

NAVAL POSTGRADUATE SCHOOL

Monterey, California



THESIS

**A HIGHLY ADAPTABLE GENERIC EVENT-BASED
MESSAGE CHANNEL DESIGN FOR LOOSELY
COUPLING SOFTWARE MODULES**

by

Cihat Eryigit

March 2002

Thesis Advisor:
Second Reader:

Geoffrey Xie
Chris Eagle

Approved for public release; distribution is unlimited.

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.			
1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE March 2002	3. REPORT TYPE AND DATES COVERED Master's Thesis	
4. TITLE AND SUBTITLE: A HIGHLY ADAPTABLE GENERIC EVENT-BASED MESSAGE CHANNEL DESIGN FOR LOOSELY COUPLING SOFTWARE MODULES			5. FUNDING NUMBERS
6. AUTHOR(S) Eryigit, Cihat			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000			8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) DARPA and NASA			10. SPONSORING/MONITORING AGENCY REPORT NUMBER G417
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.			
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.			12b. DISTRIBUTION CODE Statement A
13. ABSTRACT (maximum 200 words) Component-based software engineering is an emerging software development approach based on the fundamentals of object-oriented technology. This approach moves programmers' focus from component development to component assembly. Event-based programming is one of the techniques that can be used to assemble software components into applications. In this thesis, a new, generic, highly adaptable and flexible event channel has been designed and implemented. The main product is a Java utility package, called "channel package", which should help Java application developers create or enhance large systems using an event-based programming approach. The new channel design has several demonstrated performance advantages over existing event channel implementations. The flexibility and adaptability of the channel package has also been validated by a successful upgrade of the channel mechanism of the SAAM prototype system.			
14. SUBJECT TERMS Event Programming, Event Channel, The Server and Agent based Active network Management (SAAM) Architecture, Inter-object communication.			15. NUMBER OF PAGES
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UL

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited.

**A HIGHLY ADAPTABLE GENERIC EVENT-BASED MESSAGE CHANNEL
DESIGN FOR LOOSELY COUPLING SOFTWARE MODULES**

Cihat Eryigit
Lieutenant Junior Grade, Turkish Navy
B.S., Turkish Naval Academy, 1995

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

**NAVAL POSTGRADUATE SCHOOL
March 2002**

Author: Cihat Eryigit

Approved by: Geoffrey Xie, Advisor

C. S. Eagle, Second Reader

C. S. Eagle, Chairman
Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Component-based software engineering is an emerging software development approach based on the fundamentals of object-oriented technology. This approach moves programmers' focus from component development to component assembly. Event-based programming is one of the techniques that can be used to assemble software components into applications.

In this thesis, a new, generic, highly adaptable and flexible event channel has been designed and implemented. The main product is a Java utility package, called "channel package", which should help Java application developers create or enhance large systems using an event-based programming approach. The new channel design has several demonstrated performance advantages over existing event channel implementations. The flexibility and adaptability of the channel package has also been validated by a successful upgrade of the channel mechanism of the SAAM prototype system.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

I.	INTRODUCTION.....	1
A.	SAAM PROJECT AND SAAM CHANNEL MODEL.....	2
B.	SCOPE OF THIS THESIS.....	3
C.	ORGANIZATION OF THIS THESIS.....	4
II.	BACKGROUND.....	5
A.	JAVA EVENT MODEL.....	5
1.	Java 1.0 Event Model.....	5
2.	Java 1.1 Event Model.....	7
B.	CORBA EVENT MODEL.....	11
1.	Object Management Group (OMG).....	11
2.	Corba.....	11
3.	CORBA Event Service.....	13
a.	<i>The Push Model</i>	14
b.	<i>The Pull Model</i>	15
c.	<i>Mixing the Push and Pull Model in a Single System</i>	15
d.	<i>Types of Event Communication</i>	16
4.	CORBA Notification Service.....	17
a.	<i>Filter</i>	17
b.	<i>Quality of Service (QoS)</i>	18
c.	<i>Structured Events</i>	19
5.	Conclusion.....	20
C.	SAAM EVENT CHANNEL MODEL.....	20
1.	The Channel.....	21
III.	CHANNEL PERFORMANCE METRICS AND EVALUATION OF EXISTING SAAM CHANNEL MODEL.....	24
A.	CHANNEL PERFORMANCE METRICS AND QUALITATIVE OBJECTITIVES.....	24
1.	Channel Throughput (R).....	24
2.	Channel Work Rate (Wr).....	25
3.	Channel Access Delay (D).....	25
4.	Event Talk Time (T_i).....	25
5.	Thread Count (C).....	26
6.	Adaptability of Channel.....	27
7.	Scalability of Channel.....	28
8.	Manageability of Channel.....	28
9.	Functional Flexibility.....	28
10.	Ease of Use.....	29
a.	<i>Easy Creation of Channel</i>	29
b.	<i>Meaningful Method Names</i>	29
c.	<i>Easy Configuration of Channel</i>	29

	<i>d.</i>	<i>Sufficient and Understandable Method Overriding</i>	29
	<i>e.</i>	<i>Conflicts Between Methods</i>	29
	<i>f.</i>	<i>Conflicts Between Parameters in a Method</i>	30
	<i>g.</i>	<i>Documentation</i>	30
B.		TEST OF SAAM CHANNEL	30
	1.	Channel Throughput and Work Rate	30
	2.	Channel Access Delay	32
	3.	Event Talk Time	34
	4.	Thread Count	35
	5.	Manageability and Scalability of SAAM Channel	36
	6.	Adaptability and Functional Flexibility of SAAM Channel	37
	7.	Ease of Use of SAAM Channel	38
IV.		NEW CHANNEL MODEL AND CHANNEL PACKAGE	39
A.		INTRODUCTION	39
B.		FEATURES OF THE NEW CHANNEL	39
	1.	Generic	39
	2.	Event Buffering	39
	3.	Event and Channel Participant Priority	40
	4.	Event Delivery Order	40
	5.	Event Filtering	40
	6.	Self-Dispatching	40
	7.	Duplex Communication	40
	8.	Concatenating Channels	41
C.		CHANNEL PACKAGE	41
	1.	ChannelEvent Class	41
	2.	ChannelListener Interface	42
	3.	ChannelFilter Interface	43
	4.	ChannelScheduler Interface	44
	5.	ChannelListenerItem Class	45
	6.	ChannelEventFilter Class	46
	7.	Schedulers	49
	<i>a.</i>	<i>FIFOScheduler Class</i>	49
	<i>b.</i>	<i>PerTalker_RR_Scheduler Class</i>	49
	<i>c.</i>	<i>PriorityScheduler Class</i>	49
	8.	Channel Class	50
	<i>a.</i>	<i>Creating a Channel Object</i>	50
	<i>b.</i>	<i>Adding and Removing Listeners to Channel</i>	50
	<i>c.</i>	<i>Adding and Removing Talkers to Channel</i>	51
	<i>d.</i>	<i>Talking on Channel</i>	51
	<i>e.</i>	<i>Event Dispatching</i>	52
	<i>f.</i>	<i>Event Filtering</i>	53
	<i>g.</i>	<i>Listener Self-Dispatching</i>	54
	<i>h.</i>	<i>Duplex Communication</i>	54
	<i>i.</i>	<i>Concatenating Channels</i>	55
	9.	ChannelAccessAuthority Interface	55

10.	ChannelManager Class.....	56
V.	TEST AND RESULTS.....	59
A.	EVALUATION OF NEW CHANNEL DESIGN	60
1.	Channel Throughput and Work Rate.....	60
2.	Channel Access Delay and Event Talk Time.....	62
3.	Scalability and Thread Count.....	63
4.	Adaptability and Functional Flexibility of New Channel.....	64
5.	Manageability	64
6.	Ease of Use	65
a.	<i>Easy Channel Management</i>	65
b.	<i>Easy Self-Dispatching</i>	65
B.	INTEGRATION OF CHANNEL PACKAGE AND SAAM PROTOTYPE.....	66
1.	Removing Obsolete Classes and Interfaces.....	66
2.	PermissionTableEntry Class	67
3.	Changes to ControlExecutive Class.....	68
4.	Reducing the Number of Channels.....	68
5.	Event Priorities.....	71
6.	Additions to SAAM GUI.....	71
VI.	CONCLUSION.....	73
A.	LESSONS LEARNED	73
1.	Programming with Threads	73
2.	Integration with SAAM Prototype	73
B.	FUTURE WORK	74
1.	Communication Between Distributed Applications.....	74
2.	Automatic Sense for Self-Dispatching.....	74
	LIST OF REFERENCES	75
	INITIAL DISTRIBUTION LIST	77

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF FIGURES

Figure 2.1	Button Event Process in Java 1.0 Event Model	6
Figure 2.2	Event Life Cycles in Java 1.1	9
Figure 2.3	Java Event Class Hierarchies	11
Figure 2.4	ORB-to-ORB Communication	12
Figure 2.5	Push Model.....	14
Figure 2.6	Pull Model.....	15
Figure 2.7	Mixed Model.....	16
Figure 2.8	SAAM Channel Concept.....	22
Figure 3.1	Test Bed For Studying Impact of Listener Event Handling Time on Channel Throughput.....	31
Figure 3.2	SAAM Channel Throughput versus Listener Event Handling Time	31
Figure 3.3	Test Bed for Measuring SAAM Channel Access Delay	32
Figure 3.4	Average Access Delay versus Number of Channel Talkers	33
Figure 3.5	SAAM Channel Access Delay versus Event Delivery Time	33
Figure 3.6	Event Talk Time Versus The Number of Talkers.....	34
Figure 3.7	Test Bed for Measuring SAAM Channel Event Talk Time.....	35
Figure 3.8	Event Talk Time Versus Event Handling Time	35
Figure 3.9	Event Concatenation in SAAM Channels.....	37
Figure 3.10	Two-way communication with SAAM channel.....	37
Figure 4.1	Channel Event Structure.....	42
Figure 4.2	Channel Scheduler.....	45
Figure 4.3	Channel Listener Item	45
Figure 4.4	Self-dispatching.....	46
Figure 4.5	ChannelEventFilter Filtering Process	48
Figure 4.6	Per Talker Round-Robin Scheduler	49
Figure 4.7	Channel Event Dispatching.....	53
Figure 4.8	Channel Event Filtering Process	54
Figure 4.9	Two-way Event Communication	55
Figure 4.10	Concatenating Channels.....	55
Figure 5.1	Test Bed For Studying Impact of Self-Dispatching on Channel Throughput.....	60
Figure 5.2a	Effect of Self-Dispatching on Channel Throughput	61
Figure 5.2b	Effect of Self-Dispatching on Channel Work Rate.....	61
Figure 5.3	Event Talk Time versus Number of Talkers	63
Figure 5.4	Channel Access Delay versus Number of Talkers.....	63
Figure 5.5	Encapsulating SAAM Events in ChannelEvent Structure	67
Figure 5.6	Necessary Channels for A New Interface on Existing SAAM Prototype.....	69
Figure 5.7	Snapshot of New Channel Debug Window.....	72

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

Table 5.1	New Channel Structure in SAAM Prototype	70
-----------	---	----

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to Professor Geoffrey Xie and LCDR Chris Eagle for their guidance and help throughout this thesis research.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

Component oriented programming is an emerging technology based on the fundamentals of object-oriented technology. Component-oriented programming and design patterns have revolutionized software development. With the decomposition of monolithically built systems into independently alterable and extensible components, methodologies from other engineering disciplines are being successfully applied to software development. [Ref 15]

Assembly of components is becoming an increasingly common approach to development in the software industry. This is due in large part to its promise of considerably more productive programming. This increase in productivity results from a greater degree of independence between the development of one component and that of the other components, as well as individual component independence from the development of the larger overall assembly itself. Users of a component need only be concerned with its interface — the abstraction, or functionality, it provides — and need not be aware of the implementation underlying that function. Developers of a component need not be aware of the bigger picture of its use — their implementation simply needs to fulfill the functional obligations of the contract constituted by its interface. Different components can be developed by different teams, at different times, at different vendors or enterprises. [Ref 16]

Event-based programming is one of the techniques that can be used to assemble software components into applications. In this approach, components exchange information in an asynchronous manner. A component that produces output data does not communicate directly with the data consumer(s) to move the data. Instead, it calls up an independent delivery service (typically a system mechanism) for the data transport. The delivery service requires the data, along with pertinent information about its producer and consumer(s), to be encapsulated in generic container objects called events. A data-producing component is also referred to as an event source, a consumer is referred to as

an event listener, and the delivery service as an event dispatcher. For ease of identification and handling, events are often classified based on functionality into different types, with events of the same type tagged by the same unique identifier. Thus, a component can be an event source, a listener, or both, albeit for different event types.

Software components should be coupled as loosely as possible to maximize system modularity and to support dynamic update and replacement of components. However, in existing event handling models, event sources and event listeners are often tightly coupled. For example, the Java Event Delegation model requires that an event listener must be registered with a specific event source to be notified of events generated by that event source. This methodology results in a direct linkage between an event source and its listeners. As such, it is cumbersome to replace the event source, having to reestablish the linkage.

Another drawback of existing event models is that they are generally designed for communication between GUI components and event handlers. This causes limitations and problems when we try to adapt those event models to communication between software components other than GUI components.

A. SAAM PROJECT AND SAAM CHANNEL MODEL

The goal of the Server and Agent based Active network Management (SAAM) project is to generate a solution that will provide a guaranteed quality of service (QoS) while maintaining the relative simplicity and robustness of the underlying TCP/IP architecture. SAAM seeks to provide this QoS by introducing an additional network service that, when requested, can assist applications by reserving the necessary network resources. The SAAM architecture is designed to allow network engineers to incrementally replace existing internal infrastructure. [Ref 1]

A Java-based SAAM prototype has been built incrementally. The prototype consists of modular, self-aware, agent-based components. These components are loosely coupled, based on a novel event model called the SAAM Channel Model. In this model, the event delivery mechanism is implemented in the user domain and in the form of application-specific event channels. Each channel serves as a public communication

medium for a particular set of components that perform a particular set of functions. This association of channel with functionality eliminates the need for any direct linkage between event source and listener.

The current SAAM channel model was implemented by NPS graduates Dean Vrable and John Yarger in their thesis “The Server and Agent based Active network Management (SAAM) Architecture: Enabling Integrated Services”. The implementation provides event dispatching between SAAM components. The SAAM channel model offers several benefits over the standard Java Delegation Event Model by allowing delivery of events from one or more event sources (called talkers in the SAAM channel model) to one or more event listeners and by allowing event listeners to register with a channel that has no talkers. The details of SAAM Channel model will be described in Chapter 2.

However, the current SAAM Channel implementation has several major shortcomings. It is not a general-purpose utility as it supports only an application-specific event object. It has no event buffering capability and the performance is not optimized. For example, when an event is generated the thread of the event source is blocked unnecessarily until that event is delivered to all listeners. Further, the model does not support quality of service nor give a programmer a choice of dispatching events to multiple listeners in a specific order.

B. SCOPE OF THIS THESIS

The primary goal of this thesis is to identify and address the shortcomings of the current SAAM channel implementation. The thesis develops a highly configurable channel to loosely couple software components, following the event-based programming paradigm. This channel implementation supports multiple types of event objects and allows a programmer to choose different event dispatching schedules based on event priorities and the properties of the channel participants. Several performance metrics are defined for evaluating different event channel implementations, and the performance of the new channel implementation is optimized with respect to those metrics.

C. ORGANIZATION OF THIS THESIS

The remaining part of this thesis is organized into the following chapters.

- Chapter II: Background. Provides information about the Java Event Model, the CORBA Event and Notification Service, and the SAAM Channel Model.
- Chapter III: Channel Performance Metrics and Evaluation Of Existing SAAM Channel Implementation. Defines and explains the channel performance metrics used in this thesis and shows some experimental results on the performance of the existing SAAM channel implementation measured by these metrics.
- Chapter IV: New Channel Design and Java Channel Package. Describes the new channel design and implementation, and explains the classes and interfaces in the resulting Java channel package.
- Chapter V: Tests and Results. Presents results of performance tests of the new channel implementation and describes the integration of the new Java channel package into the existing SAAM prototype.
- Chapter VI: Conclusion. Summarizes the results from this thesis and outlines possible future work.

II. BACKGROUND

It is useful to consider some related event models before describing the new channel design. The Java event model, the CORBA event service, and the current SAAM channel model are described here.

A. JAVA EVENT MODEL

1. Java 1.0 Event Model

The Java 1.0 event model provided a methodology for processing events implemented through Java 1.0.2. It was superseded by the Java 1.1 event model with the release of JDK 1.1 in 1997.

Under the Java 1.0 event model, every type of event is encapsulated in a single class, the `Event` class, which is contained in the `java.awt` package. An event object contains information about the type of event it represents, when the event was generated, where the event occurred, and what keys were pressed when the event was generated.

In Java 1.0 event model, event types are limited and an event was delivered only to the component that generated it or one of its parent containers. The event is handed to the component's *handleEvent* method. The *handleEvent* method calls an event handler method based on the type of event it is processing. For example, if an action event is sent to the *handleEvent* method, the method will in turn call the *action* method. The event handler method will return a value of `true` if the event was completely processed or `false` if it was not. If the *handleEvent* cannot dispatch the event, the event is automatically forwarded to the component's container. If the container does not handle the event, it passes the event on to its parent container, and so on. In this way, events are propagated up the containment hierarchy until they are either consumed or reach the top level. If an event reaches the top level and a proper handler is still not found, the event will be discarded. Figure 2.1 illustrates how a button event would be processed with the Java 1.0 event model.

The system provides a default implementation for each of the event handling methods. The default implementations do nothing and return the value `false`. It is straightforward to override one or more of these methods to provide the desired event-processing methodology.

