Building Distributed Applications Using the Reactive Approach

Changgui Chen
Wanlei Zhou

School of Computing & Mathematics
Deakin University
Clayton, Vic. 3168, Australia
Email: changgui, wanlei@deakin.edu.au

Abstract

Traditional programming approaches decompose a system and its structure into smaller or manageable (and usually passive) objects. It may cause some problems in developing modern systems because they may consist of active objects. The reactive system method views the whole system as a reactor and it consists of independent active objects or actors which may reflect real objects from the world. The paper advocates an approach for building distributed applications based on the reactive model. Three reactive modules: DMM, sensor and actuator are implemented and evaluated. Two examples presented at the paper show the potential use of our reactive method. Compared with previous programming approaches, the reactive system method is more close to reality and has richer semantics.

Keywords

Reactive systems, Java applications, distributed applications, fault-tolerant computing, software development.

INTRODUCTION

Reactive systems have been defined as systems that maintain ongoing interactions with their environments, rather than producing some final results on terminations (Gerth 1997, Harel 1985). They cannot be described only as computing a function from an initial state to a terminal state. Such systems are quite different from transformational ones, which accept inputs, perform transformations on them and produce results. An adequate description of reactive systems must refer to their ongoing behaviors, which are seen as reactions to external stimuli. Typical examples of reactive systems are flight reservation systems, industrial plant controllers, operating systems, most kinds of real-time computing embedded systems, and communication systems, etc.. The more investigations of reactive systems can be found in Caspi 1994, Harel 1998 and J cIUvinen et al. 1990.

The reactive system approach views a system as a reactor that continuously interacts with its environment by receiving and sending messages, which is the nature of reactive systems. Usually, a reactive system uses sensors and actuators to implement the mechanisms that interact with its environment or applications. Its system controls or decision making managers (DMMs) are used to implement the policies regarding to the control of the applications (Boasson 1993). Figure 1 depicts the architecture of the reactive system model. It has three levels: policies, mechanisms and applications.

The major advantage of the reactive system model is the separation of policies and mechanisms, i.e., if a policy is changed it may have no impact on related mechanisms and vice

versa. For example, if a decision making condition based on two sensors was "AND" and now is changed to "OR", the sensors can still be used without any changes required, i.e., the mechanism level can remain unchanged. This advantage will lead to a better software architecture and has a great significance in developing distributed and fault-tolerant computing applications (Edwards et al. 1997, Zhou 1999). General speaking, the development of distributed and fault-tolerant computing systems is a very difficult task. One of the reasons is that, in normal practice, most fault-tolerant computing policies are deeply embedded into application programs, therefore these applications can not cope with changes in environments, policies and mechanisms. To build distributed systems that can cope with constant changes in environments and user requirements, it is essential to separate fault-tolerant computing policies from application programs. Hence, we can apply the reactive system model to develop better distributed and fault-tolerant applications.

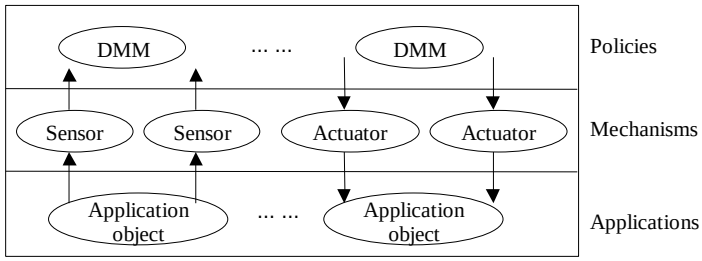

Figure 1: The reactive system architecture

Several systems, such as Meta (Wood 1994), Disco (Systa 1996) and STATEMENT (Harel 1990), and languages, such as Reactive C (Boussinot 1991) and Reactive Pascal (Quintero 1996) that are based on the reactive system concepts have been developed recently. However, most of the research on reactive systems is concentrated on process control (such as controlling a robot). In this paper, we try to implement the above reactive system architecture and apply it in developing distributed applications. We will implement DMMs, sensors and actuators that can be used in a distributed environment using Java language. Two examples, one is a web-based teamwork support system and the other is a network partitioning problem, are presented in the paper based on the reactive system concepts.

