSP_ListOfTriplets` instance. A list of triplets consists of one or more triplets and the count of triplets. The constructor method `WOSP_ListOfTriplets()` creates an empty list. The following methods are available to manipulate the triplet list.

```
getTriplet()                Get the current triplet.
getTriplet(int)             Get the triplet at position.
head()                      Head returns the first triplet from list.
insertTriplet(Triplet)      Insert node after the current in list
                            and set current to that node.
insertTriplet(Triplet, int) Insert node after position.
removeTriplet()             Remove the current node from list and
                            set current to the next node if present,
                            otherwise to the previous node (the end).
removeTriplet(int)          Remove node at position.
setCurrent(int)             Set current to position.
setCurrentToNext()          Set the current to the next node unless
                            the next node is the anchor.
setCurrentToPrev()          Set the current to the previous node
                            unless the next node is the anchor.
setTriplet(Triplet, int)    Substitutes the triplet at position with triplet.
tail()                      Tail returns the last triplet from list.
tripletCount()              Get the count of triplets.
```

6.4 WOS PID

The WOS PID (process identifier) is a mapping of the local process PID (e.g., a Unix PID) to a system independent format. It identifies each WOS process uniquely with a time-stamp. The format is the following :

```
YYYYMMDDhhmmssttt
```

where `ttt` represents milliseconds. The WOS PID will be stored in a local WOS Process Warehouse together with the local process ID.

6.5 Message ID

The message ID is used to identify a message and the sender of a message within the WOSNet (the network of all WOS nodes). Such an ID contains the IP address or the fully qualified host name (fqh) of the sender, the WOS PID (see above) of the sending WOS process, and a time-stamp (same format as WOS PID). So the format is a follows :

```
{IPaddr|fqh}:WOSPID:YYYYMMDDhhmmssttt-seqID
```

The sequence ID (seqID) is not really part of the message ID. It is added to identify a command within a message. It consists of two values – the `index` of this triplet within the message and the number of triplets `related` to this triplet. The format of a sequence ID is `-index-related`.

6.6 Class `WOS_Connection`

Once a TCP/IP connection is established between to nodes, we need to pass along the stream descriptors used to send and receive messages. The class `WOS_Connection` serves that simple purpose. The constructor method `WOS_Connection(InputStream in, OuputStream out)` creates a `BufferedReader` instance and a `BufferedWriter` instance for streams `in` and `out`, respectively. The following methods are available to manipulate the streams :

```
getIn()             Get a reference to the BufferedReader of the connection.
getOut()            Get a reference to the BufferedWriter of the connection.
```

7 Conclusion

A prototype version of the interfaces presented here was developed using jdk 1.1.6. Tests were initially performed in a Linux environment. The WOSP Parser and WOSP Analyzer were developed and tested first. Preliminary

Table 1: Performance analysis.

Mode	Message size (Bytes)	Min. transfer time (ms)	Max. transfer time (ms)	Avg. tranfer time (ms)	Std. dev.
Connectionless	1606	1009	1257	1180.0	43.3
	2606	977	1255	1031.7	67.8
	3606	1034	1278	1072.5	45.4
	4606	1078	1276	1134.2	43.3
	5606	1129	1210	1169.3	20.5
	6606	1187	1478	1250.9	74.6
	7606	1244	1315	1275.3	20.3
	8606	1284	1641	1371.7	86.5
	9606	1348	1422	1382.0	21.8
	10606	1405	1613	1457.6	51.2
Connection oriented	1600	540	576	554.9	10.0
	2600	147	161	152.6	3.8
	3600	191	207	198.7	4.1
	4600	254	266	256.1	2.2
	5600	291	473	342.0	47.3
	6600	398	671	516.5	67.6
	7600	504	773	606.3	57.3
	8600	604	849	723.8	69.4
	9600	678	903	795.1	66.5
	10600	707	1035	901.5	94.2

performance test can be found in Table 1, which shows the influence of the message size on the transfer time (in milliseconds). Fig. 11 shows how the average transfer time changes with the message size. We then worked on the WOSRP layer. Finals tests were performed in Linux, WinNT, and AIX environments. Only simple reconfiguration, which is required anyway from any WOS installation, we needed for the prototypes to work properly. This includes the path to launch Java virtual machine, the path to the WOS temporary directory, standard ports for local and remote WOS communications.

WOSRP requests and replies are realized using UDP datagrams, and are serviced one at a time. We realized that the buffer used by the UDP server was shared by all the threads launched to serve each datagram. This materialized in scrambled messages. By serving one datagram at a time, we avoid this message scrambling problem at the cost of a slower processing time. However, every datagram received is queued and eventually processed.

The connectionless and connection-oriented modes are realized using TCP connections. Although the connectionless mode is an asynchronous communication mode, the use of TCP libraries provided a more robust transmission. On one hand, connectionless messages have varying length which makes the use of UDP datagrams difficult. For instance, we would have to reassemble the datagrams to recreate the original message. On the other hand, even though the communication is asynchronous, we still want the messages to be transmitted without errors and reliable.

