Web Services –Separation of Concerns: Computation Coordination Communication

Theophilos A. Limniotes and
George A. Papadopoulos
Department of Computer Science
University of Cyprus
75 Kallipoleos St., P.O. Box 20537
1678 Nicosia, Cyprus
{theo,george}@cs.ucy.ac.cy

Farhad Arbab
Department of Software Engineering, CWI
P.O.Box 94079, 1090 GB Amsterdam
The Netherlands
farhad@cwi.nl

ABSTRACT. The purpose of this paper is to investigate the use of a new concept in component communication, expressed by the channel based coordination language called *ρέω*, in the coordination of Web Services. The role of *ρέω* is to construct and manage connectors. Connectors are patterns of connected channel communicators. The communication and coordination of components lying over a distributed address space has been dealt so far with stream or datagram connections created and controlled by the participating calculation and coordination components. Web Services can take advantage of the *ρέω* channel system that separates the communication issue using components that have independent sink and source ports that can be attached to web services components, thus overcoming the problem of compatibility in distributed systems. The flow of information is entirely regulated by the channel interconnections.

1. INTRODUCTION

In the recent years the development of distributed computing systems with the use of component engineering provoked the development of component broker systems with stub procedures and proxy messages which mainly achieve communication but not coordination between components. Such successful well-known systems are CORBA, COM/DCOM (only for Microsoft systems), and Java RMI. Their disadvantages are simply restricted to their complexity, which secures communication of heterogeneous systems (CORBA, RMI). However communication does not guarantee the exploitation of the full functionality in some products, especially between heterogeneous systems. So such systems are in fact successful only with platforms of the same vendor. Moreover, due to the multiple levels of communication transactions tend to be very slow.

On the other hand the HTTP component provides a simple interface for communicating with an HTTP server. This communication protocol is simple, inexpensive and the HTML format is common to all platforms. That creates an advantage over other systems that simply claim to be open, but in practice are inflexible and contain numerous limitations. So the next advancement happened in the area of Web Services, with an effort to establish communication between components. The initial effort was made in 1998 with the establishment of an independent to presentation code for representing data forms, the Extensible Markup Language, XML by the independent WWW Consortium (W3C). The Extensible Markup Language (XML) is a subset of SGML. Its goal was to enable generic SGML to be served, received, and processed on the Web in the way that is now possible with HTML. XML has been designed for ease of implementation and for interoperability with both SGML and HTML.

SAC '04, March 14-17, 2004, Nicosia, Cyprus.

At the present the definition of the web services classes for reuse and composition (integration of heterogeneous web service patterns) is done with the Web Service Definition Language WSDL ([2]). The Simple Access Object Protocol (SOAP) for XML is a lightweight protocol for exchange of information in a decentralized, distributed environment. It is an XML based protocol that consists of three parts: an envelope that defines a framework for describing what is in a message and how to process it, a set of encoding rules for expressing instances of application-defined data types, and a convention for representing remote procedure calls and responses. SOAP can potentially be used in combination with a variety of other protocols. Finally the Universal Description, Discovery and Integration (UDDI) protocol is one of the major building blocks required for successful Web services. UDDI creates a standard interoperable platform that enables companies and applications to quickly, easily, and dynamically find and use Web services over the Internet. UDDI also allows operational registries to be maintained for different purposes in different contexts.

Coordination is important for Web Services: such systems combine services that are located on different web sites and this combination implies the need for coordination of their activities in order to regulate the flow of information and guarantee the reliability of the shared information. The Web Services systems can accommodate the channel based coordination system called *ρέω*. The dynamic connectors of channels that the functionality of this system offers are used in the management of the communication of distant components. The primitives of *ρέω* coordination language offer a great variety of synchronous and asynchronous channels with respect to access rights, mutability, reliability, grouping of connecting nodes, and execution model ([5]). The *ρέω* system is explained further in section 2.

The aim of this study is to create a language that takes care of the transportation of the functionality of the *ρέω* system ([1]) across the Internet. The objects are all distributed across the participating machines and a web service invocation is achieved

with SOAP-XML messages over TCP/IP connections that at the bottom level implement these channels. The architecture described is similar to that of an XML based multiple heterogeneous system ([3]). The form of migration is in XML and the stages for the translation are adequately described in section 3. The next section is a report of the new-formed language (Channel Coordination Definition Language). This is followed by the case study- a distributed space object system ([6]) and its implementation with the new language.