The rest of paper is organized as follows: in the next section, we address the implementation issues of reactive modules. Their performances are presented in Section 3. Two examples as the applications of the reactive system method are given in Section 4. Section 5 summaries our work.

IMPLEMENTATION OF REACTIVE MODULES

In Figure 1, a DMM subscribes to sensors and receives reports from these sensors on the application's states. The DMM then uses actuators to change the states of the applications according to the policy it implemented. In this model, sensors can be attached to applications to obtain their states (or monitor some events about the applications). These states or events are sent to the DMMs which react to them by using actuators to change the states of applications.

There are mainly three reactive modules: DMM, sensor and actuator in Figure 1. They are very generic. We can implement them as generic classes using Java programming language. Java virtual machines, which are rapidly becoming available on every computing platform, provide a virtual, homogeneous platform for distributed and parallel computing on a global

scale. A Java application usually consists of many objects (may be developed using languages other than Java) distributed all over the network and therefore it is essential for the control part of the application to know the current states of these objects. The DMMs, sensors and actuators implemented in Java can have this function.

Two communication methods can be used in Java: multicast data-gram and stream-based communications. Zhou (1998) has addressed multicast data-gram sensors and it gives a conclusion that multicast data-grams are an unreliable method of communication and therefore may have limited application in distributed systems covering multiple subnets. However, stream-based sensors are more reliable. Hence, we will implement Java DMMs, sensors and actuators using the stream-based communication. They are all implemented as multithreaded entities using various features of the Java language.

Without losing generality, we assume that there are *m* DMMs and *n* sensors and actuators respectively in a reactive system. Each sensor can be subscribed by multiple DMMs and each DMM can subscribe to multiple sensors as well. Once a DMM makes a decision, it will be sent to relevant actuators to change the related applications' states.

Stream-based DMM class

The generic Java DMM class has the following functions:

- First, it subscribes to sensors by establishing connections with the sensors and then waits for reports from them.
- Second, once a DMM receives a report from a sensor, it will process it and make decisions according to the predefined policy. Since some applications are related, the decisions made by the DMM will be sent to the related actuators to change the related applications' states.

Using Java multiple threads, the DMM class will create a number of threads to handle each specific task. It first creates a Connector thread to build connections with sensors it subscribes to, and then creates a group of threads (Receiver) to handle each specific connection (to a sensor). Therefore, the connections between the DMM and the sensors are managed by their own threads of executions (there are as many Receiver threads as the number of sensors). The Receiver threads wait for reports from the sensors using dedicated connections. Meanwhile, the DMM creates a DmmToActuator thread for each actuator to handle its connection (to the DMM), which sends decision messages to the actuator.

The stream-based DMM class consists of multiple objects interacting with each other. These are a main DMM object that processes reports and makes decisions, a connector object that connects to sensors, a receiver object that waits for reports from sensors and a dmmToActuator object that sends decisions to actuators. The following codes achieve the DMM class:

```
public StreamDmm(int[] port) {
    //Constructs DMM and identifies the port number used to establish connections with
   //sensors,creates a connector object.}
public void decisionMaking() {
   //Empty, needs to be implemented for each specific DMM.}
class Connector extends Thread {
   //Builds connections to sensors and creates a Receiver object for each connection.}
class Receiver extends Thread {
   //Establishes communication channels between the DMM and each sensor, waits for reports from
   //the sensors and processes them, meanwhile creates a DmmToActuator object to handle
   //connections with actuators.}
class DmmToActuator extends Thread {
```

```
    //Sends decisions to actuators.}
```

Each specific DMM class can inherit from the generic stream-based DMM class and only needs to implement the decisionMaking() method.