The connection oriented mode was tested by simulating a dialog between two nodes. One problem was difficult to solve: it turned out that the communication link is broken before the end of file is detected by the process receiving the information. We had to properly catch this exception to properly terminate the interaction between the nodes.

The next step is to develop a real WOSP Version Manager, along with warehouse management tools. We also will develop a generic WOSP Analyzer to handle the execution of commands on any remote system.

References

[1] Gilbert Babin, Peter Kropf, and Herwig Unger. A two-level communication protocol for a Web Operating System (WOS^{TM}). In *Euromicro Workshop on Network Computing*, pages 939–944, Västerås, Sweden, August 1998.

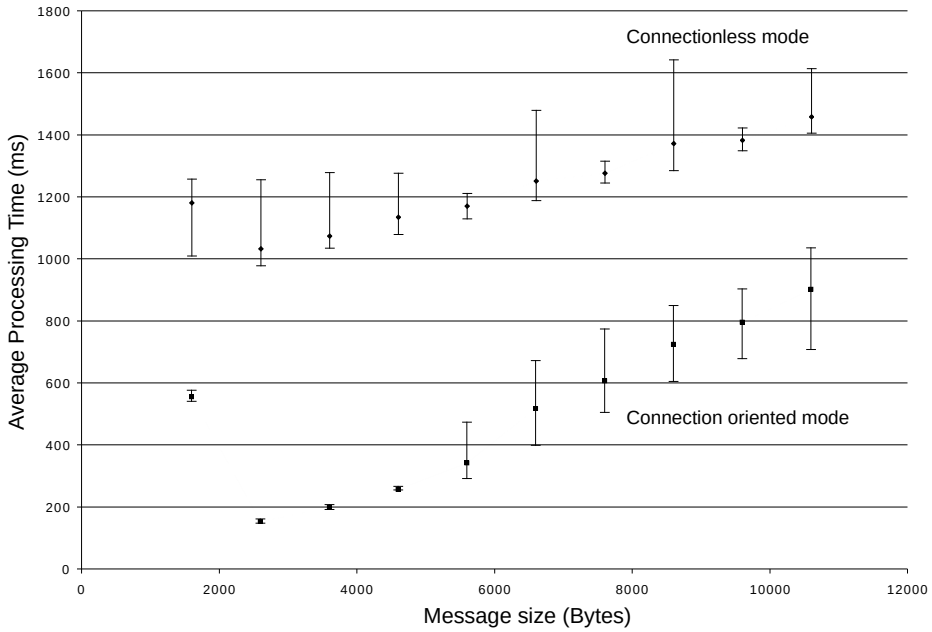

Figure 11: Average transfer time vs. message size.

[2] J.E. Baldeschwieler, R.D. Blumofe, and E.A. Brewer. Atlas: An infrastructure for global computing. In *Seventh ACM SIGOPS European Workshop on System Support for Worldwide Applications*, 1996.

[3] A. Baratloo, M. Karaul, Z. Kedem, and P. Wykoff. Charlotte: Metacomputing on the web. In *9th Conference on Parallel and Distributed Systems*, 1996.

[4] S. Ben Lamine, P.G. Kropf, and J. Plaice. Problems of Computing on the Web. In A. Tentner, editor, *High Performance Computing Symposium 97*, pages 296 – 301, Atlanta, GA, April 1997. The Society of Computer Simulation International.

[5] T. Brecht, H. Sandhu, M. Shan, and J. Talbot. Towards world-wide supercomputing. In *Seventh ACM SIGOPS European Workshop on System Support for Worldwide Applications*, 1996.

[6] N. Camiel, S. London, N. Nisan, and O. Regev. The POPCORN project: Distributed computing over the Internet in java. In *6th International World Wide Web Conference*, April 1997.

[7] H Casanova and J. Dongarra. NetSolve: A network server for solving computational science problems. *International Journal of Supercomputer Applications and High Performance Computing*, 3(11):212–223, 1997.

[8] Bernd O. Christiansen, Peter Cappello, Mihai F. Ionescu, Michael O. Neary, Klaus E. Schauser, and D. Wu. Javelin: Internet-based parallel computing using java. In *ACM Workshop on Java for Science and Engineering Computation*, June 1997.

[9] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *International Journal of Supercomputer Applications*, 1997.

[10] A.S. Grimshaw, W.A. Wulf, J.C. French, A.C. Weaver, and P.F. Reynolds. The Legion vision of a Worldwide Virtual Computer. *CACM*, 40(1), January 1997.

[11] Sun Microsystems. Jini. http://java.sun.com/products/jini/whitepapers/.

[12] Amin Vahdat, Thomas Anderson, Michael Dahlin, Eshwar Belani, David Culler, Paul Eastham, and Chad Yoshikawa. WebOS: Operating system services for wide area applications. In *Proceedings of the Seventh IEEE Symposium on High Performance Distributed Systems*, Chicago, IL., USA, July 1998.

[13] Maarten van Steen, Philip Homburg, and Andrew S. Tanenbaum. The architectural design of Globe: A wide-area distributed system. Technical Report IR-422, Vrije Universiteit, March 1997.