2. THE ρέω SYSTEM AND THE COORDINATION AMONG COMPONENTS

In many network architectures, each process in the network is either client or a server. This channel-based system is an application of synchronization and communication (through an adequate channel system) rather than combining proxies of services to build a web component. This is how the basic architecture of the system should look like.

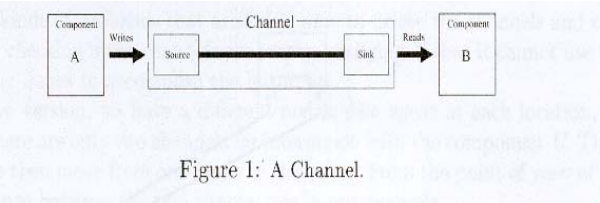

Figure 1: A Channel.

By definition every channel in *ρέω* represents a connector ([1]). More complex connectors are made out of simpler ones. The `create` operation creates channels with specified *channel end unique identification variables* (cev) that either can be source or sink end. Every cev is said to coincide on a node N that maybe connected, with the `connect` operation to one (among many) component instance. The operation `disconnect` applies to a channel end likewise, while `read` suspends the instance that performs this operation waiting for a value that can much to a data pattern *p* that is expected to be read to a variable. The operation `take` is a destructive variant of `read` and the channel loses the value that is read. Similarly the `write` operation suspends the operation of the calling instance until the value of variable `v` is written to the source channel end port. Moreover there is a `wait` operation that suspends the operation of the executing component if some predefined *nconds* conditions parameters become true. The node operation `join` merges two nodes (with their connectivity) to one, and `split` produces a new node N´and splits the set of channel ends that coincide on a node N between the two nodes N and N´according to the set of the specified edges. The node operation `hide` hides a specified node N from participation in the future node-modifying operations. Further node operations are `forget` that changes cev (or connected to it node N) so it no longer refers to the channel end it designates and `move` that relocates `cev` or `N` to a new location given as `loc.`

The ***coordination*** of web components should include predicates of *channel communication* like `create, read, write`, etc ([5]). The construction and access to a web component is performed with XML function transportation ([4]). The basic idea is that a web service should be able to receive and transmit its operation related information to a sink requestor component via a ρέω channel. Morover a third attached component would be able to influence the flow of data between the original two components with its own absorption of data ([1]). The components that execute the ρέω operations (`read write` and `take`) work with constructed components that implement a communication pattern independently, i.e. the latter do not have to know about the former. Of course this pattern can change with the execution of certain commands like `create`, `connect` and `join` (that belong to the channel management group of operations) at run time so it is not static. These can be classified as the management construction predicates while the former implement the message passing primitives. This functionality is very useful because it separates the communication components (created by web channel-services components) from the coordination concerns (incited by the web services). This in turn makes our system much more flexible with the ability for dynamic change of the communication pattern while running, as well as for dynamic change of the coordinating components. The overall architecture for such system is as follows:

There are three types of components

- The actual web services and client components
- The channel services components
- The "Channel-end Register" node components

The latter plays the role of a service register of a web service whose main task is to offer channel end references (according to a certain algorithm). So the management of this channelling system is actually undertaken by node processes which act like brokers.

The former is a candidate for using the channel-end registry and to do that it has to initially create a channel locally (with a primitive `_create` operation). The returned channel-ends IDs are stored with two respective node instances. The referred nodes are instantiated by the node-level `create` operation.

The actual channel-service components are descriptions of types of methods like `create` which implement this channel service. These descriptions have to be expressed in WSDL in order to be transmitted over the Internet via SOAP. The description should include definitions of the operations performed by the channel service, the required messages, the data types in used, and the chosen communication protocols. The purpose of the WSDL is to describe these services over the Internet. Channel services and Node brokers exchange WSDL files to restore connectivity and performed operations. SOAP comes in once a channel serviced is to be invoked. There would be no need for this intermediate conversion if the system should consider only interaction between (say) Java programs with RMI calls. The XML-based Web Services enable the definition of objects written in any language into language neutral types , and vice versa. The following figure shows an example Web service and a client invoking it in two different ways: Using SOAP and using HTTP GET. Each invocation consists of a request and a response message.

