WEBGOP: Collaborative Web Services Based on Graph-Oriented Programming

Alvin T. S. Chan, *Member, IEEE*, Jiannong Cao, *Member, IEEE*, and C. K. Chan

***Abstract*—WEBGOP is a programming architecture for collaborative Web services using graph-oriented programming. The motivation for the project comes from the realization that the integration of collaborative Web services lacks support. The aim of WEBGOP is to extend the Web from a client–server system to a structured multipoint system. A graph abstraction of the network provides the structure for the integration of Web services and facilitates their configuration and programming. Using WEBGOP, a logical graph representing a virtual-overlay network over the Internet is created to link up collaborative Web services. Web services are individually or jointly invoked through either unicast or multicast messages within the overlay network. All messages are based on the simple-object access protocol (SOAP). This forms an extension of the hypertext transfer protocol (HTTP) to support the distributed invocation of Web services. The Web services on different servers work collaboratively for a multipoint network application. This project provides a structured integration of Web services by extending the support of intermediary processing in a multipoint service. It also provides a rich network-programming interface for a new class of integrated Web applications while retaining the use of the Internet protocol and HTTP.**

***Index Terms*—Collaborative, graph-oriented programming, simple-object access protocol (SOAP), Web services.**

I. Introduction

In today's infrastructure, the Web is based on the simple client–server model. It supports only simple and direct end-to-end request–reply cycles between a client and a server. There is no provision for a client or a network service to mediate the routing of a request, nor for any mechanism to coordinate different Web entities on behalf of a distributed application. Without additional architectural support for collaborative services, the Web remains a collection of communicating computer pairs without much collaborative effort at involving more than end-to-end computing. Web services are loosely-coupled computing tasks communicating over the Internet using hypertext transfer protocol (HTTP) and extensible markup language (XML). The concept of Web services opens up a new front in service-oriented architecture for the design of Web-based Internet applications. The popularity and wide acceptance of HTTP and XML have greatly driven the concept of Web service as an effective service model for the creation of ubiquitous business-to-business e-business over the Internet. The challenge of how different Web services can be integrated together to achieve seamless operation remains to be addressed.

Manuscript received March 11, 2002; revised January 18, 2004, and March 10, 2005. This work was supported by the Hong Kong Polytechnic University's Central Research Grant G-T877 and Internal Competitive Research Grant A-PF82. This paper was recommended by Associate Editor N. Cassaigne.

The authors are with the Department of Computing, Hong Kong Polytechnic University, Kowloon, Hong Kong (e-mail: cstschan@comp.polyu.edu.hk).

Digital Object Identifier 10.1109/TSMCA.2005.851342

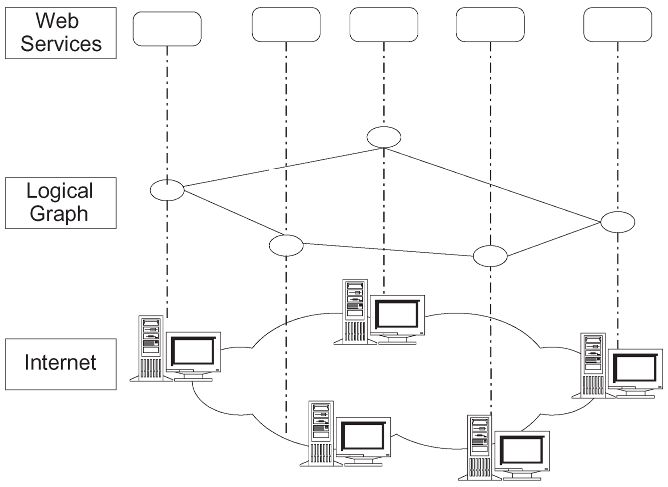

Fig. 1. WEBGOP integrates Web services through a logical graph over the Internet.

The collaboration of Web services is the key to achieving larger goals. Traditional business relies on the collaboration of companies. Companies build on the collaboration of people. Even human bodies work on the collaboration of organs. Naturally, if there is good architectural support for the collaboration of Web services that extends beyond the simple serving of localized services, this would be much more effective.

This paper proposes a programming architecture, WEBGOP, for collaborative Web services using graph-oriented programming (GOP). A logical graph representing a virtual overlay network over the Internet is created to link up the collaborative Web services (see Fig. 1). It is like a stage upon which different players perform different roles. WEBGOP is the stage and the various Web services are the players, and they work together for a collaborative application in the Web environment.

Our work is focused on the integration of Web services and GOP for the proposed programming platform. GOP was introduced by Cao *et al.* [5] as a programming model for configuring distributed systems and programming interacting distributed processes described in a graph-oriented manner. A distributed program is configured using a logical graph for defining and coordinating the interactions amongst the components of the distributed program. Each component is represented as a node in the logical graph. The graph is the unifying element for collating various distributed processes.

In WEBGOP, a similar graph is created to link up Web-server nodes preloaded with local programs (LPs) to provide collaborative Web services. Procedures in the LPs are invoked through either unicast or multicast messages within the overlay network. All of the messages are in the firewall-friendly simple-object access protocol (SOAP) format. The LPs executing in different server nodes work collaboratively to implement a multipoint network application. Each of the nodes may execute autonomously within the local execution environment, while using SOAP to perform remote invocations of LPs installed on linked nodes. The project also provides a rich network-programming interface for a new class of Web applications while retaining the simple Internet protocol (IP) and HTTP.

The remainder of this paper is organized as follows. Section II presents a brief review of related work. Section III explains the proposed architecture and the design of the WEBGOP system. Section IV describes the implementation of the system. Section V presents the results of the experiments done on the prototype. Section VI gives some examples of applications that use the system. Section VII discusses possible future work. Section VIII concludes the paper.

II. Related Work

The slow evolution of network protocols, such as transmission control protocol (TCP) and IP, has led to researchers to advocate the building of active networks that are highly programmable and that allow protocol innovations to be rapidly developed and deployed [14]. While some research into this area has had promising initial results, the feasibility of deploying such an approach on a large-scale and wide-area infrastructure, such as the Internet, is questionable. The issue of deployment arises from the complexity of introducing programmability into routing nodes and the need to address the heterogeneity of the Internet. This can be seen from the limited success of MBONE, which is the backbone virtual network for multicast in the Internet environment [8], [11]. It is layered on top of the existing IP infrastructure to support the routing of IP multicast packets. The nodes in the MBONE are connected by IP tunnels, in which a meshed network is overlaid across the existing IP infrastructure. The endpoints of the tunnel are typically workstation-class machines that have an operating system that supports IP multicast and that run the "mrouted" multicast routing daemon.

A Web proxy is an application-specific service interposed between clients and browsers. It typically functions as a gateway for Web traffic transiting between the endpoints. In its present form, a Web proxy represents an excellent example of how an intermediary node can be incorporated as part of a web service to enhance the web-browsing experience. It is equipped with a hierarchical-caching scheme that reduces the latencies experienced by individual users and aggregate bandwidth consumption.

Programming inside the Web is leveraged by many researchers to address the issues of mobility and nomadic computing. Kleinrock [12] suggested the interposition of "nomadic routers" between end systems and the network. Similarly, "nomadic agents" and "gateways" might be placed near discontinuities in the available bandwidth, e.g., at wireless base stations. Services performed at these gateways would include file caching and image transcoding. Researchers have investigated the use of mobility support routers and indirect TCP [2] to transform fixed-network datagrams for efficient wireless transmission. In the absence of architectural support, these users have adopted a variety of ad hoc services, which perform some application-specific functions. One wonders whether the techniques developed in distributed computing could be applied to the Web to provide a unified programming framework encompassing the abstraction of various Web entities and their interactions.

GOP is a distributed programming framework. It was proposed by Cao *et al.* [3], [4] to provide high-level, structured abstractions of distributed programming. With GOP, a distributed program is built using a user-specified logical graph to define and program the communication and synchronization among those components of the program called LPs. The vertices of the graph are associated with the LPs of the distributed program and the edges of the graph represent interrelationships among the LPs. The vertices of the graph are mapped to the underlying network of physical processors. With GOP, users can concentrate on the structure and the logic of the distributed program while low-level programming for message passing between physical processors, task mapping and graph operations are handled by the system. GOP enables the transparent configuration and programming of interacting distributed processes.

Cao *et al.* [3] extended GOP into *DyGOP* and applied it to the programming of a hierarchical group of Web servers. *DyGOP* provides for the dynamic reconfiguration of the logical graph. With *DyGOP*, nodes can be removed from the graph and new vertices can be added, while LPs can be unbound or rebound to a node. Located in different places, Web servers in a group may contain overlapping information and may cache and/or replicate the data provided at other server sites. They can work together to balance the overall workload of the system and cooperate to provide high-performance Web services. The configuration of a Web server group is dynamic: servers can be created and added to the group at any time; existing servers in a group may be broken down, or may leave the group, and so forth. In their paper, Cao and Chan used *DyGOP* to dynamically reconfigure a Web-server group and produced a central server-based prototypical implementation of a dynamic reconfiguration manager on a local area network of workstations.

Chan and Cao [6] also proposed the concept of using GOP to model the dynamic reconfiguration of programmable networks in the programmable active-network transport architecture (PANTA). Programmable networks are a class of evolving network architecture aimed at overcoming the slow evolution of existing network protocols by augmenting networking nodes and associating software drivers with programmability [14]. PANTA is based on a transport-splicing technique, in which two end-to-end communication paths are linked via an intermediate splice-point router at the transport level. Programmability is provided at the communication ends and the splicing points. With GOP, an active network configuration is built using a logical graph as the underlying abstraction for expressing and defining the communication and synchronization among the programmable nodes. When applied, new services can be deployed and implemented rapidly to enable customization of network properties for domain-specific applications.

The recently emerged concept of Web services has introcduced a new service-oriented architecture in the building of robust Internet applications [9]. Web services are loosely coupled computing tasks that communicate over the Internet using HTTP and XML. The challenge of seamlessly integrating collaborative Web services motivated the development of Web-services flow language (WSFL) [17]. WSFL is an XML language used to describe the composition of web services for business processes. While WSFL and WEBGOP both attempt to address the issue of the composition of web services, they differ in a number of important aspects. In WSFL, the focus is on the composition and orchestration of web services for business processes and, in particular, workflows. Being a declarative language, WSFL uses Web-service definition language (WSDL) to describe service interfaces, while the relationship between the services is externally bound. WEBGOP architecture applies a programmable framework for developing structured graph-oriented multipoint web services. The composition and the relationship between services are captured in well-defined and structured WEBGOP application programming interfaces (APIs) that use rich graph semantics. Unlike WSFL, which is layered on top of WSDL within the computing stack, WEBGOP uses WSDL to describe WEBGOP graph-oriented interfaces.

Our work bears some similarity to the work of Francis [10] on "Yoid." Yoid is a suite of protocols that allows all of the replications and forwarding required to distribute a given application to be done in the endhosts that are running the application themselves. In other words, yoid works in the case where the only replicators/forwarders (distributors) of content are themselves the consumers of the content. Yoid does not force the consumers of the content to do all the distribution—it can also be done by "servers'" in the infrastructure. The similarity between yoid and WEBGOP lies in the use of peer-to-peer edge servers to act as intermediary computation nodes and to be integrated as part of the network-service application. While yoid is designed specifically for use with multicast services over the Internet, WEBGOP acts as a general programming architecture for web services that uses the graph-oriented approach as the abstraction that defines distributed relationships amongst collaborative nodes.

