Real time dependable communication infrastructure for a collaborative groupware system

RADU DOBRESCU, DAN POPESCU, MAXIMILIAN NICOLAE, HORIA HUMAILA

"Politehnica" University of Bucharest, Faculty of Control and Computers, 313 Spl. Independentei, Bucharest, ROMANIA
radud@isis.pub.ro

Abstract. This paper addresses the problem of providing customized real-time dependable communication services for distributed and delay-sensitive applications. The proposed architecture is that of a real-time groupware system which enables meeting of dependability requirements through replication. The communication infrastructure is implemented with real-time dependable channels. A dependable communication protocol provides both stable throughput and fast delivery of multicast messages even in the presence of frequent node and link failures. The advantages of the proposed solution are sustained by test results of a parallel processing image application.

Keywords: collaborative group, object group, dependable real time channels, multicast messages, ordering protocols

1 Introduction

Research reported in this paper proposes a distributed processing framework for complex software development that can be trustworthily dependable, by offering a synergy of different dependability mechanisms and techniques process-oriented designed, for both processing architecture and communication infrastructure.

We define dependability according to Avizienis, et al. [1]. Dependability is the ability to deliver service that can justifiably be trusted. It is the system property that integrates several vital dependability attributes: availability (readiness for correct service), reliability (readiness for correct service), safety (absence of catastrophic consequences on the environment), integrity (absence of improper system alterations) and maintainability (ability to undergo modifications and repairs). Focus of the reported research encompasses software and hardware development issues of above cited branches of dependability and deals with unintended violations of a system trustworthiness, not with malicious attacks. It focuses on rectifying insufficiencies in development of network services on a platform for distributed processing.

Distributed services have to be dependable in order to perform increasingly critical functions such that errors or delays may not compromise human safety or result in economic loss. They need to perform actions or furnish results that are correct and remain available despite the unreliability of their environment. Replication is the principal technique for rendering services dependable. Most of new proposed solutions for distributed application development are based on the object group paradigm well suited for managing replication on Java platforms ([2], [3], [4], [5]).

In this paper we present an alternate solution a higher-level Computer Supported Collaborative Work (CSCW) system. It has its own, particular requirements due to its “real-time groupware” system characteristics and will be defined as Collaborative Group (CG). The framework is an interactive Distributed Virtual Environment (DVE) characterized as “real-time” groupware system due to special and stronger requirements to allow user interaction “responsiveness.” In the same time we argue why the middleware CSCW system is an effective basis for developing dependable distributed services. In this aim we decided to use for the communication infrastructure real-time...
dependable (RTD) channels. RTD channels are implemented as middleware mechanisms between the operating system and application, thereby providing a virtual machine with enhanced QoS guarantees on which applications can be built.

2 Dependability potentials of the system architecture

2.1. Design of the CSCW architecture

At the design stage, there are two possible choices for the location of the shared applications. The first one is a centralized configuration with a single copy of the application running on a server. All the events produced by the users are routed to the server and the output is broadcast to all sites. The state of the application is easily kept consistent during the session, because all the inputs produced by the users are serialized. The state changes affect only one copy. The major disadvantage of this scheme is high bandwidth consumption and unexpected delays resulting from information travelling to the central site and back again to each site with the results. Another approach is the use of replicated configurations. In this case, each site of the working session has one local copy of the application running. All the events produced by the users are processed locally and distributed to the other sites, except for those considered local. By this approach, only commands travel across the network therefore the response time is better, and the required bandwidth is small, compared with the previous approach.

As a solution with better dependability, we have implemented on our distributed platform dedicated multipoint communication mechanisms. These mechanisms differ from the point-to-point paradigm due to the number of sources and receivers involved in each transaction. In addition to the traditional reliable communication mechanism, the system should provide the session control within multicast transactions. Multicast communication involves one source and several receivers or several sources and several receivers. Broadcast 1:n and point-to-point 1:1 are particular cases of multicast transactions.

