A Model of Computation with Push and Pull Processing

by Yang Zhao

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

Professor Edward A. Lee
Research Advisor

* * * * * *

Professor Alberto Sangiovanni-Vincentelli
Second Reader

Published as:
Technical Memorandum UCB/ERL M03/51,
Electronics Research Laboratory,
University of California at Berkeley,
December 16, 2003.

Abstract

This report studies a model of computation (MoC) that supports Push-Pull communication between software components. Push represents message transfer that is initiated by the producer. On the contrast, Pull represents message transfer that is initiated by the consumer.

Push-Pull messaging mechanisms have been used in various domains, such as the CORBA Event service, the Click modular router, inter-process communication among multi-processors, data dissemination in a peer to peer network. Formalizing the computation and communication in these domains to a MoC facilitates reusability of programming models and software architectures, and allows user to model and analyze more complex systems.

This model of computation has been implemented as the Component Interaction (CI) domain in Ptolemy II, a hierarchical and heterogeneous modeling environment. Composition of the CI domain with other domains in Ptolemy II is discussed.

This report also connects CI to other data-flow models of computation, such as Process Networks (PN), Synchronous Data Flow (SDF) and Dynamic Data Flow (DDF), and discusses the uses of CI in the execution of distributed systems.

Acknowledgments

This report describes joint work with Xiaojun Liu. I would like to thank him for his significant work and for all the wonderful discussions. Without his help, I would not have finished this project.

I am deeply grateful to my advisor Professor Edward Lee for his support, encouragement and excellent guidance. I would also like to thank him for his broad research vision, deep thinking and rigorous attitude.

I would also like to thank Professor Alberto Sangiovanni-Vincentelli for reviewing this report and providing valuable comments.

Many thanks and appreciation goes to all the people in the Ptolemy group. Our stimulating discussions and the fun group activities have enriched my life at Berkeley.

Finally, I would like to thank my husband Ye Tang and my parents for their continuous love and encouragement. This report is dedicated to them.

Contents

Introduction

The push-pull communication model supports a mixture of data-driven and demand-driven communications. It addresses problems like which component can initiate an interaction, and how the interaction should be propagated to other components. An interaction is “push” if it is initiated by the data sender; and an interaction is “pull” if it is initiated by the data receiver. Components in this model can be classified as active components and passive components. Only active components will start interactions either by sending data or by sending demand requests. Comparing to pure data-driven or demand driven models, it offers more flexible communication patterns between components.

This model provides the designer a freedom of control to specify which parts of the model are data-driven and which parts are demand-driven. It also allows the designer to explicitly manage the data storage, such as where the queue is, what size it is, and what queuing mechanism is used. Data producers and consumers in this model can operate on different rates. This can be highly essential for applications dealing with network interfaces or distributed applications that have components executing on heterogeneous platforms with various processing power and memory capacities.

The report is organized as following: chapter 1 overviews some related work that has the push-pull communication pattern; chapter 2 describes the semantics of this model; chapter 3 discusses the implementation of it in Ptolemy II; chapter 4 relates this work to some

other data-flow models and illustrates that this model is easy to distribute, and future work is pointed out then.

Chapter 1

Related Work

The push-pull model is very much inspired by observing some novel communication mechanisms adopted by a variety of fields, including middleware services, The Click modular router, web applications, communication among multi-processors [6], data dissemination in a peer to peer network [7]. As an example, the interaction between a web server and a browser is mostly demand-driven. When the user clicks on a link in the browser, it pulls the corresponding page from the web server. A stock-quote service can use an event-driven style of computation. The server generates events when a stock price changes. The events drive the clients to update their displayed information. Such push/pull interaction between a data producer and consumer is common in distributed systems, and has been included in middleware services, most notably the CORBA event service. These services motivate the design of the push-pull model. Other applications include database systems, file systems, and the Click modular router.

This chapter studies hybrid push-pull communication patterns in the specification of the CORBA Event Service and the Click modular router, as the first one represents interaction between distributed components and the second one represents interaction in applications dealing with network interfaces.

1.1 CORBA

This section gives a brief overview of CORBA and the CORBA Event Service. It then discusses four representative communication patterns.