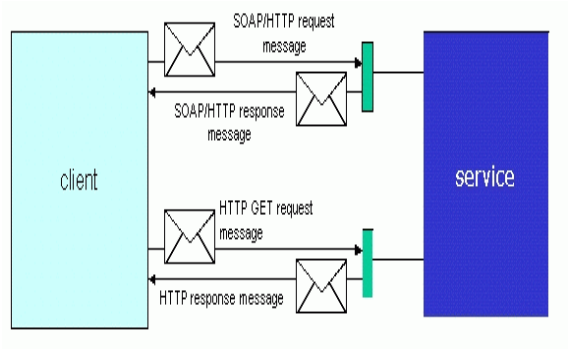

Figure 2: Client- Server invocation.

This next figure shows the same example with WSDL terminology pointing to the various things that WSDL describes. Refer to this in order to visualize the WSDL elements.

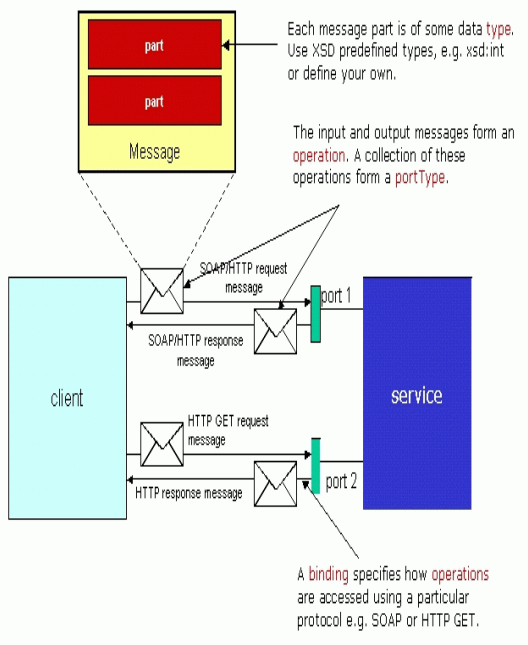

Figure 3: The WSDL message information and binding.

3. THE XML DECLARATIONS FOR THE ρέω OPERATIONS

Following here is the description of the first-level primitive methods for the construction of a Channel Composition Definition Language (CCDL). First come the DTD elements declarations as suggested in [4]. The purpose of encapsulating these primitives in XML code is to help in the implementation the system through the web services.

- **create**

```
<!ELEMENT CreateChannel(source,sink)>
<!ATTLIST channel
  name ID #REQUIRED
  type CDATA #REQUIRED>
<!ELEMENT source>
<!ATTLIST source
  id ID #IMPLIED>
<!ELEMENT sink>
<!ATTLIST sink
  id ID #IMPLIED>
```

The code in above is the data type expression of a channel itself. Its constituents are `name ID`, and `type`. Each cev has its own ID. The above structure is translated to the following XML representation. This is the XML code for creating a channel.

```
<!DOCTYPE CreateChannel SYSTEM "CreateChannel.dtd">
< CreateChannel>
  <name ID="_create" type="Synch" />
    <source id="asource">… </source>
    <sink id="asink">… </sink>
  </name>
</CreateChannel>
```

So the first message from the client can accommodate together with a cev ID a *time limit* and a *location*. If a member object indeed resides on the chosen site then a mechanism of retrieving the location of a specific service can be carried out with the use of channels. The messages are sent through a web browser, initially using an RMI call and presenting the cev ID.

In a case study a client that wishes to communicate with a ρέω based system has at first to get in touch with a sink channel. If every site that takes part in this scheme performs a CreateChannel operation then it will be in position to provide access to its participating components. The cev values should be available for potential customers.