The stream-based sensor/actuator classes

A sensor can be subscribed by many other entities. It first builds connections with its subscribers and then monitors for events and reports to its subscribers once events occur. Similarly, the Java sensor class creates a number of threads to handle each specific task. It first creates a listener thread to build a connection channel and listens to connection requests from its subscribers. Then, after a connection (requested by a subscriber) is established, the sensor will create a thread to handle it. Each connection (to a subscriber), therefore, is managed by its own thread of execution. Once an event occurs, the sensor will send it to each subscriber using a dedicated connection. Other entities can subscribe to the sensor by simply requesting a connection to it.

In order to synchronize the transmission of reports, the sensor creates a ThreadGroup, into which each new thread created to handle a connection is placed. Using the ThreadGroup, the sensor can invoke each thread to report to its DMMs at the same time, rather than the threads having to report to the DMMs individually and asynchronously. In addition, the ThreadGroup approach places the monitoring of events in one place within the sensor, rather than each thread having to monitor for an event, which duplicates processing. Figure 2 depicts the architecture of a generic stream-based sensor.

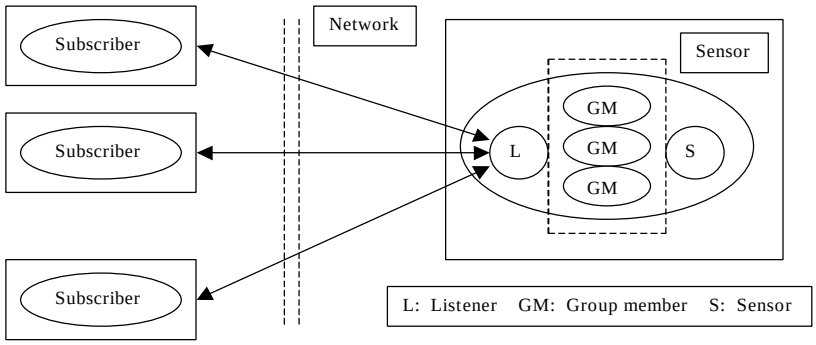

Figure 2: The generic sensor architecture

The sensor class includes a main sensor object that monitors for events, a listener object that listens for connection requests from DMMs, and a collection of zero or more member objects, each representing a connection to a DMM and responsible for communication between the sensor and the DMM. The sensor class offers the following services:

```
protected StreamSensor (int[] port)  {
    //Constructs sensor class, identifies the port numbers used to establish connections with DMMs,
    //creates a listener object and a threadGroup object}
public void run () {
    //Empty, needs to be implemented for each specific sensor.}
protected void report() {
    //Invokes each thread within the thread group to send a report.}
class Listener extends Thread {
    //Establishes connections with DMMs and creates a threadGroupMember object to handle each
    //connection and puts it into the threadGroup object.}
class ThreadGroupMember extends Thread {
```

```
    //Establishes communication channels between the sensor and DMMs, sends event messages to
    //DMMs. }
```

A specific sensor can inherit from the generic stream-based sensor and only needs to fulfill the run() method.

An actuator receives decisions from DMMs, and then changes an application's state according to the decisions. After that it may return an acknowledgement to the DMMs. The actuator class has a main actuator object that changes the application's states and a receiver object that receives decisions from DMMs.

```
protected StreamActuator(ObjectInputStream obs)  {
    //Constructs actuator class,identifies the input stream object. }
public void run() {
    //Empty, needs to be implemented for each specific actuator.}
protected void Receiver() {
    //Receives decisions from DMMs. }
```

A specific actuator can inherit from the generic stream-based actuator and only needs to re-write the run() method.

PERFORMANCE ISSUES

We have conducted a series of tests to evaluate the performance of above Java DMM, sensor and actuator. The purpose of these tests is to determine the times taken for communications among DMMs, sensors and actuators which are located in different distributed environments. According to the time they take, we can evaluate the effectiveness of them running in a distributed environment.