1.1.1 Introduction to CORBA

The Common Object Request Broker Architecture (CORBA) is a distributed object computing middleware standard specified by the Object Management Group (OMG). It has been widely accepted by the distributed computing research and industry, with the notable exception of Microsoft, which defines a competing object broker called Distributed Component Object model (DCOM). Discussion of differences between CORBA ORBs and DCOM is beyond the scope of this report. Readers can refer to chapter 14 of [1].

CORBA provides a framework where a client objects can send requests to a remote server object and get the results back (if the operation is successful). Figure 1.1 shows the basic

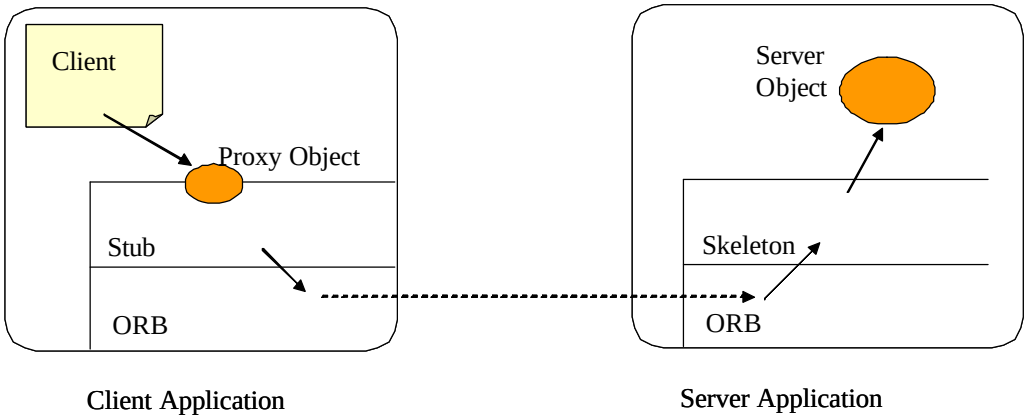


FIGURE 1.1 The basic CORBA communication model

communication between a client object and a server object. The client invokes an operation on the local proxy of a server object. The invocation usually contains data arguments that will be marshaled by the stub code. The client's ORB runtime sends a message, which contains the operation and the marshaled data, to the server's ORB. Once a client sends an invocation to a server, it is blocked waiting for the reply. The server side ORB runtime will get the message from the network and pass it to the skeleton code. The skeleton code un-marshals the data and invokes the operation on the target object. The reply of the operation will be sent back to the client object. When the client receives the result, it continues its execution. If the invocation fails, the client will receive an exception.

This basic request-reply communication paradigm is essential to the CORBA architecture and is straightforward to use. However the client and server are very coupled in this approach: they both need to be present at the time the request is invoked, the client needs to have the reference of the server object, and the client blocks for the reply. Morover, the basic model only supports uni-cast communication. In some scenarios more decoupled, many-to-many communication between clients and servers is desired.

1.1.2 Event Service

The Event Service is one service component in the OMG Common Object Services Specification (COSS). It defines a communication model that allows event data to be delivered from suppliers to consumers without requiring these participants to know about each other explicitly [3].

An event in this context means an occurrence within an object, specified to be of interest to one or more objects. The object that originates an event is called an event supplier. The