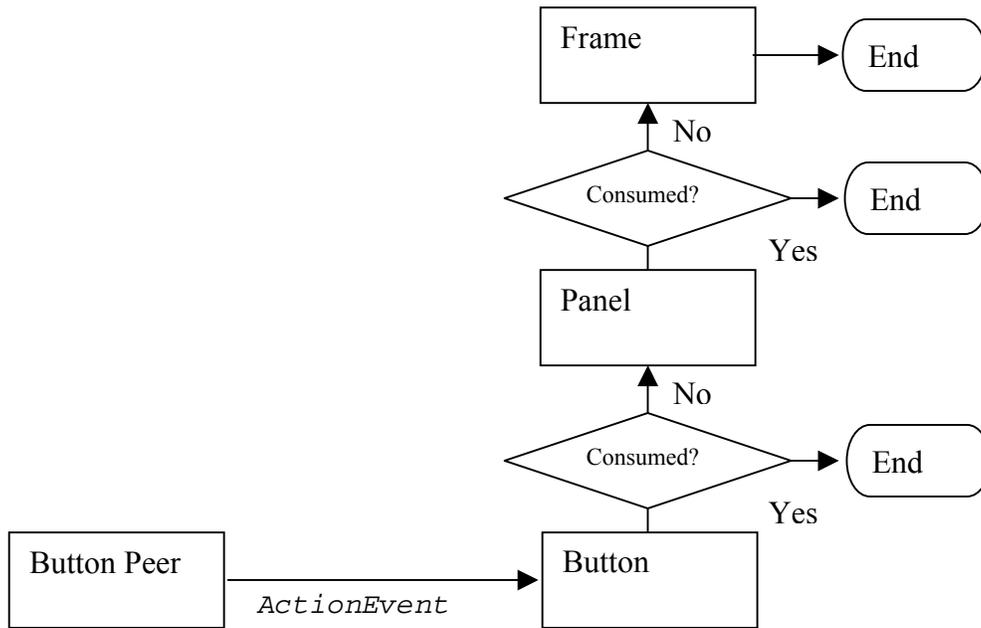


Figure 2.1 Button Event Process in Java 1.0 Event Model

One of the drawbacks of the Java 1.0 event model is that the event handler methods are defined in the `Component` class, and as such any `Component` subclasses can override these methods. This means the event handling tasks must be performed by the GUI components themselves. It is very difficult to decouple the event handling code from the GUI code.

Another drawback of this model is there is no filtering of events. Events are always delivered to the components regardless of whether the components actually handle them or not. This can be a serious performance problem, particularly with high-frequency event types. [Ref 4]

There is also no way to use this event model for inter-object communication because the model restricts an event within the scope of one `Component` class object.

2. Java 1.1 Event Model

Deficiencies of the Java 1.0 event model became readily apparent. It wasn't very efficient to have the system search for the target component every time an event was generated. Another problem was the event handling code was explicitly tied to the GUI code creating inheritance problems. For example, instead of using one `Button` class for all button objects, an application might have to create a `Button` subclass that would quit when pressed, another `Button` subclass that would open a file when pressed, and so on.

One of the significant changes from Java 1.0 to 1.1 was the way events were handled. The Java 1.1 event model employs a concept called *delegation* and cleans up many deficiencies of the Java 1.0 event model. An event source generates the event and then *delegates* the event handling process to another piece of code. The event-handling object can be completely separate from the event source. Under this model any class can serve as the event handler. [Ref 4]

An event object is an instance of a subclass of `java.util.EventObject`; it holds information about the event. The `EventObject` class serves mainly to identify event objects; the only information it contains is a reference to the event source (the object that generated the event). [Ref 3]

In the Java 1.1 event model, an event source delivers an event only to registered listener objects. Listeners that are not registered with the event source do not receive that event. For each of the events an object can generate it maintains a list of listeners that are registered to receive the event. The object registers a listener by adding a reference to the listener to the proper listener list(s). This is done using one of the object's *add-listener* methods, passing the method a reference to the listener object as an argument. An object deregisters or disconnects from an event listener by removing the listener from its listener list(s). This is accomplished by calling one of the object's *remove-listener* methods. [Ref 4]

An event is delivered by passing it as an argument to the receiving object's event handler method. `ActionEvents`, for example, are always delivered to a method called

actionPerformed in the receiver. For each type of event, there is a corresponding listener interface that prescribes the method(s) an event handler must provide to receive the event. In this case, any object that receives `ActionEvents` must implement the `ActionListener` interface.

All listener interfaces are subinterfaces of the `java.util.EventListener`, which is an empty interface. It exists only to help the compiler identify listener interfaces. Listener interfaces are required for a number of reasons. First, they help to identify objects that are capable of receiving a given type of event. This way we can give the event handler methods friendly, descriptive names and still make it easy for documentation, tools, and humans to recognize them in a class. Second, listener interfaces are useful because several methods can be specified for an event listener. For example, the `FocusListener` interface contains two methods:

- `abstract void focusGained (FocusEvent event)`
- `abstract void focusLost (FocusEvent event)`

Although these methods both take a `FocusEvent` as an argument, they correspond to different reasons for firing the event. In the example, the reason is whether the `FocusEvent` received or lost the focus. Even though the event parameter passed to the methods contains enough information to determine whether the focus was gained or lost, requiring two methods, the `FocusListener` interface minimizes the effort. Specifically, if the *focusGained* method is called, it indicates that the event type was `FOCUS_GAINED`. [Ref 3]

Under the Java 1.0 event model, the dispatching and processing of events was a linear process. An event was sent to a single target component. If the event was not completely processed by that component, it could be sent to another component, but the system was not able to broadcast an event to multiple event listeners simultaneously. Under the Java 1.1 event model, events can be sent to any number of event handler objects. Any listener class that is registered with the source component would receive the event.

Figure 2.2 shows the life cycle for events that are subclasses of the `AWTEvent` class. The `dispatchEvent` and `processEvent` methods take an `AWTEvent` object as an argument. However, many of the event classes contained in the `java.swing.event` package are not subclasses of `AWTEvent`, but instead they inherit directly from `EventObject`. The objects that generate these events will also define a `fireEvent` method that causes a given event to be delivered to all listeners in the event's listener list.

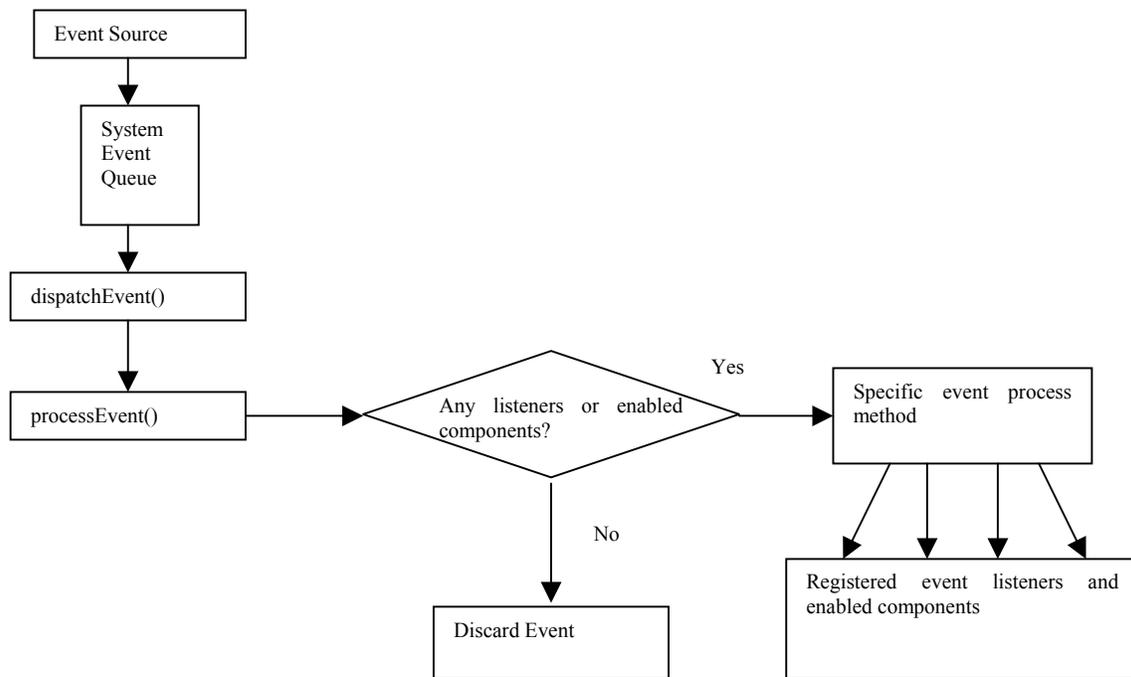


Figure 2.2 Event Life Cycles in Java 1.1

A system event queue is used to store newly generated events until they are dispatched by the `dispatchEvent` method. Normally, the operation of the system event queue are transparent to the user. It does what it does in the background, automatically. The `EventQueue` class encapsulates the Java event queue. It is possible to look into, and even manipulate, the system event queue via the `EventQueue` class.

All event-handling code executes in a unique thread, called the *event-dispatching thread*. It is the event-dispatching thread that calls any event listener methods. The event-dispatching thread retrieves and processes events from the system event queue in a First-

In-First-Out (FIFO) fashion. It finishes executing an event handler's code before invoking the next event handler. One reason this is done is to keep the component in sync with the component display and to block other activities on a component while an event is being processed. For instance, when a button is pressed it will appear to sink into its container display and its color will change. While the `ActionEvent` that is generated is being processed, the event-dispatching thread will block any other user interactions with the button.

The Java API provides a large selection of event and listener classes. It also provides the basic building blocks for creating application-specific event classes and event listeners. One can define a new event class either from scratch, using the high-level event superclasses, `EventObject` and `AWTEvent`, or write a subclass of an existing `EventObject` or `AWTEvent` subclass. The event and support class hierarchy is shown in Figure 2.3 [Ref 4].

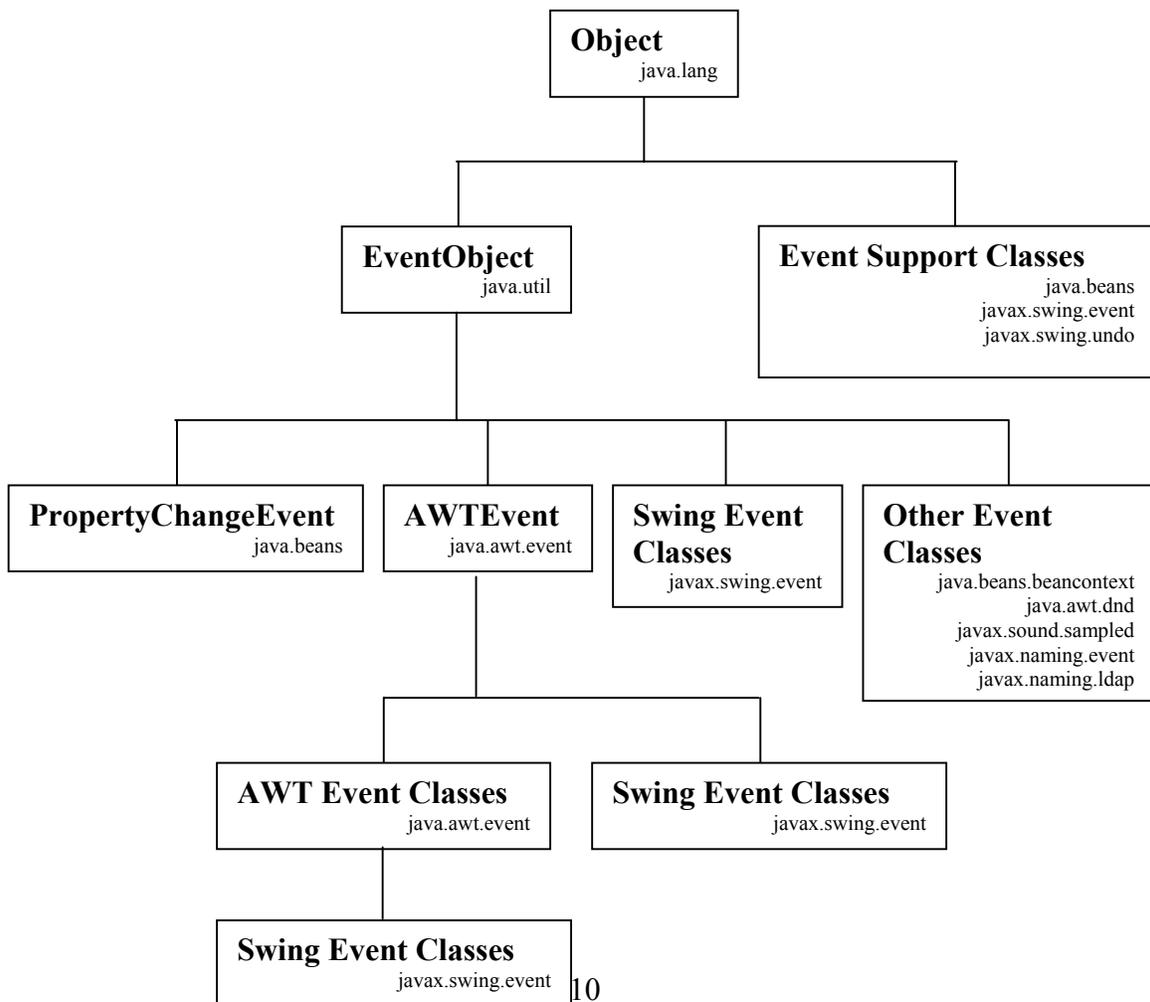


Figure 2.3 Java Event Class Hierarchy

B. CORBA EVENT MODEL

1. Object Management Group (OMG)

The Object Management Group is an international organization supported by over 800 members including information system vendors, software developers, and users. It was founded in 1989. OMG promotes the theory and practice of object-oriented technology in software development. The organization's charter includes the establishment of industry guidelines and object management specifications to provide a common framework for application development. The primary goal is to achieve high reusability, portability, and interoperability of object-based software in distributed, heterogeneous environments. Conformance to these specifications makes it possible to develop a heterogeneous applications environment across all major hardware platforms and operating systems. [Ref 6]

2. Corba

CORBA is the acronym for Common Object Request Broker Architecture, OMG's open, vendor-independent architecture and infrastructure established to enable object-oriented computer applications to work together over networks. CORBA is the Object Management Group's answer to the need for interoperability among rapidly proliferating hardware and software products. It allows applications to communicate with one another no matter where they are located or who has designed them.

The CORBA specification only defines a set of conventions and protocols that must be followed by CORBA implementations. It is left to vendors and developers to translate this specification into a working implementation. CORBA does not make any restriction on language usage or underlying operating systems.

Because CORBA is language independent, it relies on an *Interface Definition Language (IDL)* to express how clients will make a request for a service.

A client communicates to a server object through an object reference. This is a pointer to the object that allows for operations and data access requests to be sent from the client to the server via an *Object Request Broker (ORB)*.

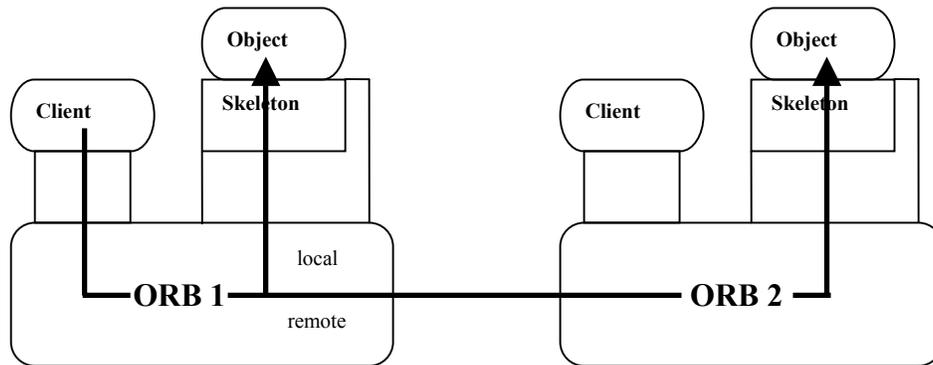


Figure 2.4 ORB-to-ORB Communication

An ORB knows whether an incoming request should be routed to local implementations or to another ORB running on a different machine. When a request reaches the ORB for which it is intended, the request is passed to an object adapter. The *Portable Object Adapter (POA)* forms a link between an object's implementation and its presence (reference) on the ORB.

It is the IDL compiler's job to turn IDL definitions into stub and skeleton files. The stubs and skeletons are all language and ORB dependent. In a case in which one is developing a heterogeneous application, the same IDL file may be used to generate the stubs and skeletons for each language and ORB implementation.

The stubs generated by the compiler will be used by the client processes to communicate with the server. A skeleton file is the companion of a stub. It is the skeleton's job to receive requests from the ORB, call the proper implementation, and return the results. [Ref 5]

3. CORBA Event Service

A standard CORBA request results in synchronous execution of an operation by an object. If the operation defines parameters or return values, data is communicated

between the client and the server. In some scenarios, a more decoupled communication model between objects is required. For example, several documents are linked to a spreadsheet. The documents are interested in knowing when the values of certain cells have changed. When the value of one of the cell changes, the documents update their presentations accordingly. Furthermore, if a document is unavailable because of a failure, it is still interested in any changes to the cells and wants to be notified of those changes when it recovers. [Ref 6]

The CORBA event service decouples the communication between objects. The event service defines two roles for objects: the supplier role and the consumer role. Suppliers produce event data and consumers process event data. Event data are communicated between suppliers and consumers by issuing standard CORBA requests.

There are two approaches to initiating event communication in CORBA. They are called the *push model* and *pull model*. The push model allows a supplier of events to initiate the transfer of the event data to consumers. The pull model allows a consumer of events to request the event data from a supplier. In the push model, the supplier is taking the initiative; in the pull model, the consumer is taking the initiative.

