Weaving Crosscutting Concerns into Inter-process Communications (IPC) in *AspectJ*

Ali Raza, Dr. Stephen W. Clyde
Computer Science Department
Utah State University
Logan, Utah, USA
ali.raza@aggiemail.usu.edu

Abstract—Implementing crosscutting concerns for message-based inter-process communications (IPC) are difficult, even using aspect-oriented programming languages (AOPL) such as *AspectJ*. Many of these challenges are because the context of communication-related crosscutting concerns is typically a conversation consisting of message sends and receives. Other challenges stem from the wide variety of IPC mechanisms, their inherent characteristics, and the many ways in which they can be implemented, even using a common communication framework. Additionally, current AOPL do not provide pointcuts for weaving of advice into high-level IPC abstractions like conversations. This paper describes an extension to *AspectJ*, called *CommJ*, with which developers can implement communication-related concerns in cohesive and loosely coupled aspects.

Keywords:modularity; aspect-oriented programming (AOPL); crosscutting concerns; *AspectJ*; software reuse and maintenance.

I. INTRODUCTION

Inter-process communications (IPC) are ubiquitous in today’s software systems, yet they are rarely treated as first-class programming concepts. Instead, developers typically have to implement communication protocols indirectly using primitive operations, such as *connect*, *send*, *receive*, and *close*. The sequencing and timing of these primitive operations can be relatively complex. For example, consider a distributed system that uses the Passive File Transfer Protocol (PFTP) to move large data sets from a client to a server. The server would enable communications by listening for connection requests on a published port, e.g., 21. A client would then initiate a conversation, i.e., an instance of the PFTP protocol, with a connection request to the server on that port. Figure 1 shows a typical sequence of messages following the initial connection request.

Neither the client’s nor the server’s side of the conversation is simple. In fact, to ensure responsiveness for end users and to handle multiple simultaneous clients, both the client and server might execute parts of a single conversation on different threads, making it even harder to follow concurrent conversations dynamically. A system using PFTP could be further complicated by communication-related requirements not central to primary objective of moving large amounts of data, such as logging, detecting network failures, monitoring congestion, and balancing load across redundant servers.

From a communications perspective, these concerns (and many others not listed above) are what Aspect-oriented Software Development (AOSD) refers to as crosscutting concerns, because they pertain to or cut through multiple parts of core or base concepts. Directly implementing these concerns in a typical system can cause the scattering and tangling of code. Scattering occurs when the same or very similar logic exists in multiple places in the software. Tangling occurs when single software component is complicated by logic for secondary concerns. Scattering and tangling often occur together.

AOSD, which first started to appear in the literature in 1997 [12, 25], reduces scattering and tangling of code by encapsulating crosscutting concerns in first-class programming constructions, called aspects [15]. In strongly typed languages, an aspect is an Abstract Data Type (ADT) with all of the same capabilities as an object class. However, an aspect can also contain advice methods that encapsulate logic for addressing crosscutting concerns and pointcuts for describing where and when the advice needs to be executed. More specifically, a pointcut identifies a set of join points, which are temporal places in the execution of the system for where and when weaving of advice takes place [15].

*AspectJ* is an AOPL that extends Java for aspects [14-17]. It allows programmers to weave advice into join points that correspond to constructor calls or executions, methods calls or executions, class attribute references, and exceptions.

It is possible for skilled software developers to create reusable, well-encapsulated crosscutting concerns in a traditional *object-oriented programming language* OOPL.
However, the difference between AOPLs and an OOPL is that AOPLs offer convenient mechanisms for separating crosscutting concerns from core functionality and for following a principle called obliviousness [18]. Although perhaps poorly named, obliviousness is the idea that core functionality should not have to know about crosscutting concerns [13].