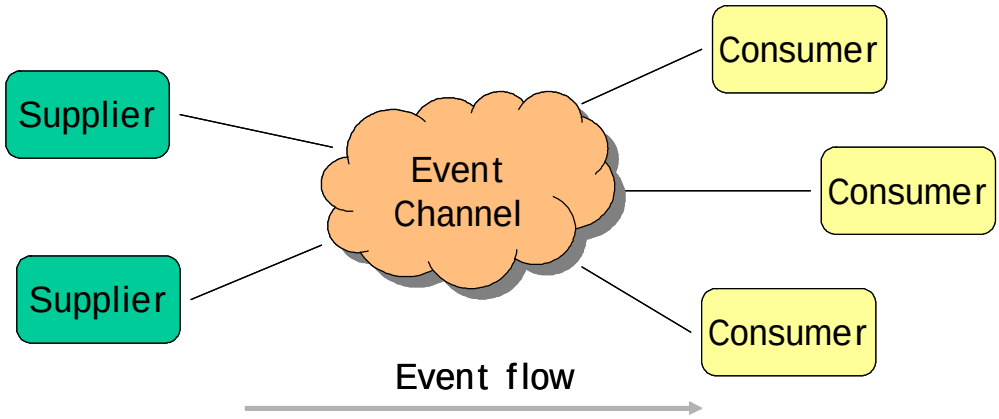


FIGURE 1.2 The event service model

object that receives an event is called an event consumer. The supplier and consumer don't talk directly. Instead, a special object called the event channel is introduced to mediate the communication. Figure 1.2 illustrates the architecture of this model. Suppliers and consumers connect to the event channel but not directly to each other. Conceptually, the event channel appears as a single consumer to the supplier and as a single supplier to the event consumers. However, to avoid name confusion, the term “supplier” is specially saved for the object that sends events to the channel and the term “consumer” is saved for the object that receives events from the channel. Suppliers do not need to know how many consumers to talk to or what their identities are, and vise versa for consumers. In this way, the event channel decouples suppliers and consumers. There is no need for the supplier and consumer to be both present, as in the reply-response model. An event channel can connect any number of suppliers and consumers. Suppliers or consumers can join or leave the system dynamically. Suppliers or consumers can also connect to multiple event channels.

1.1.3 Push -Pull Communication

An interaction between a message sender and receiver is identified as push-style if the message transfer is initiated by the sender. In contrast, a pull-style communication represents interactions initiated by the receiver. An event channel may communicate with a supplier using one style of communication, and communicate with a consumer using a different style of communication.

We discuss four typical communication models that the CORBA event service identifies: the canonical push model, the canonical pull model, the hybrid push-pull model and the hybrid pull-push model.

1.1.3.1 The push Model

In the push model as shown in figure 1.3, a supplier generates events and passes them to an event channel. The channel then transfers the events to registered consumers. In this

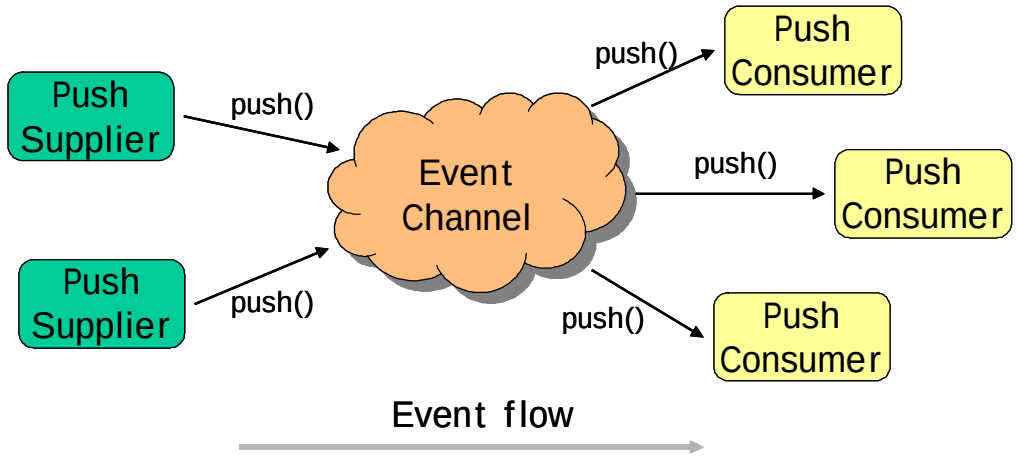


FIGURE 1.3 The push-push model

model, the interaction is started by the supplier and the consumer only passively waits for events.

1.1.3.2 The pull Model

In the pull model, a consumer actively sends request to the event channel for some event. The event channel then propagates the request to a supplier to provide the event. In this model, the interaction is initiated by the consumer, and the supplier waits for a pull request to arrive and then generates the event to be returned to the pulling consumer. The consumer may or may not block for the response. This is determined by the implementation of the supplier, consumer and event channel rather than by specification. Figure 1.4 illustrates a Pull model architecture.