Test description

The test sites consisted of a collection of networked (10M Ethernet) Sun sparc machines running the Solaris operating system. Sun's JDK1.2 Java interpreter is used to run all modules.

Three groups of tests have been performed, each in a different distribution environment. The first group of tests is conducted with DMMs and sensors/actuators located on the same host. The second group of tests is conducted with DMMs and sensors/actuators located on the same subnet but different hosts. In tests with more than one DMM and sensor/actuator, each DMM or sensor/actuator is located on a separate host. The last group of tests is conducted with sensors/actuators located on a remote subnet from the DMMs' subnet. As in the previous test group, in cases when there are more than one DMM or sensor, each of them is located on a separate host.

The time we measured to evaluate the communication among one DMM, one sensor and one actuator starts from the sensor being triggered and then reporting an event to the DMM, and ends at the actuator receiving a message from the DMM. The sensor is embedded into a test application that triggers the sensor to report to its DMM when an empty event has occurred. The DMM actually does nothing but sending the message back immediately after it receives the report from the sensor using an actuator. The actuator does nothing but sending the message from the DMM to the sensor. To simplify the tests, we embed the actuator into the sensor so that the sensor instead of actuator can directly receive the message from the DMM. The tests measure the overhead of the sensor being triggered by an application and the time taken to report the event to the DMM and receive the message from the DMM (through an actuator) as well.

In cases with multiple DMMs and sensors composition, we take an average time from those results for each DMM and sensor. Each composition is tested 1000 times with an average time taken from these results. Stream-based DMMs and sensors use a separate thread to manage each connection between a DMM and a sensor. In tests with multiple DMMs and a single sensor, the sensor invokes each thread to report to its DMMs when an event is detected. Each thread measures the time which the sensor needs to report to and receive from each DMM, and then this time is added to a total and divided by the number of DMMs to the sensor. This provides an average overhead associated with the increasing number of DMMs to the sensors. In tests with multiple sensors and DMMs, we first measure an average overhead for each sensor according to the above method, then we can get the average time for multiple sensors by adding all the overheads to a total then divided by the number of sensors.

Local host testing

The first group of tests is conducted on a local host. The tests are performed on a series of compositions of sensors and DMMs. Table 1 shows the time needed by different compositions for their communications.

M: Number of DMMs N: Number of sensors

M:N	1	2	3	5	10
1	13.712	13.698	13.802	13.742	13.132
2	19.916	20.368	21.162	20.910	21.508
3	32.310	30.796	33.088	32.952	33.442
5	51.318	51.456	51.630	51.942	50.704
10	105.834	105.706	106.422	106.656	105.798

Table 1: The time tested on a local host

We first take a look at each row on the table 1. These are tests with multiple sensors reporting to a certain number of DMMs. To our surprise, the times used by DMMs and sensors did not increase with an increasing number of sensors. The values on each row are very similar. That means, the times used by a certain number of DMMs are almost the same no matter how many sensors they subscribed to. Probably this is because each stream based sensor uses a dedicated socket to connect with its subscriber(s) and different sensors use different sockets so that their communications with their subscribers are not influenced by each other.

Then we have a look at each column on the table 1. These are tests with multiple DMMs subscribing to a fixed number of sensors. The results confirm our intuition that as the number of subscribers increases, the time used by DMMs and sensors increases as well. This is due to that more subscribers may require more resources such as sockets which sensors use to connect to subscribers.

Figure 3 shows the results tested on a local host. It shows that the times used by DMMs and sensors increase with the increasing number of DMMs, but not with the increasing number of sensors. We can see that the curves of s (number of sensors) equaling to 1, 2 and 3 are almost overlapped. That means, the time used by the system has nothing to do with the number of sensors. Hence, we can use one of these curves (e.g. 1 sensor) to represent the time curve of a local host testing.