The problem is that AspectJ, like other AOPLs, does not support the weaving of advice into high-level communication abstractions, such as conversations. Our work, called CommJ, extends AspectJ so developers can weave crosscutting concerns into IPC in a modular and reusable way, while keeping the core functionality oblivious to those concerns. See Section II for a high-level overview. Section III describes a conceptual model that provides a theoretical foundation for CommJ, namely its message event joint points (see Section IV) and event tracking (see Section V). Section VI describes base aspects central to CommJ’s implementation. To validate CommJ, we have created a library of reusable aspects for common communication crosscutting concerns and have applied them to a variety of sample systems (see Section VII). Then, Section VIII discusses how application programmers can write their own communication aspects. Related work is presented in Section IX. Finally, Section X summarizes the current state of CommJ and outlines our future work.

II. HIGH-LEVEL OVERVIEW

Overall CommJ enables the partitioning of a complex communication problem into manageable cohesive concepts and promotes greater reuse with better maintainability. Figure 2 shows an architectural block diagram that represents relevant conceptual layers and their dependencies. The following paragraphs describe the high-level components and their dependencies.

![CommJ Architectural Block Diagram](image)

In general, a universe model is a formal description of a closed universe of things, as well as their relationships, properties, interactions, and behaviors. Figure 3 shows part of our universe model for IPC, which we refer to as the UMC or Universe Model of Communication. Section III describes a portion of UMC in more detail.

CommJ is an AspectJ library that implements message-event join points and keeps track of conversations. A software developer that wants to use communication-related aspects simply has to include this library. Sections IV - VI explain how CommJ implements the join points, keeps track of conversations, and base abstractions for the application programmers, respectively.

The Reusable Aspect Library (RAL) is a toolkit-like collection of communication aspects that application programmers should find useful for in many different kinds of applications. They include aspects for measuring turn-around times, tracing conversations, and introducing behaviors into complex, multi-step protocols, like PFTP. Section VII describes this library in more detail.

Application-level Aspects are those written by the application programmers, either by using the abstractions provided by CommJ or by specializing the aspects in RAL. Section VIII discusses how these application-level aspects can encapsulate complex crosscutting behaviors in an understandable and maintainable way, without sacrificing obliviousness or flexibility.

III. UNIVERSE MODEL FOR COMMUNICATIONS

The UMC establishes a conceptual framework for discussing and reasoning about network-based communications. Figure 3 shows a portion of this model, namely the part that deals with message concepts. The full UMC includes other concepts, like connections, that we do not discuss here for brevity.

The central idea of the portion presented in Figure 3 is that of a Message Event, which is the “happening” of a message being sent (i.e., Sent Event) or a message being received (i.e., Received Event). It is a time point related to a particular message and is part of a Conversation following a Protocol. Every Received Event must have a corresponding Message Received object, which is simply a message in the role of having been received. Similarly, every Sent Event must have a Message Sent object. Also, consistent with theoretical foundations for IPC [28], all the Message Events in a system form a partial ordering; the events on a single thread are totally ordered; and a message’s Sent Event always comes before its Received Event(s).

Message in Figure 3 is an abstraction that represents data sent from one process to another as part of conversation. Each Message can be associated with at most one send and possible many receive events, which is the case for multicasts or broadcasts. The Message class contains abstract reflection methods for retrieving message identifying information (MII), which consists of message, conversation, and protocol identifiers. Application developers implement these methods for their specific types of messages and then CommJ uses those implementations in keeping track of conversations. Since these methods are abstract and are implemented in the application, developers remain in full control of their message structure.

Even though the UMC focuses on communications, it includes Channel, Thread, Node, and Process classes to help provide context information for the individual messages and conversations.
IV. MESSAGE EVENT JOIN POINTS

Communication join points fall into two general categories: message related and connection related. Since this paper is focusing on Message Events, we only discuss the former here.

As mentioned earlier, join points represent places and times where/when advice can be executed. In AspectJ, they correspond to constructors, methods, attributes, and exceptions. Advice can be executed before, after, or around these various contexts. CommJ adds conversations to the list of possible contexts, but unlike the advice contexts in AspectJ, a conversation is not tied to a single programming language construct. Instead, a context in CommJ can be either:

A - an entire conversation from a process’s perspective (see Figure 4)

B - any sequence of message send or receive events in the conversation as seen by a process

C - a single send or receive event in a conversation

MultiStepConversationJP represents join points for entire conversations, as well as join points for sequences of events within a conversation. RRConversationJP (i.e., request-request conversation join points) also represents join points for complete conversations, but only those that follow request-reply protocols. MultiStepConversationJP could be used for the same, but RRConversationJP includes optimizations for this common type of conversation. SendEventJP and ReceiveEventJP implement join points for individual message events.