Our work also bears some similarity to the "active-service" approach used by McCanne *et al.* [13] in their system that enables scalable multipoint collaboration (also known as the MASH system). One of their earlier studies was on Chawathe's reliable multicast proxy (RMX) model [7]. The RMX model consists of an RM agent that serves as the interface to a main multicast session and transformation engines to convert data from the format of the main session to the proxied session. User-defined computations are placed within the network as they are in active networks, but all of the routing and forwarding semantics of the current Internet architecture are retained. Their system works on the TCP layer. Because active services do not require any change to the Internet architecture, in today's Internet, they can be deployed incrementally.

III. WEBGOP Architecture and Design

A. Existing Web Architecture

The Web is an architectural framework for accessing hyperlinked documents distributed across the Internet environment. The initial proposal for the Web came from Tim Berners-Lee in 1989. It started off as a one-way system for retrieving static documents, in which a client initiated a request and the server responded by returning the requested document. With the advent of common gateway interface scripts, programmability became available at the server side and it is now possible to have dynamism in the generation of server documents. The introduction of Java Applet extended such programmability to the client side as well. The latest servlet based on Java technology added greater programming flexibility to the interactions between the client and the server.

The concept of Web services recently emerged and introduced a new service-oriented architecture in building robust Internet applications. Web services are loosely-coupled computing tasks that communicate over the Internet using HTTP and XML. Given the immense popularity of HTTP and XML and their wide acceptance as standard Web protocols, the Web service concept is a very promising one for effective ubiquitous e-business over the Internet. The challenge is to seamlessly integrate these different Web services with the ultimate objective of rendering highly coordinated Web services that are robust, sharable, and extensible.

Despite the desire to achieve collaborative Web services, the Web is still basically a client–server system. There is no architectural support for collaborative services involving intermediate parties other than the client and the server. The Web remains a collection of computer pairs without much collaborative effort involving more than two end-to-end computers.

B. Proposed WEBGOP Architecture

The WEBGOP architecture is a programmable framework that employs a graph-oriented programming approach for developing structured multipoint Web services. The aim of WEBGOP is to extend the Web from a client–server system to a structured multipoint system. A logical graph is defined to provide the structure of collaboration. It represents a virtual overlay network over the Internet linking up collaborative Web services.

Fig. 2 shows a more detailed view of a graph in WEBGOP with nodes and connections. Each node is a Web entity installed with the WEBGOP runtime and is associated with one or more Web services. The edge represents logical connections that form the virtual network. Web services communicate with the others through the WEBGOP systems installed on the nodes. Each WEBGOP node is composed of a server unit, a client unit, and a GOP unit. The server and client units provide communication functionalities. The GOP unit performs basic graph-oriented configuration, data transfer, the invocation of Web

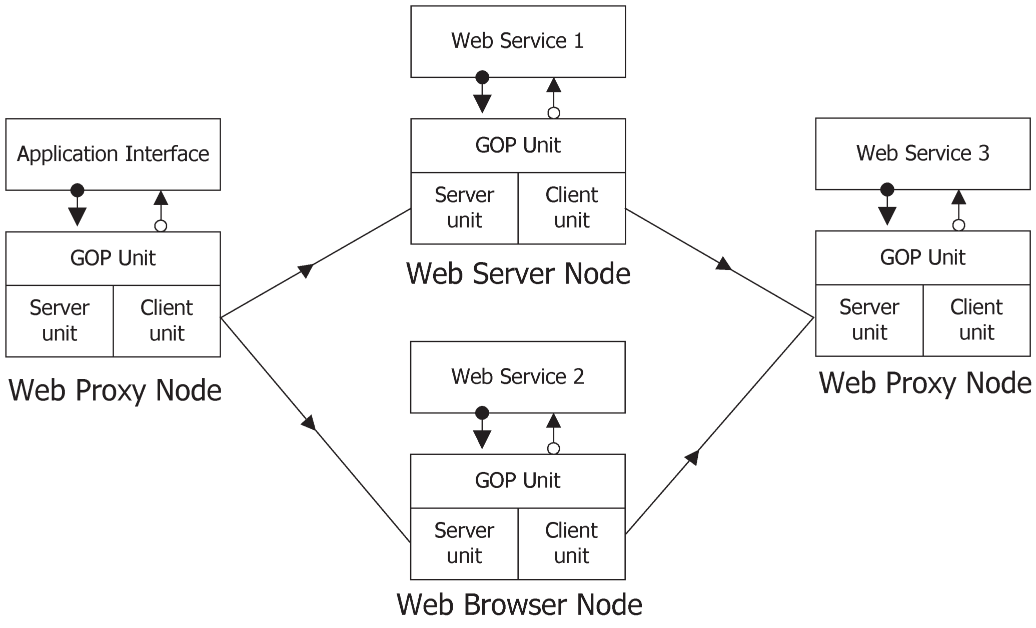

Fig. 2. Connections between nodes in WEBGOP.

services within the GOP context, and so forth. Details of the graph operations are described in Section III-C. As is shown, the context of a programmable framework can exist at different levels of Web entities, including Web servers, Web proxies, and even Web clients. It is not absolutely necessary for all of these entities to be GOP-enabled to leverage on the benefits of dynamically configurable Web services. However, maximum programmability and robustness in creating graph-oriented services can only be achieved with the GOP-enabling of all levels of entities. Each entity comprises a GOP runtime kernel, which performs basic graph-oriented configuration and data transfer, executes LP within the GOP kernel context, and so on.

C. Invocation and Collaboration of Web Services

The messaging system is the key element that controls the selective and/or cooperative invocations of individual Web services in WEBGOP. To support open interaction between entities, the invocation and collaboration of Web services are designed to be independent of specific programming languages or operating systems. Instead, they rely on an open and portable messaging technology, SOAP for remote procedure calls (RPC) [16], to pass commands and parameters to control their actions. They also need GOP to control the collaboration of Web services.

1) Invocation of Web Services: SOAP is an object-oriented, Internet-based protocol for exchanging information between applications in a distributed environment. SOAP is independent of the programming language, platform, or transport mechanism used for the exchange. SOAP's interoperability arises from a simple syntax based on XML. HTTP is the most widely used transfer protocol for SOAP messages, which are formatted as XML documents. Other protocols such as simple mail transfer protocol (SMTP) or the file transfer protocol (FTP) can also be used. The SOAP message-exchange model consists of one-way transmissions from the sender to the receiver and also of two-way communications in a request/response pattern. SOAP messages rely on XML namespaces and on the XML schema-definition language.

RPCs in SOAP are essentially client–server interactions over HTTP. Requests and responses comply with the encoding rules of SOAP. SOAP for RPC defines a mechanism to pass commands and parameters between an HTTP client and an HTTP server. The request-universal resource identifier (URI) in HTTP is typically used at the server end to map to a class or an object, but this is not mandated by SOAP. In addition, the HTTP header SOAPAction specifies the interface name (a URI) and the name of the method to be called on the server. The SOAP message is an XML document whose root element, the Envelope, specifies the overall structure, the intended recipient, and other attributes of the message. SOAP specifies an RPC convention, which includes the representation and format to be used for calls and responses. A method call is modeled as a compound data element consisting of a sequence of fields named accessors, one for each parameter. A return structure consists of the return value as well as the out and in/out parameters. SOAP encoding rules specify the serialization for primitive and application-defined data types.

SOAP for RPC alone provides the functionalities for the invocation of Web services. Extension is however needed to provide facilities for the collaboration of Web services.

2) Collaboration of Web Services: GOP is proposed for controlling the distribution of messages and the collaboration of Web services. With GOP, a distributed program is built using a user-specified logical graph to define and program the communication and synchronization of those program components called LPs. The vertices of the graph are associated with the LPs of the distributed program and the edges of the graph represent interrelationships among LPs. The vertices of the graph are mapped to the underlying network of physical processors. With GOP, users can concentrate on the structure and the logic of the distributed program while low-level programming for

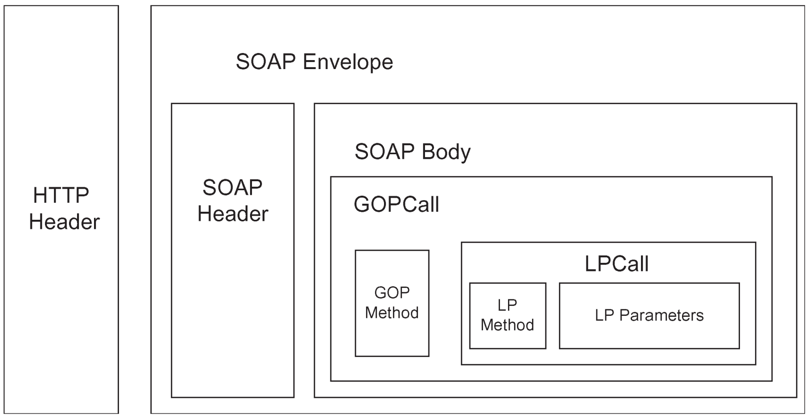

Fig. 3. Structure of a WEBGOP request message.

messages passing between physical processors, task mapping and graph operations are handled by the system. GOP enables the transparent configuration and programming of interacting distributed processes.

For each application, a graph with a unique graph identity must first be defined and made known to all concerned processors. Subsequent collaborative actions of the distributed programs can make quick references to the graph identity only.

A Web application involving a number of collaborating Web services is in fact a distributed program. The Web services are the LPs. The servers that provide the Web services are the nodes. The underlying logical structure of the relationship of collaboration can be modeled using a logical graph. By controlling the distribution of invocation calls to Web services through GOP, the collaboration of Web services can be achieved and controlled.

Similar to distributed programs, for each collaborative Web application, a graph with a unique graph identity must first be defined and made known to all of the servers concerned. Subsequent collaborative actions of the Web services can make quick reference to the graph identity only. Collaboration of services is defined according to the specified graph. Servers can also hold multiple graphs so that they can be bound to different graphs when required.

SOAP is a good protocol candidate for implementing GOP in the Web environment. It allows for hierarchically structured queries and responses, and specifies the serialization of primitive string data types, and aggregates like arrays and vectors. Sparse arrays, and protocols for sending parts of them, are also supported. In the context of GOP, new types, such as an edge, graph, and map, may be defined inside a schema definition.

SOAP has to be extended for the purpose of controlling joint invocations of collaborative Web services. Two sets of calls have to be made: one to invoke Web services, which are LPs in the context of GOP; and another for making a GOP-based distribution of the Web-service invocation calls. For simplicity, we refer to the Web-service invocation call as an LP call and the call for GOP-based distribution as a GOP call. In this project, we make use of a nested-invocation approach to format the SOAP calling convention in a GOP-based Web-service distribution. An LP call is embedded in a GOP call. Each SOAP request includes a GOP call, which in turn includes a parameter of the LP call. The GOP call controls the distribution of the message, while the LP call invokes the required Web service.

D. WEBGOP Request Message

This section describes the format for a standard WEBGOP message. A WEBGOP message is basically an SOAP message composed of an HTTP post header and an SOAP envelope with an SOAP body. The SOAP body includes a remote method call for graph-oriented operation, which in turn includes a method call for the LP as one of its arguments. Each WEBGOP message is in fact a nested RPC in SOAP. The outer RPC invokes a GOP module, which controls the distribution of the message. The inner RPC directly invokes a Web service. Fig. 3 shows the diagrammatic structure of a WEBGOP request message.