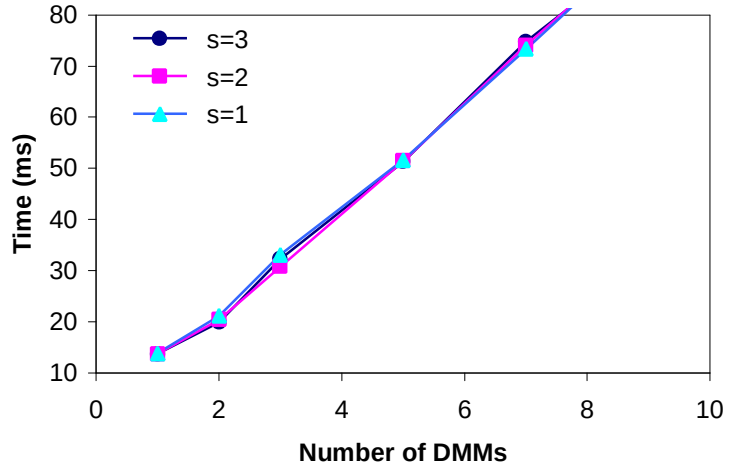

Figure 3: Test on a local host

Different hosts testing

Two groups of tests are conducted with DMMs and sensors located on different hosts but on the same subnet and remote subnets, respectively. These tests show a similar result to the local host testing, that is, with an increasing number of DMMs, the processing time for DMMs and sensors increases as well, but not with an increasing number of sensors. The only difference is that three groups take different times to process the communication among DMMs and sensors. Figure 4 shows the comparison of three groups of tests in the cases of one sensor.

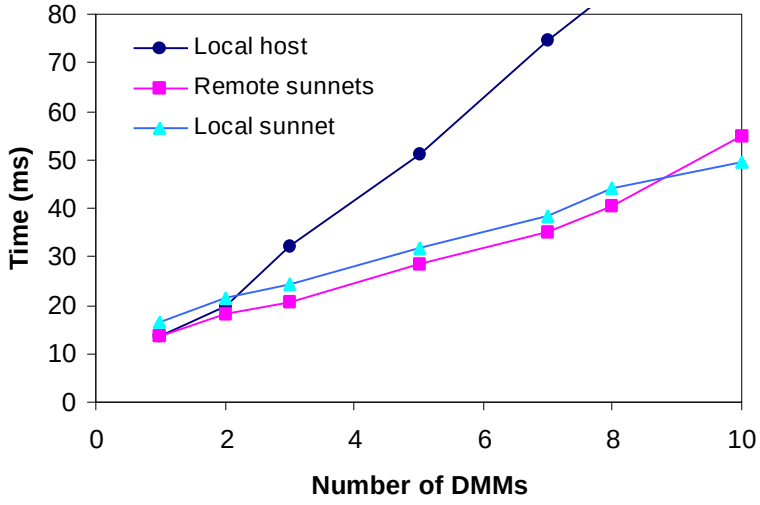

Figure 4: Three groups of tests

From Figure 4, we have three observations. First, we see that the time curve for a local host is steeper than those for a local subnet and remote subnets. That means, in the cases of a local subnet and remote subnets testing, the time for communication among DMMs and sensors does not increase very dramatically with an increasing number of DMMs. In fact, their

increases are much less than the increase in a local host! This is good news for our reactive modules, because most of DMMs require reports from sensors located in different hosts. This result shows that DMMs and sensors running in a distributed environment is more effective than them running in a local host.

Second, we can find that the DMMs and sensors running on remote subnets take more time than them running on a local subnet. But on both cases the time they took increases at an almost same speed with an increasing number of DMMs. This is because, the communications among remote subnets require more resources than that on a local subnet.

Last, we can see that, in the case of 1 DMM, sensors running on a local subnet or on remote subnets use more time to report to and receive from the DMM than sensors running on a local host. When the number of DMMs increases to 2, there is little difference for the time taken among the three groups. However, when the number of DMMs increases further, the DMMs and sensors running on a local subnet or on remote subnets use less time than their running on a local host. This is because we run each DMM or sensor on an individual host for local subnet and remote subnets testing, while for local host testing, all DMMs and sensors are executed on the same host.