A developer can implement crosscutting concerns, define conversation-related pointcuts, and weave advice into any of above join points by specializing the corresponding abstract CommJ aspects, shown in yellow in Figure 5.

V. EVENT TRACKERS AND REGISTRIES

Behind the scenes, CommJ uses JoinPointTrackers, which are monitors [22] that perform pattern matching on communication events, to track individual events and to organize them into high-level conversation contexts. Since the monitoring of communications is itself a crosscutting concern, JoinPointTrackers are implemented as aspects that weave the necessary monitoring logic into places where communication event may take place. Although CommJ can support many different kinds of JoinPointTrackers, Figure 5 only shows one special kind of tracker, namely MessageJoinPointTracker, which has been specifically designed for send and receive events on standard JDK sockets and channels.

When a MessageJoinPointTracker discovers a relevant communication event, it creates a join point instance, e.g., a SendEventJP, correlates it with other events in the same conversation, and then adds it to a registry, namely the MessageJPRegistry shown in Figure 5. Advice in a communication aspect can access these join point objects to obtain context information, like the conversation’s start time, channel, or the protocol.
VI. BASE ASPECTS

All communication aspects are ultimately derived from abstract MessageAspect, which provides concrete pointcuts that dynamically track send and receive events (See Figure 6 for more details). For space considerations, the full definitions of the pointcuts are not shown, and are not necessary for understanding their purpose. However, it is important to note that they take join point objects as parameters, because this is how advice based on these pointcuts can access communication contexts.

```java
public abstract aspect MessageAspect{
    public pointcut MessageSend(SendEventJP jp) ....
    public pointcut MessageRecieve(ReceiveEventJP jp) ....
}
```

Figure 6: Pointcuts in MessageAspect

The four specializations of MessageAspect in Figure 5 correspond to four different kinds of conversation contexts, as mentioned earlier, and extend MessageAspect with pointcut abstractions that are meaningful to those contexts (see Figures 7a-7d). Developers can create their own application-level communication aspects that inherit from these aspects and include advice based on these pointcuts.

The OneWaySendAspect is relatively trivial because it represents a simple one-message conversation from the message sender’s perspective. Similarly, the OneWayReceiveAspect represents a one-message conversation from the message receiver’s perspective.

The RRConversationAspect extends MessageAspect with pointcuts for conversation beginnings and ends. Developers can use this aspect to weave advice before, after, or around simple request-reply conversations, either from a conversation initiator or responder perspective.

The MultistepConversationAspect is the most complex of the four. In addition to pointcuts for conversation beginnings and ends, it provides a way for applications to specify arbitrarily complex communication protocols, which define the message patterns that comprise conversations. A process can participate in a conversation with one or more ProcessRoles. See Figure 8.

The key to working with complex protocols is that an aspect developer can formally define them in terms of ProcessRoles and then ProcessRoles in terms of finite state machines (see State Machine in Figure 9). For example, consider a communication protocol that involves three processes, A, B, and C, and where A starts a conversation by sending a message to B and waits for a response. When A receives a response from B, it then sends a message to C and waits for a response. When A receives a
response from \( C \) it sends a final message to both \( B \) and \( C \). Figure 9 shows a finite state machine for the \textit{A ProcessRole} of this protocol. The \( B \) and \( C \) \textit{ProcessRoles} are considerably simpler and are not shown here.

