A Reusable Software Framework for Distributed Decision-Making Protocols

Sudharshan Vazhkudai H. Conrad Cunningham
Department of Computer and Information Science
University of Mississippi
University, MS 38677 USA

Abstract

Developers of programs for distributed systems spend considerable time designing solutions to various complex decision-making problems. The work described in this paper applies the techniques of design patterns and software frameworks to the problem of designing decision-making protocols in distributed systems. The goal is to reduce the time and effort involved in implementation.. The paper presents a reusable software- framework and applies it to a sender-initiated distributed scheduling protocol.

Keywords: *Frameworks, Distributed Systems, Distributed Algorithms, Object Orientation, Patterns.*

1.0 Introduction

Often the complexity of a distributed system is a hindrance to the rapid development of safe and effective software. Distributed systems do not have a central point of control with direct access to the states of all nodes, thus decisions often require an exchange of state information and a complex "negotiation" among the nodes. The coordination of the distributed components requires complex message-based protocols for synchronization and communication. The basic complexity is the same whether the system is a multicomputer, a cluster of workstations or a worldwide network.

Object-oriented techniques and technologies have opened up new approaches to the design of programs for distributed systems. Of particular interest are design patterns and software frameworks. According to Buschmann, a design pattern "describes a particular recurring design problem that arises in specific design contexts and presents a well-proven generic scheme for its solution." [1] The application of patterns potentially can reduce the complexity of the process of designing and implementing distributed systems software [2]. Whereas a pattern is a reusable design idea, a framework is a collection of software entities. According to Johnson, "a framework is a reusable design expressed as a set of abstract classes and the way their instances collaborate" [3]. Although construction of a software framework may require a complex software development effort, once developed and tested, a framework reduces the complexity of developing software within its domain.

The work described in this paper applies the techniques of patterns and frameworks to the problem of designing decision-making protocols in distributed systems. The goal is to reduce the time and effort involved in implementation. The types of protocols covered include:

A *mutual exclusion* protocol, where a node requests permission from all other nodes to enter its critical section [4].

A *consensus* protocol, where nodes have to agree upon certain issues. [5]

A *leader election* protocol, where nodes are selected to coordinate certain tasks [4].

A *sender or receiver initiated* protocol for distributed scheduling, where nodes advertise their need/desire to schedule jobs and decide based on the response to the advertisement [6].

A *Byzantine* protocol, where a faulty node is detected based on recursive message passing (announcing the status) and decision making [7].

A *distributed deadlock detection* protocol, where messages are sent to nodes that hold resources in an attempt to break a deadlock [8].

2.0 Framework Architecture

To build an object-oriented framework, we must identify the common aspects of the protocols and design an appropriate abstraction. This abstraction can then be expressed as a set of abstract and concrete classes that collaborate to form the basic structure upon which specific protocol implementations can be built. The first choice in design is to select the architectural style [9] that will guide the development. Then abstract classes that capture the desired high-level behaviors can be defined. These abstract classes will include certain abstract behaviors represented by not-yet-defined operations that conform to the specified interface. These operations can be defined by building a class that inherits from the abstract class or by delegating the operation to other classes. In this work, we can be guided by design patterns such as the Template Method (which uses inheritance) or the Strategy pattern (which uses delegation) [10].

2.1 Horizontal View

The framework consists of two pivotal "horizontal" entities, an Initiator and a Listener. Instances of these reside in each node in the distributed system and all instances execute concurrently. The Initiator and the Listener entities are composed of several component objects. These objects collaborate to perform the protocol-specific and platform-dependent tasks.

The Listener entity launches at system startup and listens for requests from other nodes. It responds to each request by creating a separate child process. This enables the parent to continue listening for additional requests. Thus the Listener behaves as a concurrent server.

A node launches its Initiator entity when it must make a decision that requires coordination with other nodes. This decision making might use any of the above distributed protocols. Typically, the Initiator entity at a node will collaborate with the Listener entities in all the other nodes to arrive at a decision. The Initiator and Listener execute concurrently; while the Initiator at a node X is interacting with the Listeners at other nodes, the Listener entity at node X can interact with Initiators at other nodes. This type of architecture supports the dynamic nature of a distributed system, where any node can initiate a dialogue randomly.

2.2 Vertical View

Since this problem domain involves a mix of high- and low-level issues, it seems natural to employ the Layered Architecture pattern in building the framework [1]. We identify three "vertical" layers as shown in Figure 1: an Application Layer, a Protocol-Specific Layer, and a Platform-Dependent Layer. The horizontal entities are themselves layered, with their constituent objects distributed across the three layers with interaction only on the lowest layer. The Layered Architecture specifies the responsibilities of each layer and defines the interlayer collaborations.