APPLICATIONS

After implementing and evaluating the reactive modules, we want to apply them in developing distributed applications. Two examples are presented in this section for the demonstration of applications of the reactive system model.

The Web-based Teamwork Support System

Teamwork is a key feature in any workplace organization. In this computer era, many tasks can be carried out by team members, who may be physically dispersed, cooperatively on the Internet via the Web. In a teamwork system, team members prepare their work individually in parallel which can be viewed as parallel steps. How to manage such steps or sub-tasks, in another word, coordination, is the key issue for completion of the entire task.

Figure 5 is the client-server architecture of a Web-based teamwork support system, where a centralized server site plays the key role for management/coordination of a task. A Java application for a particular teamwork-oriented task runs at the server site as a daemon all the time serving for the entire life-span of the task. At the client site, only an appropriate Web browser is required and no other particular software needs to be installed since each team member uses Web pages on the Internet and Java applets downloaded from the server site on-the-fly to carry out the sub-tasks allocated.

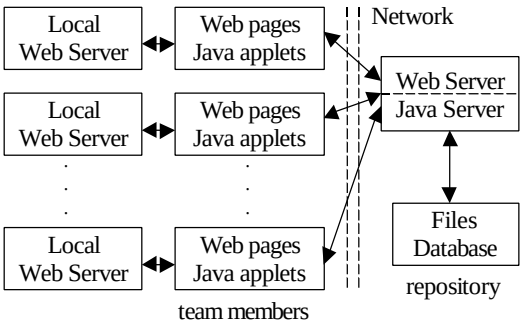

Figure 5: Information flow for supporting teamwork

The most common mechanism used for coordination in process support is a dynamic to-do list for each team member to inform the associated sub-tasks which need to be done. Once a team member has finished a sub-task on the to-do list, the notification should be made to the server. Then, at the server side, the appropriate coordination for process control can be adjusted to generate updated to-do lists for related team members. The client side is the key to getting the real work done. How to pass information between the server and clients, i.e., download data from and/or upload data to the server site, is very important, especially in a Web-based environment which often has restrictions for doing so. Furthermore, how to keep the system continuously running even in the presence of failures should be considered.

To resolve these problems, we embed the reactive system architecture into the teamwork support system. We use the sensor/actuator mechanism to pass information between the server and team members for coordination, as depicted in Figure 6, where a DMM (decision making manager) will be embedded into the Java application at the server site for making decisions to determine whether a sub-task can be started on the basis of the related client states. At the client site, sensors are attached to the Java applets for each team member to obtain their states and then report to the DMM. After it makes decisions, the DMM will use actuators to change the clients' states. The more detailed coordination mechanism about teamwork support is addressed in Chen (1999).

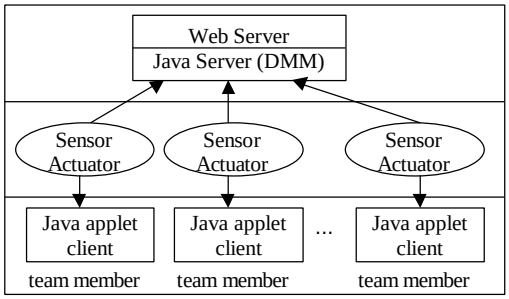

Figure 6: The architecture of teamwork coordination

In this case, the DMM stays same no matter what changes the sensors have, i.e., they are separated and not influenced by each other so that they can be maintained easily.

The Network Partitioning Problem

A network partitioning failure is a major threat to the reliability of distributed database systems and the availability of replicated data. A network partitioning occurs when failures fragment the network into isolated sub-networks called partitions, such that sites or processes within a given partition are able to communicate with one another but not with sites or processes in other partitions. If processes continue to operate in the disconnected partitions, they might perform incompatible operations and make the application data inconsistent.

Network partitioning failures most likely happen at a wide area network. We assume that the network environment is consisted of different subnets connected by gateways. At each subnet, we have database server groups which are comprised of replicas. All database servers (or replicas) store identical information initially and each of them can accept client requests (organized as transactions) that read or update stored information independently. The task of the replicated system is to maintain the data consistency among all the replicas throughout the whole network, even in the case of failures. Figure 7 depicts the architecture of such a distributed replication system.

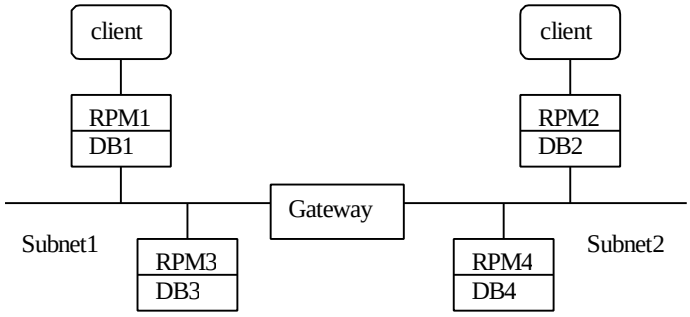

Figure 7: A distributed replication system

In this architecture, there is a replication manager (called RPM) running on each host where a database server (DB) or replica is running. A client issues transactions through the local replication manager. A transaction request issued by a client can consist of different sub-transactions each of which is to be serviced by a group of servers or replicas. Replication managers, which receive the requests from clients, divide the transactions into sub-transactions and pass them onto different replicas. Among one group of replicas, a Primary Replica leads other Non-primary Replicas. The transaction processing policy is to treat the Primary Replica for every sub-transaction, or service, as the checkpoint for fully commit mode. Any replica can execute a service freely but a partial commit mode is returned if it is a Non-primary Replica. Only those transactions checked by Primary Replicas will be finalized by either being upgraded to a fully commit mode or downgraded to an abort if conflict exists. Coordination among replica groups is carried out by replication managers to finalize transactions after collecting results from different service executions.

A network partitioning happens when gateways between subnets fail. This leads to a situation where server group members distributed in different subnets cannot communicate with one another and may stop a transaction processing. To detect and analyze partitioning failures, we embed the reactive system architecture into the replicated system. To do so, we add a dedicated decision making manager (DMM) as a server group component in each subnet and it will subscribe to sensors in each server member to find out the partition existence and help in transaction processing. Sensors are attached to each server member to report their states to the DMMs. Figure 8 shows the system modeled with the reactive system architecture. For

simplicity, we only include two subnets connected by one gateway in our network configuration.

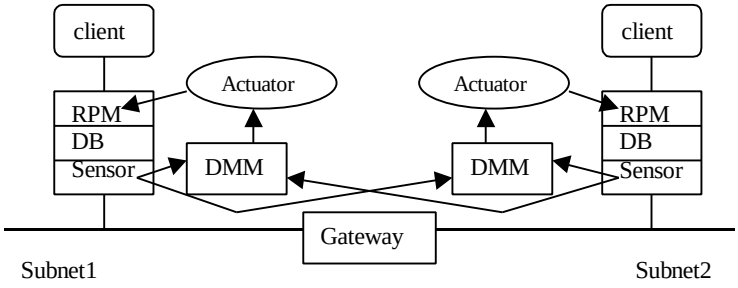

Figure 8: System architecture for partition-tolerant applications

In this architecture, each server group member attaches with a sensor which reports its state to the DMMs in different subnets. DMMs will decide whether a partition happens according to the reports received from sensors and then make decisions to instruct RPM how to process transactions using actuators. In Figure 8, RPMs deal with transaction processing while DMMs deal with failure handling and coordination between replica groups. A more detailed description of this application can be found in Chen (2000).

Similarly, in this example, the DMMs and sensors stay same no matter what changes each other has. The main mask of the DMMs is to make decisions, and the main task of the sensors is to monitor for clients' states.