The \textit{CommJ StateMachine} class includes a \texttt{buildTransitions} method that allows developers to define state machines in terms of states and message-event transitions. Figure 10 shows the implementation of this method to define a \textit{StateMachine} for the sample \textit{ProcessRole} shown in Figure 9.

```
public class SampleProcessRole extendsStateMachine{
    ....
    @Override
    public void buildTransitions(){
        addTransition("Initial", "S", "M1", "WaitingRspFromB");
        addTransition("WaitingRspFromB", "S", "M2", "M3", "WaitingRspFromC");
        addTransition("M1", "S", "M2", "S1", "M3", "S2");
    }
    ....
}
```

Figure 10: State Machine configuration for sample Process Role

VII. REUSABLE ASPECTS LIBRARY

Aspects in the RAL are also derived from the base aspects in \textit{CommJ}. They represent general crosscutting concerns commonly found in applications with significant communication requirements. Table 1 lists some of the aspects currently in the RAL and Figure 11 shows part of the implementation of first one, \textit{TotalTurnAroundTimeMonitor}. Note how the advise in this aspect follows the \textit{Template Method} pattern [29]. This allows developers to quickly adapt it to the specific needs of their application by overriding the \textit{Begin} and \textit{End} methods. Other aspects in the RAL make use of this and other reuse techniques to easily integrate them into existing or new applications.

We expect that RAL will continue to grow as new generally applicable communication aspects are discovered, implemented, and documented.

```
public aspect TotalTurnAroundTimeMonitor
    extends MultistepConversationAspect{
    private long startTotalTime = 0;
    private long turnAroundTime = 0;
    before(MultistepConversationJP jp):
        ConversationBegin(jp){
            startTime = System.currentTimeMillis();
            Begin(jp);
        }
        after(MultistepConversationJP jp): ConversationEnd(jp){
            long turnaroundTime = (System.currentTimeMillis() - startTime)/1000;
            End(multiStepJP);
    }
    public getTurnAroundTime { return turnAroundTime; }
    protected void Begin(MultistepConversationJP jp){
        // Specialization of this aspect should override the method
    }
    protected void End(MultistepConversationJP jp){
        // Specialization of this aspect should override the method
    }
    ....
```

Figure 11: A code snippet of \textit{TurnAroundTimeAspect}

VIII. APPLICATION-LEVEL COMMUNICATION ASPECTS

As mentioned, aspect developers implement and add application-level aspects into core application logic by either reusing RAL aspects or specializing the base aspects in \textit{CommJ}. As an example, this section describes the implementation of an application-level aspect that weaves performance measurements in the multistep protocol,

<table>
<thead>
<tr>
<th>Aspect Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TotalTurnAroundTimeMonitor</td>
<td>Provides virtual helper methods for conversations which help programmers to override RAL aspects in their applications</td>
</tr>
<tr>
<td>MessageLoggingByConversation</td>
<td>Log messages by conversations in a developer-defined format and repository</td>
</tr>
<tr>
<td>MessageEncryption</td>
<td>Add session-level encryption/decryption to communication protocols</td>
</tr>
<tr>
<td>NetworkNoiseSimulator</td>
<td>Allows developers to add noise, message log, and message duplication to network communications, which is useful for system testing</td>
</tr>
<tr>
<td>NetworkLoadBalancer</td>
<td>Helps programmers balance message loads across two more communication channels</td>
</tr>
<tr>
<td>VersionControlAspect</td>
<td>Helps programmers manage multiple version of messages structures for their applications</td>
</tr>
</tbody>
</table>
introduced in the previous section. For discussion
purposes, assume that the performance measurements are a
rolling window of throughput and average-conversation
turn-around time statistics. Also, assume that the core
application considers a unit of work to be the completion
of a conversation that follows this protocol. So, we can
measure throughput for a unit of time, say 1 minute, by
simply counting the number of these conversations
completed in that minute. The average turn-around time is
the average of timespans from conversation start times to
conversation end times. The rolling window keeps track
of these statistics for the current minute and 10 previous
minutes. Figure 12 in the next page shows the key snippets
of an aspect that implement this performance measure
crosscutting concern.