The message sent to connect to a participating site (and therefore bind to it) is a `channelEnd` *"`connect`"* message and should include the ID of a channel's end. A source or a sink is connected to the component instance that performs the primitive operation `connect`, which along with `disconnect`, `forget` and `move` form the group of *channel end* operations `ChannelEnd` with the following DTD:

- **connect, disconnect, forget, move**

```
<!ELEMENT ChannelEnd (timelimit?, location?)>
<!ATTLIST ChannelEnd
  cev ID #REQUIRED>
<!ELEMENT timelimit (#PCDATA)>
<!ELEMENT location (#PCDATA)>
```

and this is the XML code for channel end operations.

```
<!DOCTYPE ChannelEnd SYSTEM "ChannelEnd.dtd">
< ChannelEnd>
  <cev ="_connect" />
    <timelimit>…</timelimit>
    <location>…</location>
  </cev>
</ChannelEnd>
</CreateChannel>
```

- **read, take, write**

The next group of primitives defines the operations of `_read`, `_take` and `_write`, which have attributes the appropriate cev and a variable v for read or for write. This is the DTD code for the read/write group of operations, followed by the XML code for the read/write group of operations.

```
<!ELEMENT UseChannel (timelimit, variable)>
<!ATTLIST UseChannel
  name ID #REQUIRED
  portType CDATA #REQUIRED
  filterType CDATA #REQUIRED>
<!ELEMENT timelimit #REQUIRED>
<!ELEMENT variable #REQUIRED>
```

```
<UseChannel>
  <name="_read" type="asource" filter="pat"/>
    <timelimit>…</timelimit>
    <variable>v</variable>
  </name>
</UseChannel>
```

The three types of primitive DTD declarations can be transferred to the corresponding *node* predicates. The referred *nodes* are not created (like channels) but are initially constructed by sets of channel ends that are connected to components by the primitive operation `connect` and are enhanced by the `join` node operation. The latter merges two or more nodes together.

- **move**

The move operation can relocate a channel end to the referred location `loc`. This is the DTD code for the move operation, followed by the XML code for the move operation.

```
<!ELEMENT MovChannel (cevariable, loc)>
<!ATTLIST MovChannel
  name ID #REQUIRED
  portType CDATA #REQUIRED
  filterType CDATA #REQUIRED>
<!ELEMENT loc #REQUIRED>
<!ELEMENT cevariable #REQUIRED>

<MovChannel>
  <name="_move" type="asource" filter="pat"/>
    <variable>cev</variable>
    <location>loc<location>
  </name>
</MovChannel>
```

The result of this operation is to pass the message to perform the `move` operation for the particular cev.

- **`join, split`**

Suppose that a site has several cevs connected to it. Then another site (which is member) can provide to it the class `join`(sender, receiver) which performed by the receiver will result in redirecting all messages to more combinations of sink and source ends. This the DTD code for the `join` operation, followed by the XML code for the `join` operation.

```
<!ELEMENT join (N1, N2)> //Nodes N1 and N2
<!ATTLIST join
  name ID #REQUIRED
  portType CDATA #REQUIRED>
<!ELEMENT cevs #REQUIRED>
<!ELEMENT N1 #REQUIRED>
<!ELEMENT N2 #REQUIRED>

<join>
  <name="join" type="asource"/>
    <variables>cevs</variables>
    <node>N1</node>
    <node>N2</node>
  </name>
</Join>
```

At the nodes operations level, cevs are represented by nodes N that are able to contain one or more of them, forming the so-called connectors. The construction of these connectors is achieved by the node-level-operation of `join`(). The `split`() method works similarly but in an opposite way splitting the channel ends between a new formed node and the old one. Notice that channels do not support message passing with the method call semantics. This is the DTD code for the `join` operation, followed by the XML code for the `join` operation.

```
<!ELEMENT split (N, quoin)> //Nodes N - set of edges
quoin
<!ATTLIST split
  name ID #REQUIRED
  portType CDATA #REQUIRED>
<!ELEMENT cevs #REQUIRED>
<!ELEMENT N #REQUIRED>
<!ELEMENT edges #REQUIRED>

<split>
  <name="join" type="asource"/>
    <variables>cevs</variables>
    <node>N</node>
    <edges>quoin</edges>
  </name>
</split>
```

4. THE DESCRIPTION OF A NEW CHANNEL COMPOSITION DEFINITION LANGUAGE

The above construction of XML function structures can be used in the definition of a new service composition specification language. The characteristics of this language are:

- The use of the *ρέω* system by all the participating parts (installed on every machine locally).
- The sharing of information regarding the use of the *ρέω* primitives. The exchange of messages is done at the web service level, and concerns the channel end structures and references, because the ends of a channel must internally know each other to keep the identity of the channel and control communication.
- If the type of a channel is asynchronous it must also have a reference to the buffer that implements every channel.
- An interface reference from a component to a channel end restricts the actions of the component to only the predefined operations on the channel.
- The predefined operations are `create`, `connect`, `disconnect`, `forget`, `read`, `write`, `move`, `join`, `split`. These can be classified in the following categories:

1. The `create` primitive creates a new channel with a specified channel type.
2. The `connect`, `disconnect` primitives connect and disconnect respectively a specified node to the calling component instance.
3. The `forget` changes the specified channel end id and `move` changes the channel end to the new node.
4. The `read`, `write` and `take` operations perform a read, a write, or a destructive read from/to a specified variable to/from the connected specified node.
5. The `join` operation is a node merging operation producing a new node from two other specified nodes. The `split` creates a new node and splits the attached channel ends between the new and the old one according to a specified list.

The type of the chosen channels plays an important role in the outcome of the mode of execution of the node operations. The Synch channels offer synchronous unbuffered transmission, and the FIFO channels offer asynchronous unbounded buffered transmission. Buffers can generally be used as sequencers and Synch/SynchDrain channels as flow regulators. The mapping of the messages is achieved by the references to the buffers and/or the channel ends.

5. AN IMPLEMENTATION OF WEB SERVICES (A CASE STUDY)

Using the definitions of the previous section the web resources can be encapsulated in distributed objects, and the web can be transformed from a collection of clients and servers (serving web pages) into an object space of distributed objects ([6]). The requests between such objects should be carried out via a web browser as shown in fig. 4.

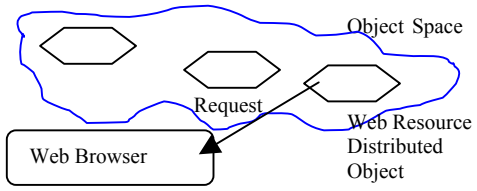

Figure 4: Distributed objects in an object space.

The web browser communicates with another through a Local Representative that each DSO has as shown in fig 5. The details of this architecture will be explained in the next section.

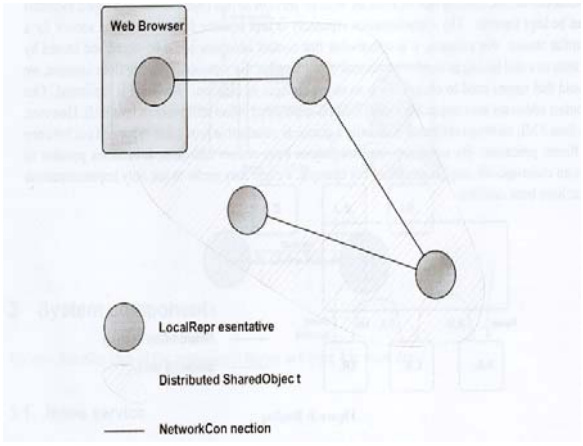

Figure 5: A distributed shared object.

5.1. The Use of Channels in a Web Component Case Study

Channels offer the necessary subtle connectivity required in the implementation of such diversity and complexity as the Distributed Object Space over the Internet. Moreover the primitives of ρέω channels can be used as a language for the coordination of concurrent services or as a connector constructing language for the binding of the component spaces (connectors) in a component based system.

The use of WSDL representation of component classes and its corresponding XML expression should be regarded as the means for the construction of a message that will incite a channel operation at a remote (hosting) side. In this case it could relate to a class or method associated with the use of ρέω channels that exist on this remote site. Each information source should have a channel where requests can be issued. Clients to a channel end have to be aware of their cev ID (reference). The XML message contains the code for some ρέω operations with the corresponding cev references. These messages cause one or more remote execution in the sites hosting the ρέω services. In the present web component case study we deal with the formation of Distributed Shared Objects (DSO), which reside in different web sites.

