PROGRAMMING DYNAMICALLY RECONFIGURABLE WEB SERVER GROUPS USING THE DYGOP MODEL

JIANNONG CAO*, ALVIN CHAN
Department of Computing, Hong Kong Polytechnic University
Hung Hom, Kowloon
Hong Kong
{csjcao | cstschan}@comp.polyu.edu.hk

KANG ZHANG
Department of Computing, Macquarie University
Sydney, NSW 2109
Australia
kang@mpce.mq.edu.au

This paper describes the issues in providing support for programming dynamically reconfigurable Web server groups. A graph-oriented model, DyGOP, is presented which provides an integrated framework for dynamic configuration specification, programming, and management. Various requirements of dynamic reconfiguration are discussed and how the DyGOP model accommodates these requirements are illustrated. We also report the implementation status of a prototype of the DyGOP framework in programming applications of dynamic reconfigurable Web server systems.

1 Introduction

It has been widely accepted that electronic commerce over the Internet should be facilitated on a global basis. More and more organizations and enterprises build Web servers expanding multiple sites over the world. We consider a hierarchical group of WWW servers, which cooperate to provide some electronic commerce services on the Internet, which is often quite natural, mimicking the physical structure of the organizational administration. A typical scenario is the group of servers of a large organization, which is organized into a logical tree structure, where the root of the tree is the server for the headquarter office, and the other servers provide services pertinent to the regional, country, and city branch offices of

* This research is partially supported by the University Grant Council of Hong Kong under the CERC Project 9040229 and by Hong Kong Polytechnic University Research Grant PA24
the organization. Figure 1 illustrates some examples of such Web server groups. Organizing a group of Web servers into a hierarchical structure provides an effective approach to implementing high-performance Web services so that the specified degree of reliability, availability, scalability, and efficiency can be achieved [2]. For example, it can be effective in reducing the scope and numbers of broadcasting of synchronization and system state information.

This paper is concerned with the problem of how to program dynamically reconfigurable hierarchical Web server groups. The members in a Web server group perform autonomous tasks and also cooperate with some other group members. The cooperation is necessary because the servers in the group may contain overlapping information, cache and/or replicate the data provided at other server sites, and work together to balance the overall system workload. The members need to coordinate with each other so that the global system state consistency can be maintained and high performance can be achieved. Therefore, in programming the Web server group, the group member servers are configured into a hierarchy, which represents the relationship and interactions among the group member servers. The configuration of a Web server group is dynamic in nature. Servers can be created and added into the group at any time; existing servers in a group may be broken down and thus leave the group or is replaced by another server; servers in a group may form subgroup; servers may move from one subgroup to another. Thus we need support for programming and managing dynamic changes to the group configuration.

In general, the configuration of distributed system is concerned with structuring the program components and their interactions [14]. It is performed by instantiating software components, binding them together, and allocating them to a hardware configuration. Reconfiguration is the process of changing the current configuration to a new one, which may involve changing the set up of components or their allowed interactions. Dynamic reconfiguration permits the development of distributed programs whose configurations change as execution proceeds, while at the same time, maintains the correctness of the operations [9, 11, 13, 16]. Programming and managing dynamic configuration changes are the two essential aspects of dynamic configuration programming [12], which allows the configuration of a system to be specified and changed while the system is in execution. Since the program modules for the group member servers need to contain code for sending messages to other members and for synchronizing their activities, if we use the conventional message-passing model for programming Inter-Process Communications (IPC), dynamically changing the Web configuration while maintaining the group operations would be impossible. In this paper, we propose a novel framework for programming and managing dynamic configuration of Web server groups, based on the graph-oriented programming (GOP) model [3, 4].

The GOP model aims at providing high-level, structured abstractions for building distributed programs. In GOP, user-specified logical graphs are used as a
control structure for implementing structured communication and synchronization of distributed cooperative processes. The GOP model consists of a language-level logical graph construct and a collection of software facilities that extends the capability of existing programming languages and facilities by directly supporting distributed graph operations. The programmer can write distributed programs using primitives in terms of the graph construct, in a way very much similar to that of writing sequential programs. GOP is developed as a coordination construct that can be integrated into existing programming languages (e.g., C and Java) and systems (e.g., PVM and CORBA [7, 5]) and therefore is widely applicable. However, the original GOP model has no provision for including system evolution and reconfiguration as a part of the software development process. This paper shows that the GOP model can be extended to accommodate the requirements of supporting dynamically configurable distributed programs. The improved model is called DyGOP. We discuss how DyGOP can be used to programming dynamically configurable Web server groups by developing graph-oriented primitives and operations specific to hierarchically-structured Web server groups.

![Diagram](image)

Figure 1: Hierarchical Web server groups. (a) a hierarchical group of headquarter, regional, and local Web servers. (b) a tree structure of replicated Web servers [2].

2 GOP: A Graph-oriented Model for Distributed Programming

We have developed an integrated approach, called GOP, for high-level programming of distributed systems based on programmer-specified logical graphs. GOP provides a set of language-level constructs that extend the capability of existing programming languages and systems by directly supporting distributed graph operations.
With GOP, a distributed program is built using a logical graph as the underlying mechanism for defining and programming the communication and synchronization among the program components. The components are local programs (LPs) that may execute on several processors. The three elements of the language-level graph construct are (a) A conceptual graph whose vertices are associated with the LPs of the distributed program and whose edges represent inter-relationships among LPs; (b) A LPs-to-vertices mapping which allows the programmer to assign LPs to the vertices; (c) An optional vertices-to-processors mapping which allows the programmer to explicitly specify the mapping of the logical graph to an underlying network of processors. Figure 2 depicts the three components. Programming based on the graph construct consists of creating instances of the DIG construct and writing codes for LPs using the graph primitives provided by the system. We have defined a rich set of primitives for various operations on the graphs [3], including communication among the vertices of a graph, subgraph generation, graph update, and query. These operations provide the necessary facilities for graph-oriented distributed programming.

GOP helps programmers in providing high-level abstractions in programming distributed systems. Many distributed algorithms and programs are modeled, specified, and structured using logical graphs. Using GOP, since programmers can manipulate the distributed system based on the logical graph, they are relieved from the burden of coding low-level system functions. Instead, they can concentrate on the logic of their distributed programs. Furthermore, programmers are given enough flexibility to exploit the semantics of the graph construct to deal with different aspects of distributed programming in an integrated way. For example, multiple graphs of different kinds may be created for different purposes at different times; mappings can be changed during the program execution; LPs can be bound to multiple logical graphs. In [3], we described a graph instance by a 3-tuple (Functionality, Time, Version), which defines the contexts space of a graph-based distributed program. A program can dynamically switch to different contexts. In different contexts, a graph can carry different meanings and graph operations can be used to implement different functions. In this sense, in terms of intensional programming [15], graphs are intentions and graph operations depend on the values of graphs in different program contexts. The intensional approach to programming may provide us valuable techniques to define the semantics of graph operations so that they can be implemented in different ways but provide the same interface to programmers [6].

3 Graph-oriented Programming of Dynamic Reconfigurable Web Server Groups

In this section, we describe an approach to programming dynamically reconfigurable Web server groups, based on DyGOP, which is an extension to the GOP model.
DyGOP provides an integrated framework that facilitates dynamic reconfiguration of distributed systems.

Program Level:
LP types

Logical Graph Level:
Instances of LP types mapped to logical graph vertices

Physical Network Level:
Graph vertices mapped to network nodes

Figure 2: Mappings between levels in GOP

DyGOP extends GOP for dynamic reconfiguration in a natural way in the sense that the logical graph is used as a means of configuration specification and the graph construct is made a dynamic one. With DyGOP, nodes can be removed from the graph, new vertexes can be added to the graph, LPs can be unbound from or rebound to a node. We also want to make sure that such a dynamic change of the configuration during the system execution would not affect the whole system operation. For example, when adding a new Web server into a group, we need to add a new vertex to the corresponding configuration graph and then bind the server module LP to the new vertex. The operations of other vertexes in the graph would not be disturbed if their code were written completely using the graph-oriented primitives. The correctness of the overall group process would not be affected neither if the underlying run-time system can ensure the consistency of such a configuration change.

Dynamic reconfiguration adds another dimension to the context space of GOP in terms of intensional programming [15]. In terms of intensional programming, the reconfiguration-programming primitives, as well as those defined for communication and synchronization, can be thought of as a set of intensional operators. Their operations are expressed in terms of the logical graphs but their semantics are dependent on the current Web server group configuration [15].
3.1 Dynamic Web Group Configuration Specification and Programming

First of all, in DyGOP, rather than developing a new, separate configuration language as in the existing approaches [9, 12, 13, 16], we use the logic graph construct as an integrating mechanism for configuration specification and component programming. In this way, we can incorporate the configuration specification into the LP program design while at the same time eliminate the dependency of the program computation code on the program configuration information. It is believed that it is important to provide a notation and technique in which the two aspects of dynamic configuration specification and non-reconfiguration functionality can be described in a single formalism, while keeping them as separate "views" [1].

Initially, when we configure a group of Web servers, we create a logical graph which, in the context of this paper, will be a hierarchical graph where vertices are organized into levels. The created logical graph instance describes the relationships between the member servers in the group. It consists of a name, a directed logical graph, an LP-to-vertex mapping, and an optional vertex-to-processor mapping. An example is a tree in Figure 1.b, which can be specified as follows:

\[
\text{Dgraph Tree} = \{(0..6), \{0,1\}, \{0,2\}, \{1,3\}, \{1,4\}, \{2,5\}, \{2,6\}\};
\]

The declaration associates the Dgraph instance with a name Tree. The first component on the right hand side of the declaration specifies the vertices of the tree. The second component consists of a set of edge descriptions, each defines an edge between two vertices in adjacent levels in the tree.

Then we can write LPs, which define the functionality of each Web server, using the graph-oriented primitives for inter-LP communications and synchronization. For example, we define three LP types, namely PrimaryServer, ReplicatedServer1, ReplicatedServer2. Then instances of these two LP types can be mapped to the vertices of Tree.

\[
\text{LV-MAP M1} = \{(0, \text{"PrimaryServer"}), (1, \text{"ReplicatedServer1"}), (2, \text{"ReplicatedServer1"}), (3, \text{"ReplicatedServer2"}), (4, \text{"ReplicatedServer2"}), (5, \text{"ReplicatedServer2"}), (6, \text{"ReplicatedServer2"})\};
\]

The three different LPs implement different caching and replication functions. They need to interact with other LPs to maintain state consistency and to achieve high performance. In the DyGOP model, communications and synchronization between Web servers in a group are programmed in a structured interaction approach, i.e., in terms of the underlying logical graph rather than the identifiers of individual servers. The communication and synchronization primitives can be either vertex-oriented or edge-oriented.
Vertex-oriented primitives are \textit{Send\_to\_Children()}, \textit{Send\_to\_Parent()}, \textit{Send\_to\_Neighbors()}, \textit{Receive\_from\_Parent()}, \textit{Receive\_from\_Children()}, and \textit{Receive\_from\_Neighbors()}.

Edge-oriented primitives are \textit{Send\_on\_Edge()}, \textit{Send\_on\_AllEdges()}, \textit{Receive\_from\_Edge()}, \textit{Receive\_from\_AllEdges()}. Edge-oriented primitives can support more flexible and more general reconfiguration requirements, especially when different functions are associated with adjacent vertices. Sending messages to these vertices using edges is more natural and convenient in programming.

As we can see, these graph-oriented primitives and operations are defined purely in terms of the underlying logical graph construct so that programming the functionality of the Web server modules requires no knowledge of and thus does not depend on the current group configuration information. This property of context independency gives flexibility of the system components being manipulated at runtime, minimizing the inter-dependency on configuration. Therefore, in case that a configuration change occurs, execution of the Web servers needs not be stopped; rather, only the separate configuration information needs to be updated.

Furthermore, when writing the code for the communication and synchronization among the Web servers, DyGOP allows us to refer to only the local names and objects in each Web server but not to shared names and/or objects. This naming in terms of the vertices and/or edges of the logical graph allows the de-coupling of IPC interface (each provided at a particular vertex) from the actual Web server module, which responds to the messages. Each server internally refers to the communication peers only by purely local names, which will not change for different configurations of the group. The binding of the graph vertices to the server members in the group describes the substitution of externally defined peers for locally specified ones. This property is desirable for dynamic configuration, because direct naming or referencing shared objects would limit the flexibility of adding and removing the Web servers into the group and prevent them from being reallocated to different groups. Any names of components and communication entities should not be embedded in the context of a Web server module but left to the configuration specification, which describes the logical structure of a group. Otherwise, when reconfiguration is needed, programmer must stop the whole group and then re-edit and re-compile the Web server modules; this makes dynamic reconfiguration impossible.

With the DyGOP reconfiguration model, the programmer should be able to specify both static and dynamic configurations of Web server groups and to program both planned and un-planned group reconfigurations in a systematic way. We have seen that initial group configuration specification can be done by declaration of a logical graph construct. We add more programming primitives for supporting dynamic reconfiguration specification and programming.

Both declarative and constructive primitives can be provided for specifying dynamic changes to the logical graph and the mapping between the LPs and the
graph. A declarative specification describes what the structure is, not how it is constructed. It is mainly used for planned reconfiguration. Constructive specification, on the other hand, can be used for both planned and unpredictable changes. Planned reconfiguration is also called operational, i.e., the requirement to change, such as dynamic expansion of worker processes in a master-slave paradigm, can be programmed into the distributed program since the requirement is present at the time the program is designed. On the other hand, evolutionary changes, such as adding new Web servers into a group, expanding and upgrading server functionality, and handling of faults, by their nature cannot be predicted by the designer of the program. Configuration changes of Web server groups are usually evolutionary and unpredictable, which can only be accommodated by structural modification to the group configuration – e.g., by the replacement, addition and reconnection of components and by binding them to the vertices of the logical graph. Primitives for structured changes, such as `AddVertex()`, `DelVertex()`, `AddEdge()`, `DelEdge()`, `MapLPtoVertex()`, `UnMapLPtoVertex()`, etc., have been developed. For instance, in our example, if we want to remove the server associated with vertex 4, the primitives `DelVertex(4, Tree)` and `DelEdge(1,4, Tree)` can be invoked, which will lead to the termination of the LPs mapped to the vertices.

With DyGOP, deletion of vertices and edges would not affect existing vertices’ operations. When a vertex wants to interact with a vertex that has been deleted, the underlying run-time system will automatically detect the absence of the deleted node and return an error message. As long as the LPs are programmed with exception handling code, the existing vertices can continue their executions even though some of their partners have quit.

It is also possible to add a new vertex into a graph without affecting the operations of existing vertices. For example, the new vertex can be added to the graph and then edges can be added between the new vertex and some other existing vertices. After this, whenever the existing sends a message to its neighbors or children, the new vertex will also receive that message.

LPs can be dynamically mapped to and un-mapped from graph vertices. This facilitates the modification of the programs executed at the vertices without changing the system configuration. Other parts of the Web server group may not need to know such changes at all. Together with the dynamic addition and deletion of nodes and edges, DyGOP allows us to dynamically change the interfaces of the LPs, which usually remain fixed in existing dynamic reconfiguration systems.

DyGOP is powerful enough to specify more dynamic reconfiguration requirements. For example, it is possible to merge two graphs into a single graph and it is also possible to bind a vertex to more than one graph.

Dynamic reconfiguration can be initiated by either an independent configuration process or by operations embedded in Web server LP code. In the former case, the configuration process can be bound to the Web server group at any time and initiate the reconfiguration procedure. In the latter case, dynamic reconfiguration of a Web
server group can be initiated by any member in the group, although in most cases they are triggered by the master Web server. The master Web server can be designed to monitor the execution of the whole group and the underlying system and detect the conditions for initiating dynamic configuration. Invocation of the reconfiguration primitives will result in an exception to the run-time system or be handled by part of the distributed program.

3.2 Validation and Coordination of Dynamic Web Server Group Configuration Changes

Since reconfigurations in a Web server group are mostly evolutionary changes [14], the sequence of change operations may be applied over time to a group. Accordingly, the state of the configuration has to be considered and modeled. The operations must be specified in the context of the configuration states, which can be maintained in a configuration database by the underlying run-time support system. Upon reconfiguration, validation of the specification of the new configuration needs to be performed to determine whether it describes a valid configuration change and, if it is the case, the dynamic changes should be carried out so as to maintain correctness and consistency of the Web group operations.

In the DyGOP approach, the transformation of graphs, which represents dynamic changes to the Web server group configurations, can be formally modeled and controlled with well-defined graph operations. Such operations include validation function primitives, such as IsGraphExist(), IsVertexExist(), IsEdgeExist(), etc., and may concern the query of graph properties, merging of graphs, and comparison of graphs.

![Diagram](image)

**Figure 3: The process of dynamic reconfiguration**

Much of existing research in dynamic reconfiguration has concentrated on providing tool and language support to help the programmer express and execute the desired changes. In most cases, the running assumption is that the system is already in the safe state for reconfiguration by the time the change actions are carried out, or the system will be able to recover from any inconsistencies the change actions may introduce. Recently, several algorithms have been proposed to ensuring component consistency in the presence of dynamic reconfiguration [8, 10]. These algorithms
can be adapted to our system. Furthermore, since reconfiguration transitions move between configurations, additional synchronization constraints on these transitions may need to be specified to prevent interference between reconfiguration transitions (see Figure 3). We are investigating various schemes for coordinating the reconfiguration process to achieve such consistent reconfiguration transitions. The reconfiguration process should be executed in an atomic manner and thus synchronization is needed among the group members.

4 Implementation Status

Based on a prototype of GOP [3], we have developed and implemented dynamic reconfiguration primitives described in the last section and a framework for dynamic reconfiguration management [4]. Figure 4 shows the architecture of the management framework. The shaded area shows the data flow in dynamically reconfiguring a distributed system. Configuration change requests are sent to the validation module which checks the validity of the requests. Validation module needs to enquire the configuration information maintained in the configuration database. If the sequence of requests is valid, reconfiguration module will update the corresponding graph construct and the configuration specification. Graph construct and configuration specification on the left-hand side represents current system status; after the change operations, the system status is updated and is represented on the right-hand side.

A prototype of the DyGOP framework has been developed, based on a central-server based implementation (see Figure 5). The central server maintains the configuration specification of the graph instances. The validation and processing of a reconfiguration plan is in a client-server manner, where the central-server receives the requests from the program module initiating the reconfiguration.

Currently, we are developing applications of dynamically reconfigurable Web server groups using the prototype. Examples include a group of replicated Web servers cooperating for providing certain information service.

5 Conclusions

The main contribution of this paper is the design and initial implementation of a graph-oriented framework to support dynamic configuration programming of Web services. Dynamic configuration is required for many purposes including dynamic creation and replacement of Web server modules, system evolution and upgrade with new functionality and new technology, and fault tolerance. We have discussed the dynamic configuration programming issues, presented the DyGOP dynamic configuration model, and described the design for dynamically reconfigurable Web server groups. We have also reported the status of the development of a prototypical implementation of the proposed framework.
Figure 4. The dynamic reconfiguration framework

Figure 5. The central-server based prototype
References