First notice how this advice is derived from TotalTurn-
AroundTimeAspect and in doing so, it can reuse its
implementation of the conversation turnaround time

```
public aspect MyApplPerformanceMonitor
    extends TotalTurnAroundTimeMonitor{

    private Stats[] statsList = new ArrayList[11];
    private int currentStatsIndex = 0;

    @Override
    protected void End(MultistepConversationJP jp) {
        // Get number of elapsed minutes since beginning of current
        Stats long elapsedMinutes = Min(Stats[currentStatsIndex].
        getMinutesSinceStartTime(), 10);
        // Roll Stats window forward, if necessary
        for (int i=0; i<elapsedMinutes; i++) {
            currentStatsIndex++;
            if (currentStatsIndex>10)
                currentStatsIndex=0;
                Stats[currentStatsIndex].Reset();
        }
        currentStats.addCompleteConversation(getTurnaroundTime); }

    class Stats{
        private long start_time;
        private int completeConvCount;
        private double avgTurnaroundTime;

        public Stats{
            Reset();
        }

        public Reset(){
            current_time = current_time;
            completeConvCount = 0;
            avgTurnaroundTime = 0;
        }

        public long getMinutesSinceStartTime() { 
            // using current time, compute and return the number of
            minutes since the start time of this Stats object. A zero means
            we still in the same minute
        }

        public void addCompleteConversation(double
            newTurnaroundTime) {
            avgTurnaroundTime = ((completeConvCount*avgTurnaroundTime) +
                newTurnaroundTime)/(++completedConvCount);
        }
```

Figure 12: performance measure crosscutting concern

concept directly. Then, it adds the Stats array for holding
the rolling window of statistics and some additional
behavior to the ending of a conversation to compute the
statistics.

IX. RELATED WORK

We found many papers that talk about using aspect-
oriented technology for communication-related cross-
cutting concerns, such as replication [5], persistence [9],
synchronization [8, 16], and remote pointcuts [6]. To date,
we have not found any other work that extends the
possible contexts and join points for aspects to
conversations or sequences of events in a specific
conversation. The closest idea discusses the composition
of communication abstractions by separating out definition
of communications from the definition of other aspects [7].
Although this work is of value, we believe that CommJ
enables better modularity while preserving obliviousness.

Marco, et al., describe a Java-based communication
middleware, called AspectJRMI, that applies AOPL
concepts to modular design and the implementation of
RMI. Their primary contribution is the decomposability of
RMI into small crosscutting concerns.

Other related ideas deal with the definition of reusable
communication constructs in languages, like Erlang,
which is based on processes communicating via
asynchronous message passing [26, 21]. However, these
approaches do not inherently encourage the separation of
crosscutting concerns from core application requirements.

Gary, et al., describe an approach for building
customized protocols using Cactus – a system in which
micro-protocols are implementing individual attributes of
transport [1]. More complex protocols can then be
composed from these micro-protocols. Dirk, et al., show
how to separate the definition of communication from the
definition of other system functionality [2]. A paper on
extensible client-server software by Coady, et al., talks
about requiring a clear separation of core services from
those that should be customizable [3]. Remi, et al., talk
about concurrent event-based AOPL and define an
approach of writing concurrent aspects [11]. All these
works address research objectives different from CommJ
and only indirectly related to our research.

X. SUMMARY AND FUTURE WORK

This paper introduced the notion of communication
aspects and discussed an AspectJ framework, i.e., CommJ,
for weaving aspects into inter-process communications. It
then describes the design and implementation of some of
CommJ key components, namely the base communications. It also
provides an overview of a toolkit that consists of reusable
communication aspects and doubles as a proof of concept,
since these aspects can be directly applied to a wide range
of existing applications.

Based on preliminary evidence, we believe that
CommJ is capable of encapsulating a wide range of
communication-related crosscutting concerns in modular
aspects. However, more research and experimental
evidence is needed. We plan to conduct real world
experiments using CommJ to verify its benefits in software reuse and maintenance. We also hope to gather more empirical evidence of CommJ value by increasing the number of aspects in the RAL and by continuing to expand the number and types of applications that use CommJ.

Those interested in trying out CommJ or contributing to it can obtain a copy of the framework from http://commj.cs.usu.edu.

REFERENCES