The following is a simple illustrative example that instructs the recipient server to invoke the method *setData* of an LP defined in the graph with Graph ID, *local.hello.2*; and relay the message to other descendents in the graph.

```
1  POST /webgop/servlet/webgop.server.
            WebgopServlet HTTP/1.0
2  Host: localhost:8001
3  Content-Type: text/xml
4  Content-Length: 629
5  SOAPAction: “”
6
7  <SOAP-ENV:Envelope xmlns:SOAP-ENV
   = “http://schemas.xmlsoap.org/soap/envelope/”
   xmlns:xsi = “http://www.w3.org/1999/
   XMLSchema-instance”
   xmlns:xsd = “http://www.w3.org/1999/
   XMLSchema”>
8  <SOAP-ENV:Body>
9  <ns1:relay2Descendents xmlns:ns1
   = “urn:webgop:service” SOAP-ENV:encodingStyle
   = “http://schemas.xmlsoap.org/soap/encoding/”>
10 <gid xsi:type = “xsd:string”>local.hello.2</gid>
11 <source xsi:type = “xsd:int”>0</source>
12 <ns2:setData xmlns:ns2
```

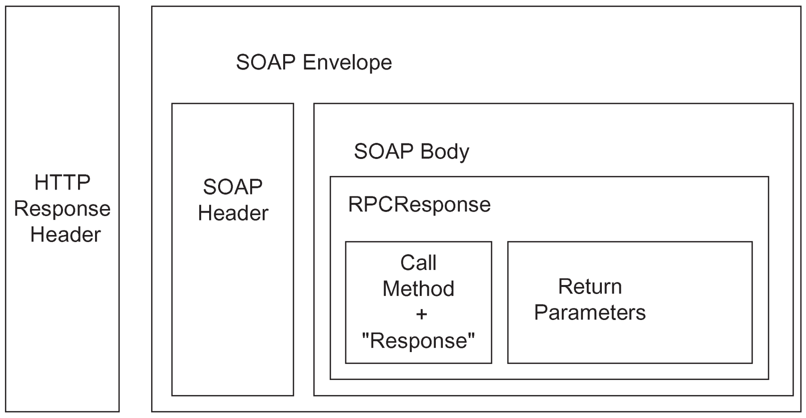

Fig. 4. Structure of a WEBGOP response message.

```
       = “urn:webgop:lp” xmlns:ns3
       = “urn:gop_encoding” xsi:type
       = “ns3:LPCall”>
13  <data xsi:type
       = “xsd:string”>Hello Webgop</data>
14  </ns2:setData>
15  </ns1:relay2Descendents>
16  </SOAP-ENV:Body>
17  </SOAP-ENV:Envelope>
```

Lines 1–5 are the HTTP header. The HTTP header shows that the message is posted to the */webgop/servlet/webgop.server.WebgopServlet* of the host, *localhost:8001*. Lines 7–17 are the SOAP envelope. There is no SOAP header in this example. Inside the SOAP envelope, lines 8–16 are the SOAP body. Lines 9–15 are the GOP call that shows that the invocation object is *urn:webgop:service* under the namespace, *ns1*, and the GOP method to be invoked is *relay2Descendents*. Line 10 shows the Graph ID, *local.hello.2*. Line 11 shows that the call originated from Node 0 of the graph. Lines 12–14 are the LP call nested within the GOP call. The LP object to be invoked is *urn:webgop:lp* under the namespace, *ns2*; and the LP method to be invoked is *setData*. Line 13 is the string parameter, *Hello Webgop*, to be sent to the LP program.

In this project, HTTP and SOAP are proposed as the baseline protocols. We extend them to provide the functionalities required by WEBGOP. The advantages of HTTP and SOAP are their immense popularity in the Web community. The ubiquitous support of these protocols makes them the ideal candidates to offer the transferring of messages across wide-area Web infrastructures and the invocation of Web services, in a truly open and highly accessible programming platform. The conformance to standards opens the possibility of reusing many of the readily available libraries that have been implemented. Furthermore, there is an added advantage to using the security and error controls provided by HTTPs and the lower TCP layers.

E. WEBGOP Response Message

The response to an SOAP for RPC request is defined in the SOAP specifications. WEBGOP follows the specifications. Fig. 4 shows the diagrammatic structure of a WEBGOP response message.

This project follows the convention of naming the structure of the return data after the name of the method, with the string *Response* appended. For instance, the response to a *setData* call to a node is as follows:

```
1   HTTP/1.0 200 OK
2   Content-Type: text/xml; charset = UTF-8
3   Content-Length: 427
4   Servlet-Engine: Tomcat Web Server/3.2.1
    (JSP 1.1; Servlet 2.2;
    Java 1.3.1_01; Windows 98 4.10 x86;
    java.vendor = Sun Microsystems Inc.)
5
6   <SOAP-ENV:Envelope xmlns:SOAP-ENV
    = “http://schemas.xmlsoap.org/soap/envelope/”
    xmlns:xsi = “http://www.w3.org/1999/
    XMLSchema-instance” xmlns:xsd
    = “http://www.w3.org/1999/XMLSchema”>
7   <SOAP-ENV:Body>
8   <ns1:setDataResponse xmlns:ns1 = “urn:webgop:lp”
    SOAP-ENV:encodingStyle = “http://schemas.
    xmlsoap.org/soap/encoding/”>
9   <return xsi:type=“xsd:string”>
    Hello Webgop</return>
10  </ns1:setDataResponse>
11  </SOAP-ENV:Body>
12  </SOAP-ENV:Envelope>
```

Lines 1–4 are the HTTP header. Lines 6–12 are the SOAP envelope, and lines 7–11 are the SOAP body. Lines 8–10 are the response to the LP Call that called the invocation object *urn:webgop:lp* under the namespace, *ns1*. The LP method invoked is *setData* and a string *Response* is appended to the name to show that this is a response. Line 9 shows the return data, *Hello Webgop* of the string type.

F. Fault Reporting

Fault reporting is defined in SOAP for RPC. The system is extended in WEBGOP. In SOAP, exceptions are returned as faults to the client. Extension is provided for GOP errors. The following is an example of a fault report in which the specified Graph ID is not available from the server.

```
1  HTTP/1.0 500 Internal Server Error
2  Content-Type: text/xml; charset = UTF-8
3  Content-Length: 428
4  Servlet-Engine: Tomcat Web Server/3.2.1
   (JSP 1.1; Servlet 2.2; Java 1.3.1_01;
   Windows 98 4.10 x86; java.vendor
   = Sun Microsystems Inc.)
5
6  <SOAP-ENV:Envelope xmlns:SOAP-ENV
   = "http://schemas.xmlsoap.org/soap/envelope/"
   xmlns:xsi = "http://www.w3.org/1999/
   XMLSchema-instance" xmlns:xsd = "http://
   www.w3.org/1999/XMLSchema">
7  <SOAP-ENV:Body>
8  <SOAP-ENV:Fault>
9  <faultcode>SOAP-ENV:Server</faultcode>
10 <faultstring>Graph ID not found</faultstring>
11 <faultactor>/webgop/servlet/webgop.server.
   WebgopServlet</faultactor>
12 </SOAP-ENV:Fault>
13 </SOAP-ENV:Body>
14 </SOAP-ENV:Envelope>
```

In the above example, lines 8–12 show that a fault occurred. Line 9 is the fault code for automatic machine classification. Line 10 shows the fault message, *Graph ID not found*. Line 11 shows that the fault actor is */webgop/servlet/webgop.server.WebgopServlet*.

IV. Prototype Implementation

A. Development Environment

This section describes the design and development of the WEBGOP system, which provides a programming framework for the rapid development of multipoint collaborative Web services. The system is developed in JAVA to provide strong portability using Java Development Kit 1.3.1. It builds on the following components of development: Jakarta-Tomcat-3.2.1, Apache SOAP 2.0, and Apache Xerces Java Parser 1.4.3 [1]. The Jakarta-Tomcat-3.2.1 provides the HTTP server communication functionalities, while Apache SOAP 2.0 provides the SOAP encoding and decoding functionalities. The Apache Xerces Java Parser 1.4.3 is needed for the XML parsing of the SOAP messages.

B. Node Architecture

WEBGOP is implemented as a network of SOAP-enabled nodes with multiple-incoming and multiple-outgoing links. In essence, a node in WEBGOP is composed of a server unit, a GOP/invocation unit, and a client unit. The server unit is responsible for receiving incoming messages. The GOP/invocation unit is responsible for distributing messages according to the specified graph and for the invocation of LPs. The client unit is for outgoing communications. The node architecture is depicted in Fig. 5.

C. Node Operation

A node can operate as a requester, in which case WEBGOP request messages are first initiated; or as an intermediate or end receiver, in which case WEBGOP messages are received, processed, and forwarded to other nodes, as appropriate.

When an application wants to initiate a WEBGOP request, it specifies a graph identity for distribution destinations and calls the appropriate method in the *webgop.invocation.Service* class in the invocation unit. Based on the graph identity, the GOP unit is consulted for detailed information on destinations. A WEBGOP message consisting of an LP call nested within a GOP call is formed. The request is then encoded into the HTTP–SOAP format and sent out through the client unit. Upon receipt of the corresponding responses, the WEBGOP system is responsible for decoding and routing the message to the application program.

On the receiver side, a WEBGOP request is received at the server unit and decoded. The system extracts a GOP call, which invokes an appropriate method in the *webgop.invocation.Service* class in the GOP/invocation unit. The graph registry is consulted for any detailed forwarding destinations based on the graph identity. If necessary, a forwarding WEBGOP message is formed, encoded into the HTTP–SOAP format, and sent out through the client unit. Meanwhile, an LP call is extracted from the GOP call and the appropriate method in the LP is invoked. Finally, a WEBGOP response is prepared based on the results of the execution and passed back to the requester through the server unit.

D. Issues of Implementation

1) Basic Data Types: In implementing the WEBGOP framework, three basic types of data are defined: *Graphs*, *Edges*, and *Maps*. Graphs are the underlying abstraction behind collaboration in the architecture. A *Graph* refers to a logical graph that shows the connections of the computers involved in a collaborative Web service application. It consists of a set of *Map*s representing the computers and a set of *Edge*s representing the connections. The key parameters of the *Graph* type are the identity of the graph, the number of nodes, the number of edges, the set of *Map*s, and the set of *Edge*s. These new types are not included in Apache SOAP. They are added to the SOAP mapping registry so that the WEBGOP system can encode such data in an SOAP format and decode them when required.

2) Graph Definition: The setting up of a graph amongst Web services is an important step in defining the channels of communication and relationships for subsequent collaboration. In the implementation of the prototype, when a programmer defines a graph, the WEBGOP system distributes the graph to all of the nodes concerned along the edges of the graph. If a fault occurs in any step of the distribution and graph-definition process, the whole graph set-up process is regarded as a failure, and a fault is reported. Responses to the results of the graph

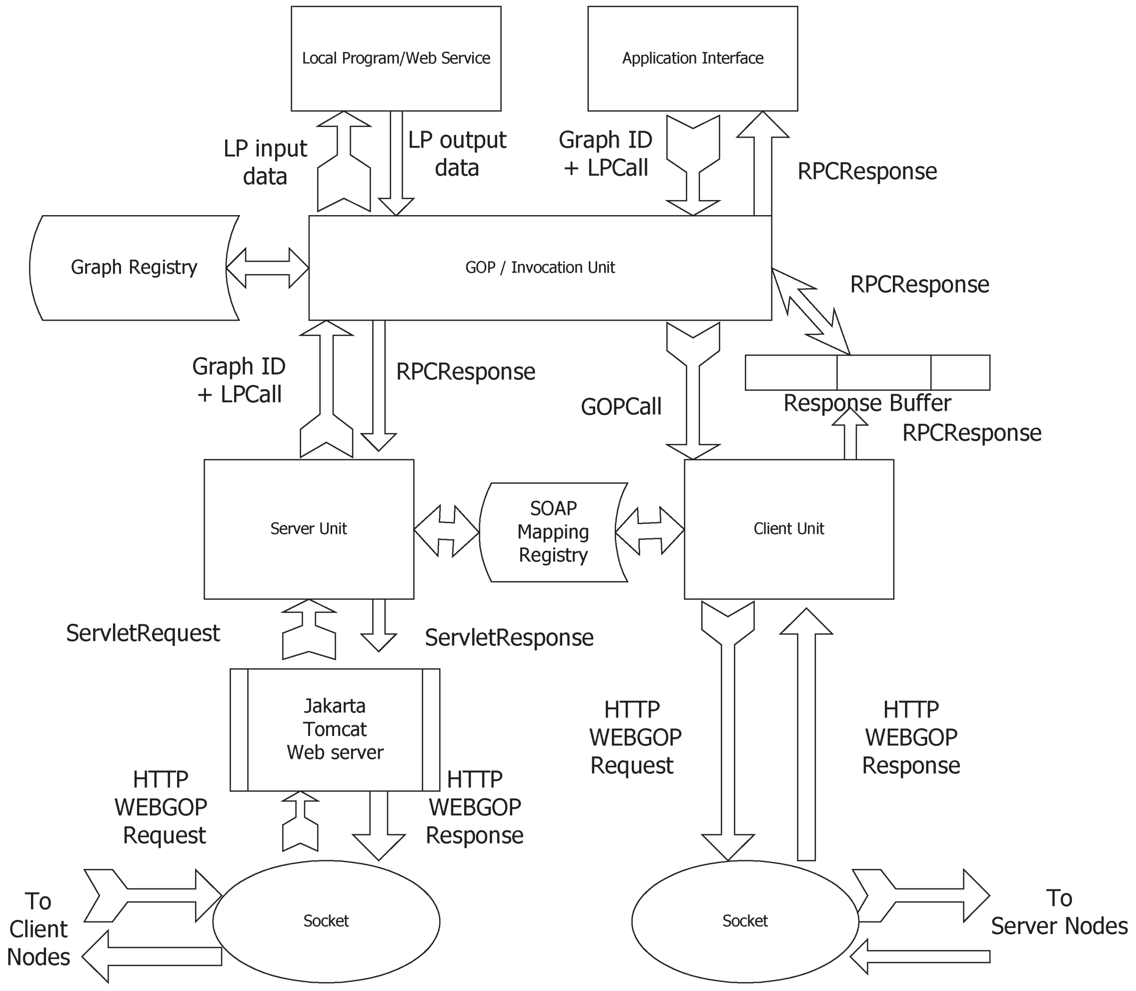

Fig. 5. Internal architecture of a WEBGOP node.

set-up on each server are therefore required, but none of them need to relay to the originator. To reduce the number and size of responses, a node with multiple children does not relay all of the responses from its children to its parent, but instead only provides a consolidated response.

Once the unifying graph is established amongst WEBGOP servers with a graph identity, subsequent GOP operations can be performed by making a reference to the graph identity. Multiple logical graphs can be defined for a server node to link up to a number of different networks. The different graph identities provide the differentiation in graphs. Each graph identity must be unique. The graph information is stored in a Graph Registry maintained by each server participating in the logical graph. Fig. 6 shows an example of a server, *host4*, participating in two virtual networks through two logical graphs, namely, *polyu.ring* and *polyu.tree*.

3) Spanning Tree: When a server defines a graph or sends a message to all nodes of a graph, there is a possibility that the graph is closed. As a result, the messaging process will enter an endless loop. A spanning tree is employed to overcome this problem. When a message is sent to all nodes starting from a node, a spanning tree is first created starting from that node. Any edge in the graph that ends with a node that has already been visited is discarded in the formation of the spanning tree.

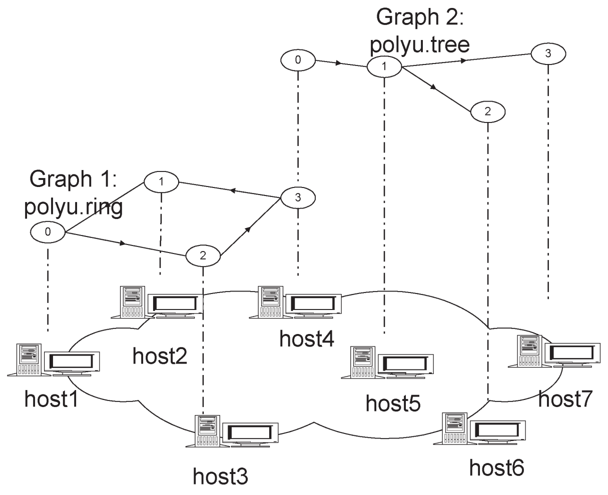

Fig. 6. A host can hold multiple logical graphs.

The spanning tree so formed thus includes all of the necessary and sufficient edges for the proper distribution of messages to all of the nodes.

4) Graph Modification: The prototype implementation allows changes to be made to the configuration of a graph after it is defined amongst servers. We cater to changes in edge configuration as well as in node configuration. The changes are expressed in terms of obsolete edges, new edges, obsolete nodes, and new nodes.

If no additional hosts are involved, the changes are broadcast to all of the hosts, and their graph information is updated. If new hosts are involved, the graph information is first dispatched to the new hosts. The changes alone are meaningless to the new hosts because they do not hold any previous information about the graph. Subsequently, the changes are broadcast to the remaining hosts of the graph. The prototype implementation provides separate methods, namely, *addHost()* and *modifyGraph()*, for users to use in modifying graph configurations with and without adding new hosts, respectively.

5) Local Program: In this prototype implementation, an LP method is invoked through reflection. Basically, this provides a one-way transfer of data from the GOP/invocation unit to the LP. To ensure that the LP can send back data to the GOP unit, each LP must implement a method called *setGopService()*. When an LP is instantiated, the GOP unit automatically invokes this method to relay its own Object ID to the LP. Once instantiated, the object ID of the LP is stored together with the Graph ID in the Graph Registry. This facilitates continued communications with other Web services on other nodes. To completely end an LP, the method *end()* of an LP shall be called. The method should include all of the cleanup procedures for ending the LP. Each LP must follow a standard interface that implements both the *setGopService()* method and the *end()* method.

6) RPC and Responses: The response mechanism of WEBGOP requires special consideration. In WEBGOP, many servers are involved. With a single request from a client, it is possible to invoke LPs residing on a number of child servers. There may also be further invocations of LPs residing on descendent servers down the graph. As a result, a multitude of outputs are generated and the cumulative execution time can be quite long. A simple request–response mechanism is not suitable.

In the prototype implementation, the return of responses is generally limited to one-level calls. Responses are collected only from children not from descendents because of the response-implosion problem. Intermediate nodes do not relay responses from distant nodes back to the originator. The originator would be overloaded if it has to handle all of the responses from its descendents.

When multicasting to all levels of descendents, the sender does not wait for full responses requiring only simple acknowledgments from its children one level below. The simple acknowledgment system works between adjacent levels. Each server keeps track of the distribution to its children. Acknowledgments are provided only for the purpose of monitoring the multicast operation. The unidirectional multicast approach offers a realistic framework for the distribution of invocations while minimizing the potential drain in computing resources.

One problem with this one-way multicast approach is that the sender has little control over the distribution of data and the invocation of programs. For mission-critical messages, programmers may consider more complicated control mechanisms, such as those based on a consolidated response. Consider the example of a graph-definition process. The originator is concerned if a fault occurs in the distribution and graph-definition processes of all of the participants. Responses are required, but not all of them need be relayed to the originator. To reduce the number and size of the responses, a node with multiple children does not relay all of the responses from its children to its parent, instead providing only a consolidated response.

7) Response Buffer: The WEBGOP system may generate many responses to a host. This is different from normal programming for a single client–server communication, in which a process can stop and wait for the response. A method must be devised to receive multiple responses. To overcome this problem, a response buffer is provided. The responses are passively stored in the buffer and must be actively retrieved.

When a host sends a call to multiple recipients, it sets up a communication thread and a response buffer for each recipient. Retrieval access to the buffers is blocked until the buffers are filled up by responses returned by the recipients. When a response arrives, it is stored in the buffer. The blocking of retrieval access to the buffer is removed and all awaiting threads are notified.

E. Application Programming Interface

WEBGOP provides a rich application-programming interface for collaborative Web-services applications. The methods in Table I are the main APIs defined in *webgop.invocation. Service*, the class providing the runtime service of WEBGOP. They are the WEBGOP primitives to be invoked by application servlets and LPs/Web services.

The APIs can be broadly classified as follows:

- **Graph Operations**: defineGraph, deleteGraph, modifyGraph, addHost;
- **Graph Information**: listGraph, getGraphInfo, getCurrentNode, getNodewhere, isSTLeaf, isRTRoot;
- **Web Service Invocation**: invokeLP, call2Node, call2Children, call2TreeChildren, call2Parents, call2TreeParents, send2Descendents.

A typical WEBGOP application operates as follows. The starting point of execution is usually an application servlet. First, it invokes the *defineGraph* method to link up all of the relevant servers. The WEBGOP system will automatically pass the graph to the relevant nodes along the edges in the graph. The application servlet then invokes Web-service invocation primitives, such as *call2Children*, *call2Parents*, and *send2Descendents*, to selectively or jointly invoke the Web service/LPs located on the other servers. The WEBGOP system will automatically generate and distribute invocation messages to the destination nodes.

If modifications to a defined graph are necessary, the methods, *addHost*, *modifyGraph*, and *deleteGraph* should be called. The WEBGOP system will pass graph-modification messages to the relevant nodes and update the graph copies stored in them. *ListGraph*, *getGraphInfo*, *getCurrentNode*, and *getNodewhere* are the methods provided for checking the graph information stored in a server.

By using these WEBGOP primitives, programmers are relieved of the burden of coding low-level system functions such

TABLE I
MAIN APIs DEFINED IN webgop.invocation.Service

RPCResponse `addHost`(java.lang.String gid, `Edge`[] obEdge, `Edge`[] nwEdge, `Map`[] obMap, `Map`[] nwMap)	The addHost method distributes a graph to all new hosts and modifies graph copies in other hosts. Valid only at the root of an old spanning tree.
ResponseBuffer `call2Children`(java.lang.String gid, `LPCall` lpCall)	The call2Children method makes a remote-procedure call to the local programs residing on the children of a graph.
ResponseBuffer `call2Node`(java.lang.String gid, int destination, `LPCall` lpCall)	The call2Node method makes a remote-procedure call to the local program residing on a node of a graph.
ResponseBuffer `call2Parents`(java.lang.String gid, `LPCall` lpCall)	The call2Parents method makes a remote-procedure call to the local programs residing on the parents of a graph.
ResponseBuffer `call2TreeChildren`(java.lang.String gid, int source, `LPCall` lpCall)	The call2TreeChildren method makes a remote-procedure call to the local programs residing on the children of a spanning tree derived from the graph.
ResponseBuffer `call2TreeParents`(java.lang.String gid, int source, `LPCall` lpCall)	The call2TreeParents method makes a remote-procedure call to the local programs residing on the parents of a reverse spanning tree derived from the graph.
RPCResponse `defineGraph`(`Graph` graph, int source, int currentNode)	The defineGraph method distributes a graph to all of its nodes according to the spanning tree, starting at the source.
RPCResponse `deleteGraph`(java.lang.String gid, int source)	The deleteGraph method distributes a graph-deletion message to all nodes according to the spanning tree, starting at the source and deleting all graph copies.
int `getCurrentNode`(java.lang.String gid)	getCurrentNode gets the current node ID for a given graph.
GraphInfo `getGraphInfo`(java.lang.String gid)	getGraphInfo gets the current node ID for a given graph.
gid `getNodewhere`(java.lang.String gid)	getNodewhere gets the location of the current node for a given graph.
RPCResponse `invokeLP`(`GraphInfo` graphInfo, `LPCall` lpCall)	The invokeLP method invokes the local program residing on the current server.
boolean `isRTRoot`(java.lang.String gid, int source)	isRTRoot checks whether the current server is a root on a reverse spanning tree, starting from a given source node.
boolean `isSTLeaf`(java.lang.String gid, int source)	isSTLeaf checks whether the current server is a leaf on a spanning tree, starting from a given source node.
Graphlist `listGraph`()	listGraph gets all of the entries in the Graphinfo registry.
RPCResponse `modifyGraph`(java.lang.String gid, int source, `Edge`[] obEdge, `Edge`[] nwEdge, `Map`[] obMap, `Map`[] nwMap)	The modifyGraph method distributes a graph-modification message to all nodes according to the old spanning tree, starting at the source.
ResponseBuffer `send2Descendents`(java.lang.String gid, `LPCall` lpCall)	The send2Descendents method casts a remote-procedure call to the local programs residing on the descendents of a graph.

as the passing of messages and can concentrate on the logic of their programs.

V. Experiments and Evaluation

This section presents the experiments performed on the prototype implementation of the WEBGOP system. The purpose is to characterize and evaluate the performance of the proposed system.

A. Test Setup

Tests were conducted on a network of UNIX machines connected via a local area network. All of the machines were installed with the prototype implementation of WEBGOP running on top of TOMCAT Web servers. Tests were conducted on a number of logical-graph configurations. The detailed setup configurations of the experiments are shown in Table II.

A series of parallel- and serial-graph configurations was tested. Fig. 7 shows the serial-graph configurations and Fig. 8 shows the parallel-graph configurations. Each circle in the figures represents a UNIX machine. Each one has a unique IP address. The graphs represent the virtual networks in which the machines interact under WEBGOP.

The outgoing message is timestamped by the originator. On receiving the message, the LP, *CheckTime*, compares its own local time and the timestamp to compute the unadjusted

TABLE II
SETUP OF EXPERIMENTS

Graph ID	No. of UNIX Machines Involved	No. of Children Connected to Originator	No. of Levels of Descendents	Graph Configuration
polyu.time.1	2	1	1	parallel/serial
polyu.time.s2	3	1	2	serial
polyu.time.s4	5	1	4	serial
polyu.time.s6	7	1	6	serial
polyu.time.s8	9	1	8	serial
polyu.time.s10	11	1	10	serial
polyu.time.p2	3	2	1	parallel
polyu.time.p4	5	4	1	parallel
polyu.time.p6	7	6	1	parallel
polyu.time.p8	9	8	1	parallel
polyu.time.p10	11	10	1	parallel

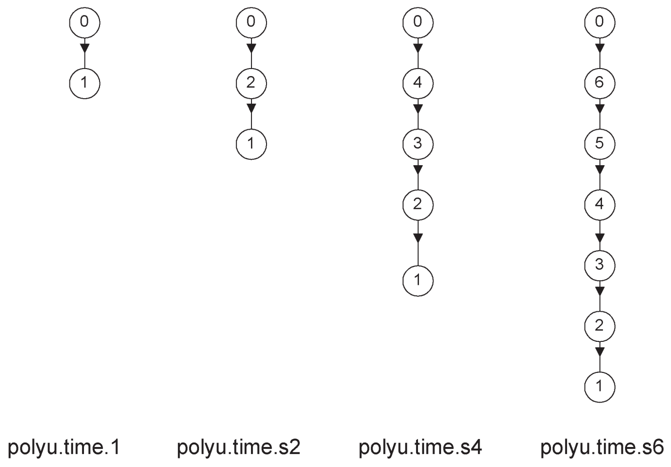

Fig. 7. Tested serial-graph configurations.

invocation latency. The following is the listing of the method *invokeTime*.

```
public long invokeTime(long clientTime, String data){
long serverTime = System.currentTimeMillis();
long invokeTime = serverTime-clientTime;
System.out.println(serverTime +
“>>LP:CheckTime>>Unadjusted
   invocation time(ms): ”
+ invokeTime);
System.out.println(“Data: ” + data);
return invokeTime;
}
```

Therefore, the invocation time that is computed must be adjusted for the discrepancies between the clock of the originator and that of the recipient. To do so, the originator sends a separate message using a WEBGOP primitive, *call2Node,* directly to each descendent to check the required clock adjustment. The adjustment is taken to be the difference between the mean time of the dispatching of the message as recorded by the originator and the mean time of the receipt of the message as recorded by the descendent. Fig. 9 diagrammatically shows the proposed clock adjustment.

B. Setting Up Graphs Amongst Servers

Before any application can run normally, a graph has to be set up among the relevant servers to initialize the system. Experiments were conducted using the WEBGOP primitive, *defineGraph*. In setting up a graph among servers, the originator distributes the graph to all other servers. In return, they acknowledge the receipt of the graph. Fig. 10 summarizes the graph setup times for serial graph configurations and Fig. 11 summarizes the graph setup times for parallel-graph configurations. The processes were repeated ten times for each graph. The first cycle of graph definition generally took a longer time due to the initialization of the servlets on each of the servers. Subsequent cycles generally remained steady, but there were fluctuations because of the other workloads of the servers. In a serial configuration, the processes happened sequentially and therefore more time was required to define a graph with more levels of descendents. The setup times for parallel configurations were generally shorter than those for serial configurations. This was because the graphs in the descendents were set up in parallel rather than in sequence.

C. RPCs by call2Children

In the second stage of the experiments, tests were made on the WEBGOP primitive, *call2Children*. The objective of this set of tests was to check the time required to get returns from LPs installed on the children nodes. Upon receipt of a request for call to the children, the originator server looks up the specified graph in its graph registry, sends *call2Children* messages to the children of the graph, and waits for returns. On receiving the RPC, the children look up their own graph registries and invoke the LP specified in the graph. Returns are then provided for the originator to pick up.

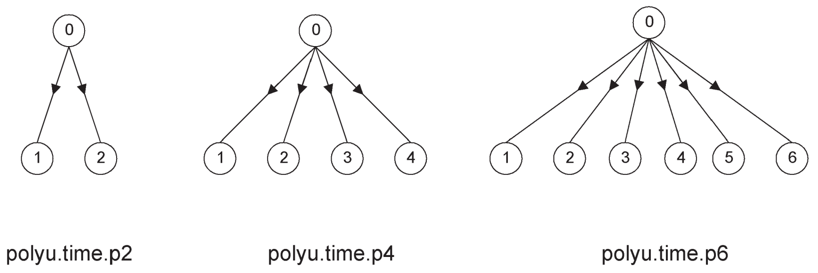

Fig. 8. Tested parallel-graph configurations.

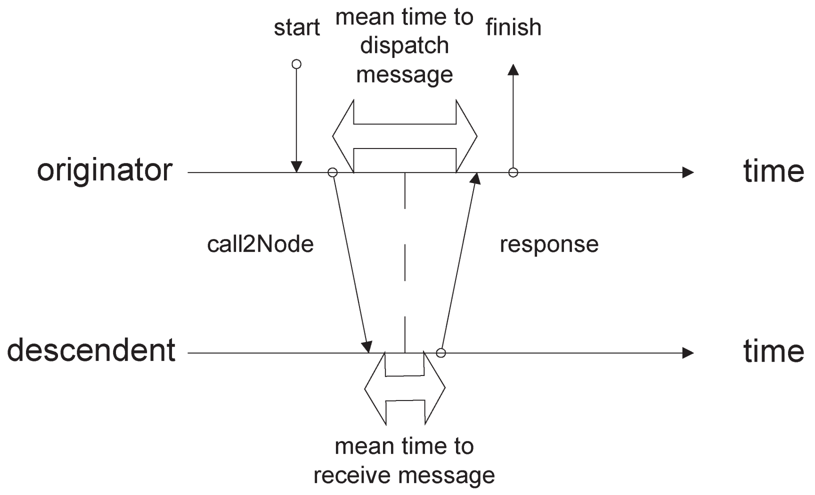

Fig. 9. Clock adjustment between the originator and a descendent.

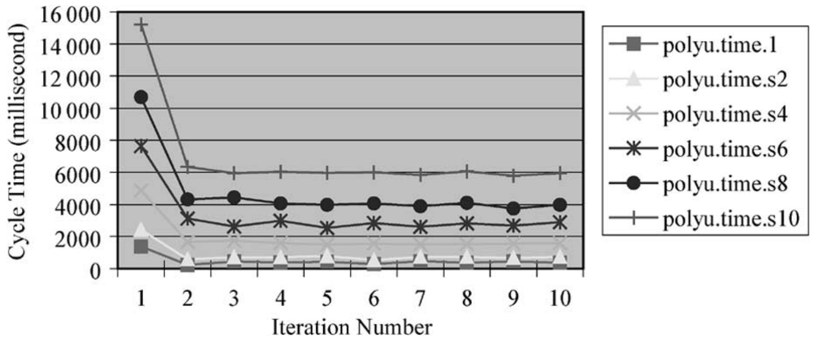

Fig. 10. Tests on setting up serial graphs among servers.

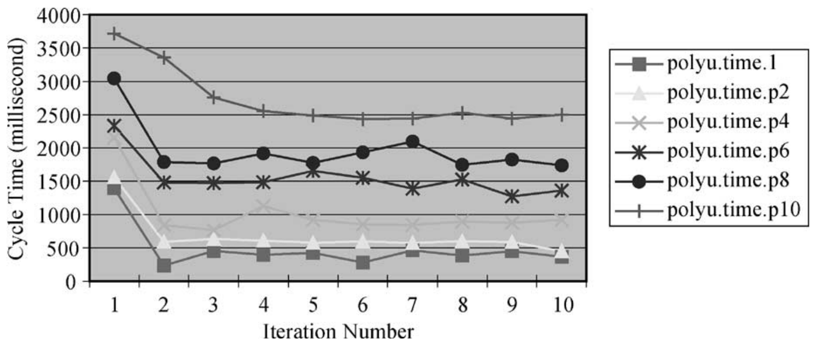

Fig. 11. Tests on setting up parallel graphs among servers.