2.2.1 Application Layer

The top layer of the architecture is the Application Layer. Component objects at this layer are aware of the needs of the user's application. The primary task is to configure the Protocol-Specific Layer below. Since a group of compatible objects must be created together, this is an opportunity to use the Abstract Factory pattern [10]. Thus, the algorithm that needs to make a decision and the abstract factory are both part of the Application layer.

2.2.2 Protocol-Specific Layer

The middle layer of the three-tiered architecture is the Protocol-Specific Layer. The abstract classes provide a generic framework from which protocol- specific classes can be derived.

To define the appropriate abstract structure, we first analyze typical protocols from the domain and identify the "hot spots" and "frozen spots" [11]. That is, we identify the points at which the protocols are different and at which they are the same. We define concrete classes and operations to implement the shared functionality of the frozen spots. We then specify the behavior at the hot spots as abstract operations and classes. We encapsulate the differences among the protocols inside the abstract operations.

The Template Method pattern [10] helps us structure the abstract classes so that we can use inheritance to provide the needed concrete definitions for a specific protocol [10]. We express the

high-level protocol-independent aspects of the algorithm in concretely defined operations of a class. These concrete operations call the abstract operations to carry out the protocol-specific tasks. To implement a new decision-making protocol, we provide appropriate new concrete definitions of the abstract operations.

The Strategy pattern [10], can provide delegation-based plug-points in the framework corresponding to the hot spots. In this approach, we capture the high-level, protocol-independent aspects of the algorithm in the specification of an abstract class. We then define a concrete class that implements the abstract class for a specific protocol and plug an instance into the framework. The overall framework then delegates the protocol-specific details to the instance.

In the decision-making framework, this layer consists of the DispCol and the Decision-Maker classes in the Initiator (i.e., at the client end) and the MsgMgr and the MsgProc classes in the Listener (i.e., at the remote end).

2.2.3 Platform-Dependent Layer

The bottom layer of the architecture is the Platform-Dependent Layer. This layer consists of components that are not affected by the specifics of a protocol. For example, a change in protocol will likely require recompilation of the code for the top two layers but will not require recompilation of this layer.

This layer is, however, dependent upon the computing platform (hardware, operating system, and network) upon which it executes. This layer must be changed when the platform changes. The layer consists of objects that maintain the system state. It encapsulates the mechanisms for gathering the relevant information about a node for sending this information to other nodes, and for receiving similar information from other nodes. The layer maintains this information in the Comm and SST objects.

2.3 Framework Components

The hot spots in the framework appear in the Protocol-Specific Layer. Thus we concentrate our attention on the classes in that layer, beginning with the Initiator component (Figure 2).

2.3.1 Components of the Initiator

The Dispatcher/Collector class *DispCol* is the main entity in the Protocol-Specific Layer. The responsibilities of this class are as follows:

1. Construct a Message from the information provided by the Application Layer.
2. Contact the Platform-Dependent Layer to fill in details such as to whom to send, how long to wait after dispatching the message, etc.
3. Dispatch the Message built to the Comm object of the Platform-Dependent Layer.
4. Collect responses from the Platform-Dependent Layer and wait for responses.
5. Tabulate the responses.
6. Invoke the DMM (Decision-Maker) object to make the needed decision.
7. Convey the decision to the client.

A fragment of the C++ class definition of DispCol class is shown below.

```
class DispCol {
protected:
        Message *msg;
        Comm *tpg;
        int timer_has_expired;
        int timeout_replies;
        int no_replies_xptd;
        int no_arrived;
        Message **response_list;
        ……
public:
        virtual int ComposeMesg() = 0;
        virtual int Tabulate(Message *msg) = 0;
        virtual int InitResponseList() = 0;
        int Dispatch();
        void doit(DecisionMaker *dmm);
        int StartTimer();
        int EnterWaitLoop();
        void DispColTimeOut(int sig);
        ……
};
```

Let us look at code fragments for a few of the member functions to illustrate the functionality.

```
void DispCol::doit (DecisionMaker *dmm) {
        ……
        this->ComposeMesg();
        this->InitResponseList();
        this->Dispatch();
        dmm->Decide(response_list, no_arrived);
}
```

We define the behavior of the hot spots using abstract operations. In the code above, the

methods defined as virtual are identified as portions of the framework that vary between protocols. These methods include the ComposeMesg(), Tabulate(), and InitResponseList() operations. Similarly, examples of the frozen portions of the framework are the Dispatch() and EnterWait-