CONCLUSION

Reactive systems concepts are an attractive paradigm for system design, development and maintenance because it separates policies from mechanisms. This paper has presented the implementation and performance analysis of the reactive system for building distributed applications. The main advantage of reactive system concepts is the separation of mechanisms and policies in software development. The performance of Java DMMs, sensors and actuators shows that they can be used in a distributed environment effectively. Two examples of their applications in the teamwork support and the network partitioning problem have showed the potential benefits of the Java DMM, sensor and actuator classes. In both cases, the DMM modules stay the same. Their main task is to make decisions according to the reports from the sensors they have subscribed. This shows the advantage of separating mechanisms from policies.

REFERENCES

Boasson, M. (1993) Control Systems Software. *IEEE Transactions on Automatic Control*, vol. 38, nr. 7, 1094-1107

Boussinot, F. (1991) Reactive C: An extension of C to program reactive systems. *Software - Practice and Experience*, 21(4): 401-428

Caspi, Paul, Alain Girault and Daniel Pilaud (1994) Distributing reactive systems. *The ISCA International Conference on Parallel and Distributed Computing Systems (PDCS'94)*. Las Vegas, USA

Chen, C. and W. Zhou (2000) An Architecture for Resolving Network Partitioning. *Proceedings of the ISCA 15th Int'l Conf. for Computers and Their Applications (CATA-2000)*, 84-87, New Orleans, USA

Chen, C., W. Zhou and Y. Yang (1999) Coordination Mechanism for Teamwork Support *Proceedings of the 1999 Asia Pacific Decision Sciences Institute Conference*, 389-391, Shanghai, China

Davila Quintero, J. A. (1996) Reactive PASCAL and the event calculus. *Proc. of the Workshop at FAPR'96: Reasoning about Actions and Planning in Complex Environments.* Darmstadt, Germany, June

Edwards, S., et. al. (1997) Design of embedded systems: Formal models, validation, and synthesis. *Proceedings of IEEE*, 85(3):366-390

Gerth, Rob and Orna Grumberg (1997) Abstract Interpretation of Reactive Systems. *ACM Trans. on Programming Languages & Systems*, v.19 n2, 253-239

Harel, D. and A. Pnueli (1985) On the development of reactive system. *Logics and Models of Concurrent Systems.* Krzysztof R. Apt. Spring-Verlag, Berlin, Heidelberg, New York, Tokyo. 477-498

Harel, D. and A. Shtul-Trauring (1990) STATEMATE: A working environment for the development of complex reactive systems. *IEEE Trans. on Software Engineering*, 16(4): 403-414

Harel, D. and Michal Politi (1998) *Modeling Reactive Systems with Statecharts: The Statemate Approach.* McGraw-Hill Companies, January

J cIUvinen, H. M., R. Kurki-Suonio, M. Sakkinen and K. Systa (1990) Object-oriented specification of reactive systems. *Pro. 12th International Conference on Software Engineering*, IEEE Computer Society Press, 63-71

Selic, Bran, Garth Gullekson and Paul T. Ward (1994) *Real-Time Object-Oriented Modeling*. John Wiley & Sons, Inc.

Systa, K. (1996) *The Disco tool.* Tampere University of Technology, Tamphere, Finland, http://www.cs.tut.fi/laitos/Disco/tool.fm.html.

Wood, M. and K. Marzullo (1994) The design and implementation of Meta. In K. P. Birma, editor, *Reliable Distributed Computing with the Isis Toolkit*, pages 309-327. IEEE Computer Society Press

Zhou, W. (1999) Detecting and tolerating failures in a loosely integrated heterogenerous database system. *Computer Communications*, 22, 1056-1067

Zhou, W. and E. Eide (1998) Java Sensors and Their Applications. *Proceedings of the 21st Australian Computer Science Conference (ACSC 98)*, 345-356, Perth, Australia

COPYRIGHT