2.2. Collaborative Group Distributed Object Model

The context of our work is a distributed system composed of client and server objects interconnected through a communication network. The system is asynchronous in the sense that neither the computational speeds of objects nor communication delays can be bounded. Furthermore, the system is unreliable and failures may cause objects and communication channels to crash whereby they simply stop functioning. Once failures are repaired, they may return to being operational after an appropriate recovery action. Finally, the system is partitionable in that certain communication failure scenarios may disrupt communication between multiple sets of objects forming partitions. Objects within a given partition can communicate among themselves, but cannot communicate with objects outside the partition. When communication between partitions is re-established, we say that they merge. Developing dependable applications to be deployed in such a CG system is a complex task due to the uncertainty resulting from asynchrony and failures. The CG middleware system has been designed to simplify partition-aware application development by transforming complex system properties such as failures, recoveries, partitions, merges and asynchrony into simpler, high-level abstractions with well-defined semantics. CG promotes dependable application development through replication implemented as object groups [6]. Distributed services that are to be made dependable are replicated among a collection of server objects that implement the same set of remote interfaces and form a group in order to coordinate their activities and appear to clients as a single server. Client objects access a distributed service by interacting with the group identified through the name of the service. In particular, clients are unaware of the number, location or identity of individual servers in the group. Communication between clients and groups takes the form of group method invocations, which result in methods being executed by one or more servers forming the group, depending on the invocation semantics.

2.3. Collaborative Group Method Invocation Service

CG differs from existing object group systems due to its uniform communication interface based entirely on group method invocations. Clients and servers alike interact with groups by remotely invoking methods on them. In this manner, benefits of object-orientation are extended to internal communication among servers. Although they share the same intercommunication paradigm, we distinguish between internal group method invocations (IGMI) performed by servers and external group method invocations (EGMI)
performed by clients. There are several reasons for this distinction: visibility (methods to be used for implementing a replicated service should not be visible to clients), transparency (clients are not required to be aware that they are invoking a method on a group of servers rather than a single one) and efficiency (in CG, external group method invocations have semantics that are slightly weaker than those for internal group method invocations [7]).

When developing dependable distributed services, internal methods are collected to form the internal remote interface of the server object, while external methods are collected to form its external remote interface. The next step in server object development is the creation of appropriate proxy classes for invoking external and internal group methods. Proxies for external group method invocations are called stubs (as in standard Java RMI [8]), while internal group method invocations are delegated to the group managers that handle all communication between server objects through group method invocations. Stubs implement the same external interface for which they act as a proxy, while group managers implement a multi-response interface derived from the internal remote interface. Multi-response interfaces differ from the corresponding internal remote interface due to the fact that internal invocations may return an array of results, rather than single values.

3 Dependability potentials of the communication infrastructure

3.1. RTD Channels
A channel is an abstraction for communicating between two or more application-level processes in a distributed system. A dependable channel provides guarantees related to the reliability of message transmission, while a real-time channel provides timeliness guarantees. A real-time dependable (RTD) channel has a combination of dependability and timeliness guarantees. The RTD channel service offers a simple Application Programming Interface (API) consisting of operations to open a channel, push a message into a channel, and close a channel. Messages are delivered to the receiver using an upcall to a specified function. The API is the same regardless of the chosen channel properties.

A large number of properties can be defined for channels, but for brevity, we describe only a representative set here. In particular, we consider real-time – i.e., whether each message sent on a channel will be delivered to its destinations by a deadline, reliability – i.e., whether each message reaches its destinations, message ordering – i.e., in what order messages are delivered to the application, and jitter – i.e., the variance in message transmission time.

Real-time properties. The real-time behavior of a channel is specified by a deadline probability \( p_d \) (the required probability that a message sent on the channel reaches its destination by a specified deadline). Various techniques such as admission control, scheduling, congestion control and retransmissions are used to provide the desired \( p_d \).

Reliability properties. The reliability of a channel is specified by a reliability probability \( p_r \), defined as the required probability that a message sent on a channel eventually reaches its destination. Two different types of reliability guarantees can be identified: bounded reliability (based on a fixed maximum number of retransmissions) and absolute reliability (a message is retransmitted until it has been received).

Ordering properties. Message ordering properties define constraints on the relative order in which messages are delivered to the application on the receiving site. Message ordering properties are orthogonal to real-time and reliability.

Jitter control properties. Jitter control properties restrict the variance in message transmission time along a channel. Jitter can be reduced by delaying the delivery of early messages until the message deadline.

3.2. Implementing customizable RTD channels
Each RTD channel is implemented as a composite protocol consisting of selected micro-protocols and uses system resources allocated as paths. Resource allocation has been divided into the channel control module (CCM), which is specific to channels, and the admission control module (ACM), which manages resources across multiple types of services.