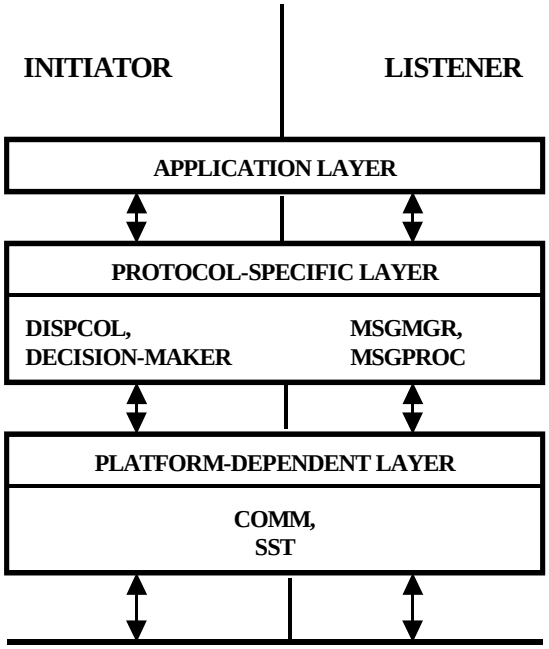

Figure 1

Loop() operations.

The DispCol object combines these methods to provide a template (using the Template method pattern) that is inherited by concrete protocol implementations. The doit() method is a template for specific protocol implementations. The DispCol class also delegates the actual decision making to the DecisionMaker object, dmm. This delegation of responsibility for a protocol-specific algorithm to another object is an application of the Strategy pattern. The Dispatch() method is a concrete method that uses the Comm object to send and receive messages.

```
int DispCol::Dispatch() {
        ……..……..…..
        this->tpg = new Comm(this->msg);
        ……..……..…..
        this->tpg->Send();
        ……..……..…..
        StartTimer();
        EnterWaitLoop();
        ……..……..…..
}
```

The EnterWaitLoop() method is a concrete method that waits either for all messages to arrive or for the timer to expire.

```
int DispCol::EnterWaitLoop() {
        ……..……..
        while((no_arrived != no_replies_xptd)    &&
        ! timer_has_expired) {
                msg = this->tpg->Receive();
                no_arrived++;
                Tabulate(msg);
                ……..……..…..
        }
}
```

Other methods such as StartTimer() and DispColTimeOut() are concerned with setting up timers and time-out values.

Note that a Message class is used in the DispCol class. The Message Class is the basic message entity that is passed back and forth between various component objects and nodes.

```
class Message {
protected:
        int protocol_id;   // MutExcl, Scheduling, etc.
        int from;          // from id
        int whom2send;  // could be ID of a group
        ……..……..…..
};
```

The above framework elements are the most common among protocols. Specific protocols would inherit from this class to add other elements. It is the responsibility of the DispCol object to fill in a Message object. It achieves this by contacting the SST object, which is described later.

The *DecisionMaker* class is strictly concerned with arriving at a decision based on the responses collected; it is invoked by the DispCol object. The C++ class definition is shown below.

```
class DecisionMaker {
public:
        DecisionMaker();
        ……..……..…..
        virtual int Decide(Message **res,int arrv)=0;
};
```

The abstract method Decide() is the most important element. Specific protocols could implement different types of algorithms. Specific implementations apply different algorithms to the response list as part of the Decide() method. For

example, the Decide() method could perform a sort on process and memory values for a scheduling protocol and then decide the validity of the token for a mutual exclusion protocol.

2.3.2 Components of the Listener

Figure 3 shows the components of the Listener entity.

The *MsgMgr* class is the DispCol equivalent of the Listener. Its responsibilities include:

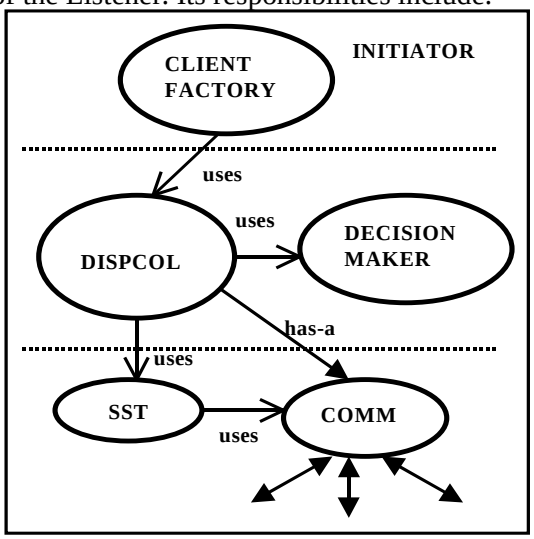

Figure 2

1. Composing a message from the arrived data.
2. Invoking the MsgProc object with the message.
3. Composing a message for reply.
4. Replying to the Initiator, using the Comm object.