Fig. 12 summarizes the time required to make calls to children and to get responses. Each call was repeated ten times. As the LP was small and did not require much initialization, the response time of the first cycle was not much longer. It was observed that more time was required to complete the RPCs for more children. The delay was mainly due to the handling of

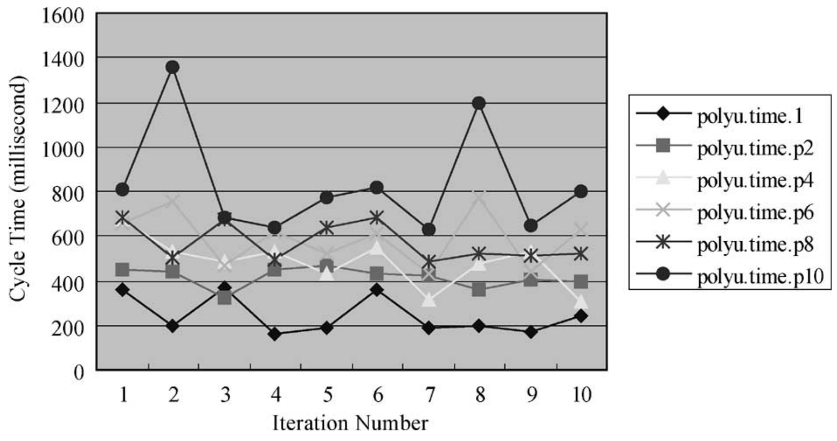

Fig. 12. Tests on remote-procedure calls to children.

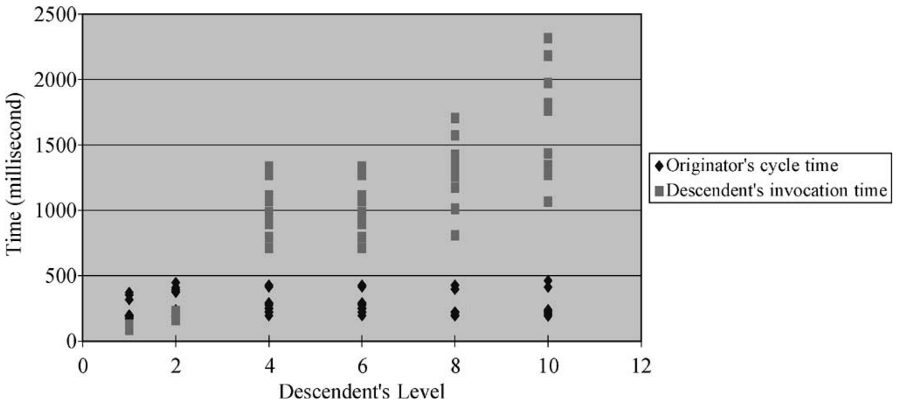

Fig. 13. Tests on *send2Descendents* for serial-graph configurations.

responses. Although a multithreaded implementation was used, the responses all returned to the same node, requiring sequential decoding of the responses.

D. Web Service Invocations by send2Descendents

In the third stage of the experiments, tests were conducted on the WEBGOP primitive, *send2Descendents*. The objective of this set of tests was to check the time required to invoke LPs installed on the descendents. Contrary to *call2Children*, responses were not awaited for messages sent by the *send2Descendents* command.

When the server originator receives an instruction of *send2Descendents*, it looks up its graph registry for the specified graph, and sends the message to the intermediate server. The intermediate server then looks up its own graph registry, invokes the LP specified in the graph, and relays the message to its child. Without waiting for further responses from its own child, the intermediate server immediately returns a response to its parent. This process is repeated for each descendent in the graph. In so doing, the message is sent sequentially from one level of descendent to the next.

Fig. 13 summarizes the invocation times and response return times for serial-graph configurations, and Fig. 14 summarizes the invocation times and response return times for parallel-graph configurations. The delay to the Web service invocation increased almost linearly with the number of levels of descendents. The invocation latencies of the Web service on the descendents were about 100 to 200 ms per level of descendent. By having multiple children on one node, the amount of delay experienced in the invocation of Web services on the children was reduced to insignificance. The dispatches were made almost spontaneously because of the multithreaded implementation. However, the response return times tended to be longer because of the implosion of the responses from the children nodes. There is also a limit on the number of connections a node can handle. For a parallel-graph configuration, the messages are sent to the other nodes in parallel rather than in sequence. The operation is similar to that for *call2Children* in Fig. 14, except that the children will check whether they have any descendents of their own and act accordingly.

E. Observations

Based on the tests, the following observations were made.

The time required to establish graphs is much longer than that required to make calls to children or to send messages to descendents. After a graph is defined, the efforts required to invoke collaborative Web services are significantly smaller than those for reestablishing the graph. This shows that although there may be overheads in establishing collaborative links amongst Web services, if repeated use of the linkages is envisaged, the efforts are well justified.

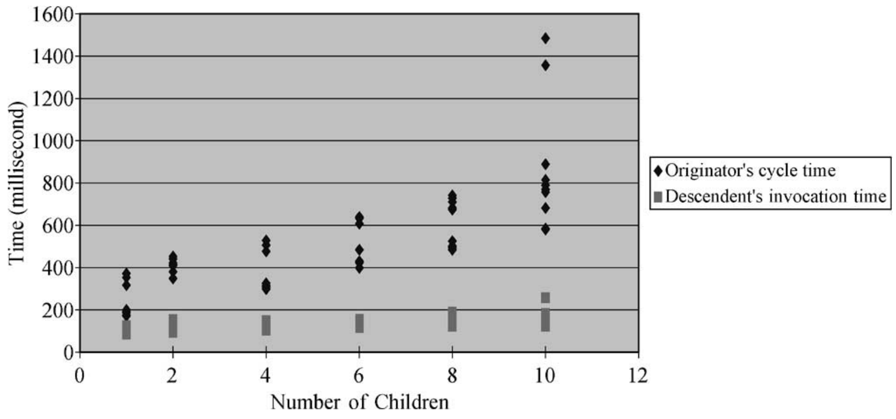

Fig. 14. Tests on *send2Descendents* for parallel-graph configurations.

The parallel-graph configuration generally provides for faster invocation and responses than the serial-graph configuration. However, significant computing resources are required to handle multiple messages and responses. There may also be a response-implosion problem if too many children are attached to a node. The serial configuration eases the workloads to other servers. For each application and hardware configuration, an optimum combination of a parallel and serial arrangement of the graph structure should be carefully considered and deployed. WEBGOP allows the programmer to choose the best collaboration structure for his/her application.

WEBGOP enables the processing workload to be spread from a single server to multiple servers. As each participating server can actively process data, the amount of data transfers can also be reduced and significant network bandwidth can be saved. The invocation latencies are of the same order of magnitude as the general network latencies experienced by Web users.

Web-services architecture works on an emerging technological paradigm: interoperability. In this project, we take the same approach and use the XML-based SOAP for communication. The main difference between the Web-services architecture in universal description, discovery, and integration of business for the Web [15] (UDDI) and the approach adopted in this project is in the way components are integrated. The UDDI's Web-service architecture promotes significant decoupling and runtime integration of components. This has the disadvantage of incurring large overheads in locating and binding components with each runtime instance. For an application that repeatedly uses the same set of components, the overheads of finding and binding components should be done once. By adopting GOP, the structure and dependence of components can be established and described in a simple graph formation. To reconnect and redeploy the same components, we need only make reference to a single-graph instance, which significantly simplifies the programming effort and speeds up the process. While reconfiguration may incur additional resources, it is not expected that reconfigurations would be frequent.

VI. Deploying WEBGOP Applications

This section presents the development of Web-based applications that are based on the WEBGOP programming architecture. The purpose of the experiments is to provide us with an ideal opportunity to study the complex interactions among the end-to-end services as well as the data flow among the entities. The examples demonstrate the benefits of WEBGOP in providing graphical constructs that allow an intermediary node program to directly describe its relationship among its neighbors in the form of various relational axes. In particular, WEBGOP focuses on the tree relational structure to provide a way of representing the hierarchical nature of an application structure in a graphical form. Importantly, developers of WEBGOP can easily model the relationships between distributed nodes as family relations. A typical WEBGOP application is comprised of three parts.

- A graph specification that describes the graph structure of the nodes according to the spanning tree starting at the source.
- Direct annotations of graph primitives within each node that defines the executable graph statements.
- A runtime WEBGOP execution environment that maintains and manages the graph operations. The mapping and routing of messages are carried out automatically by the runtime based on the graph definition.

A. *Global Sum on a Three-Dimensional (3-D) Hyperpyramid*

This application computes the global sum on a 3-D hyperpyramid as illustrated in Fig. 15. Each node is a server and bears a value. The purpose of the application is to produce the sum of all values on the nodes. It provides a simple application example of distributed computing and, more importantly, it demonstrates the ease with which such an application can be constructed using the WEBGOP approach.

Before the main program is run, a graph named *local.sum* is defined and distributed among the servers.

The four nodes are defined as follows.

Node ID	Host name	Location	Local Program
0	host0	localhost : 8080	localprog.Subtotal
1	host1	localhost : 8001	localprog.Subtotal
2	host2	localhost : 8002	localprog.Subtotal
3	host3	localhost : 8003	localprog.Subtotal

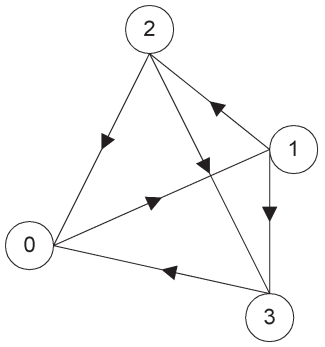

Fig. 15. 3-D hyperpyramid.

The six edges are defined as follows.

EdgeID	Start Node	End Node
0	0	1
1	1	2
2	1	3
3	2	3
4	2	0
5	3	0

The graph is established amongst the servers by calling a graph-related operation servlet, *GraphOperation*. The servlet uses the WEBGOP primitives, *defineGraph*, *modifyGraph*, *addHost*, and *deleteGraph* to perform graph-related functions. The graph information stored in a server is recalled by the WEBGOP primitive servlet, *webgop.server.MonitorServlet*.

The principal application program is *GlobalSum*, which is a servlet operating on Node 0. On every other node, an LP, *localprog.Subtotal*, is installed. The principal application program collects all of the subtotal sums of its children by calling the LP installed on them. It makes use of the spanning tree in WEBGOP to avoid duplication in the summing nodes. The WEBGOP primitive, *isSTLeaf()*, is called to check whether the node concerned is a leaf node on the spanning tree starting from Node 0; and the WEBGOP primitive, *call2TreeChildren()*, is used to get subtotals from the children of the spanning tree. The application is run by invoking the servlet, *GlobalSum*, as follows:

http://localhost:8080/[installedpath]/servlet/GlobalSum

With WEBGOP, the programmer is concerned with only the program logic while the system handles the low-level burdens of passing messages and forming the spanning tree. WEBGOP enables ease of programming to overcome this problem over the Web. Without GOP, it would be very difficult for nodes to know how to sum node values without engaging in double counting.