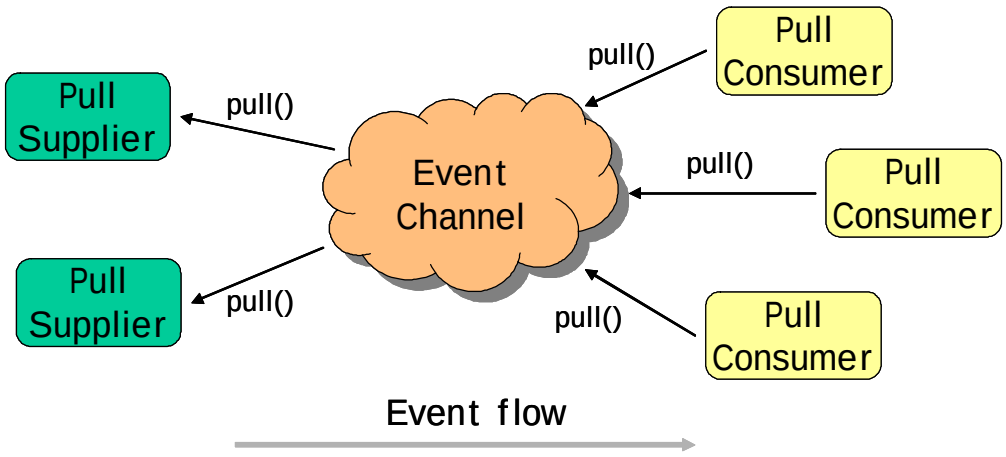


FIGURE 1.4 The pull-pull model

1.1.3.3 Hybrid push-pull Model

In this model, the interaction between a supplier and the event channel is push, and between a consumer and the event channel is pull. A supplier generates an event and passes it to the event channel. The channel does not transfers the event to registered consumers until consumers pull for the event data. In this model, both suppliers and consumers actively initiate the interaction with the channel and the event channel acts as a queue component. If suppliers and consumers operate at different rates, the channel may need an unbounded queue to store the events. If this is not practical, the channel may need to implement some dropping mechanism or blocking when the queue is full. The model is shown in figure 1.5.

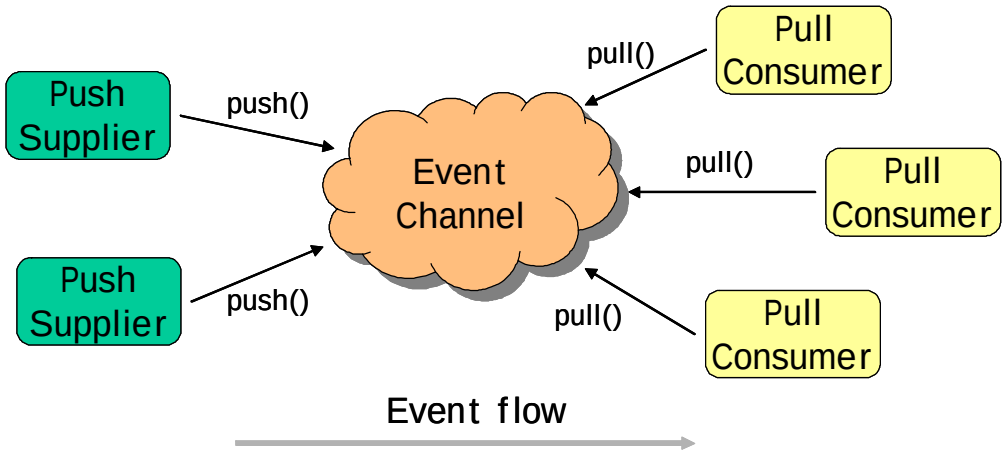


FIGURE 1.5 The push-pull model

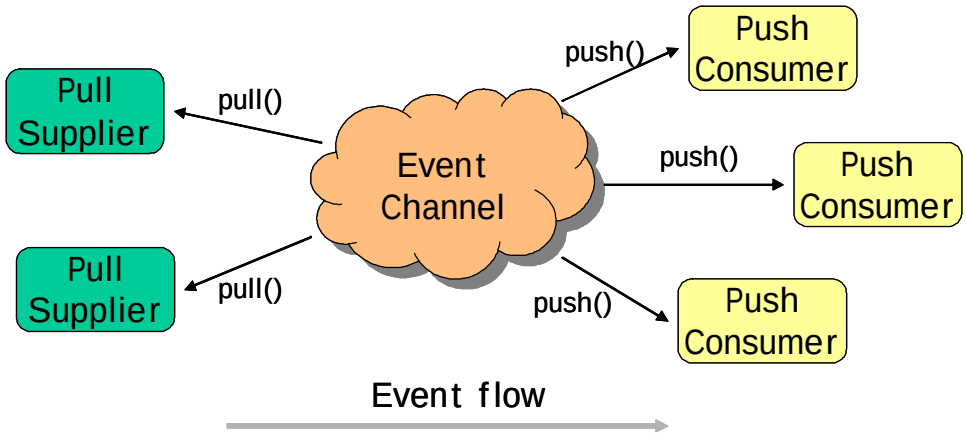


FIGURE 1.6 The pull - push model

1.1.3.4 Hybrid pull-push Model

Another combination is shown in figure 1.6. The interaction between a supplier and the event channel is pull, and between a consumer and the event channel is push. In this model, both suppliers and consumers are passive, but the event channel shall actively request data from the suppliers and pass them to consumers.

Although this model may be a bit strange, it has some interesting aspects. Conceptually, the event channel can connect to any number of suppliers and consumers that can execute at various frequencies. This model gives the channel more control on how to coordinate these components. Different scheduling algorithms may be implemented in the event channel to manage the interactions.

1.1.3.5 More mixed model

There are still other combinations of push-pull in a system. An example is shown in figure 1.7. The interaction in these models becomes more complex.

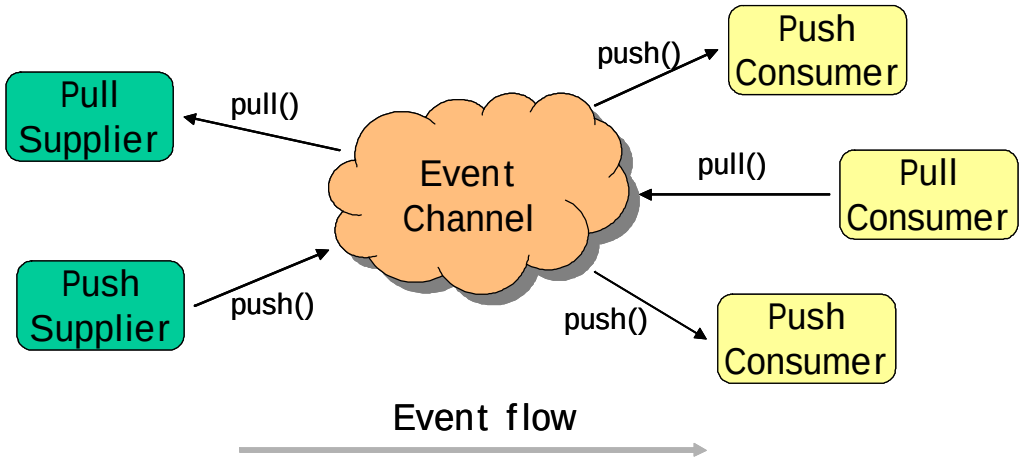


FIGURE 1.7 An example of a mixed model

1.2 Click

Click is a software architecture for building flexible and configurable routers and other packet processors, such as firewalls (dropping inappropriate packets), network address translators (rewrite packet headers) [4]. It presents a modular architecture and software toolkit for building packet processors at a proper abstract level.

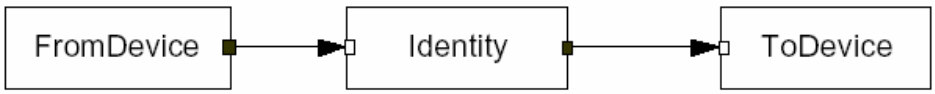
Click supports both push and pull processing. While push is for responding to packet arrivals at the input network interface, pull is for requesting data for the output interface and for scheduling. In the original design, a click router shares Linux's interrupt structure and device handling. The push processing is triggered to execute when input packets arrive. On the other hand, an output device generates an interrupt when its transmit queue has space for packets and triggers the pull processing to execute. When there are packets with different priority, a click router may use some scheduling control to pull data from

different priority queues. To overcome interrupt overhead and improve performance, the current click design uses polling to examine whether there are packets in the input device's packet queue or whether there is free space in the output device's transmit queue, and it executes push or pull processing accordingly.

We discuss the software architecture and the semantics in the following sections.

1.2.4 Click Software architecture

Click routers are built from fine-grained packet processing components called elements. Each element is a C++ class and represents a software component that performs particular computations, including classifying, queueing, IP rewriting. An element may have several input and output ports. Figure 1.8 shows an Identity element with one input port and one output port, and part of its code is also shown under it. An Element can connect to other elements via ports. In figure 1.8, the Identity element is connected to a FromDevice element and a ToDevice element. The arrow shows the direction of the dataflow.




```
class Identity: public Element { public:
Identity() { add_input(); add_output(); }
......
void push(int port, Packet *p) { output(0).push(p); }
Packet *pull(int port) { return input(0).pull(); }
};
```

FIGURE 1.8 The Click Identity Element

Port defines the interface that an element uses to interact with other elements. A port has a push method and a pull method. For an input port, its push method can call the element's push method and pass the packet as an argument; its pull methods can be called by the element when the element wants to pull data from its upsteam element. For an output port, its push method can be called by the element when it wants to pass data to its downsteam element; its pull method can call the element's pull method.

An element sets its ports to be push or pull according to the configuration string when the router is initialized. Elements may also contain agnostic ports, which act as push when connected to push ports and as pull when connected to pull ports. A connection connects an output port to an input port. The types of ports at the endpoints of a connection need to be the same, or at least one must be agnostic. A connection can either be push or pull, determined by the ports it connects to. On a push connection, packet transfer is initiated by the source element and is passed downstream to the destination element. On a pull connection, in contrast, the destination element initiates packet transfer and the source element provides the packet as the returned result to the request.

In Click, push and pull connections are complementary mechanisms for packet transfer. Push connections are appropriate for communication triggered by asynchronous events, for instance, reacting to unsolicited packets arrivals at an input device. On the other hand, pull connections instruct the source element to send data only when the destination elements are ready to process the data, for example, to transmit a packet only when the transmitting device is ready.

1.2.5 Semantics

A Click router is executed by a single thread, and the execution is managed by the scheduler. The scheduler maintains a list of elements to be scheduled. An element can be put into the list by an I/O interrupt handler or a software trigger, such as a timer. The data structure in the list contains the element's name and a pointer to the function in the element that will be called when it is scheduled. And the scheduler will loop on this list and schedule elements sequentially according to the scheduling mechanism implemented in the scheduler. There is currently no preemption mechanism in Click.

When an element is scheduled, the scheduler calls the method that the element specified to call, which may in turn calls other methods. When the method returns, the scheduler will try to schedule the next element in the list to run.

To understand the semantics better, let us look at the example shown in figure 1.9. Assume that:

- A1 is related to a hardware interrupt for receiving data, and when the interrupt happens, it puts A1 to the scheduler's list and a pointer to A1's push method.
- A2, and A3 are related to hardware interrupts that happen when the devices are ready to send data. When the interrupt for A2 happens, it puts A2 to the list and a pointer to A2's pull method. Similarly for A3.
- Element B is implemented to work as follows:

  if configured as push input push output, then:

  when the push method of the *in* port is called, it calls B's push method, whih calls *out1'*s push method, *out2'*s push method and *out3*'s push method;

if configured as pull input pull output, then:

when *out1*'s pull method is called, it calls B's pull method, which calls *in*'s pull method. And it is similar for *out2* and *out3*.

- D, E and F are implemented similarly to B except that each of them only has one output port.
- C works as follows: when its input's push method is called, it calls C's push method.
- queue1 and queue2 are configured as push input pull output and work as follows:

  when the push method of queue1's (queue2's) input is called, it calls queue1's (queue2's) push method;

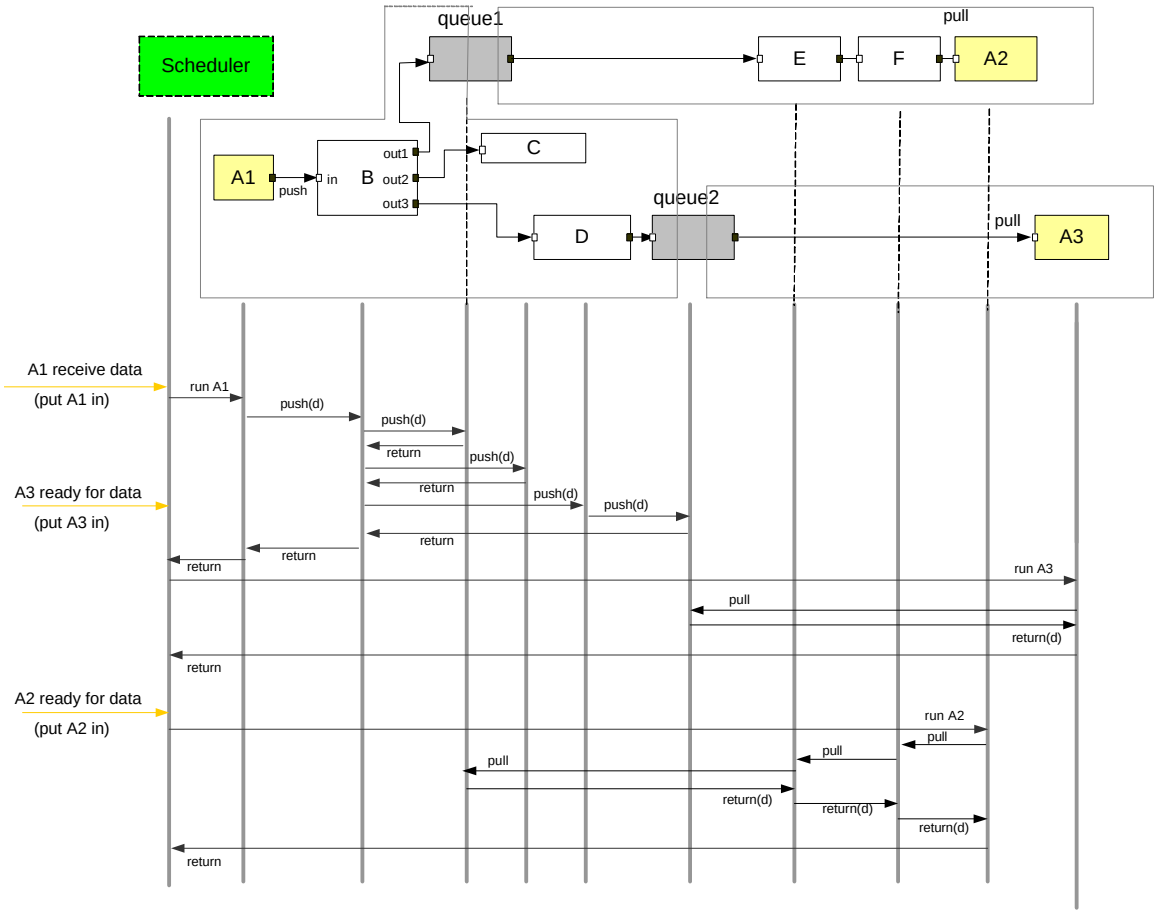


FIGURE 1.9 An simple model to illustrate the Click semantics

when the pull method of queue1’s (queue2’s) output is called, it calls queue1's (queue2’s) pull method.

Based on these assumptions, the corresponding sequence diagram for a particular order of interruptions for A1, A2, and A3 are shown in figure 1.9.