The C++ class definition is shown below.

```
class MsgMgr {
protected:
        Message *msg;
        Comm *tpg;
        ……..
public:
        ……..
        virtual int ComposeMesg() = 0;
        void doit(MsgProc *mproc);
        int Reply();
};
```

The concrete Reply() and doit() methods are quite similar to the methods in DispCol class, except there is no timer involved. The ComposeMesg() abstract method must be specialized by different protocols to build protocol specific messages.

The *MsgProc* class is similar to the DecisionMaker class. The MsgProc also has an abstract Decide() method. The Decide() method is specialized by the specific implementation.

2.3.3 Common Components

The classes in the Platform-Dependent Layer, Comm and SST, are two components that are common to both the Initiator and the Listener.

The *Comm* class is a generalization of the communication aspects of distributed systems. It encapsulates the architecture (multiprocessors, etc), communication media (shared memory, etc), topology (bus, ring, tree, etc), header and packet building mechanisms, time-out mechanism, and packet transmission, re-transmission, reception and synchronization issues [6, 8].

The *SST* is the System State Table [4]. Each node maintains a copy of the SST; it holds the necessary information about the machines in the distributed system. Thus, all nodes communicate among themselves to update their SST's. Listed below are a few elements of the SST.

```
class SST {
private:
        int whoami, cluster_size, whoiswho;
        int do_I_have_token;
        int nrprocs, freemem, netreqs;
        ……..
};
```

3.0 An Implementation

We have implemented two protocols, a Machine Election for Sender-Initiated Distributed Scheduling [6] and Mutual Exclusion [4]. Here we look at the scheduling protocol.

3.1 Sender-Initiated Scheduling

A sender-initiated scheduling protocol is a load-sharing process initiated by a heavily loaded node that attempts to send a task to a lightly loaded node. The overloaded node, X, queries all other nodes to determine whether they have enough resources to take on a task from the overloaded node. The requirements for the process (say a node with available memory >= M and with processes < P) are encapsulated in the mes-

sage sent to a group of nodes in the system. Other nodes reply (along with their memory and processes values) if they have enough resources. Node X then selects the optimal one [6, 8].

Let us discuss a few important methods. We name the derived classes with the suffix MCELE (machine election). Thus we have DispColMCELE, DecisionMakerMCELE, and so forth. This implementation has a MessageMCELE class that inherits from the Message class. It has the following elements.

```
class MessageMCELE : public Message {
private:
        int proc_req, mem_req;
        ……..……..……..……..
};
```

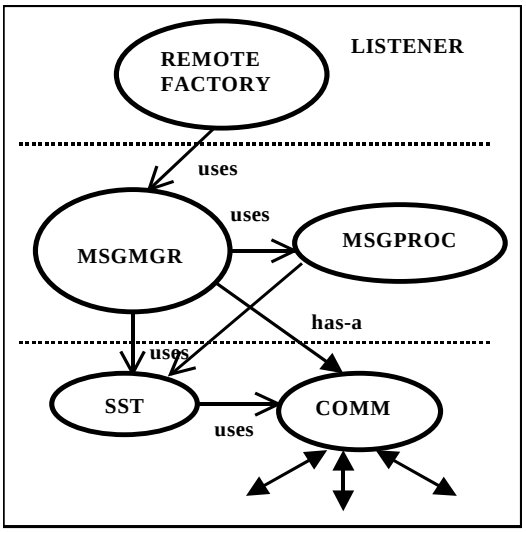

Figure 3

The Tabulate() virtual method creates a list of all the responses. A variation could be a hashing function that hashes responses into the list based on their process and memory values.

```
int DispColMCELE::Tabulate(Message *msg) {
        MessageMCELE *msgmc = new MessageMCELE(msg);

        this->response_list[no_arrived-1]= (MessageMCELE *)msgmc;
        ……..……..……..……..
}
```

The Decide() method finds the optimal response (the node with the most memory and fewest processes) from the list of responses.

```
int DecisionMakerMCELE::Decide
        (Message  **response_list, int noarr) {
        for(i = 0; i < noarr; i++) {
        // A simple sort on response_list[ i]
        }
}
```

The Decide() method of the MsgProcMCELE class queries the SST object to find the number of processes currently running and the amount of memory available. It compares these values with the proc_req and the mem_req in the message. If the condition is satisfied, it copies the values into the message. Thus, using the framework, we could develop efficient protocols easily, thereby minimizing the effort required otherwise.

```
virtual int MsgProcMCELE::Decide(Message *msg)
{
        SST *sst = new SST();
        int p, m;

        sst->GetProcMem(&p, &m);
        if((p <= msg->GetProc() &&
                        (m >= msg->GetMem())) {
                msg->SetProcMem(p, m);
                return 1;
        }
        else
                return 0;
        }
}
```

3.2 Application Layer Issues

The Application Layer uses a structure based on the Abstract Factory pattern [10] to manage the different protocols. Given a set of related abstract classes, this pattern enables us to create compatible groups of objects drawn from the concrete subclasses without the user knowing the specific subclasses being instantiated. The need for the pattern stems from our desire to support multiple protocols in each node. All these protocols have a common set of abstract classes. We employ the pattern to create concrete factories, which handle specific protocol requests.

In the Initiator, the concrete factory receives a request from the client and instantiates the protocol-specific classes such as DispColMCELE and DecisionMakerMCELE. Likewise, in the Listener, the concrete factory receives a message from the Comm object and instantiates the MsgMgrMCELE and MsgProcMCELE

classes. The Comm object delivers messages to the factories for different protocols. The factory would typically create a separate thread of execution for each request.

4.0 Discussion

System developers must invest significant effort in design and implementation of decision-making protocols. To decrease the needed investment, we developed a reusable software framework for distributed decision-making protocols. We organized the framework using a layered architecture and presented two complementary views, horizontal and vertical.

The horizontal view consists of two types of entities, Initiators and Listeners. They represent the high-level, inter-node collaborations necessary to make the decisions. An Initiator and a Listener execute concurrently on each node. Each entity consists of component objects spread across the various layers of the architecture.

The vertical view is of three interacting layers, organized according to the information "hidden" inside each. At the top, the Application Layer encapsulates the details of the user's computation. In the middle, the Protocol-Specific layer encapsulates the specific details of a decision-making protocol. At the bottom, the Platform-Dependent layer encapsulates the details of the hardware/software infrastructure.

We focussed was on the Protocol-Specific Layer. We factored it into generic aspects that are the same for all protocols in the domain (i.e., frozen spots) and aspects that vary among the protocols (i.e., hot spots).

The simple framework described here is a white-box framework [12]. That is, its design and implementation must be understood before it can be used. We applied the Template Method pattern to structure the generic algorithms for distributed decision making. The framework includes several abstract classes that encapsulate these algorithms. In these algorithms, certain key operations are left abstract. For a specific application users must subclass the abstract classes, providing appropriate concrete definitions for the abstract operations. As more implementations are done using the framework, it may be possible to evolve it toward a black-box framework. A black-box framework [12] includes a library of concrete, application-specific components that can be plugged into hot spots in the generic application. The components for a particular hot spot must adhere to the given abstract interface and have the required behaviors. However, these components can be used without the user understanding the details of their internal construction; the users only need to understand the external interfaces of the components.

References

[1] F. Buschmann, R. Meunier, H. Rohnert, P. Sommelad, M. Stal, *Pattern-Oriented Software Architecture, A System of Patterns*, Wiley, 1996.

[2] D.C. Schmidt, "ASX: An Object Oriented Framework for Developing Distributed Applications," *Proceedings of the 6th USENIX C++ Conference*, Cambridge, MA, April 1994.

[3] R. Johnson, "Frameworks Home Page," Dept of Computer Science, University of Illinois,1997.http://st-www.cs.uiuc.edu/users/ johnson/ frameworks.html

[4] N.A. Lynch, *Distributed Algorithms*, Morgan Kaufmann, 1996.

[5] D. Dolev, R. Strong, "Distributed Commit with Bounded Waiting," *Proceedings of the 2nd IEEE Symposium on Reliability in Distributed Software and Database Systems*, July 1982.

[6] M. Singhal, N.G. Shivaratri, *Advanced Concepts in Operating Systems*, McGraw Hill, 1994.

[7] L. Lamport, "The Weak Byzantine Generals Problem," *Journal of the ACM*, vol. 30, no. 3, July 1983, pp. 669-676.

[8] A.S. Tanenbaum, *Distributed Operating Systems*, Prentice Hall, 1995.

[9] M. Shaw, D. Garlan, *Software Architecture. Perspectives on an Emerging Discipline*, Prentice Hall, 1996.

[10] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns, Elements of Reus-*

able Object-Oriented Software, Addison-Wesley, 1995.

[11] W. Pree, *Design Patterns for Object-Oriented Software Development*, Addison-Wesley, 1995.

[12] R. Johnson, B. Foote, "Designing Resusable Classes," *Journal of Object-Oriented Programming*, 1988.