Basic micro-protocol. The core micro-protocol on which all others build is the Basic Channel, which implements the abstraction of a simple channel with no reliability, ordering or jitter guarantees. This functionality can be augmented using different retransmission, ordering, and jitter control micro-protocols. This micro-protocol implements the basic function of passing a real-time message through the channel composite protocol.
Ordering micro-protocols. Ordering micro-protocols impose ordering constraints on message delivery to the application. A multicast ordering scenario is defined as a view $V$. For any two messages $m_i$ and $m_j$, multicast in view $V$, the following condition holds:

**FIFO Multicast.** If there is a correct process $p$ which FIFO-multicasts message $m_i$ before $m_j$, then all correct processes $q$ deliver message $m_i$ before $m_j$.

**Weak Totally Ordered Multicast.** If there is a correct process $p$ which delivers $m_i$ and $m_j$ in the order $m_i$ before $m_j$, then every correct process delivers $m_i$ and $m_j$ and in the same order as $p$.

**Uniform Multicast.** If any process, whether correct or not, delivers message $m$, then every correct process delivers message $m$.

**Strong Totally Ordered Multicast.** If there is any process of view $V_i$ which delivers $m_i$ and $m_j$ in the order $m_i$ before $m_j$, then every correct process which delivers $m_i$ and $m_j$ in the order $m_i$ before $m_j$.

**Hybrid Total Order.** This order incorporates the strong and the weak totally ordered multicasts. For any two processes $p$ and $q$ (whether correct or not), where process $p$ has delivered messages $m_i$ and $m_j$ in the order $m_i$ before $m_j$, process $q$ has also delivered $m_i$ and $m_j$ in the order $m_i$ before $m_j$.

**Global Order Multicast.** For every two correct processes $q$ and $r$, which both deliver the global multicast $mG$, the sets of messages delivered before the delivery of $mG$ are identical and include message $m$.

4 Testing a dependable, ordered, multicast communications framework

For testing the dependability potentials of our solution, a simple Parallel Image Processing (PIP) application, that consist in still image segmentation, clustering, data block distribution, image processing on individual hosts and then processed blocks transfer and grouping in order to obtain a whole processed image was performed. A number of experiments have been performed to test both functional and performance aspects of the service. All tests were run on an experimental platform dedicated for parallel processing [9].

Our platform consists of a “head” machine and a number of six cluster nodes. Communication between cluster nodes is available through high-speed Gigabyte networking, offering a very efficient and cost-efficient solution. The “head” provides all services for the cluster nodes – IP allocation, booting services, File System (NFS) for storage of data, facilities for updating, managing and controlling the images used by the cluster nodes. Increased computing performance is provided by optimization for balanced and distributed processing using several workload distribution mechanisms. Cluster users can easily migrate processes from one node to another, and decision about which node to be used is made upon workload information from all nodes.

We have implemented and tested several communication primitives provided by the ordered multicast communication framework implemented by using the different dependability mechanisms and techniques described in the previous sections. Common to all implementations is the key idea to delay the delivery of a received message until some condition is satisfied, so the solution is for message buffering and delivery delaying. In the next examples describing the implementation of different ordered multicast primitives, the delivery of received messages is often dependent on the reception of other messages, view changes and/or local state information in the layer. Received messages, whose delivery has to be delayed are buffered in the layer until their delivery condition is satisfied. In the following the term received means received from the view synchronous communication layer by the ordered multicast communication layer and the term delivered means the delivery from the ordered multicast communication layer to the application layer.

**Uniform Multicast**

A process receiving a uniform multicast $m$ does not deliver it immediately, but buffers it until it receives information (in form of a message $delv(m)$) that all the processes have received $m$. Let consider a message $m$ multicast by $p$ to a group. Process $p$ multicasts the message $m$ using a view synchronous multicast, including additional information for the ordered multicast communication layer of the destinations, not to deliver the message immediately. The destination processes which receives the message, buffer it and send back an acknowledgement to $p$ which then multicasts a second message $delv(m)$ indicating that the buffered message $m$ can be delivered (fig.1).

![Fig.1. Implementation of the Uniform Multicast](image-url)
Weak Totally Ordered Multicast

Suppose a process $p$ issues a message $m$ using the \textit{wto-mcast} primitive. At the destination this multicast is recognized as a weak totally ordered multicast and buffered until the reception of a sequence number associated with the multicast. A sequencer process, for example the process with the smallest rank number, is responsible for generating a sequence number for each weak totally ordered multicast it receives. The condition to deliver a message multicast with the weak totally ordered multicast primitive \textit{wto-mcast} is the reception of the message and the reception of the sequence number from the sequencer process. Upon reception of the sequence number for message $m$, a process can deliver the message $m$, provided that it has already delivered all weak totally ordered messages preceding $m$. In case of a failure, an arbitrary but deterministic order on the remaining, undelivered messages is constructed and the messages are delivered in this order. This a priori order is based solely on data contained in the messages without any further message exchanges. It is based first on the rank number of the sender, and then on the sequence number included in the message identifier (fig. 2).