An event channel is an intervening object that allows multiple suppliers to communicate with multiple consumers asynchronously. An event channel is both a supplier and a consumer of events. Event channels are standard CORBA objects and communication with an event channel can be accomplished using standard CORBA requests. An event channel mediates the transfer of events between the suppliers and consumers as follows:

- The event channel allows consumers to register interest in events, and stores this registration information
- The channel accepts incoming events from suppliers.
- The channel forwards supplier-generated events to registered consumers.

Suppliers and consumers connect via the event channel and not directly to each other. From a supplier's perspective, the event channel appears as a single consumer; from a consumer's perspective, the event channel appears as a single supplier. In this way, the event channel decouples supplier and consumers.

Any number of suppliers can issue events to any number of consumers using a single event channel. There is no correlation between the number of suppliers and the number of consumers, and new suppliers or consumers can connect to more than one event channel.

a. The Push Model

In the push model a supplier generates events and actively passes them to a consumer. In this model a consumer passively waits for events to arrive.

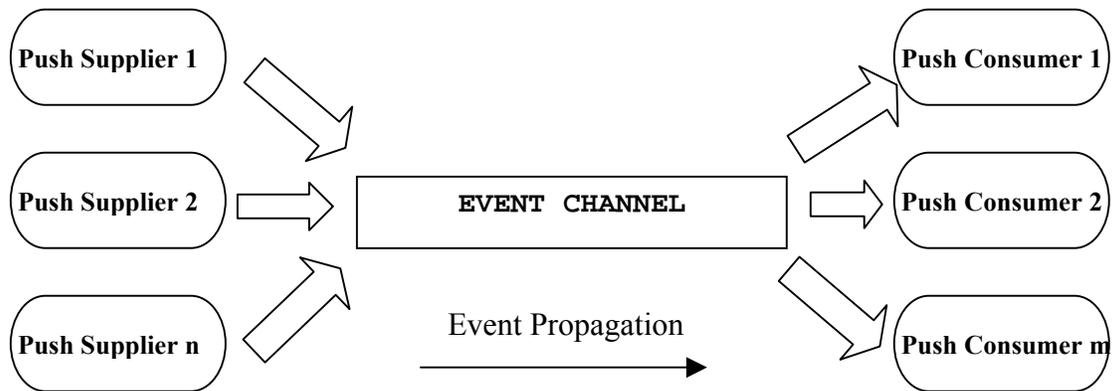


Figure 2.5 Push Model

In this architecture, a supplier initiates the transfer of an event by invoking an IDL operation on an object in the event channel. The event channel invokes a similar operation on an object in each consumer that has registered with the channel.

b. The Pull Model

In the pull model, a consumer actively requests that a supplier generate an event. In this model, the supplier waits for a pull requests to arrive. When a pull request arrives, event data is generated by the supplier and returned to the pulling consumer.

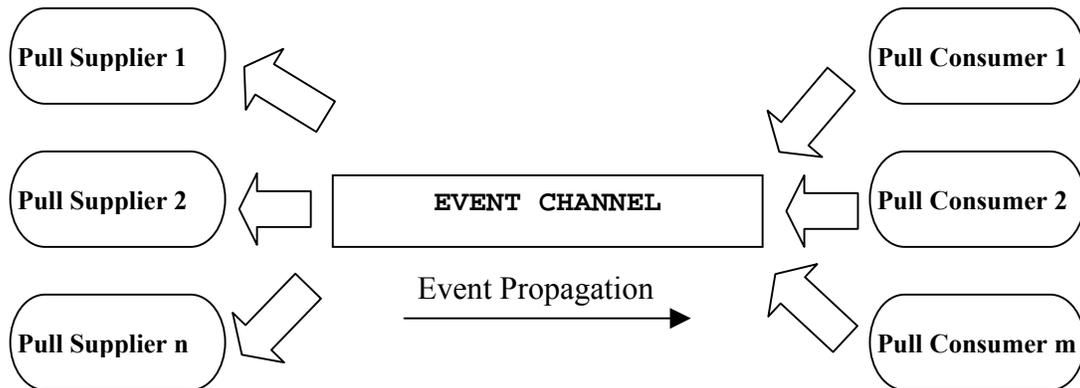


Figure 2.6 Pull Model

In this architecture, a consumer initiates the transfer of an event by invoking an IDL operation on an object in the event channel application. The event channel then invokes a similar operation on an object in each supplier. The event data is returned from suppliers to the event channel and then from the event channel to the consumer, which initiated the transfer.

c. Mixing the Push and Pull Model in a Single System

Because suppliers and consumers are completely decoupled by an event channel, the *Push* and *Pull* models can be mixed in a single system. For example, suppliers may connect to an event channel using the *Push* model, while consumers connect using the *Pull* model, as shown in Figure 2.7.

In this case, both suppliers and consumers must participate in initiating event transfer. A supplier invokes an operation on an object in the event channel to transfer an event to the channel. A consumer then invokes another operation on an event channel object to transfer the event data from channel. Unlike the case in which consumers connect using the *Pull* model, the event channel takes no initiative in forwarding the event. The event channel stores events supplied by the push supplier until some pull consumer requests an event, or until a push consumer connects to the channel.

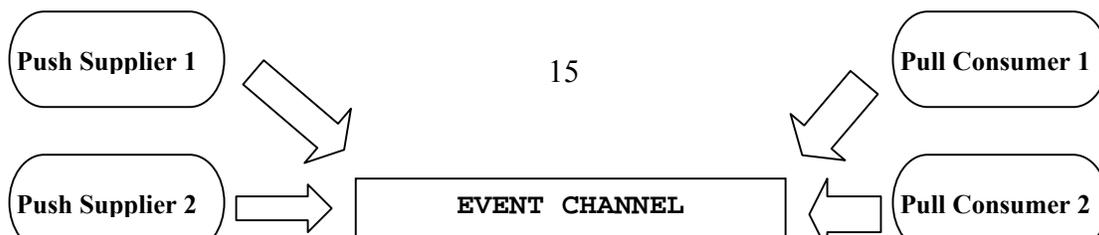


Figure 2.7 Mixed Model

d. Types of Event Communication

The CORBA Event Service maps an event to a successfully completed sequence of operation calls. The operation and the sequence of calls are clearly defined for both *Push* and *Pull* models, and data related to an event can be passed as operation parameters or return values. Event communication can take one of two forms, *typed* or *untyped*.

(1) **Untyped Event Communication.** In untyped event communication, an event is propagated by a series of generic *push* or *pull* operation calls. The *push* operation takes a single parameter, which stores the event data. The event data parameter is of type `any`, which allows any IDL defined data type to be passed between suppliers and consumers. The *pull* operation has no parameters but transmits event data in its return value, which is also of type `any`. Clearly, in both cases, the supplier and consumer applications must agree on the contents of the `any` parameter and return value if this data is to be useful.

(2) **Typed Event Communication.** In typed event communication, a programmer defines application-specific IDL interfaces through which events are propagated. Rather than using *push* and *pull* operations and transmitting data using an `any` argument, a programmer defines an interface that suppliers and consumers use for the purpose of event communication. The operation defined on the interface may contain parameters defined in any suitable IDL data type. In the *Push* model, event

communication is initiated simply by invoking operations defined on this interface. The *Pull* model is more complex because event communication is initiated by invoking operations on an interface that is specially constructed from application-specific interface that the programmer defines. Event communication is initiated by invoking operation on the constructed interface. [Ref 8]

4. CORBA Notification Service

CORBA Event Service falls short in several areas. There is no Quality of Service (QoS) support that can be utilized to set the certitude of notification transmission. Many times, event notifications need to be processed so consumers can meet requirements of reliability, priority, ordering, and timeliness. Also, there is no capability to filter events so a consumer will only get a subset of events. Moreover, existing Event Service implementations are usually filled with proprietary mechanisms. Once a set of events is implemented within a system, it becomes impossible to integrate those events with another application that uses the same set of events but a different Event Service implementation. In order to make this service more transferable, progress has been made to achieve greater agreement on implementing some of these mechanisms that have become proprietary. There have been efforts to enhance the functionality and interfaces of the Event Service. The enhancements have resulted in a new CORBA service -- the Notification Service.

a. Filter

One of the most promising additions to the Notification Service is filtering. Filters allow consumers to subscribe to particular events by matching events to be delivered to them against constraint expressions. This allows the Notification Service to cut down on traffic and improve the scalability of CORBA event handling. Filters are CORBA objects that enable interfaces to add, remove, and modify constraints that match event message values. Constraints refer to variables that are bound to parts of the event notification message and are specified using event types and/or expressions written in a constraint language.

b. Quality of Service (QoS)

The Notification Service provides better support for QoS by defining standard interfaces that allow control over the notification delivery's Quality of Service characteristics. Name/value pairs are used to represent service characteristics at different levels of the protocol stack. Each characteristic has some impact on notification delivery. Therefore, at the channel, one may want to set a discard policy for determining which notifications are thrown away as resource limits are breached. Or one may want to set a maximum number of notifications that may be queued for a single consumer. At the message level, one may want to set the reliability of event delivery. This could be a "best effort" value, where no guarantees of delivery are made, or it could be set as "persistent," in which case the Notification Service stores the notification until the connection is re-established. Some of the Quality of Service properties defined by the specification include:

- **Discard policy:** Specifies policies for discarding events when the queues are full.
- **Earliest delivery time:** Specifies how long an event is to be held in the channel before it is delivered.
- **Expiration time:** Indicates the time range in which an event is valid. If an event is not delivered within a specified time then an event channel should discard it.
- **Maximum events per consumer:** Provides an upper bound to the number of events the channel will queue on behalf of a given consumer. This QoS value helps to relieve the pressure that can grow within a channel because of misbehaving consumers that prevent some of the events from being consumed as scheduled. A huge number of notifications that are constantly being saved and queued for a single misbehaving consumer can use up a great deal of valuable system resources that would otherwise be available for well-behaved consumers.

- **Order policy:** Specifies the order in which notifications are buffered for delivery.
- **Priority:** Specifies the order in which events are delivered to consumers so that more important events take precedence.
- **Reliability:** Is linked to both event reliability and connection reliability to specify fault-tolerance properties. If these properties are in place, the Notification Service will reconnect to all its clients and deliver all non-expired events to its consumers after a crash.

c. *Structured Events*

With all this added information for each event, there is a need for a general event structure, in which a wide variety of event data can be stored. The structured event is a more strongly typed event message, known to the Notification Service and its clients. This structure will allow more efficient processing of the event notification.

Structured events consist of a header and a body. The header contains two sections. The first has fixed information such as *domain_name*, *type_name* and *event_name*. The second section, which is the variable portion of the header, contains optional information about the event. This is constructed as a sequence of properties to hold QoS information related to the individual event notification. The separation of the header into two sections is designed so that a lightweight notification can be created. Such a notification would not include a body, as all pertinent information is contained in the header, thus reducing the amount of extraneous information which must be transmitted .

The body of the structured event contains the actual event data, and is also divided into two parts: the filterable data, and the remaining, or payload, data. The filterable part is a sequence of properties. These properties contain the fields of the notification on which consumers are most likely to base filter decisions. All other data is contained in the remainder of the body. This data is of type CORBA `any`. Again, this design was created to streamline throughput. There is no reason that an event could not

be filtered on data in the payload portion of the body, but performance would not be optimal.

Also complete contents of the notification could be contained within the optional header fields, leaving the body empty. This would allow a more streamlined event. The structured event was organized to make filtering as simple and efficient as possible while not forcing one to use it in a specific manner. [Ref 9]

5. Conclusion

The CORBA Event and Notification Services provide powerful capabilities to handle events in a distributed, heterogeneous environment. However, CORBA only specifies a set of conventions and protocols that must be followed by CORBA implementations - it is left to developers to implement these specifications. Further, the CORBA Event Model is an integrated part of CORBA, so that it is difficult to separate the event model from CORBA and make it a generic tool for other projects that do not use CORBA.

C. SAAM EVENT CHANNEL MODEL

The current SAAM channel model was designed by the SAAM research team, headed by Dr. Geoffrey Xie, and implemented by NPS graduates Dean Vrable and John Yarger in their thesis “The Server and Agent based Active network Management (SAAM) Architecture: Enabling Integrated Service”. The channel model provides event dispatching between SAAM components. It was designed to overcome some limitations of the Java event model.

One of the limitations is the order of instantiation. In the Java event model, the order of instantiation of objects is important. For instance, an event listener cannot register with an event source that has not been instantiated.

Another limitation has to do with event source replacement. If one event source is to take over notification of a certain type of event, listeners for that event must de-register with the old source and then register with the new source. This process involves a great deal of object passing and quickly becomes cumbersome.

1. SAAM Channel

A SAAM channel is a mechanism for delivering SAAM protocol events from one or more event sources to one or more event listeners (Figure 2.8). Similar to the CORBA event model, a SAAM channel decouples the sources and listeners of an event.

A SAAM channel is currently implemented as a Java class, and is thus allowed to have attributes that constitute the state of the channel and methods that affect the state. A channel contains a vector of *listeners* and a vector of *talkers*. Under this model, event listeners become listeners to a channel. They register with a channel as they would register with any event source. The key difference here is event sources now must register as talkers on the channel. To send an event notification, a registered talker merely calls the `talk` method of the desired channel. The channel then notifies all listeners on that channel.

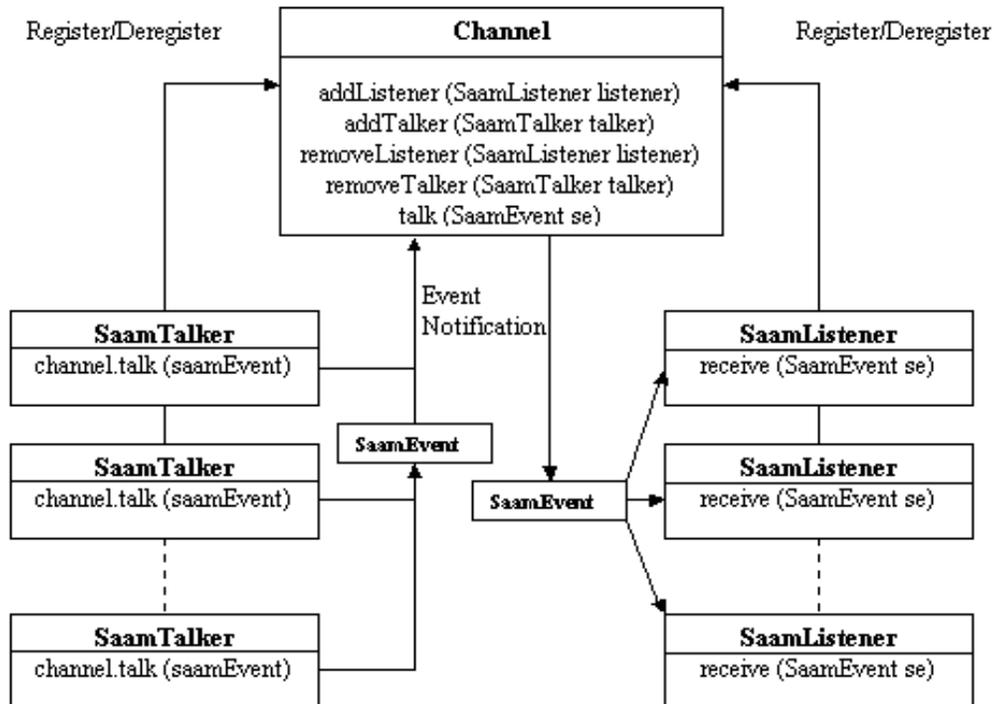


Figure 2.8 SAAM Channel Concept

In the channel registration process, the order in which objects register is not important. Listeners are allowed to register with a channel that has no talkers and vice versa. If a listener registers with a channel that has no talkers, the listener will not receive any events on that channel until a talker registers with the channel, and then talks to it. If a talker registers on a channel that has no listeners, the events sent by the talker will be dropped. [Ref 1]

In the SAAM prototype, a set of well-known channels with unique and static identification numbers is defined to provide the communication between SAAM components. Each channel serves a particular set of SAAM components that perform a well-defined task. For example, *channel 80004* is used to pass SAAM control messages to the *PacketFactory* component for processing. All components, which want to pass a SAAM control message to *PacketFactory*, may at any time register with this channel as a talker and pass their messages to the channel regardless of which instance of *PacketFactory* is listening. Therefore, associating channels with well-defined tasks and static identifiers facilitates loosely coupled communication between SAAM components.

The existing SAAM channel model offers several benefits over the standard Java Event Delegation model. However, there are some shortcomings in the current SAAM channel design and implementation. These deficiencies will be discussed in the next chapter.

III. CHANNEL PERFORMANCE METRICS AND EVALUATION OF EXISTING SAAM CHANNEL MODEL

To motivate the new channel design, a set of performance metrics and qualitative objectives that help to characterize a channel model and implementation are defined in this chapter. The performance of the current SAAM channel model have been measured and qualified according to the defined metrics. The results are reported in this chapter. They raised several performance issues that have guided the design and implementation phase of the new channel model.

A. CHANNEL PERFORMANCE METRICS AND QUALITATIVE OBJECTIVES

In this section, several channel performance metrics used in this thesis will be defined and explained.

1. Channel Throughput (R)

A channel's main task is to deliver events generated by registered senders (channel talkers) to registered receivers (channel listeners). The performance of this task can be measured by channel throughput, defined as the number of events delivered per second. An effective and efficient delivery mechanism is required to maximize the throughput.