B. Stock-News Active Multicast

In this test application, a single piece of stock-news information is multicast to the following six servers, which provide different Web services. The first server is an HTML page server of Company A. It transforms the stock information into an HTML page bearing the company's title and stores it for distribution to desktop-based client browsers. The second server is a wireless markup language (WML) page server residing in the same company, which transforms the stock information into the company's customized WML page for distribution to wireless application protocol (WAP)-based users. The third server is an e-mail server, which generates e-mails from the stock information and sends them to users via SMTP. The services are repeated for Company B. The situation is depicted in Fig. 16.

The nine nodes are defined as that shown in the table at the bottom of the next page.

The eight edges are defined as follows.

Edge	Start Node	End Node
0	0	1
1	0	2
2	1	3
3	1	4
4	1	5
5	2	6
6	2	7
7	2	8

The application can be executed by calling the principal servlet, *Multicast*, on Node 0 as follows:

http://158.132.8.171:8080/[*installedpath*]/servlet/Multicast

The principal application servlet, *Multicast*, includes a graph-definition module and a stock-news message-sending module. It operates on Node 0. On other nodes, a different LP is installed based on individual requirements. The *localprog.HtmlFileA* transforms a message into an HTML page customized for Company A, while the *localprog.WmlFileA* transforms the message into a WML page customized for the company. The *local.EmailA* is responsible for sending e-mails to e-mail users of the company. The *localprog.HtmlFileB* transforms a message into an HTML page customized for Company B. Similarly, the *local.WmlFileB* transforms the message into a WML page customized for company B. All of these LPs have one thing in common. Their principal methods are all named *setData*, which will be remotely invoked.

The graph-definition module makes use of the WEBGOP primitive, *defineGraph*, to set up a virtual network amongst servers. After the graph is defined and set up, a stock message can be multicast to the servers using the WEBGOP primitive, *send2Descendents*. The methods with a common name, *setData*, are invoked on all the LPs as each distributed program processes autonomously. Each server will perform different functions according to different LPs. The *localprog.HtmlFileA* is responsible for generating an HTML page customized for Company A, the *localprog.WmlFileA* for generating a WML file for mobile users. The *localprog.EmailA* serves to send e-mails to the end users. Similarly, over at Company B, the corresponding LPs perform the same functions (as Company A) to serve different outputs. The messages are actively multicast across the WEBGOP-enabled servers residing in both companies, with the adaptation of contents carried out by the respective LPs serving different formats of the web contents.

This example demonstrates WEBGOP in enabling user-defined computations to be placed within the Web as with active

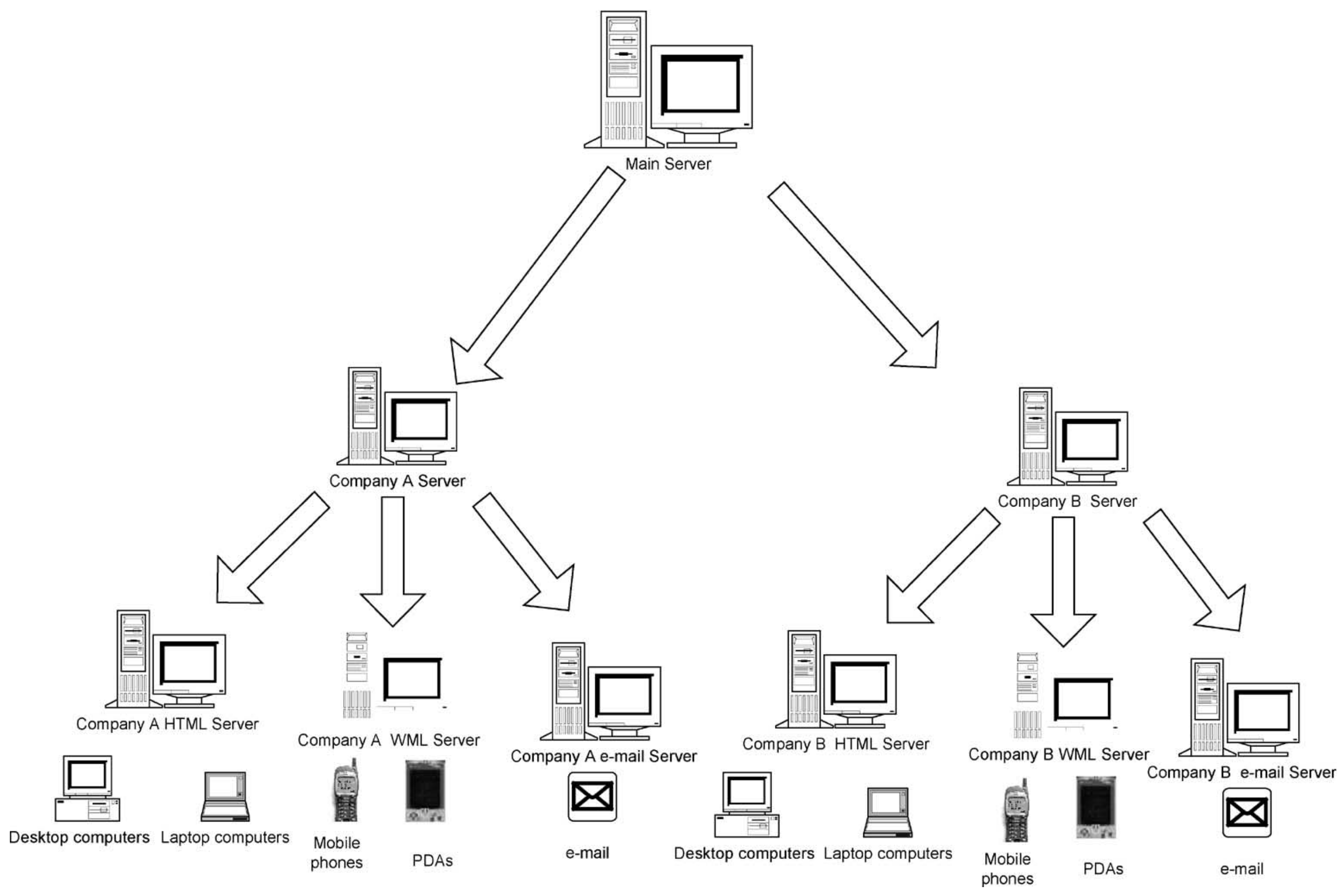

Fig. 16. Active multicast configuration.

networks, while retaining all of the routing and forwarding semantics of the current Internet architecture. Because the service does not require any change to the Internet architecture, it can be deployed incrementally on today's Internet. This provides an incremental approach to developing active networks using today's technology in the Web environment. In short, WEBGOP presents a novel computing model with the support of graph-oriented semantics in programming distributed web services.

C. Hierarchical Caching

In this test application, three servers form a hierarchical tree to provide a Web-caching service. An abstracted fragment of code is listed in the Appendix. When a server receives a request for a Web page, it searches its cache. If the search fails, it sends a message to its parents using the WEBGOP primitive, *call2TreeParents*. The parents repeat the same process with the grandparents and so on within the hierarchical tree. If the page is found, its location is returned to the requester and the servlet is redirected to the page. This program works even for interconnected servers forming closed rings. WEBGOP automatically generates a reverse spanning tree for searching and the programmer does not need to worry about the duplication or endless looping of searches.

In the test application, we make use of a single machine with different ports to simulate different servers. Therefore different LPs with different search paths have to be employed.

Node ID	Host Name	Location	Local Program
0	u5x − 171	158.132.8.171 : 8080	localprog.HtmlFile
1	u5x − 172	158.132.8.172 : 8080	localprog.HtmlFileA
2	u5x − 173	158.132.8.173 : 8080	localprog.HtmlFileB
3	u5x − 174	158.132.8.174 : 8080	localprog.HtmlFileA
4	u5x − 175	158.132.8.175 : 8080	localprog.WmlFileA
5	u5x − 176	158.132.8.176 : 8080	localprog.EmailA
6	u5x − 177	158.132.8.177 : 8080	localprog.HtmlFileB
7	u5x − 130	158.132.8.130 : 8080	localprog.WmlFileB
8	u5x − 131	158.132.8.131 : 8080	localprog.EmailB

The three nodes are defined as that shown in the table at the bottom of the page.

The two edges are defined as follows.

Edge	Start Node	End Node
0	0	1
1	1	2

The graph can be defined by calling the graph-definition servlet, *GraphOperation*, as follows:

http://localhost:8080/[*installedpath*]/servlet/GraphOperation

After a graph is defined, the application can be run by calling the principal servlet, *HierarchicalCache*, on Node 2, as follows:

http://localhost:8002/[*installedpath*]/servlet/HierarchicalCache

The concept of hierarchical caching has been around for some time but its implementation is complicated. A hierarchical-caching scheme reduces the latencies experienced by individual users and the aggregate bandwidth that is consumed. WEBGOP provides an easy means of programming for this type of application.

VII. Future Work

This project has demonstrated the benefits of employing a graph-oriented approach to the development of distributed Web services. Importantly, the strong semantics of graph grammar in modeling inherently graph-oriented services have proven to be very useful in developing distributed Web services. Despite the successful implementation of the prototype design, there are challenges and issues that remain to be addressed. It is the intention of this section to highlight these issues and to make recommendations for future work.

- Integration with WSDL—The current system allows one Web service for each graph on a server. WSDL is emerging as a potential new standard that provides universal naming. Future work may investigate the standardized naming of Web services using WSDL to differentiate between Web services/LPs for each graph.
- Incorporation of more types of data in the messaging—Today, our implementation supports the transfer of strings, integers, long, Booleans, and arrays. The transfer of more types of data such as graphic data should be investigated.
- Experimenting with other transport protocols—Currently, HTTP is used. Future work may explore the possibility of extending WEBGOP to support SMTP and FTP.
- Attachment—Under WEBGOP, every bit of information in a message has to be encoded and decoded. The use of attachments may reduce the workload involved in encoding and decoding the bulk of the information in a message.
- Standardization of graph identity—The graph identity must be unique in differentiating between different collaboration networks. A standard naming convention for graph identity would be useful.
- Automatic reconfiguration of the collaboration structure—The prototype implementation allows for programmer-defined insertions and deletions of nodes. In future work, the automatic reconfiguration of the collaboration structure may be investigated, taking into account network-traffic conditions, characteristics, and so forth.
- Although HTTPs may provide a certain level of security, security issues should be further investigated because the consequences of the improper activation of procedures on servers could be catastrophic.

VIII. Conclusion

In this paper, we have presented the design and implementation of WEBGOP, which represents a programming architecture for collaborative Web services using GOP. In this architecture, a logical graph representing a virtual-overlay network over the Internet is created to link up strategic hosts, which are preloaded with LPs providing collaborative Web services. Procedures in the LPs are invoked either through unicast or multicast messages within the overlay network. All messages are in the firewall-friendly SOAP format.

The complete WEBGOP framework has been implemented, tested, and evaluated on a Java execution platform. We conducted several experiments over a cluster of distributed workstations to validate the functionality of WEBGOP in providing collaborative graph-based coordination among participating web services. Overall, the time required to set up the initial graph session among web services incurred higher overheads than the graph operations. This is not surprising, since state initializations and host discovery and integration are required to configure the graph session. To demonstrate the versatility of WEBGOP in modeling collaborative Web services, this paper presented implementations of three trial applications using the prototype system: one for computing the global sum of the distributed nodes, one for providing active multicast services, and one for hierarchical caching. The ease with which these services are described using graph semantics within the application facilitates rapid development because the structural relationships among the services are directly embedded in the programs.