![Fig. 2. Implementation of the Weak Total Order](image)

Strong Totally Ordered Multicast

The key idea of the implementation is to add control information to strong totally ordered multicasts; this is used to define the total order. In contrast to the weak totally ordered multicast, this order is independent of the order in which the strong totally ordered multicasts are received. Each process keeps a round counter which is incremented each time it issues a strong totally ordered multicast, which is tagged with this round number. Upon reception of such a strong totally ordered multicast, the receiver recognizes the message as a strong totally ordered message, buffers it and issues an acknowledgement to the coordinator using the reliable send primitive of the reliable communication layer. A process orders the received messages in the following way: (1) two messages of different rounds are ordered by the round number in which they were issued; (2) two messages issued in the same round are ordered using the rank number of the sender of the message. Figure 3 illustrates the algorithm; the failure cases are discussed below.

![Fig. 3. Implementation of the Strong Total Order](image)

As the coordinator waits for the acknowledgements of all processes, before issuing the delivery messages, the uniform property of the message delivery is guaranteed. For each round, the coordinator receives from each process at least one acknowledgement per round, indicating whether the process has issued a concurrent multicast in that round or not. As a process can only send at most one strong totally ordered multicast or an acknowledgement for each message issued to a given round, the coordinator receives from each process either a message or an acknowledgement. Thus the order is defined as follows: a message from round $n$ is delivered before a message from round $n + 1$, and for messages issued in the same round $n$, they are ordered using the rank of the sender. In the case of a failure, the uniformity is guaranteed in the same way as in the uniform multicast, and the delivery order on messages which are not yet ordered can be determined locally by each process.

Implementation of Hybrid Order

Assume that the sequencer of the weak and the coordinator of the strong totally ordered multicasts are the same process. All totally ordered multicasts (strong and weak) are issued into rounds, i.e. weak totally ordered multicasts are tagged by the round counter for strong totally ordered multicasts, but this counter is only incremented when a strong totally ordered multicast is issued. Thus, for a given round there can be more than one weak totally ordered multicast, but only one strong totally ordered multicast per process (property of the implementation of the strong totally ordered multicast). Thus, the coordinator orders all the weak totally ordered multicasts of one single round in the order the coordinator receives them and before the strong totally ordered multicasts of the same round. This order is applied to the weak and strong totally ordered multicasts of all rounds. Each process includes the number of weak totally
ordered messages it has issued in a given round in the acknowledgement for the strong totally ordered message. Therefore, the coordinator knows exactly how many weak totally ordered messages have been issued in a round, and it can order the messages appropriately.

In the case of failure, the view synchronous communication layer guarantees that every correct process has delivered the same set of messages. This set of messages includes in particular the weak and strong totally ordered message as well as the delivery order messages from the coordinator. For messages to whom no delivery order message is received, all remaining total order messages are ordered in the ways described above, i.e. first by considering round numbers of all messages, then weak totally ordered multicasts before strong totally ordered multicasts and weak totally ordered multicast of the same round by using multicast sequence number and for strong totally ordered multicasts (one per round) the rank number of the sender.

5 Conclusion

The ability to make scheduling decisions at run-time is a key aspect of means that may be exploited to provide dependable real-time communication. This paper aims to put together several solutions to achieve efficient fault tolerance and support non-periodic data or messages with partially known timing requirements, attributes which are of increasing concern in distributed control systems. The proposed architecture is that of a real-time groupware system where services are implemented by groups of remote objects that cooperate in order to maintain the consistency of their state. The communication infrastructure is implemented with real time dependable channels that guarantee messages sent are eventually delivered at the destination without failures and unacceptable delays. Simulations and analysis of the experimental results obtained in a parallel image processing application executed on a cluster with six processing units show that high reliability can be obtained through the use of the designed dependable capabilities. The functional tests indicate that ordering is preserved as expected for FIFO, causal, and total ordering despite out-of-order and dropped messages.

The next step is to move the RTD channel service into the kernel, which we expect will have a number of advantages. First, the performance of channels should improve considerably. Second, and more importantly, predictability should improve greatly. This can be used to prevent other system activity from interfering with the execution of RTD channels.

ACKNOWLEDGEMENTS

This work was partially supported by the Romanian Ministry of Education and Research under Grant No. 81-060/2007.

References: