Control Message
An Object Behavioral Pattern for Managing Protocol Interactions

Joe Hoffert and Kenneth Goldman
{joh,kjg}@cs.wustl.edu
Distributed Programing Environments Group
Department of Computer Science,
Washington University, St. Louis, MO. 63130, U.S.A.

Abstract

Often in distributed applications, protocols are used to negotiate functionality between distributed components. It can be difficult to manage the interactions of protocol messages when they are negotiating functionality between these distributed components. The Control Message pattern simplifies message management by allowing messages to suspend and resume their executions based on replies received. This pattern can be generalized to objects waiting for events. An example usage is shown along with the benefits and liabilities of using the pattern. An implementation outline is also provided along with some sample code. Finally, patterns related to the Control Message pattern are listed.

1.0 Intent

Encapsulate a protocol message as an object to allow it to wait for and handle certain events during its execution. This allows an object using the Control Message pattern to suspend itself waiting for events and to resume execution once a desired event has occurred. It also allows the object to determine for itself the events which are of interest.

2.0 Also Known As

Microthread pattern?, Protocol pattern?, Rendezvous pattern? [maybe not applicable yet?]

3.0 Classification

Object Behavioral for Distributed or Multi-Threaded Applications.

4.0 Motivation & Context

Often in distributed applications, protocols are used to negotiate functionality between distributed components. As part of a protocol one component receives a request for some distributed functionality. It will then collaborate with one or more other components to provide the requested functionality.
Protocol negotiations may be needed, for example, to determine which options (if any) are supported for a particular functionality. They may also be needed to determine if the functionality can be realized between the targeted components. The protocol determines the valid messages and their interactions. It also defines valid negotiations of functionality. It specifies how message interactions should occur and what the legal possibilities are.

The interaction of components to enable some specified functionality is illustrated in Figure 1.

One component sends a request to another component. To achieve the desired functionality the enabling component collaborates with a third component. It sends a message to this collaborating component saying "Here are all the options I support for the requested functionality. Which of these do you support if any?" The collaborating component may then respond "I do not support what you need", "There's not enough common functionality between us to do the job", or "I support what you need but with these qualifications." The enabling component needs to wait on the collaborating component's response to proceed.

There may be several negotiation or collaboration stages between distributed components to accommodate certain functionality. The distributed components may need to do some processing on their own, send off a message to the other coordinating components, and then wait for a reply. This cycle may occur several times before the desired capability is enabled. It is important to note here that messages between components may cross process and machine boundaries.

*Command* objects [1] are good for encapsulating requests and replies. However, *Command* objects by themselves are unable to support negotiations (i.e., the interactions of protocols) between distributed components. They lack the ability to suspend and resume their executions. They also lack the ability to check for events that might be of interest to them. The *Control*
Message pattern uses the Command pattern to encapsulate requests as objects. Additionally, it allows these objects to execute for a time, suspend execution to wait for events, and resume execution when events of interest have arrived.

Complex protocols can be simplified using the Control Message pattern. A Protocol creates the appropriate ControlMessages. The ControlMessages are then told to send themselves to the appropriate components. When an initial ControlMessage is received by another component it is told to run itself. The ControlMessage will then execute until it needs to coordinate with another component or components. At this point, it will suspend its execution waiting for the appropriate coordination event. The Control Messages determine how and where they will be suspended.

Since Control Messages suspend themselves waiting for certain events, there needs to be an object that services events. Each component in the application using the Control Message pattern needs to notify ControlMessages when new events that might be of interest occur. In this regard, it makes sense to have an EventMonitor. There may be several different types of EventMonitors depending on the need of the application.

EventMonitors are aware of any relevant events that are received by a component. When an event is received, the EventMonitor will pass along the event to any waiting ControlMessages of which it is aware. For example, the EventMonitor may have a queue of suspended ControlMessages that are interested in events. This functionality is encapsulated in the waitingFor method.

When a new event arrives the EventMonitor will iterate through its ControlMessages invoking their takingOver methods with the event as an argument. The ControlMessage’s takingOver method checks if the ControlMessage wants to take over the event. If it does, it consumes the event. Thus, the EventMonitor will stop querying its remaining suspended ControlMessages for that particular event (cf. the External Chain of Responsibility pattern [2]).

When a ControlMessage receives an event of interest it checks to see if it has all the information or resources needed to continue execution. If so, it will unregister itself from the EventMonitor and resume its execution. Otherwise, it will continue to be suspended waiting for events. The current execution state of the ControlMessage is kept so that the object knows the appropriate context when resuming execution.

5.0 Example Usage

Some of the protocols supported by the C++ class library in the Playground distributed programming environment [3] are non-trivial and involve several different types of messages. Some of the messages need to wait for replies at several different stages of their executions. For example, in the lifetime of a ProposedLinkMessage it can wait on up to three different kinds of messages depending on its current execution state. It also knows what types of responses are appropriate for its particular execution state. It will report an error if the response is inappropriate for a given state. Moreover, several protocol negotiations may be occurring simultaneously. In the case of the Playground C++ class library where this pattern is used the only type of events that are of interest are incoming messages. Hence, the EventMonitors in Playground are MsgMonitors.
When a ControlMessage gets to the point where it needs to wait for a reply from another process it suspends itself on a MsgMonitor where replies are received. The ControlMessage keeps track of its execution state internally so that it knows how to resume execution when it processes a reply. When a reply comes in each suspended ControlMessage is asked if it wants to handle the reply. If it does then it resumes its execution based on its current execution state. This cycle of a ControlMessage suspending and resuming itself may happen several times before the message completes its execution.

FIGURE 2. Playground Example

6.0 Applicability:

Use the ControlMessage pattern when you want to:

- enable protocols where messages need to interact with and wait for each other; and
- facilitate the protocol messages crossing machine, process, or thread boundaries; or
- have objects that compete for scarce and/or non-shareable resources. One object starts with the resource. When it is done it relinquishes the resource. This could be noted as an event by an EventMonitor and the EventMonitor can pass this event to any suspended ControlMessages waiting for that resource; or
- process multiple negotiations simultaneously without blocking. Using the ControlMessage pattern, an application can process multiple protocol negotiations at the same time without one of the negotiations blocking all the others. A ControlMessage does not need to run to completion before any other ControlMessage is also allowed to run. Several negotiations, represented by several ControlMessages, may be ongoing concurrently.

Do not use the ControlMessage pattern if you:

- only need simple messages passed between components with no replies or negotiations.
7.0 Structure

8.0 Participants

- Protocol (ConnectionProtocol)
  - creates the actual concrete ControlMessages applicable to a protocol.

- ControlMessage (ControlMessage)
  - defines the interface for all the concrete ControlMessages.

- ConcreteControlMessage (ConnectRequestMessage, ProposedLinkMessage)
  - implements the ControlMessage interface.

- EventMonitor (MsgMonitor)
  - passes events to ControlMessages that are suspended waiting for events.

9.0 Collaborations

- ConcreteControlMessages add themselves to EventMonitors (via the addMonitoring method) when they are interested in incoming events. They remove themselves from the EventMonitors (via the removeMonitoring method) when they are no longer interested in incoming events.

10.0 Consequences

10.1 Benefits

The ControlMessage pattern offers the following benefits:

Separation of Concerns: The Control Message pattern decouples messages from their receivers. This not only alleviates the overhead of tying messages to specific receivers but also accommodates multiple receivers for any one message.
It decouples the object that monitors events (i.e., EventMonitor) from the object that determines interest in events (i.e., ControlMessage). ControlMessages can change the type of events in which they are interested simply by unregistering themselves with one type of EventMonitor and registering with another type of EventMonitor. Additionally, ControlMessages can register themselves with multiple EventMonitors if they are interested in several types of events.

Moreover, the policy of deciding how ControlMessages are notified of events is separated from the processing of that event. The EventMonitor decides how the ControlMessages are notified of an event but the ControlMessage decides how to proceed with execution once the event has been delivered. It is easy to change the policy of how messages are notified without affecting how the event is processed.

**Flexibility:** The Control Message pattern allows ControlMessages to suspend and resume execution any number of times before completing their executions.

**Localization of Functionality:** The Control Message pattern allows ControlMessages to have a life of their own. They need not be managed by any other object and need not have any long-term dependencies to any objects. All the information needed to execute, suspend execution, and resume execution (including selection of pertinent events) is encapsulated within the ControlMessages.

### 10.2 Liabilities

The ControlMessage pattern has the following liabilities:

**Potential Interface Bloat:** The ControlMessage pattern increases the size of the interface for messages due to the extra methods of suspend, unsuspend, resume, and takingOver.

**Dangling Suspended Messages/Message “Leaks”:** Suspended ControlMessages may never resume execution if the events for which they are interested are never passed to them. These messages are still queried when new events arrive which takes up processing time. Timeouts can be used to remove “outdated” suspended messages but there is always the issue of how long the timeout should be.

**Heavyweight Messages:** ControlMessages may be too heavyweight for some applications. Some protocols only need simple messages passed between components. They may not require replies or negotiations.

### 11.0 Implementation

This section describes how to implement the Control Message pattern in C++. The implementation described below is influenced by the Playground distributed programming environment.

- **Determine the applicable protocols:** Each protocol creates certain types of messages. It also supports message interactions. Determine which protocols will involve negotiations of messages or will require messages who executions will be suspended waiting on events. All of the concrete ControlMessages for a relevant protocol are derived from the abstract ControlMessage class. This allows the EventMonitors to treat all concrete ControlMessages uniformly.
• **Define the ControlMessage methods for relevant messages:** Each applicable ControlMessage will need to implement the methods as declared by the ControlMessage interface. Determine how each concrete ControlMessage will suspend its execution - namely to which EventMonitor or EventMonitors will it suspend itself. Determine what applicable state information is needed so that ControlMessages can resume execution appropriately.

The state information is relevant because each negotiation or ControlMessage is not a separate thread. Each ControlMessage must essentially encode its “program counter” in its state. It then saves this “program counter” when it suspends itself and branches to this “program counter” when its execution is resumed.

It may be that not all concrete ControlMessages for a protocol will need to implement all the ControlMessage methods. There may be only certain messages in a protocol that need this capability. The other messages need do nothing. If one of the ControlMessage methods is invoked on a concrete ControlMessage subclass object that did not define that method a compilation error will be generated indicating that the method is declared but not defined.

• **Determine events of interest:** Typically, for any distributed application there are several different types of events that occur. Determine which events will be of interest for the ControlMessage subclasses. Specifically, determine on which events ControlMessages will want to wait.

• **Determine which EventMonitors will handle which events:** Once the relevant events have been determined, the programmer needs to decide how the events will be handled. Specifically, determine which EventMonitors will handle which events. There are several different approaches.

One approach is to have a single EventMonitor handling all events. This may be appropriate for applications that do not anticipate the queue of suspended messages to be very large. If typically there are only a few messages that are waiting for events and there are few events being passed between components, this strategy probably makes the most sense.

However, if there will be several messages waiting for events and many events coming in to a component then having a single EventMonitor may create a processing bottleneck. An undesirable amount of time may be spent in the EventMonitor iterating through all the suspended messages for each incoming event. Additionally, this time is compounded with many events coming in.

A second approach is to have one EventMonitor for each type of event. If there will be many types of events that will be monitored and many events coming in to a component, it may make sense to have a separate EventMonitor for each type of event. This will speed up event dispatching since events will only be passed to messages that are interested in that type of event.

This approach does add some complexity since the incoming events will need to be demultiplexed to their appropriate EventMonitors. Additionally, information will be needed for the incoming events to determine where they should be sent which may increase coupling and reduce information hiding. The Reactor pattern [4] can be helpful in demultiplexing events. Each EventMonitor would be a ConcreteEventHandler in this pattern.

• **Determine default processing for events:** It may be appropriate for a component to receive events where no ControlMessage is waiting for it. For this case it makes sense to define default processing for these events. For example, incoming ControlMessage events may not be replies to other ControlMessages but instead may be initial requests themselves. In this case, it may be appropriate to have these ControlMessages execute. EventMonitors could have the default behavior of telling ControlMessages to run themselves if no other ControlMessages are waiting on them. EventMonitors could also simply return whether or not the event was consumed. Some other object could then handle default processing.
For other components, it may never make sense to have incoming events that do not have ControlMessages waiting for them. In this case, it may be appropriate to ignore the event and optionally report a warning or error.

Variations:

Lists of Event Types: EventMonitors can have lists of different types of suspended ControlMessages that are only applicable for a specific type of event. This assumes certain types of events are only applicable to certain types of ControlMessage subclasses. This can decrease processing time at the cost of coupling the EventMonitors with concrete ControlMessage classes since now EventMonitors must know about specific ControlMessage subclasses.

Event Notification Without Consumption: Some ControlMessages may want to be made aware of an incoming event but are not interested in consuming the event. This may occur for monitoring purposes, for example. These ControlMessages would be notified of the event but would leave it for some other ControlMessage to consume. This can easily be facilitated by adding the monitoring functionality to the ControlMessage’s takingOver method. The ControlMessage could do whatever bookkeeping it wanted to do with the event and the method’s return value would indicate that the event was not consumed.

Event Transformation: Some ControlMessages may want to transform the event and return it to the EventMonitor so that it instead passes the transformed event on to subsequent waiting messages. For example, this may be desirable in the case where events are encrypted and need to be decrypted for further processing. The decrypting ControlMessage checks if the event is decrypted. If it is, it will decrypt it and return it to the EventMonitor to pass along in place of the original event. This can be facilitated by changing the return value of the takingOver method from a boolean to a pointer to an event. If the return value is a NULL pointer the event has been consumed. Otherwise, the returned event would be passed to the remaining waiting ControlMessages.

With this approach there may need to be an ordering placed on suspended ControlMessages. Clearly in the case of a ControlMessage waiting to decrypt applicable events it should be passed any incoming events before other ControlMessages that are expecting decrypted events. When ControlMessages suspend themselves on EventMonitors they can pass a priority. The EventMonitor will then know which suspended messages should be queried first when new events arrive.

Merging ControlMessage and EventMonitor Functionality: A ControlMessage could be its own EventMonitor. This may make sense if there is only ever a single ControlMessage waiting for events at one time. However, it is conceptually cleaner and simpler to separate the two roles into different objects if several ControlMessages wait for the same types of events.

12.0 Sample Code

When an EventMonitor receives an incoming event it queries all its known suspended messages. The EventMonitor class header and the implementation of waitingFor might look something like this:

class EventMonitor {
public:
EventMonitor();
virtual ~EventMonitor();

// Check if the monitor is waiting for this event
bool waitingFor(Event* message);

// Add this message to the list of monitored messages
void addMonitoring(ControlMessage* message);

// Remove this message from the list of monitored messages
void removeMonitoring(ControlMessage* message);

private:
    List<ControlMessage *> msgList_;
};

bool EventMonitor::waitingFor(Event* event)
{
    // Iterate through the list of waiting messages to see if one of them wants
    // to take control of the passed-in event
    for (msgList_.begin(); !msgList_.atEnd(); msgList_++) {
        if ((*msgList_)->takingOver(event)) {
            return true;
        }
    }
    return false;
}

In this code example, the EventMonitor does not handle an unconsumed event. Instead it returns
whether or not the event was consumed. The default behavior is the responsibility of the calling
object.

When a concrete ControlMessage’s takingOver method is called to query it about an incoming
event it might handle the event in the following manner:

bool
ConcreteControlMessage::takingOver(Event* event)
{
    bool takenOver = false;

    if (interestedIn(event)) {
        // Unmonitor this message
        getEventMonitor()->unmonitor(this);

        // Resume running of the control message
        resume(event);

        takenOver = true;
    }
    return takenOver;
}

When a concrete ControlMessage resumes execution it can check its internal state and process
the event accordingly:

void
ConcreteControlMessage::resume(Event* event)
{
    switch (internalState) {
    case INTERNALSTATE1:
        processInternalState1Event(event);
        break;
    case INTERNALSTATE2:
        processInternalState2Event(event);
        break;
    default:
        throw "Bad internal state for ConcreteControlMessage";
        break;
    }
}

In this code example, the valid execution states for the ConcreteControlMessage is denoted by an enumeration.

### 13.0 Known Uses

The ControlMessage Pattern is used in the Playground C++ class library. It is used to negotiate connections between distributed components. *Does TCP do something similar or is it hard coded for specific messages when handshaking?*

### 14.0 Related Patterns

The following patterns relate to the ControlMessage Pattern:

- The *Command* pattern [1] is used to encapsulate protocol requests and replies as objects. A key component of the ControlMessage pattern is to extend Commands to facilitate suspension and resumption of execution.

- The *External Chain of Responsibility* pattern [2] can be used to pass incoming events to potential receivers.

- The *Strategy* pattern [1] can be used to determine how events are handled/dispatched for different types of EventMonitors.

- The *Factory Method* pattern [1] can be used by the Protocols to create appropriate concrete ControlMessages.

- The *Iterator* pattern [1] may be used in the EventMonitor to iterate through the suspended ControlMessages.

### References