The architecture can be applied to distributed computing on the Web as well as to the coordination of Web services. The project provides a structured integration of Web services in

Node ID	Host Name	Location	Local Program
0	host0	localhost : 8080	localprog.SearchPage0
1	host1	localhost : 8001	localprog.SearchPage1
2	host2	localhost : 8002	localprog.SearchPage2

different servers that work collaboratively for a multipoint network application. It also provides a rich network-programming interface for a new class of integrated Web applications while retaining the simple IP and HTTP.

APPENDIX

```
/∗
 ∗
 ∗   File:   HierarchicalCache.java
 ∗   Creation date:   December 2001
 ∗
 ∗/
 import javax.servlet.∗;
 import javax.servlet.http.∗;
 import java.io.∗;
 import webgop.invocation.Service;
 import webgop.invocation.LPCall;
 import webgop.gop.Graph;
 import org.apache.soap.rpc.Parameter;
 import webgop.invocation.RPCResponse;
 import webgop.messaging.ResponseBuffer;
 import java.util.StringTokenizer;
 import java.util.Vector;

/∗∗
 ∗ HierarchicalCache is the servlet for getting cached pages
 ∗ from a graph-linked hierarchy of servers.
 ∗
 ∗
 ∗/
 public class HierarchicalCache extends HttpServlet
 {
    static Service service;

   /∗∗
    ∗ init method is called when servlet is first loaded.
    ∗/
    public void init() throws ServletException
    {
       if (service == null)
       {
          service = new Service();
       }
    }

   /∗∗
    ∗ doGet method is called in response to any GET request.
    ∗ Returns an HTML form that allows the user to choose to
      define or modify a graph.
    ∗
    ∗/
    public void doGet(HttpServletRequest request,
       HttpServletResponse response) throws IOException
    {
       //MIME type to return is HTML
       response.setContentType("text/html");

       //get a handle to the output stream
       PrintWriter out = response.getWriter();

       //create HTML form to allow user to select an operation
       out.println("<HTML>");
       out.println("<HEAD>");
       out.println("<TITLE>Hierarchical
          Caching</TITLE>");
       out.println("</HEAD>");
       out.println("<BODY>");
       out.println("<H2>Hierarchical Caching</H2>");
       out.println("<BR>
         <FORM METHOD=\"POST\">");
       out.println("<P>GraphID: <INPUT TYPE =
          \"TEXT\"NAME = \"GRAPHID\" "
           + "size = \"25\"></P>");
       out.println("<P>Request Page: <INPUT TYPE =
          \"TEXT\"NAME = \"PAGE\" "
           + "size = \"40\"></P>");
       out.println("<P><INPUT TYPE =
          \"SUBMIT\" NAME = \"Submit\" "
           + "VALUE=\"Submit\">");
       out.println("<INPUT TYPE = \"RESET\"
          NAME = \"Reset\" "
           + "VALUE = \"Reset\"></P>");
       out.println("</FORM>");
       out.println("<P>Note: search method of the local
          programs on parents will be invoked</P>");
       out.println("</BODY></HTML>");

       //close output stream
       out.close();
    }

    /∗∗
    ∗ doPost method is called in response to any POST
    ∗ request. This method determines if it was called in
    ∗ response to the user selecting to define the distribution
    ∗ graph, collect a message or send the message.
    ∗/
    public void doPost(HttpServletRequest request,
       HttpServletResponse response) throws IOException
    {
       String gid, page, location;

       //get info from HTML form submitted by user
       gid = request.getParameter("GRAPHID");
       page = request.getParameter("PAGE");
       try
       {
          location = getPage(gid, page);
          if (location!=null){
             response.sendRedirect(location);
          }
          else
          {
             //page not found
             PrintWriter out = response.getWriter();
```

```
            response.setContentType("text/html");
             //html output

            out = response.getWriter(); //get handle to output
                stream
            out.println("<HTML>");
            out.println("<HEAD><TITLE>Hierarchical
                Caching</TITLE></HEAD>");
            out.println("<BODY>");
            out.println("<P>Page not found in hierarchical
                cache</P>");
            out.println("</BODY>");
            out.println("</HTML>");
        }
    }
    catch (Exception e)
    {
        //exception occurred
        PrintWriter out = response.getWriter();
        response.setContentType("text/html"); //html output
        out = response.getWriter(); //get handle to output
            stream
        reportError(out,e);
        out.close();
    }
}

/**
* getPage identifies the location of cached page in parents.
*
* @param out Client output stream
* @param request HttpServletRequest object
*/
private String getPage(String gid, String page) throws
    Exception
{
    int source = service.getCurrentNode(gid);
    if (service.isRTRoot(gid, source) == true) return null;
    Vector lpParams = new Vector();
    lpParams.addElement (new Parameter("gid",
        String.class, gid, null));
    lpParams.addElement(new Parameter("source",
        int.class, new Integer(source), null));
    lpParams.addElement(new Parameter("requestPage",
        String.class, page, null));
    LPCall lpCall = new LPCall("search", lpParams);
    ResponseBuffer buffer[] = service.call2TreeParents
        (gid, source, lpCall);
    int cnt = buffer.length;
    for (int j = 0; j < cnt;j++)
    {
        RPCResponse res = buffer[j].get();
        if (res.generatedFault())
        {
            System.out.println("Fault "+ j +" = "
             +res.getFault()+"<BR>" );
        }
        else
        {
            if (res.getReturnValue()!=null)
            {
                String location = (String)res.
                    getReturnValue().getValue();
                return location;
            }
        }
    }
    return null;
}

/**
* reportError method returns stack trace to clienton error.
*
* @param out Printwriter output stream
* @param e Exception
*/
private void reportError (PrintWriter out, Exception e)
{
    StringWriter errorSW = new StringWriter();
    PrintWriter errorPW = new PrintWriter(errorSW);
    e.printStackTrace(errorPW);
    String stackTrace = errorSW.toString();
    out.println("<H1>WEBGOP Error</H1>
        <H4>Error</H4>" + e +
        "<H4>Stack Trace</H4>" +
              stackTrace + "<BR>");
}
}
```

ACKNOWLEDGMENT

The authors would also like to take this opportunity to thank the reviewers for their insightful comments and suggestions on improving the quality of the paper.

REFERENCES

[1] Apache XML Project. [Online]. Available: http://xml.apache.org
[2] A. Bakre and B. Badrinath, "Implementation and performance testing of indirect TCP," *IEEE Trans. Comput.*, vol. 46, no. 3, pp. 260–278, Mar. 1997.
[3] J. Cao, A. Chan, and K. Zhang, "Programming dynamic reconfigurable web server groups using the DyGOP model," in *Proc. 12th Int. Symp. Languages Intentional Programming*, Jun. 1999, pp. 65–77.
[4] J. Cao, E. Chan, C. H. Lee, and K. W. Yu, "A dynamic reconfiguration manager for graph-oriented distributed programs," in *Proc. Int. Conf. Parallel and Distributed Systems (ICPADS)*. Seoul, Korea: IEEE Comput. Soc. Press, Dec. 1997, pp. 216–221.
[5] J. Cao, L. Fernando, and K. Zhang, "Programming distributed systems based on graphs," in *Intentional Programming*, M. A. Oregun and E. A. Ashcroft, Eds. Singapore: World Scientific, 1996, pp. 83–95.
[6] A. Chan and J. Cao, "PANTA: A graph-oriented programmable active network transport architecture," in *Proc. IEEE Wireless Communications and Networking Conf.*, New Orleans, LA, Sep. 1999, vol. 3, pp. 1293–1297.
[7] Y. Chawathe, S. A. Fink, S. McCanne, and E. A. Brewer, "A proxy architecture for reliable multicast in heterogeneous environments," in *Proc. 6th ACM Int. Conf. Multimedia*, Bristol, U.K., 1998, pp. 151–159.
[8] Y. H. Chu, S. G. Rao, and H. Zhang, "A case for end system multicast (keynote address)," in *Proc. Int. Conf. Measurements and Modeling Computer Systems*, Santa Clara, CA, 2000, pp. 1–12.
[9] DeveloperWorks' Web Services Zone. [Online]. Available: http://www.ibm.com/developerworks/webservices

[10] P. Francis. (2000, Apr.). Yoid: Extending the Internet Multicast Architecture. [Online]. Available: http://www.aciri.org/yoid/docs/ycHtmlL/htmlRoot.html

[11] Frequently Asked Questions (FAQ) on the Multicast Backbone (MBONE). [Online]. Available: http://www.cs.columbia.edu/~hgs/internet/mbone-faq.html

[12] L. Kleinrock, "On some principles of nomadic computing and multi-access communications," *IEEE Commun. Mag.*, vol. 38, no. 7, pp. 46–50, Jul. 2000.

[13] S. McCanne, E. Brewer, R. Katz, E. Amir, Y. Chawathe, T. Hodes, K. Mayer-Patel, S. Raman, C. Romer, A. Schuett, A. Swan, T. L. Tung, T. Wong, and K. Wright, "MASH: Enabling scalable multipoint collaboration," *ACM Comput. Surv.*, vol. 31, no. 2es, Article 2, Jun. 1999.

[14] D. L. Tennenhouse *et al.*, "A survey of active network research," *IEEE Commun. Mag.*, vol. 35, no. 1, pp. 80–86, Jan. 1997.

[15] Universal Description, Discovery, and Integration of Business for the Web (UDDI) 2.0. [Online]. Available: http://www.uddi.org

[16] W3C Simple Object Access Protocol (SOAP) Version 1.1. [Online]. Available: http://www.w3.org.TR/SOAP

[17] Web Services Flow Language (WSFL 1.0). (2001, May). Edited by Prof. Dr. Frank Leymann (Distinguished Engineer; Member IBM Academy of Technology, IBM Software Group).

Alvin T. S. Chan (M'92) graduated from the University of New South Wales with the Ph.D. degree in 1995.

He was employed, after graduation, as a Research Scientist by the CSIRO, Australia, and is currently an Associate Professor at the Hong Kong Polytechnic University, Hong Hum, Hong Kong. From 1997 to 1998, he was employed by the Center for Wireless Communications, National University of Singapore, Singapore, as a Program Manager. He is one of the founding members of a university spinoff company, Information Access Technology Ltd. He is an active consultant and has been providing consultancy services to both local and overseas companies. His research interests include mobile computing, context-aware computing, and middleware for adaptive computing.

Dr. Chan is a member of ACM.

Jiannong Cao (M'93) received the B.Sc. degree in computer science from Nanjing University, Nanjing, China, in 1982, and the M.Sc. and Ph.D. degrees in computer science from Washington State University, Pullman, WA, in 1986 and 1990, respectively.

He is currently an Associate Professor in the Department of Computing at Hong Kong Polytechnic University, Hung Hom, Hong Kong. He is also the Director of the Internet and Mobile Computing Lab in the department. He was on the faculty of computer science at James Cook University and University of Adelaide in Australia, and at City University of Hong Kong, Hong Kong. His research interests include parallel and distributed computing, networking, mobile computing, fault tolerance, and distributed software architecture and tools. He has published over 150 technical papers in the above areas. He has served as a member of editorial boards of several international journals, a reviewer for international journals/conference proceedings, and also as an organizing/program committee member for many international conferences.

Dr. Cao is a member of ACM. He is also a member of the Computer Architecture Professional Committee of the China Computer Federation.

C. K. Chan, photograph and biography not available at the time of publication.