So in this the web resources can be encapsulated in distributed objects, and the web is transformed from a collection of clients and servers (serving web pages) into an object space of distributed objects ([6]). In our implementation this is achieved via the requests between such objects that are carried out via a web browser shown in fig. 2.

The web browser communicates with another through a Local Representative that each DSO has as shown in fig 3. The details of this architecture will be explained in the next section.

5.2 The Binding of a Candidate Object

To communicate with a Globe DSO, clients must bind to the object. This causes a new *Local Representative* (LR) of the DSO to be created in the client´s address space, effectively connecting that address space to the rest of the DSO. To do this the binding process has two main phases:

- To find where a host side of the DSO is and
- To initialise a Local Representative.

The shared objects of this system are all hosted under a common class (Globe) name ([6]). A proxy called *the translator* accepts requests from the *DSO browser* that the client uses. A filter can sort out Globe related URLs. Such names are forwarded to a special Globe gateway, which performs the binding to the object and the callings of the appropriate methods. The aim of every client request and binding is to obtain its own Local Representative in its own address space, effectively connecting that address space to the rest of the DSO

The name given by the client is passed to the name service (NS). At this point there is an inter process communication between the translator site and the Globe Gateway. For the mobile channel system to be operable we consider that a channel has already been created on every site with a Local Representative.

- The first stage of the clients *binding* process begins by sending the name ID of a DSO to the name service (NS) which maps names to location transparent object handles (OH)
- The name service returns an object handle.
- The object handle is passed to the location service.
- The location service retrieves a contact address.

A contact address represents a contact point of the DSO. Contact addresses identify a LR that should be loaded into the client's address space. The contact address contains an implementation handler. This is sent to the *implementation repository*, which in turn returns a *class archive*.

The class archive is in turn used by the *class loader* for extracting the implementation code in order to create the actual LR, so that the client's address is **connected** to the rest of the DSO.

The overall view of information exchange is shown in the figure below:

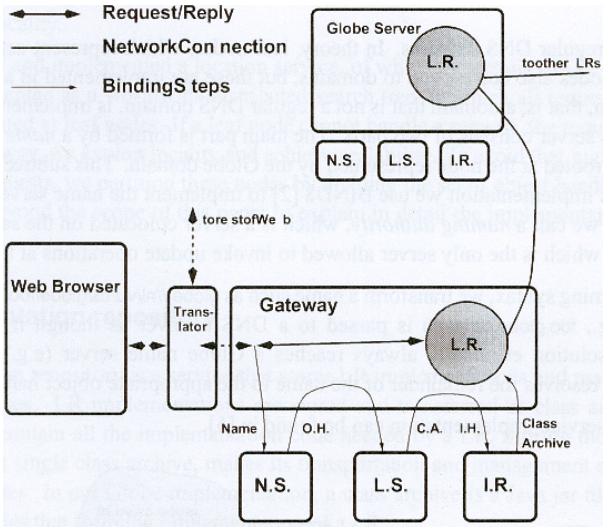

Figure 6: The overall view of the binding process.

6. THE ACHIEVEMENT OF THE DSO COMMUNICATION THROUGH ρέω CHANNELS

The communication between the DSO address space and the potential member is done with the use of channels, one from each object of a particular address space. Every member address (site) possesses the ends of two channels One of them is ready to receive applications from potential new members. The

XML-connect predicate has to be used by every applicant to supply the cev ID to a DSO site. This connects the specified channel end, cev, to the component instance that contains the active entity, which performs this operation. The XML message sends the cev ID to the site that 'owns' this channel. The execution of connect at one site results to the attachment of the calling procedure to the called channel end.

The **ρέω** commands are executed locally, i.e. all channels are created at member sites or candidate member sites. For example to have the *binding* operation performed with the use of channels both the *web browser* and the *translator* have to own at least one channel each, one for requesting and one for replying respectively. The sites are remote and the *web browser* component sends its cev ID to the *translator* component along with a connect XML message. The channel-end level primitives are indicated by the underscore.

```
ChannelEnd(timelimit,translator_loc)
```

The *translator* performs a connect operation with the received cev ID and connects to the *WB'*s channel end.

```
_connect(timelimit,cev_wb)  //primitive channel operation
```