The channel event delivery period is the time necessary for the channel to deliver an event to all registered channel listeners. It may vary from event to event. Also, when a listener uses a separate thread for event handling, delivery of an event to that listener means just passing the event to the listener object. It does not involve in the actual handling of the event. Therefore, channel throughput does not always measure how fast listeners process events. The maximum channel throughput is directly related to the event delivery time. The following explicitly defines the relationship between the maximum channel throughput and the minimum event delivery time:

R_{max} : *Maximum Channel Throughput*

T_d : *Minimum Channel Event Delivery Time*

$$R_{max} = \frac{1}{T_d}.$$

2. Channel Work Rate (W)

Channel Work Rate is defined as the total number of events processed by the channel's listeners per second. Similar to the channel throughput, it is possible to improve channel work rate by using efficient event dispatching techniques. The key is to maximize the degree of parallelism in the handling of events by different listeners.

3. Channel Access Delay (D)

Channel Access Delay is defined as the latency between a channel talker's *talk request* and its actual access to the channel. A channel can only process one talk request at a time. It is assumed that a talk request will be blocked and put into a waiting queue if the channel is busy processing another talk request. The channel will serve talk requests in the queue following a specific algorithm (e.g., FIFO). The total waiting time a talk request spends in the queue is defined as the *Channel Access Delay of that Talk Request*. The average channel access delay of events from a talker is defined as the *Channel Access Delay of That Talker*. These delays are directly related to the total number of registered talkers and the talk frequency and period of each talker.

4. Event Talk Time (T_t)

Event Talk Time is the amount of time the channel spends in processing a talk request. This metric is important for event talkers as well as the channel itself. Excessive event talk time may decrease the performance of the talkers and the channel dramatically. This is because all talker threads will be blocked during each event talk period.

From the perspective of channel, access delay will increase and channel throughput will decrease as a result of long event talk time.

5. Thread Count (C)

Multi-threading a channel object may increase the performance of the channel with respect to the metrics defined above. However, it is not always a good idea to add more threads to the design of a channel. As systems differ with respect to the number of threads supported, for a channel design to be general purpose, it is important to make the channel scalable in terms of the thread count. Also, a large number of threads may incur a significant amount of resource overhead. Each `Thread` object consumes memory. In addition, each thread has two execution call stacks allocated for it by the Java Virtual Machine. One stack is used to manage Java method calls and local variables, while the other stack is used to keep track of native code.

A thread also consumes processor resources. There is inherent overhead in the scheduling of threads by the operating system. It happens when one thread's execution is suspended, its state stored, and another thread is given access to the processor so its execution may be resumed. This event is called a *context switch*. CPU cycles are required to do the task of context switching and can become significant if numerous threads are running. [Ref 10]

Another important issue, which should be considered when using threads, is synchronization of thread executions. The consequences of failing to properly synchronize threads which access shared resources are severe: data corruption and race conditions. These can cause programs to crash, produce incorrect results, or behave unpredictably. Even worse, these conditions are likely to occur only rarely and sporadically, making the problem hard to detect and reproduce. If the test environment differs substantially from the production environment, either in configuration or in load, these problems may not occur at all in the test environment, leading to the erroneous conclusion that the tested programs are free of major failures, when in fact the conditions triggering the failures simply have not yet been encountered.

Improper or excessive synchronization of threads can lead to other problems, such as poor performance and deadlock. While poor performance is certainly a less severe problem than data corruption, it can still be a serious problem. Writing good multithreaded programs requires walking a fine line, synchronizing enough to protect the data from corruption, but not so much as to risk deadlock or impair program performance unnecessarily.

Synchronization comes at a cost, also. A synchronized block in the Java language is generally more expensive, in terms of execution time, than the critical section facilities offered by many platforms, which are usually implemented with an atomic "test and set bit" machine instruction. Even when a program contains only a single thread running on a single processor, a synchronized method call is still slower than an unsynchronized method call. If the synchronization actually requires contending for the lock, the performance penalty is substantially greater, as there will be several thread switches and system calls required. [Ref 13]

When adding additional threads to the design of the channel, the issues and costs must be considered carefully. At a minimum, synchronization of built-in channel threads should be transparent to a developer who uses the channel objects, and the developer should not be asked to synchronize an application specific (talker or listener) thread with a built-in channel thread in the code.

6. Adaptability of Channel

Adaptability measures the ease with which a channel model and implementation can be adapted to other projects and applications. A channel design with high adaptability meets software reusability and extendibility goals.

Reusability is the ability of software products to be reused, in whole or in part, for new applications. On the other hand, extendibility is the ease with which software products may be adapted to changes to the specification. [Ref 14]

An easily adaptable channel must be reusable for other projects and must be extendible according to the needs of developers and the requirements of applications.

7. Scalability of Channel

When the number of talkers and listeners registered with the channel grows or the individual load of channel participants increases, the channel should be able to tolerate the changes. The channel should be able to be configured by developers to solve scalability problems.

When applications and the number of necessary channels grow, another scalability issue arises. Developers should be able to add new channels to their application without unduly impacting performance and manageability.

8. Manageability of Channel

For small projects, the number of channels required will also be small and the manageability of the channels will not be a big concern.

On the other hand, large applications will require a relatively large number of channels and developers will have to spend a substantial amount of effort to manage these channels. Channels should be designed to minimize this effort.

Each channel object must have unique properties that differ from other channel objects. These properties will help developers to reach, retrieve, and trace the required channel objects in their code.

9. Functional Flexibility

The major goal of a channel is to deliver events generated by talkers to registered listeners in a fast, reliable manner.

However, a channel should be flexible enough to dynamically change the behavior of the event delivery mechanism. For example, event delivery order and service priorities of channel participants (channel talkers and listeners) can be configured or tuned in accordance with the needs of participant objects.

10. Ease of Use

A channel should provide an easily usable application-programming interface (API) for developers. The following discussions assume a nominal Java `channel` class that embodies this API.

a. Easy Creation of Channel

Developers should be able to create a channel with default properties by using the default constructor of `Channel`. The `Channel` class should provide overloaded constructors to create new channel objects with different attributes.

b. Meaningful Method Names

Choosing meaningful method names makes programs readable and helps avoid excessive use of comments. It also helps developers to explore the channel functionality readily. Method names must be related to tasks fulfilled by the methods.

c. Easy Configuration of Channel

The `Channel` class must provide class methods which enable the channel to be configured or tuned to improve its performance and functionality.

d. Sufficient and Understandable Method Overriding

Method overriding eases the developers' task when programming with the `Channel` class. The *print* method of the Java `PrintStream` class is a good example for this metric.

e. Conflicts Between Methods

Semantics and functionality of the methods in the `Channel` class must be as clear as possible. Interactions between the methods must be well defined and well documented.

f. Conflicts Between Parameters in a Method

Method parameter names must be descriptive and must identify the objects expected. The name of a method and the type and order of its parameters generate the method signature. Method signatures must be designed to avoid confusion.

g. Documentation

The Channel class should be well documented so that developers can study, use, and modify it easily.

B. TEST OF SAAM CHANNEL

The existing SAAM channel was tested with respect to the defined performance metrics and qualitative objectives. In this section, the test results are reported.

1. Channel Throughput and Work Rate

Throughput and work rate of the existing SAAM channel was measured with different event handling periods for listeners. In the test scenario, three talkers and three listeners were registered with a SAAM channel (Figure 3.1). All channel talkers generated events based on *Poisson* distribution and the average event generation rate was 200 events per second. The event handling time of *Listener 1* and *Listener 2* was set to 0.5 milliseconds per event achieved by an idle *for-loop* in the listeners' *receiveEvent* methods. On the other hand, the event handling time of *Listener 3* was increased gradually by forcing the execution thread to sleep for an increasing amount of time in the listener's *receiveEvent* method.

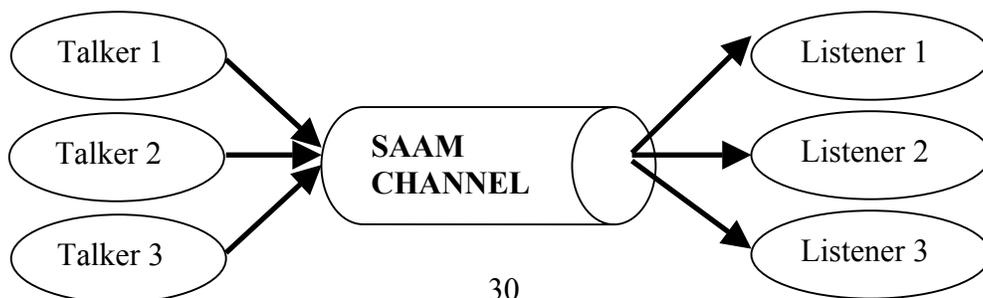


Figure 3.1 Test Bed For Studying Impact of Listener Event Handling Time on Channel Throughput

As shown in Figure 3.2, the channel throughput and work rate decreased severely when the event handling time of Listener 3 was increased. These results show that the throughput and work rate of the existing SAAM channel is dependent on the total. It is important to recognize that an event's delivery time depends on the number of registered channel listeners and the sum of their event handling times.

Event filtering can be used to multicast the events to interested listeners instead of broadcasting to all listeners. On the other hand, new threads can be created by the channel and dedicated to serve listeners that have a long event-handling time. Event filtering and dedicated event handling threads help reduce the total event delivery time and allow an event to be handled concurrently by different listeners. As a result, the channel throughput and work rate increases. The existing SAAM channel offers none of these capabilities and thus its throughput is not optimized.

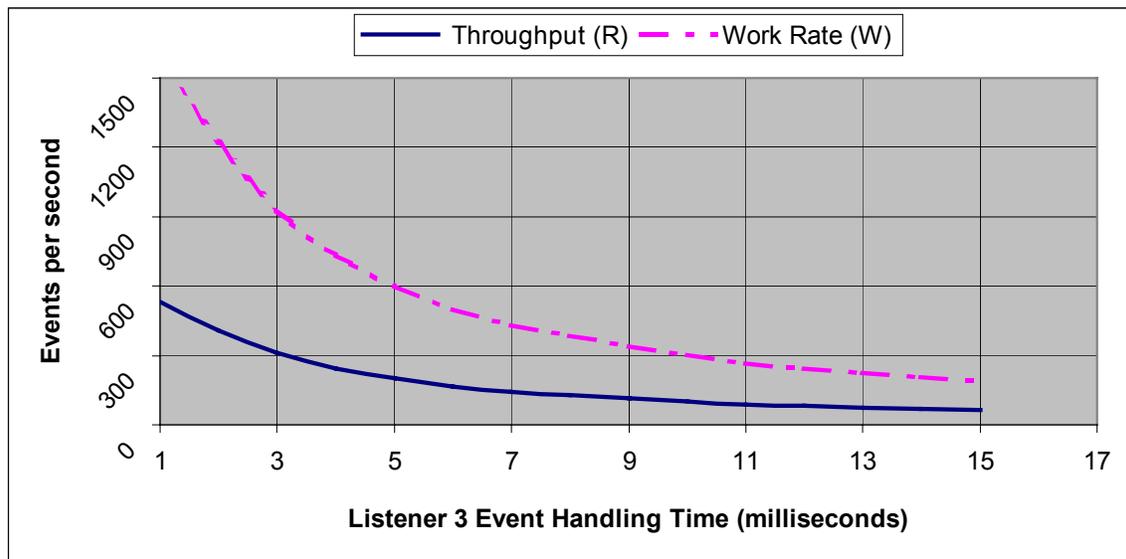


Figure 3.2 SAAM Channel Throughput and Work Rate versus Listener Event Handling Time

2. Channel Access Delay

Access delay of the existing SAAM channel was measured using a varying number of registered talkers and different event handling times for listeners. To measure the effect of the number of talkers on the channel access delay, three listeners with one millisecond event handling time (simulated by an idle *for-loop* in their *receiveEvent* method) were registered with the channel and the number of talkers was initially set to one and then increased steadily. Each talker generated 500 events per second, following a *Poisson* distribution. Figure 3.3 shows the test bed for the measurement.

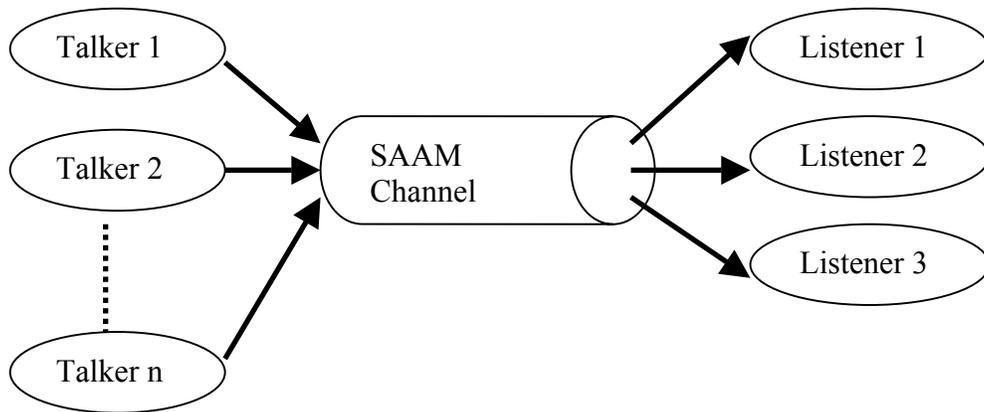


Figure 3.3 Test Bed for Measuring SAAM Channel Access Delay

The channel access delay for the case of a single talker was very small. This was an expected result because the channel was available to serve this talker all the time. The average channel access delay of talkers grew linearly when new talkers were added to channel. (Figure 3.4)

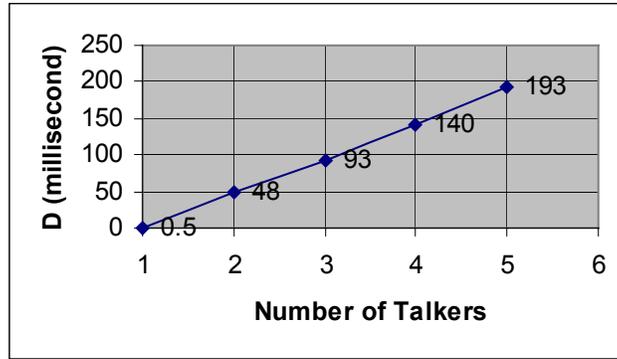


Figure 3.4 Average Access Delay versus Number of Channel Talkers

In the second scenario, the event handling time of *Listener 3* was increased gradually to see the effects of event handling time on the channel access delay. The rise in the total event delivery time caused the channel access delay to increase. (Figure 3.5)

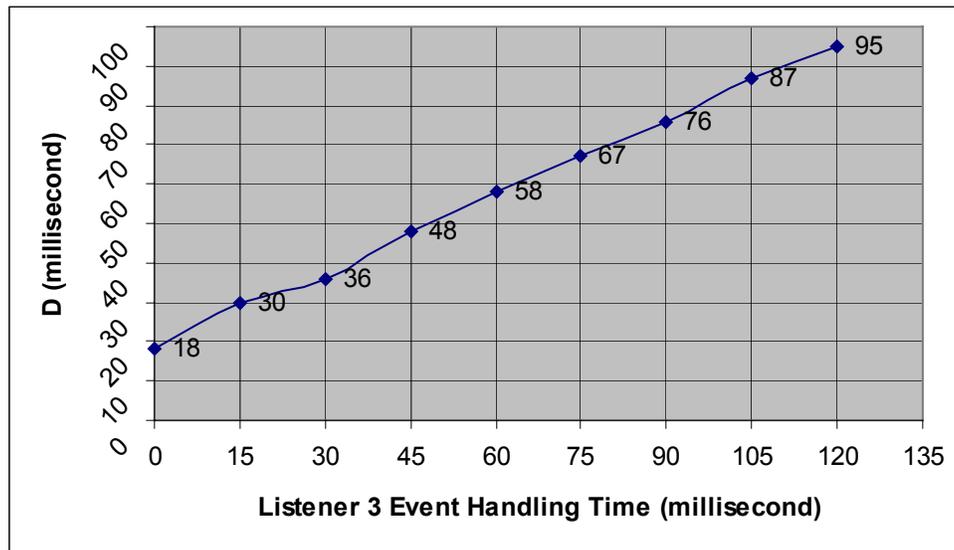


Figure 3.5 SAAM Channel Access Delay versus Event Handling Time

Therefore, the talker count and total event delivery time have a significant impact on the channel access delay of the existing SAAM channel. Specifically, the event handling times of channel listeners affect the channel access delay directly. This is because the existing SAAM channel has no buffering capability and it cannot accept another event until the current one is delivered to all listeners. This means that a channel

is not accessible by talkers while it is dispatching an event. Consequently, the performance of the channel talkers is poor and unpredictable due to high and variable channel access delays.

3. Event Talk Time

As identified earlier, the existing SAAM channel has no event buffering capability and talkers cannot continue their job by sending their events to the channel. Talkers have to wait until the current event is delivered to all listeners. Actually, it is the thread of the current talker that executes channel dispatching and event handling code, except for those listeners which are separately threaded.

The Event Talk Time of the SAAM channel includes the channel access delay and total event delivery time. Therefore, it should vary according to the number of talkers and the event handling times of its listeners. A simple experiment was performed to verify this fact.

The test bed depicted in Figure 3.3 was again used. Event Talk Time data was collected for an increasing number of talkers.

Figure 3.6 shows the results. Clearly, there is a linear rise in the event talk time when new talkers are added to the channel.

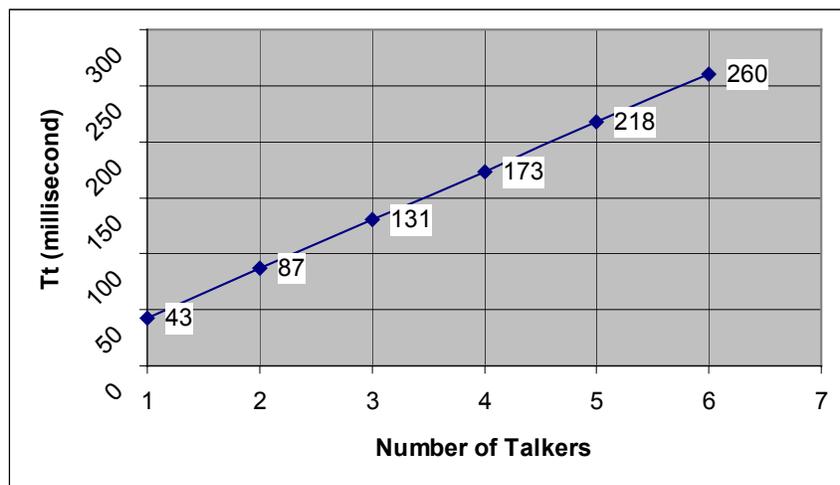


Figure 3.6 Event Talk Time Versus The Number of Talkers

More measurements of event talk time were collected using different event handling times. In this scenario, three talkers and one listener were registered with the SAAM channel. The event generation rates of the talkers and the event handling scheme of the listener are the same as the scenario used to study the channel access delay.

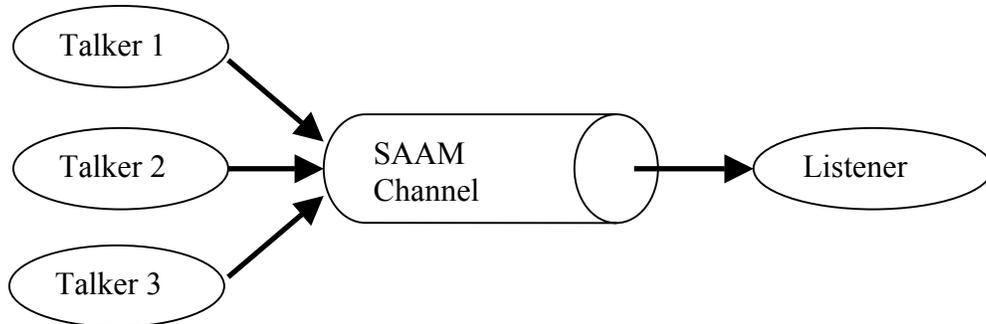


Figure 3.7 Test Bed for Measuring SAAM Channel Event Talk Time

The listener's handling time was increased steadily and the resulting event talk times recorded. Figure 3.8 shows the experiment results. It can be concluded that with the current SAAM channel design, event handling times affect event talk times directly.

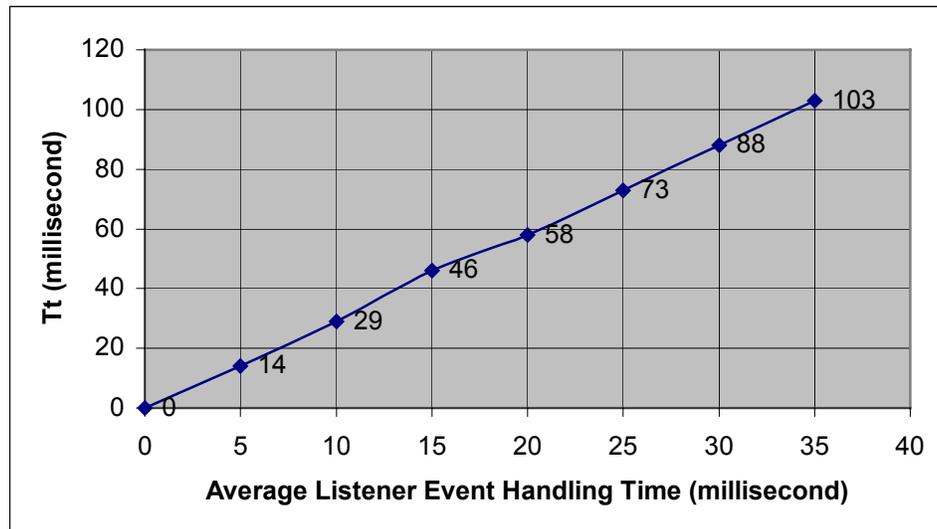


Figure 3.8 Event Talk Time Versus Event Handling Time

4. Thread Count

The existing SAAM channel does not require any thread to fulfill its functionality. This reduces its complexity. But, due to this threadless design, it cannot buffer events resulting in high and variable channel access delays and event talk times. The design may also cause some scalability problems, which will be discussed at the next section.

5. Manageability and Scalability of SAAM Channel

SAAM channels can be accessed by their channel identification numbers. An object wishing to manage these channels must create its own data structure, in order to keep records about existing channels and control the communication between them. In SAAM, the `ControlExecutive` class acts as a channel manager. SAAM talkers and listeners cannot register with or talk to channels directly. They interact with channels via `ControlExecutive`. `ControlExecutive` is implemented in an ad hoc manner. As a result, the current SAAM channel design does not provide much support for channel management.

Scalability is one of the most critical drawbacks of the existing SAAM channel design. Adding new talkers and listeners to a channel increases the channel access delay and event talk time. The channel cannot compensate for these effects without performance penalties.

In addition, interaction between multiple SAAM channels should be considered carefully when the number of channels grows in an application. Consider the scenario depicted in Figure 3.9.

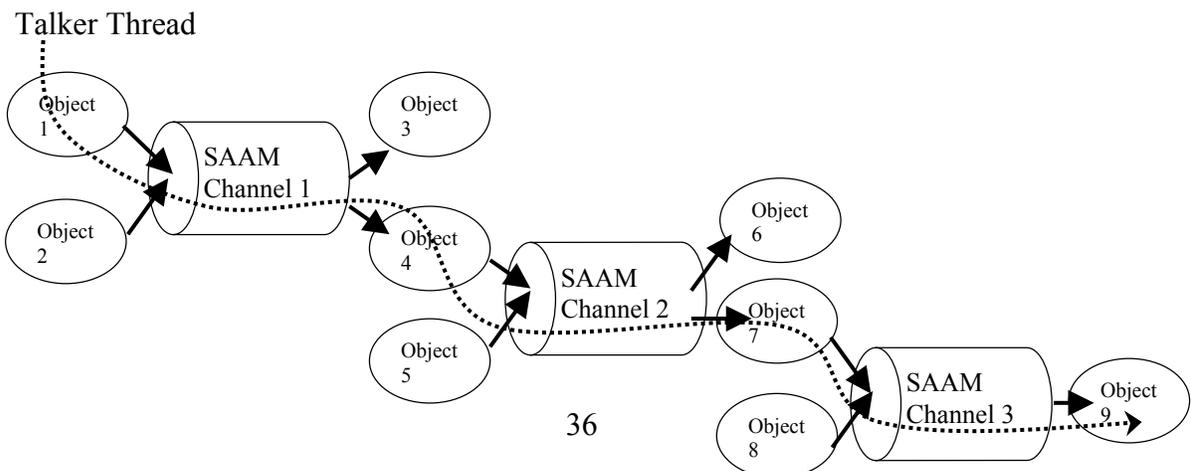


Figure 3.9 Event Concatenation in SAAM Channels

Object_1 sends an event to SAAM Channel_1. Object_4 talks to SAAM Channel_2 when it gets that event from Channel_1. Object_7 talks to SAAM Channel_3 when it gets the same event from Channel_2. This concatenation of channels blocks the thread of Object_1 until all three SAAM channels deliver the event to their listeners. At the same time, all SAAM channels in this scenario are inaccessible by other talkers until completion of Object_1's event delivery.

Another scalability issue is that the existing SAAM channel provides one-way communication. At least two channels are required to create a duplex communication between two objects. (Figure 3.10)

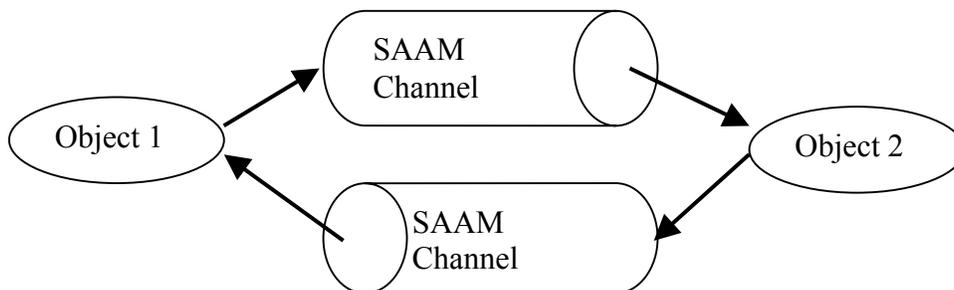


Figure 3.10 Two-way communication with SAAM channel

6. Adaptability and Functional Flexibility of SAAM Channel

The existing SAAM channel has been designed for communication between components of the SAAM prototype. Thus, it can handle only `SAAMEvents`. It is difficult to adapt the current SAAM channel model and implementation to other projects because

it has been intended to meet the requirements of the SAAM prototype. It does not adapt to the varying needs of different applications.

Event dispatching of the existing SAAM channel is inflexible and cannot be changed. Events cannot be delivered to listeners in a specific order according to application needs. Listeners are served in their registration order and it is not possible to give precedence to listeners that have specific requirements for event handling.

In addition, the existing SAAM channels broadcast the current event to all registered listeners without regard to specific interests of the listeners. This increases the channel traffic unnecessarily and forces channel listeners to filter and drop uninterested events themselves.

7. Ease of Use of SAAM Channel

An instance of the SAAM channel can be created by providing an integer channel identification number and a channel participant, either a listener or talker. Later, other listeners and talkers can be added to the channel by using the *addListener* and *addTalker* methods. An additional constructor, which allows creation of a SAAM channel without any participant, would enhance the usability of the channel design.

IV. NEW CHANNEL MODEL AND CHANNEL PACKAGE

A. INTRODUCTION

The existing SAAM channel model offers several benefits over the standard Java Event Delegation model by allowing event delivery from one or more event sources to one or more event listeners and allowing event listeners to register with a channel that has no talkers.

However, there are some shortcomings in the current SAAM channel design and implementation. The model supports just one type of event class. It has no buffering capability. The thread of an event talker is blocked unnecessarily until the event is delivered to all listeners. The model does not give a programmer a choice of dispatching events to multiple listeners in a specific order. Consequently, the performance of the SAAM event model is not optimized.

As the main contribution of this thesis, a new channel model was designed to mitigate the shortcomings in the current SAAM channel implementation. The resulting channel package provides Java developers a generic, highly adaptable, and flexible communication mechanism between loosely coupled components.

B. FEATURES OF NEW CHANNEL DESIGN

The key features of the new channel design are summarized below. The implementation details of these features will be described in the next section where the Java channel package is introduced.

1. Generic Event Structure

The new channel supports multiple types of event objects per channel. Further, it uses a simple and extendible event structure.

2. Event Buffering

The new channel is capable of buffering events. This allows a channel to dispatch events to multiple listeners in a specific order and reduces the channel access delay and event talk time significantly.

3. Event and Channel Participant Priority

In the new channel design, a priority value is assigned to each event and each channel participant (channel talker and channel listener). Therefore, it is possible to give precedence to important events or serve particular listeners or talkers first.

4. Event Delivery Order

The new channel allows developers to choose different event dispatching schedules based on the priorities of events and channel participants. The dispatching can be customized based on application needs.

5. Event Filtering

The new channel design provides an event filtering mechanism, which allows channel listeners to subscribe to particular events. A channel will deliver to a listener only those events for which the listener is subscribed.

6. Self-Dispatching

The new channel design offers a self-dispatching mechanism for creating separate threads for channel listeners with large event handling times. This mechanism can be used to mitigate the negative effect on channel performance of long event handling times for specific listeners.

7. Duplex Communication

The new channel design supports two-way event communication between objects through a single channel. In other words, an object can be both a channel talker and a channel listener on the same channel, but for explicitly different events.

8. Concatenating Channels

The new channel design allows concatenation of channels. While a channel cannot generate events, it may be forwarded events from another channel. In this sense, a channel can be a listener on another channel.

C. CHANNEL PACKAGE

The new channel design was implemented in Java and all classes and interfaces were gathered under a Java utility package, called *channel*.

➤ Classes

- `Channel.java`
- `ChannelEvent.java`
- `ChannelListenerItem.java`
- `ChannelEventFilter.java`
- `FIFOScheduler.java`
- `PerTalker_RR_Scheduler.java`
- `PriorityScheduler.java`
- `ChannelManager.java`

➤ Interfaces

- `ChannelListener.java`
- `ChannelFilter.java`
- `ChannelScheduler.java`
- `ChannelAccessAuthority.java`

1. ChannelEvent Class

`ChannelEvent` class provides a data structure to encapsulate channel events. *Talker* and *event* data members of this class are instances of the `java.lang.Object` class, which is the base class of all Java objects. As such, it is the primitive class from which all other classes are ultimately derived. Therefore, `ChannelEvent` class can encapsulate all objects as either an event or an event talker.

References to an event and its talker are required to create an instance of `ChannelEvent`. The value of the priority, for both the event and the talker, is an integer . If no value is provided, then a default value of zero is assigned. A higher integer value implies higher priority.

Another data member of `ChannelEvent` class is the timestamp. It is generated automatically, using the system's time in milliseconds.

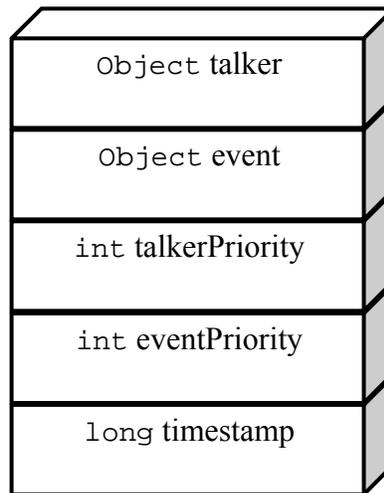


Figure 4.1 Channel Event Structure

2. ChannelListener Interface

All channel listeners must implement the `ChannelListener` interface to register with a channel and obtain events from that channel. `ChannelListener` contains a single method:

- `public void receiveEvent (ChannelEvent event)`

This method is called by the channel to deliver an event to the registered channel listeners. This method should be synchronized to guarantee that no more than one channel dispatcher thread is allowed when a channel listener registers with multiple

channels. The reason for this is that each channel creates its own event dispatcher thread and synchronization is critical to prevent the occurrence of hazards described earlier.

3. ChannelFilter Interface

Event filtering can be used by a channel to multicast its events to interested listeners instead of broadcasting to all listeners. Multicasting reduces the total delivery time for an event. Since the throughput is the reciprocal of the average event delivery time, the channel throughput will increase.

Without event filtering, channel listeners have to check all incoming events and choose the pertinent ones themselves. It can be guaranteed that channel listeners get only the events in which they are interested when event filtering is employed.

The `ChannelFilter` interface defines one simple method that must be implemented.

- `public boolean isAccepted (ChannelEvent event)`

Basically, if this method returns a value of true then this event passes the filter. Channel listeners can implement `ChannelFilter` themselves or use another object that implements this interface to take advantage of event filtering. Channel listeners can pass their filters to a channel when they register with the channel or later when filtering is needed.

Another benefit of event filtering is that channel listeners can control event flow between the channel and themselves dynamically. A listener can suspend or stop event acceptance from a channel without deregistering with channel. It is also possible to change the range and characteristics of accepted events to adapt to changes in system conditions and requirements.

The new channel design also allows concatenating event filters. A channel maintains the list of filters for a listener in a vector. New filters can be added to the end of the vector upon requests from the listener. The channel dispatches an event to that

listener only if this event is acceptable by all of the filters in the vector. Channel listeners can add a new filter or remove an existing filter by using the *addFilter* and *removeFilter* methods of `Channel`.

The `channel` package contains a built-in, ready-to-use, class named `ChannelEventFilter`, which implements the `ChannelFilter` interface and provides a filtering capability based on the class types of either the talker or the event, or simply the talker's name. `ChannelEventFilter` will be described in detail in Section 6 of this chapter.

4. ChannelScheduler Interface

A customizable channel scheduler allows a channel to buffer and dispatch events to multiple listeners in a specific order. A channel event can be categorized by either its talker, the event's type, or the respective priority of either the talker or the event itself. This categorization is used by the scheduler to determine the event delivery order.

The channel scheduler buffers and orders channel events according to its scheduling policy. The channel dispatches events to listeners by pulling them from the scheduler. The channel scheduler was implemented as an interface. The `channel` package includes three built-in channel schedulers; *FIFO Scheduler*, *Per Talker Round-Robin Scheduler*, and *Priority Scheduler*, all of which implement the `ChannelScheduler` interface. Developers may also implement and deploy their own schedulers to meet their specific requirements. This provides functional flexibility for the channel. The `ChannelScheduler` interface contains two methods:

- `public void push (ChannelEvent event)`
- `public ChannelEvent pull ()`

A channel forwards events to a scheduler by calling its `push` method and dispatches events by extracting them from the scheduler via the `pull` method.

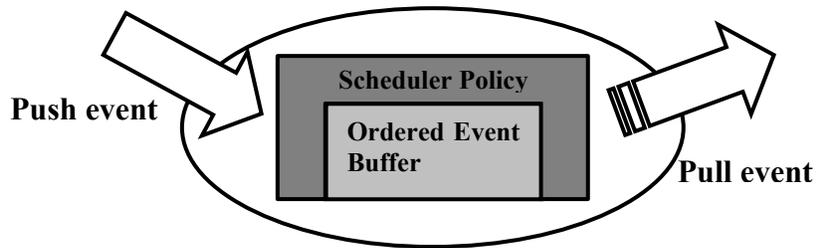


Figure 4.2 Channel Scheduler

5. ChannelListenerItem Class

ChannelListenerItem is the data structure that a channel uses to manage channel listeners. As depicted in Figure 4.3, each ChannelListenerItem object encapsulates a ChannelListener object, its filters, and its priority.

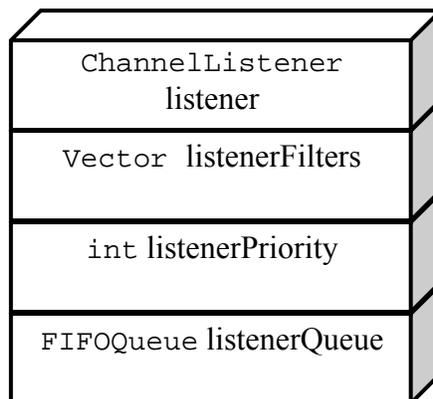


Figure 4.3 Channel Listener Item

The object also hosts an event buffer. The queue is used when a channel creates a dedicated thread for dispatching events for this listener. Without the option of having a separate dispatching thread, a listener with a long event handling time would delay the

channel dispatcher thread and the other channel listeners unnecessarily. This self-dispatching mechanism is implemented to eliminate a channel's dependence on the event handling periods of listeners.

After having registered with a channel, a listener may make a self-dispatch request by calling the *startListenerSelfDispatch* method of that channel. The channel relays this request to the channel listener item that it has created for the listener. The channel listener item allocates an event buffer and creates a thread for dispatching events from this event buffer to the listener. When the thread and the event buffer are ready, the self-dispatching flag of the channel listener is set to true. After it is set to true, the channel does not directly call the *receiveEvent* method of the listener for event delivery. Instead, the channel puts events into the event buffer of the corresponding listener item. If a channel listener has a variable event handling time in specific situations, self-dispatching can be suspended and resumed temporarily or can be stopped permanently.

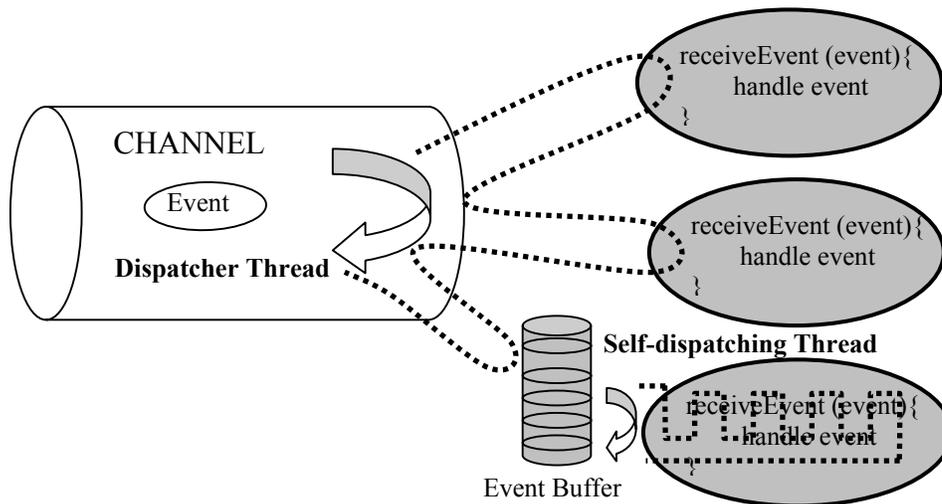


Figure 4.4 Self-dispatching

6. ChannelEventFilter Class

The channel package includes a built-in event filter class, *ChannelEventFilter*, which implements the *ChannelFilter* interface. Using this built-in filter, events may be filtered based on class types or talker name.

ChannelEventFilter class denies all events initially. In order for an event to be accepted, that is, to be passed by the filter, properties which allow its acceptance must be set by the listener. ChannelEventFilter provides four methods to set acceptance criteria:

- addAcceptedTalkerName (String talkerName)
- addAcceptedTalkerClass (Class talkerType)
- addAcceptedEventClass (Class eventType)
- addAcceptedTalkerEventPair (Class talkerType,
Class eventType)

The first method requires the name of the event talker as a parameter. The name of the talker is obtained by calling the *toString* method of the talker objects. All events whose talker names match this particular name will be accepted by the filter. The second and third methods are used to specify acceptable event class types and talker class types. The last method provides a more sophisticated filtering capability. A developer may specify acceptable events by their class types and their talker class types at the same time. The following methods are used to remove the accepted event properties, which were added previously.

- removeAcceptedTalkerName (String talkerName)
- removeAcceptedTalkerClass (Class talkerType)
- removeAcceptedEventClass (Class eventType)
- removeAcceptedTalkerEventPair (Class talkerType,
Class eventType)

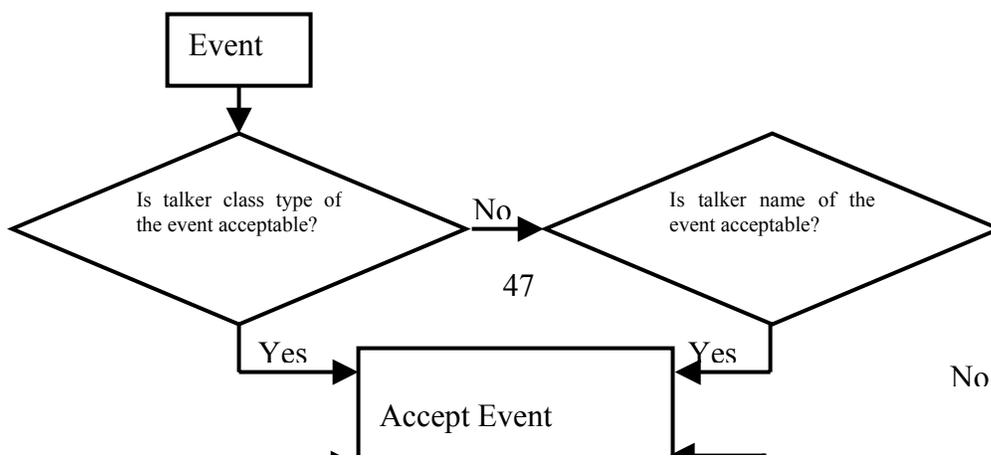


Figure 4.5 ChannelEventFilter Filtering Process

Figure 4.5 shows the filtering process for ChannelEventFilter class. There are also two useful methods: one to stop a filter from accepting events temporarily and the other to resume normal operation.

- `suspendAccepting ()`
- `resumeAccepting ()`

These methods do not change the event permission settings of the filter. All events are denied regardless of the filter settings when the *suspendAccepting* method is called. On the other hand, the *resumeAccepting* method returns the filter to its normal operation and events are accepted according to the existing filter settings.

7. Schedulers

As previously stated, the channel package contains three different built-in schedulers that implement `ChannelScheduler` interface: *FIFO Scheduler*, *Per Talker Round-Robin Scheduler*, and *Priority Scheduler*.

a. *FIFOScheduler Class*

The `FIFOScheduler` class simply buffers arriving events into a First In First Out (FIFO) queue. Events are served by their arriving orders, regardless of talker and event priorities. `FIFOScheduler` is the default scheduler for `channelsand` and will be installed when no scheduler is specified at the time of channel creation.

b. *PerTalker_RR_Scheduler Class*

The Per Talker Round-Robin Scheduler offers a fair service distribution between channel talkers. However, a priority queue is created for each talker. Arriving events are dispersed to separate queues based on their talker identity. The *pull* method extracts events from talker queues using a round-robin algorithm.

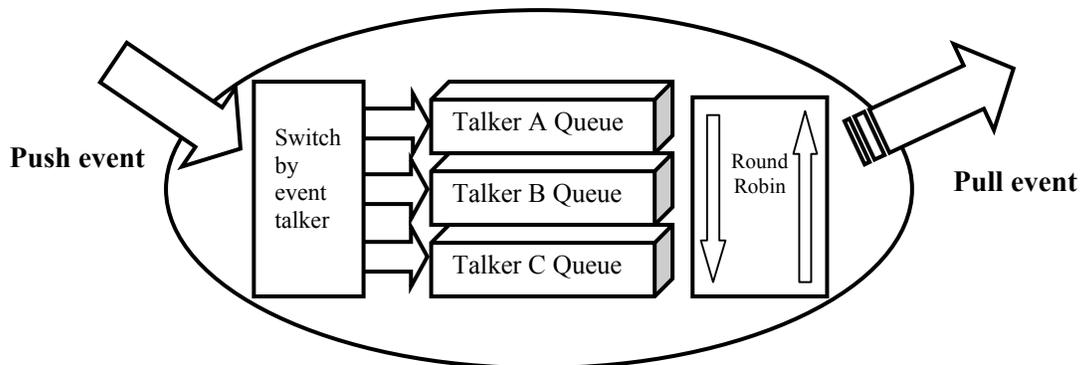


Figure 4.6 Per Talker Round-Robin Scheduler

c. *PriorityScheduler Class*

Events are buffered in a priority queue in accordance with their talker and event priorities. Events with the highest talker priority will be served first. If the talker priorities of two events are equal, then their event priorities are compared, and the one with a higher priority will be served first. When both the event and talker priorities are

equal, the first arriving event will be served first. This scheduler can be used to give event delivery precedence to particular event and talker types.

8. Channel Class

The Channel class is the core class that provides an event delivery service between channel talkers and channel listeners. Its main tasks include scheduling, filtering, and dispatching of events. Developers may customize how these tasks will be carried out for individual channels according to their application requirements.

a. Creating a Channel Object

There are two constructors to create an instance of Channel class:

- `public Channel (int channel_id)`
- `public Channel (int channel_id,
ChannelScheduler scheduler)`

The first one constructs a channel with a specified integer channel identification number. This constructor creates a channel with the default built-in FIFO scheduler. The second constructor can be used to create a channel with a specific scheduler. This scheduler can be chosen from one of the built-in schedulers or developed to meet particular requirements of the current application.

b. Adding and Removing Listeners to Channel

All objects that want to register with a channel as a listener must implement the ChannelListener interface. New listeners are added to a channel by invoking the channel's addListener method. This method is overloaded to allow the priority and filter objects of the listener to be specified as options.

- `addListener (ChannelListener listener)`
- `addListener (ChannelListener listener,`

- int priority)
- addListener (ChannelListener listener,
ChannelFilter filter)
- addListener (ChannelListener listener,
ChannelFilter filter, int priority)

Existing registered listeners are removed from a channel by calling the following method:

- removeListener (ChannelListener listener)

c. Adding and Removing Talkers to Channel

Any object may register as a talker of a channel simply by invoking one of the channel's *addTalker* methods.

- addTalker (Object talker)
- addTalker (Object talker, int talkerPriority)

Talkers can be easily removed from a channel by using the following method:

- removeTalker (Object talker)

d. Talking on Channel

After registration with a channel, a talker calls one of the *talk* methods of the channel to send an event to the channel.

- `public void talk (Object talker, Object event)`
- `public void talk (Object talker, Object event, int eventPriority)`
- `public void talk (ChannelEvent event)`

The first *talk* method is the most generic and the talker can use it to send any Java object to the channel as an event without restriction. The second would be used when the talker wants to assign a particular priority to the event. The third *talk* method makes it possible to deliver an instance of `ChannelEvent` directly to the channel. In the first two *talk* methods, the channel encapsulates the event object, the talker object, and the event priority (the default event priority is used if it was not specified) in a new `ChannelEvent`, while the third method requires a `ChannelEvent` as an input parameter. Each method pushes a `ChannelEvent` to the channel scheduler.

e. Event Dispatching

A dispatcher thread is created for a `Channel` object to pull events out of the channel scheduler and deliver them to each registered channel listener. Listeners are served in the order of their priorities. If two listeners have the same priority, the one that registered first will be served first (FIFO within priorities). The dispatcher thread continues to dispatch events continuously until the scheduler's event buffer is empty. Whenever the event buffer becomes empty, the dispatcher thread goes to sleep. It will be awakened upon arrival of a new event to the scheduler.

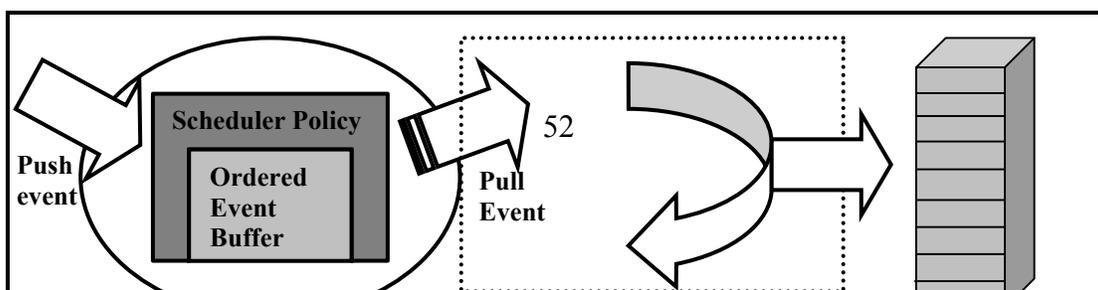


Figure 4.7 Channel Event Dispatching

f. Event Filtering

In the new channel design, a listener of a channel does not have to receive and handle all events that go through the channel. It can use event filters to identify specific types of events that it intends to receive. If one or more event filters are installed for a listener, the channel dispatcher always checks whether the current event is acceptable by those filters before delivering the event to the listener. If the event is not acceptable, the dispatcher will not deliver it to the listener. A listener is not required to specify a filter. Unless a listener specifies a filter, it will receive every event that goes through the channel. The use of a built-in filter with no specified acceptance criteria, as noted above, will prevent all events from being send to the listener. Figure 4.8 shows the event filtering process of a channel, which is repeated for every listener on a per event basis.

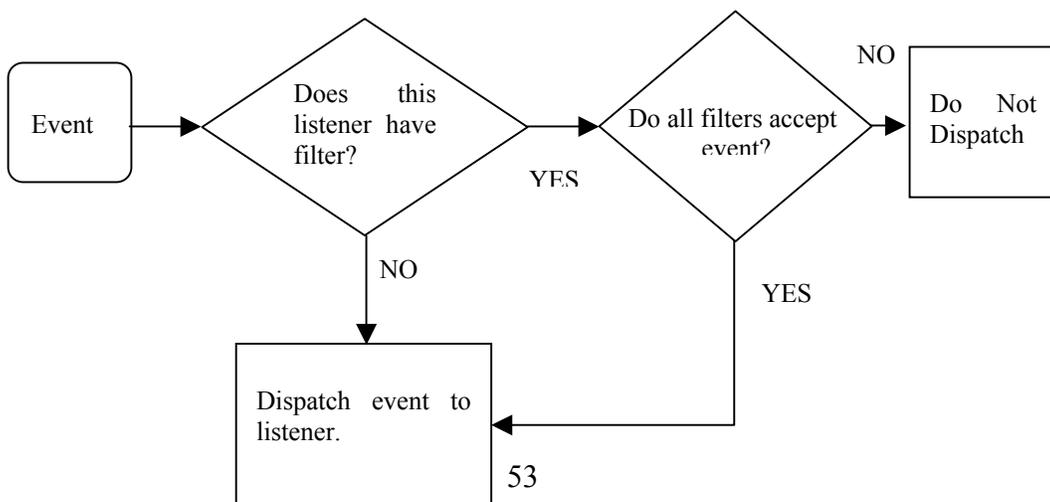


Figure 4.8 Channel Event Filtering Process

g. Listener Self-Dispatching

The ChannelListenerItem class supports self-dispatching, as discussed earlier. The Channel class provides the necessary API, i.e., public methods, to turn on and off self-dispatching for a listener. These methods are:

- startListenerSelfDispatch (ChannelListener listener)
- suspendListenerSelfDispatch (ChannelListener listener)
- resumeListenerSelfDispatch (ChannelListener listener)
- stopListenerSelfDispatch (ChannelListener listener)

A channel can start or resume self-dispatching for a listener without delay. However, suspending or stopping self-dispatching requires coordination between the channel dispatcher thread and the self-dispatching thread. The channel waits until the event buffer of the self-dispatching listener is empty to carry out a suspending or stopping request.

h. Duplex Communication

The new channel design supports two-way event communication between objects. In other words, an object can be a channel talker and channel listener on the same channel, but cannot receive its own events. Figure 4.9 illustrates the two-way communication between Object A and Object B.

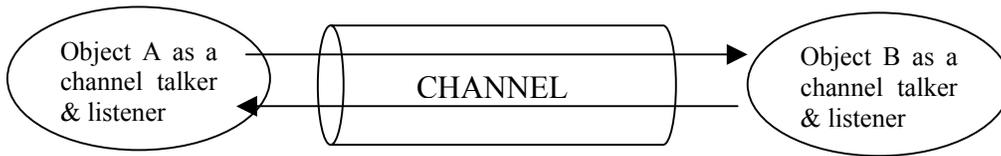


Figure 4.9 Two-way Event Communication

i. Concatenating Channels

The Channel class, itself, also implements the ChannelListener interface so that a channel can be a listener of another channel. This provides support for concatenating channels as shown in Figure 4.10. Only talkers can initiate events. A channel simply forwards events, submitted by a talker, to another channel as necessary, treating that receiving channel as a listener. The receiving channel must register as a listener using the source channel's *addListener* method.

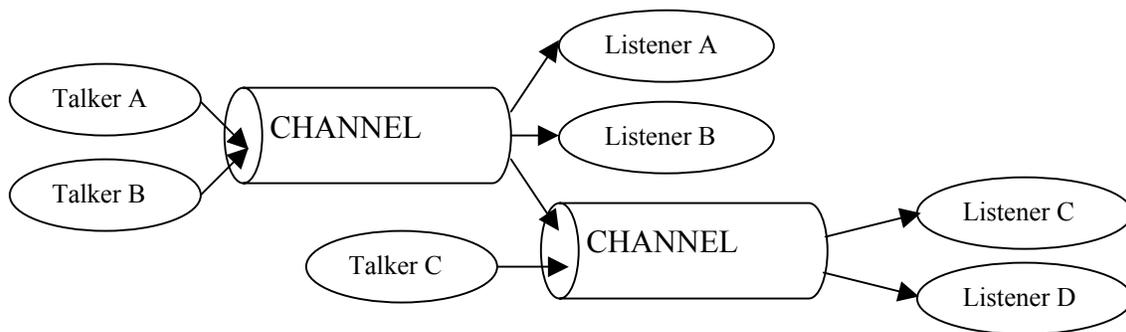


Figure 4.10 Concatenating Channels

9. ChannelAccessAuthority Interface

This interface provides a framework to control and organize all channels, and each channel's participants, of an application in a centralized manner. It defines five methods to be implemented by such a centralized channel controller.

- `boolean isTalkerAuthorized (Object talker, int channel_id)`
- `boolean isListenerAuthorized (ChannelListener listener int channel_id)`
- `int getTalkerPriority (String talkerName)`
- `int getListenerPriority (String listenerName)`
- `ChannelScheduler getSchedularForChannel (int ch_Id)`

The first and second methods allow a channel controller component to specify permissions for talking and listening on all the channels. The controller may stipulate talker and listener priorities, the values of which may be retrieved via the third and fourth methods. The last method ensures that the proper channel scheduler is installed for a new channel. In this way, the channel authority can determine the event dispatching order for the channel in accordance with the requirements of the application and channel participants.

10. ChannelManager Class

The SAAM channel model completely decouples event sources (channel talkers) and event listeners (channel listeners). This decoupling raises an interesting question: “How can channel participants obtain a reference to the channel with which they want to interact?” This is not a big concern for small applications. Channel references can be passed to a participant object when the object is created, or set later by calling the appropriate method of the channel participant of interest.

When the number of channels is large, it is necessary to create a common, easy-to-use, interface for use by all channel participants instead of passing channel references to participants individually. Channel talkers and listeners interact with this common interface to access their channels.

The `ChannelManager` class was developed to provide a common interface for channel participants in large applications. Basically, this class manages channels by keeping a table of existing channels, creating new channels when required, and overriding the `Channel` class methods with an extra argument to allow the desired channel to be identified simply by its `channel_id`. A `ChannelManager` can only control object access to channels if it contains a reference to a `ChannelAccessAuthority` object. A `ChannelManager` can only obtain this reference during instantiation and through the control object control access to channels and enforce predefined priorities for channel talkers and listeners. Thus two constructors are necessary:

- `public ChannelManager ()`
- `public ChannelManager (ChannelAccessAuthority authority)`

The first method constructs a channel manager without any access control authority while the second method creates one with an access control authority.

When an object wants to register with a channel it sends a request to the channel manager with the channel identification number included. If the channel manager contains no reference to a channel access control authority object, it proceeds to check whether a channel object with the given ID exists. If no such channel exists, the manager creates a new channel object with the specified channel ID and adds its information to the channel table. Once the target channel object is located, or created if necessary, the channel manager calls one of the channel's *addTalker* or *addListener* methods to complete the object registration.

However, if the channel manager contains a reference to a channel access control authority object, it first verifies that the requesting object is authorized access to the requested channel. This is accomplished by calling the control object's *isTalkerAuthorized* or *isListenerAuthorized* method. If the access is authorized, the

channel manager will perform the same registration steps as described in the paragraph above. Otherwise, it will reject the request. Also, the channel manager will use the control object's *getTalkerPriority* or *getListenerPriority* method to set the priority of an authorized talker or listener. The required argument *talkerName* or *listenerName* is obtained by calling the requesting object's *toString* method. Finally, when creating a channel object, the channel manager will call the access control object's *getSchedulerForChannel* method to determine the appropriate channel scheduler for the new channel.

THIS PAGE INTENTIONALLY LEFT BLANK

V. TEST AND RESULTS

Simple example applications have been built with the Java channel package. The performance of the new channel design was evaluated through experiments using these applications. The results are presented in this chapter. To demonstrate the suitability of the new channel design for large application development, the SAAM prototype has been updated to use the new channel package. The steps of this update are also described in this chapter.

A. EVALUATION OF NEW CHANNEL DESIGN

1. Channel Throughput and Work Rate

The new channel design supports event filtering and self-dispatching for channel listeners. When activated, both of these mechanisms may reduce the total delivery time for an event and allow events to be handled concurrently. As a result, the channel throughput and work rate should improve. The same experiment scenario used for measuring the SAAM Channel throughput and work rate was repeated for the new channel to verify this hypothesis. Figure 5.1 shows the test bed used in the experiment.

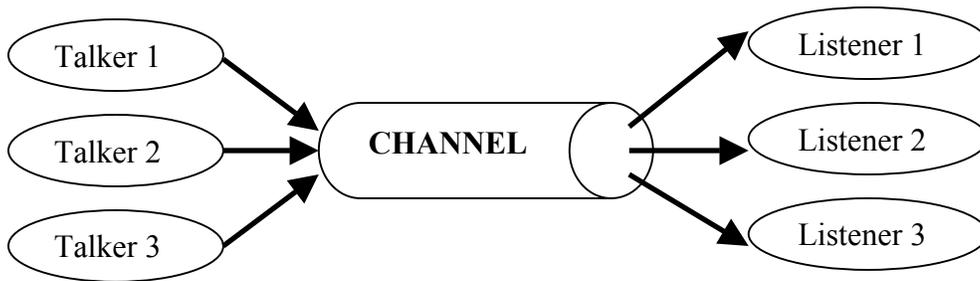


Figure 5.1 Test Bed For Studying Impact of Self-Dispatching on Channel Throughput and Work Rate

Self-dispatching was started for *Listener 3* when its event handling time reached 10 milliseconds. The channel throughput and work rate were monitored for the duration

of the experiment. The results are compared to those of the SAAM Channel in Figure 5.2a and Figure 5.2b.

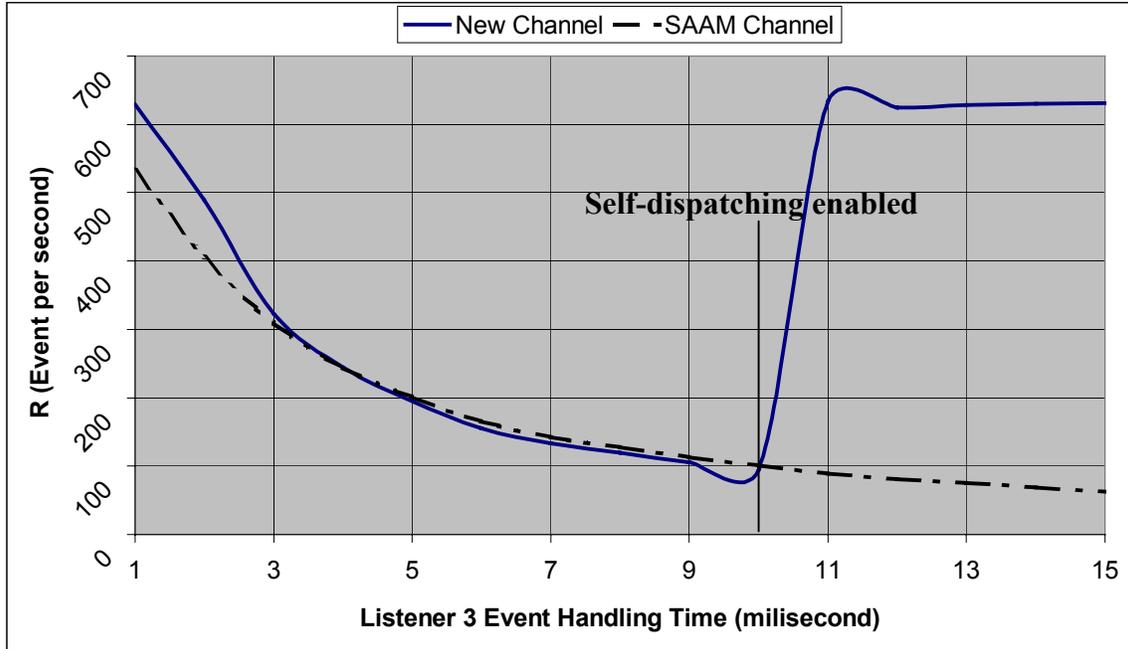


Figure 5.2a Effect of Self-Dispatching on Channel Throughput

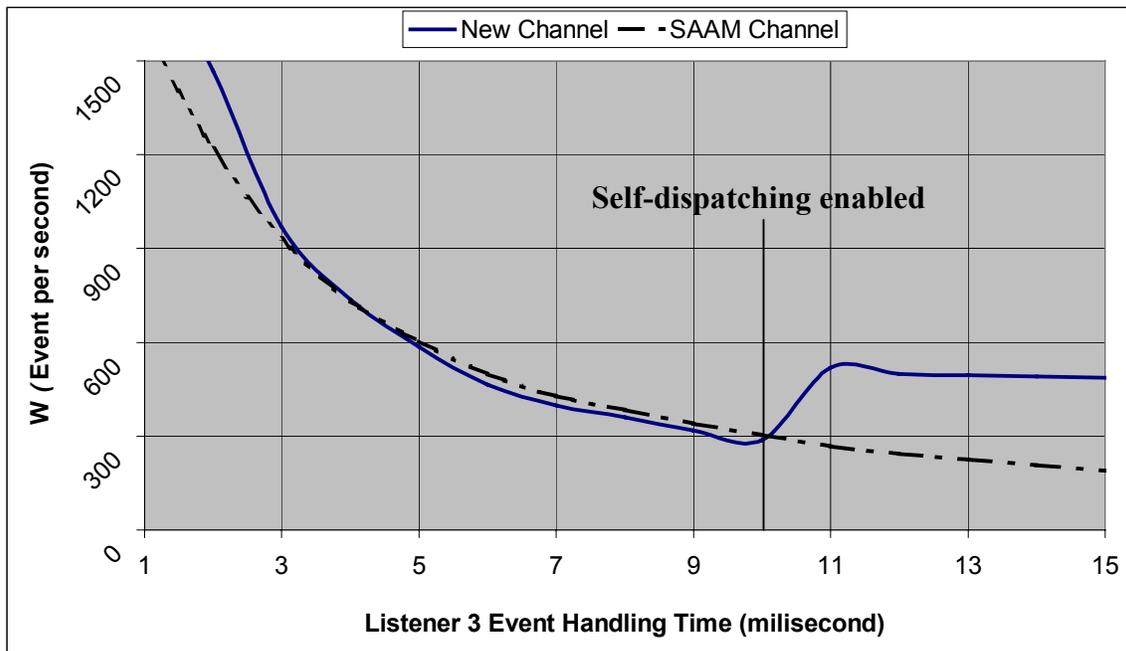


Figure 5.2b Effect of Self-Dispatching on Channel Work Rate

As anticipated, in the first phase of the experiment, the channel throughput and work rate decreased steadily as the event-handling time for *Listener 3* was increased without self-dispatching. As soon as self-dispatching was enabled for *Listener 3*, the average event delivery time decreased resulting in an increase to the channel throughput. The throughput quickly returned to the level observed before the event-handling time of *Listener 3* was increased. Additionally, the channel work rate increased significantly and remained stable despite steadily increasing the event handling time of *Listener 3*.

As discussed in Chapter 3, a current SAAM channel's throughput and work rate depend on the event handling time of each of its listeners. The new channel allows a developer to mitigate this dependency by providing a self-dispatching mechanism.

2. Channel Access Delay and Event Talk Time

The new channel design reduces event talk times significantly by buffering events in the channel scheduler. Invoking the *talk* methods only incurs delay for pushing the event into the channel scheduler. Unlike in the existing SAAM channel model, a talker does not have to wait until its event is delivered to all listeners. The reduction of individual event talk time consequently causes channel access delays to decrease.

Another experiment was performed to evaluate the channel access delays and event talk times of the new channel design. The test bed shown in Figure 5.1 was again used. For the experiment, the number of talkers was increased gradually, and the corresponding average channel access delay and event talk time were measured.

The results are depicted in Figure 5.3 and Figure 5.4. The event talk time and channel access delay were a few microseconds in contrast to the millisecond range for the existing SAAM channel. These results show that the new channel design can handle a larger number of channel talkers without causing scalability problems.

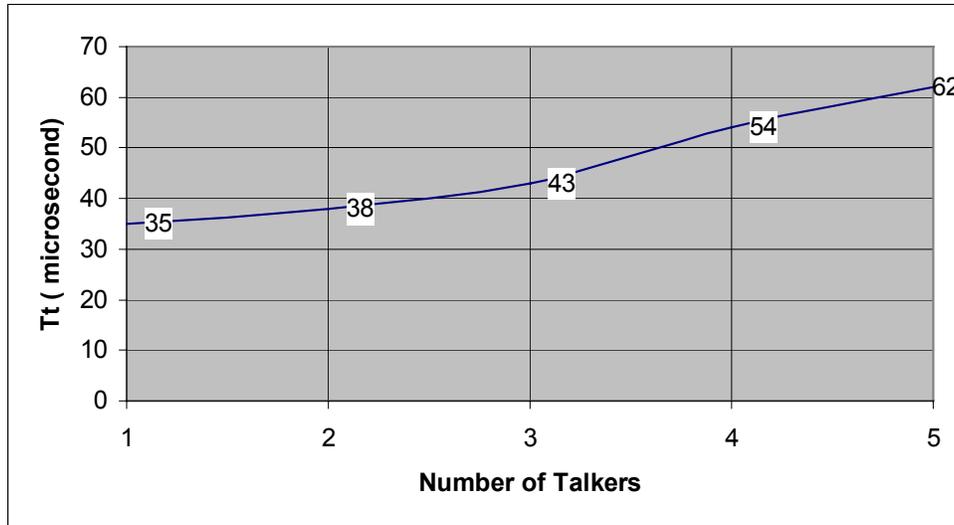


Figure 5.3 Event Talk Time versus Number of Talkers

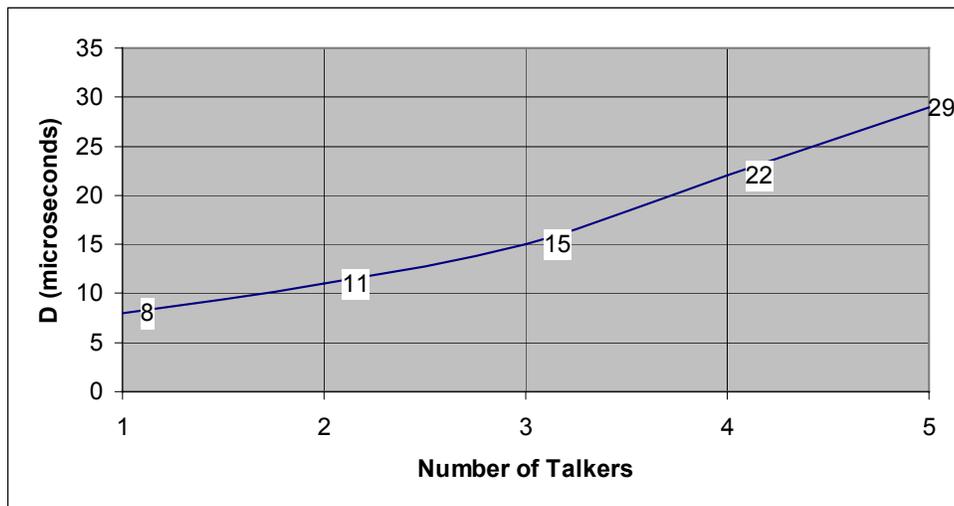


Figure 5.4 Channel Access Delay versus Number of Talkers

3. Scalability and Thread Count

In the new channel design, each channel employs one thread to perform its functionality. The thread dispatches events from the channel's scheduler to the channel's

listeners. This is necessary for event buffering, which is fundamental to minimizing the event talk time and channel access delay.

Since some systems can support only a limited number of threads, minimizing the channel (thread) count may be an important consideration. The new channel design addresses this scalability issue by providing support for event filtering and two-way communication. It does not restrict the number of participants for a channel and an object can be connected to both ends of a channel (being both a talker and listener) at the same time. As a result, it is feasible to reduce the number of required channels for an application, and thus the thread count, by increasing the number of participants on instantiated channels for that application.

4. Adaptability and Functional Flexibility of New Channel

The new channel design supports all event types (Java classes). It uses a simple and extensible event model instead of using a detailed, highly structured, and application-specific model. This generic event structure makes the channel package easily adaptable to either existing or new projects.

As discussed earlier, the new design partitions channel service into independent tasks, such as scheduling, filtering, and event dispatching. Developers may customize the functionality of each of these tasks to meet their specific needs. They may choose to use a particular scheduler for a channel. They may also use a specific set of event filters or activate self-dispatching for a listener. This functional flexibility makes the new channel configurable and tunable to meet different requirements for different projects.

5. Manageability

The new channel package offers a `ChannelAccessAuthority` interface and a `ChannelManager` class to help organize channels and control access of channel participants to these channels. They provide the necessary flexibility for managing the channels of a large application. A controller component based on them can offer a

common interface for channel participants to request channel service, control the access to existing channels, and ensure proper priorities are assigned to channel participants.

6. Ease of Use

A new channel can easily be constructed with the default channel scheduler by providing a channel identification number.

Method names of the channel package were carefully chosen according to the tasks performed by the methods, in order to help developers understand the channel functionality. The documentation files of the channel package were created in HTML format by using the *javadoc* utility.

a. Easy Channel Management

In SAAM, a `ControlExecutive` object acts as a channel manager. SAAM talkers and listeners cannot register with or talk to channels directly. They can only interact with channels via the `ControlExecutive` for monitoring and delivery of all traffic on the existing channels. The `ControlExecutive` class overrides the methods of the SAAM Channel to achieve this purpose. However, this design adds extra cost and complexity to the `ControlExecutive` class.

One of the ideas behind the `ChannelManager` class in the new channel package is to provide a built-in and ready-to-use common interface for all channel participants when developers want to manage all channels and event traffic from a central component.

b. Easy Self-Dispatching

A listener with a long event handling time blocks the channel dispatcher thread and the other channel listeners. With the new channel design, developers can eliminate this problem by enabling self-dispatching for that listener.

The same situation also appears when programming with the Java Event Model. This is because all Java events are, by default, executed in a single thread, *the event dispatching thread*. Java creators documented this and left the creation and handling of new threads to programmers when there is a need to perform lengthy

operations as resulting response to an event. In comparison, the self-dispatching mechanism described herein is much more programmer friendly. All thread creation and synchronization issues are handled internally and re transparent to the developers.

B. INTEGRATION OF CHANNEL PACKAGE AND SAAM PROTOTYPE

In order to integrate the new channel design into the SAAM prototype, some modifications and additions were made to the existing SAAM code. In this chapter, these modifications and additions are described.

1. Removing Obsolete Classes and Interfaces

The new channel package was added into the `org.saamnet` directory. The following classes and interfaces that are part of the old channel implementation were removed from the existing SAAM code:

- `org.saamnet.saam.control.Channel` class
- `org.saamnet.saam.event.ChannelException` class
- `org.saamnet.saam.event.SaamListener` interface
- `org.saamnet.saam.event.SaamTalker` interface

The new channel event model implementation can encapsulate any Java object in an event. Therefore the current SAAM event and message structure were preserved. As depicted in Figure 5.5, SAAM events are now encapsulated in a generic `ChannelEvent` object. Preserving the existing event structure reduced the complexity of the integration effort.

All SAAM classes that implemented the `SaamListener` interface have been modified to implement the new `ChannelListener` interface.

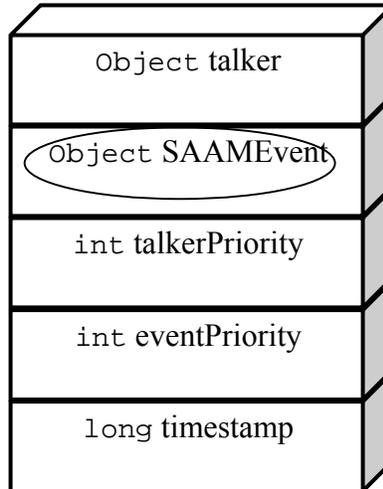


Figure 5.5 Encapsulating SAAM Events in ChannelEvent Structure

2. PermissionTableEntry Class

The `PermissionTableEntry` class was created to determine whether or not a channel participant has access to a given channel. A channel permission entry is created by providing the identification number of the channel and specifying the channel participants allowed to access this channel. The following method constructs an instance of `PermissionTableEntry` class:

- ```
PermissionTableEntry (int channel_id,
 String [] allowedTalkers,
 String [] allowedListeners)
```

It contains three simple methods for accessing the entry attributes for a channel. These methods are *getValidChannel\_Id*, *isValidTalker* and *isValidListener*. Additionally, its *getSaamPermissions* method returns a static permission table, as an array of permission table entries, containing the permissions for all channels, which are currently used in SAAM prototype, after the integration.

### **3. Changes to ControlExecutive Class**

As mentioned earlier, the existing `ControlExecutive` class overrides methods of the SAAM Channel to control the monitoring and delivery of all traffic on the existing channels. This adds extra complexity to the `ControlExecutive` class.

For the integration, all channel management related methods were removed from the `ControlExecutive` class. An instance of the `ChannelManager` class was added to the `ControlExecutive` class as a new data member, providing a common interface for channel access. The `ControlExecutive` class was also modified to implement the `ChannelAccessAuthority` interface. When the channel manager object is instantiated it includes a reference to the `ControlExecutive` object as a channel access control authority object. A *getChannelManager* method was added to the `ControlExecutive` class to provide a reference to the embedded channel manager for other SAAM components.

All SAAM components now request channel service via the channel manager, with authorization performed by the `ControlExecutive` object.

### **4. Reducing the Number of Channels**

The number of channels in the existing SAAM prototype depends on the number of interfaces and application agents in a node. When a new network interface is added to the node, a scheduler agent and a Network Interface Card (NIC) instance are created for the new interface automatically. In order to provide the event communication between these newly created components and the routing algorithm component, five new channels must be created. This scenario is illustrated in Figure 5.6 with each arrow representing a channel.

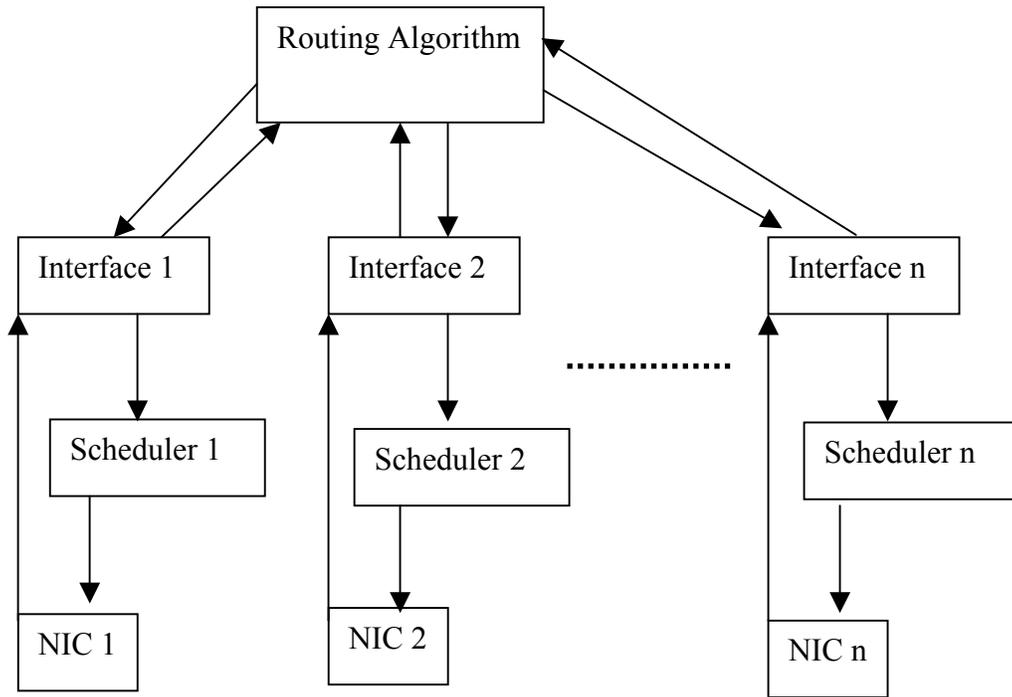


Figure 5.6 Necessary Channels for A New Interface on Existing SAAM Prototype

With integration of the new channel package to the SAAM prototype, a group of existing channels that are used for the same type of components were aggregated into one single channel by using the event filtering capability of the new channel design. A new channel is created only when new functionality (e.g., an application agent) is installed on a SAAM node. For example, only one channel is now used between all interfaces and their schedulers. Consequently, the number of channels for a given node no longer depends on the number of interfaces on that node.

Table 5.1 shows all the channels and their participants in the SAAM prototype after the integration. All talkers and listeners have the default priority of zero in each

channel. To ensure fairness between registered channel participants a PerTalkerRoundRobin scheduler is used.

| CHANNEL ID | CHANNEL TALKERS                                             | CHANNEL LISTENERS   | CHANNEL SCHEDULER      |
|------------|-------------------------------------------------------------|---------------------|------------------------|
| 80010      | Control Executive                                           | Transport Interface | Priority               |
| 80020      | Transport Interface<br>All Interfaces                       | Routing Algorithm   | Per Talker Round Robin |
| 80030      | Routing Algorithm                                           | Transport Interface | Priority               |
| 80040      | Routing Algorithm                                           | All Interfaces      | Priority               |
| 80050      | All Interfaces                                              | All Schedulers      | Per Talker Round Robin |
| 80060      | All NICs<br>Routing Algorithm                               | All Interfaces      | Per Talker Round Robin |
| 80070      | All Schedulers                                              | All NICs            | Per Talker Round Robin |
| 80080      | All NICs                                                    | Translator          | Per Talker Round Robin |
| 80090      | Translator<br>TranslatorPortListener                        | All NICs            | Priority               |
| 80100      | Translator<br>TranslatorPortListener<br>Transport Interface | Packet Factory      | Per Talker Round Robin |

Table 5.1 New Channel Structure in SAAM Prototype

## 5. Event Priorities

SAAM fulfills its functionality by communicating control messages between SAAM nodes. Therefore, losses of control traffic should be minimized to increase the reliability of the SAAM network and to recover from component failures quickly. For example, the loss of a *flow-response message*, which is sent to a SAAM router by an active SAAM server, would cause network resources to be wasted.

In order to give precedence to SAAM control traffic, a two-level event priority scheme was defined for SAAM events in the `ControlExecutive` class:

- `public static final SAAMEVENT_PRIORITY_HIGH = 10;`
- `public static final SAAMEVENT_PRIORITY_LOW= 0;`

The `ControlExecutive` and `Translator` classes were modified to assign high priority to SAAM control traffic and low priority to data traffic. Other classes were also modified to propagate priorities of events within the SAAM protocol stack.

## 6. Additions to SAAM GUI

The `Channel` and `ChannelManager` classes record channel activities for debugging purposes. The existing SAAM GUI shows only the existing channel identification numbers and participants of these channels. The main GUI of a SAAM node was modified to show debugging records of channels and the channel manager as illustrated in Figure 5.7.

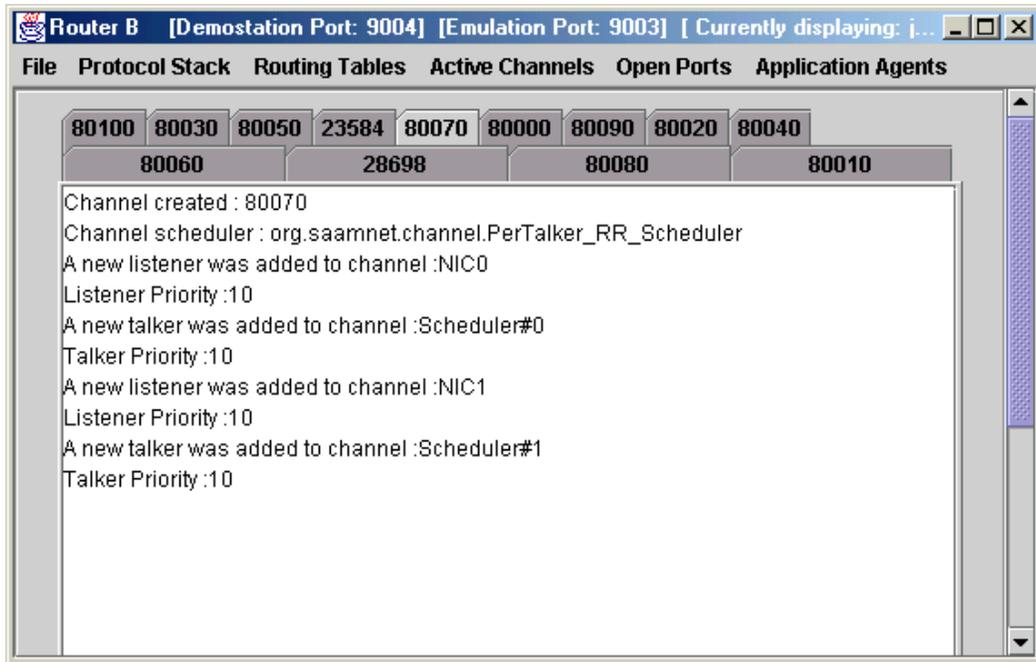


Figure 5.7 Snapshot of New Channel Debug Window

## VI. CONCLUSION

In this thesis, a new, generic, highly adaptable, and flexible event channel was designed and implemented. The main product is a Java utility package, called “channel.” This package was designed to help Java application developers create or enhance large systems using an event-based programming approach. The new channel design has several demonstrated performance advantages over existing event channel implementations. The flexibility and adaptability of the channel package was also validated by a successful upgrade of the channel mechanism of the SAAM prototype system. The remaining sections describe several lessons learned from this thesis effort and identify potential future work required to enhance the channel package.

### A. LESSONS LEARNED

#### 1. Programming with Threads

In the implementation phase of this work, many thread issues arose. Although using multiple threads in a program offers important benefits, it also requires a good understanding of thread related concepts like synchronization, race conditions, and deadlock avoidance. This understanding took considerable time to acquire. Multi-thread programming appears, on the surface, to be easy with Java. However, it is difficult to get it right.

#### 2. Integration with SAAM Prototype

Integration of the new channel package into the SAAM prototype turned out to be one of the more difficult parts of this thesis. This is because the SAAM prototype has more than two hundred Java classes and most of the SAAM components use event channels to communicate with each other. All of these components must be modified to work properly with the new channel design. It was necessary to understand the SAAM

concepts and functionality completely in order to configure and test the new channels properly.

## **B. FUTURE WORK**

### **1. Communication Between Distributed Applications**

The current channel model provides an event communication mechanism between components of one application only. However, the channel model can be extended to support event communication between different, even distributed, applications, as well. To do so, it would be necessary to define an appropriate channel naming convention and develop a communication protocol to deliver events across process or network boundaries.

### **2. Automatic Sense for Self-Dispatching**

For a channel, the event handling times of its listeners affect the channel throughput directly. Self-dispatching helps mitigate this problem. A channel listener may request its events to be dispatched by a separate thread by invoking the channel's *startListenerSelfDispatch* method.

Alternatively, an auto-sense mechanism can be embedded into the channel implementation to measure the event-handling time of each listener and automatically activate self-dispatching for a listener when specified criteria are met. In this manner, developers would no longer have to determine a priori which listeners require self-dispatching to optimize the throughput of a channel. This results in greater flexibility as it eliminates the need to hard-code when and where to initiate self-dispatching behavior on the part of specific listeners. Instead, the developers can focus on setting appropriate criteria for self-dispatching.

## LIST OF REFERENCES

- [1] Vrable, Dean J. and Yarger, John W., “The SAAM architecture: enabling integrated services”, Computer Science Department, Naval Postgraduate School, Monterey, September 1999.
- [2] Berg, Daniel J. and Fritzinger, Steven, “Advanced Techniques for Java Developers, Proven Solutions from Leading Java Experts”, John Wiley & Sons, Inc., 1997.
- [3] Niemeyer, Patrick and Knudsen Jonathan, “Learning Java”, O’reilly, 2000.
- [4] Grant, Palmer, “Java Event Handling”, Prentice Hall Ptr, 2001.
- [5] Siegel, Jon, “CORBA 3 Fundamentals and Programming”, John Willey & Sons, 2000.
- [6] Object Management Group, “Corba Event Service Specification Version 1.0”, June 2000.  
  
[[http://www.omg.org/technology/documents/spec\\_catalog.htm](http://www.omg.org/technology/documents/spec_catalog.htm)]
- [7] Object Management Group, “Corba Notification Service Specification Version 1.0”, June 2000.  
  
[[http://www.omg.org/technology/documents/spec\\_catalog.htm](http://www.omg.org/technology/documents/spec_catalog.htm)]
- [8] Iona, “Orbix Event Programmer’s Guide”, 2001.  
  
[[http://www.iona.com/docs/manuals/orbix/33/html/orbixevents33\\_pgguide/intro.html](http://www.iona.com/docs/manuals/orbix/33/html/orbixevents33_pgguide/intro.html)]
- [9] Bartlett, Dave, “Corba Junction: Corba 3.0 Notification Service”, May 2001.  
  
[<http://www-106.ibm.com/developerworks/components/library/co-cjct8/index.html>]
- [10] Paul, Hyde, “Java Thread Programming”, Sams, 1999
- [11] Moon, Jae-Chul, Park Jun-Ho, Kang Soon-Ju, “An event channel-based embedded software architecture for developing telemetric and teleoperation

systems on the WWW”, Real-Time Technology and Applications Symposium, Proceedings of the Fifth IEEE, 1999.

- [12] Zhou, Dong, Schwan, K., Eisenhauer, G., Chen, Yuan, “JECho - interactive high performance computing with java event channels”, Parallel and Distributed Processing Symposium, Proceedings 15th International, 2001
- [13] Goetz, Brian, “Threading lightly: Synchronization is not the enemy”, July 2001  
[<http://www-106.ibm.com/developerworks/library/j-threads1>]
- [14] Meyer, Bertrand, Object-Oriented Software Construction, Prentice Hall, 1988
- [15] Rege, K., “Design patterns for component-oriented software development”, EUROMICRO Conference, 1999. Proceedings. 25th, Volume: 2, 1999
- [16] Karin Högstedt, Doug Kimelman, V.T. Rajan, Tova Roth, Mark Wegman, Nan Wang, “Optimizing Component Interaction”, ACM SIGPLAN Notices Volume 36 Issue 8, August 2001

## INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center  
Ft. Belvoir, Virginia
2. Dudley Knox Library  
Naval Postgraduate School  
Monterey, California
3. Deniz Kuvvetleri Komutanligi  
Personel Daire Baskanligi  
Bakanliklar  
Ankara, TURKEY
4. Deniz Harp Okulu Komutanligi  
Kutuphanesi  
Tuzla  
Istanbul, TURKEY
5. Chairman, Code CS  
Naval Postgraduate School  
Monterey, California 93943-5118
6. Prof. Geoffrey Xie, Code CS/Xg  
Naval Postgraduate School  
Monterey, California 93943-5118
7. LCDR Chris Eagle  
Naval Postgraduate School  
Monterey, California 93943-5118
8. LTJG Cihat Eryigit  
Arastirma Merkezi Komutanligi  
Pendik  
Istanbul, TURKEY
9. Tolga Demirtas, LTJG  
Deniz Harp Okulu Komutanligi  
Tuzla  
Istanbul, TURKEY