From then on the client's *web browser* will receive information from its created channel. Likewise the *web browser* can perform the operation connect with the cev ID of the *translator* and so get a request through the client's channel, e.g the *translator* performs:

```
ChannelEnd(timelimit,webBrowser_loc)
```

And the *web browser* responds with:

```
_connect(timelimit,cev_trans)
```

The *translator* belongs to the object space where the channels are already created and connected. So the following read and write operations have to be performed. The *gateway* has a channel for receiving (read) information for binding while the components *name service, location service* and *implementation repository* have a special channel that reads requests from the *gateway*. The channel of the *gateway* moves from the *name service* to the *location service* and from there to the *implementation repository* in order to perform the binding.

```
_connect(timelimit,cev_nameService)// to NS site
UseChannel_write(timelimit,var)
_write(timelimit, cev_nameService)
_connect(timelimit,cev_locService)// to LS site
UseChannel_write(timelimit,var)
_write(timelimit,cev_locService)
_connect(timelimit,cev_implemService)// to IR site
UseChannel_write(timelimit,var)
_write(timelimit, cev_implemService)
```

At every binding stage NS, LS, and IR components respond with performing a write operation on the channel that the gateway owns.

```
_connect(timelimit,cev_gateway)// to Gateway
UseChannel_write(timelimit,var)
_write(timelimit, cev_gateway)
```

Once the binding is completed the join operation is executed by the web browser site's local representative and the local representative of the gateway in order to Join the node that contains the rest of the object space channels.

```
ChannelEnd(timelimit,translator_loc)// to translator site
_connect(timelimit,cev_gateLR)
join(node_webBrowser, node_translator)
```

7. CONCLUSION

In this paper it is investigated the use of a channel operation system in distributed systems. The new-formed system is process oriented in the sense that the processes participating in the construction of a distributed object perform the channel operations.

- The flow of data does not decide the execution of a read or a write. On the other hand the relocation of a channel does not influence the reliability of the data carried. The outcome from this is a configurable net of components that connect/disconnect to and from nodes at run time.
- At the node level nodes with their attached channel ends can join and split between them. This allows the dynamic reconfiguration of connections at real time and the redirection of flow of data between the member objects.

The Channel Composition Definition Language mainly provides a solution for the composition of a communication pattern among components through the XML encapsulation of the ρέω primitives. The major advantage is that a web server is enough for the passage of the execution messages. The construction and management of the communication concerns of the created channels can take advantage of the fast evolving web communication protocols.

8. REFERENCES

1. F. Arbab, F. Mavaddat, "Coordination through Channel Composition", *5th International Conference on Coordination Models, Languages and Applications, (Coordination 2002)*, York, UK, April 8-11, 2002, LNCS 2315, Springer Verlag, pp. 22-39.
2. J. Yang and M. Papazoglou. "Web Component: A Substrate for Web Service Reuse and Composition"*, 14th International Conference on Advanced Information Systems Engineering, (CAiSE 2002),* Toronto, Canada, May 27-31, 2002, LNCS 2348, Springer Verlag, pp. 21-36
3. G. Gardarin, F. Sha, Tram Dang Ngoc. "XML-based components for Federating Multiple Heterogeneous Data Sources", *18th International Conference on Conceptual Modeling*, *(ER '99),* Paris, France, November, 15-18, 1999, pp 506-519.
4. S. Szykman, J. Senfaut, R. Sriram. "The Use of XML for Describing Functions and Taxonomies in Computer-based Design", *Proceedings of the 1999 ASME Design Engineering Technical Conferences (19th Computers and Information in Engineering Conference)*, Las Vegas, NV, 12-15 September, 1999, Paper No. DETC99/CIE-9025.
5. F. Arbab, F. S. de Boer, J. G. Scholten, M. M. Bonsangue, "MoCha: A Middleware Based on Mobile Channels", *26th International Computer Software and Applications Conference (COMPSAC 2002)*, Oxford, England, 26-29 August 2002, Proceedings. IEEE Computer Society 2002, pp 667-673.
6. I. Kuz, P. Verkaik, Ivor. van der Wijk, Maarten van Steen, A. S. Tanenbaum. "Beyond HTTP: An Implementation of the Web Globe", Technical Report, Delft University of Technology, November 1999.