From the sequence diagram, we can see that when A1 is scheduled, the execution starts from A1's push method, and propagates to call other corresponding push methods. It won't return until it reaches a queue element or a push sink element, such as the C element. Similarly, when A2 is scheduled, it starts from A2's pull method and propagates to call other corresponding pull methods and returns the data or null until it reaches a queue element (it could also return when it reaches a pull source element).

Let us define a push island to be a set of elements connected by push connections. For example, the island that A1 belongs to (shown by a gray box around it in figure 1.9) is a push island. Correspondingly, a pull island is a set of elements connected by pull connections. For example, the island that A2 belongs to, which is shown by another gray box around it in figure 1.9, is a pull island. Any Click model can be divided into push and pull islands. The execution in each island is synchronous method calls, but the execution between islands is asynchronous. The queue element forms the boundary between islands. From this point of view, it is quite similar to the globally asynchronous locally synchronous (GALS) execution in TinyGALS [5].

Since the queue actors affect the partitioning of a model, where to put a queue element can have important influence on the behavior of the whole model. Consider a chain of elements as shown in figure 1.10. If the throughput of the `FromDevice` element varies with

a bursty period and an idle period, but the `ToDevice` element sends data with a constant rate that is much lower than the bursty rate of the data source, where should the queue element be? In this case, putting the queue closer to the data source is preferred, since packets that have to be dropped should be dropped early. What if the `ToDevice` has a higher throughput? In this case, some pull requests for data may return null; we then may prefer to put the queue closer to the data sink so that it does not need to pull through several elements to determine that there is no data to send at all. So there are trade offs for different situations.

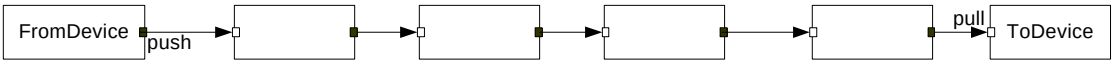


FIGURE 1.10 Example illustrating the question of where to put a queue element

Chapter 2

Push-Pull Model

In this section, we define a model of computation (MoC) for design and analyzing systems using push and pull communications.

A model of computation describes how components in a concurrent system can communicate and compute data. Different systems may desire different MoCs and some complex systems need more than one MoC. For example, dataflow MoCs (such as process network, synchronous dataflow) are suitable for signal processing systems, discrete event models are suitable for event driven systems, finite state machine models are appropriate for control systems, and hybrid systems often demand multiple MoCs.

Most of MoCs defined to date describes constraints on communication protocols between components (ex. by rendezvous or FIFO queue) and ordering of component execution. The execution in these MoCs usually starts from the data source or event source and propagates to downstream components. From the discussion in Chapter 1, we see that there is a set of systems, including distributed systems, packet processors, database systems, processor networks and peer-to-peer systems, that demand a MoC that supports both data driven and demand driven interaction and provides the abstraction to specify which component in

the system can initiate the computation and how related components will be invoked. We study these issues in the Push-Pull model.

In the following sections, we first introduce the actor component model used in the Ptolemy project [8]. Then we define a set of concepts and classify different component types used in the Push-Pull model. Finally we discuss the operational semantics.

2.1 Component model

Actor-oriented design is a component methodology that is particularly effective for system-level design [10]. The term *actor* was first introduced by Carl Hewitt in the 1970's to describe the concept of autonomous reasoning agents [11], and it evolved through the work of Agha and others to describe a formalized model of concurrency [12]. Agha's actors each have an independent thread of control and communicate via asynchronous message passing. The term is further developed by Edward Lee and the Ptolemy group to embrace a larger family of models of concurrency that are often more constrained than general message passing. The Ptolemy actors are still conceptually concurrent, but they need not have their own thread of control.

Figure 2.1 shows a simple Ptolemy model. An actor encapsulates its computation and internal state. It interacts with its environment and other actors via a well-defined interface, which includes *ports* and *parameters*. Ports represent the points of communication. Parameters are used to configure the computation of an actor. And the channel of communication between actors is implemented by a *receiver* object contained by ports. A receiver

defines the communication protocol between actors. The *director* defines the ordering of actors execution. A MoC is implemented by a particular director and receiver.

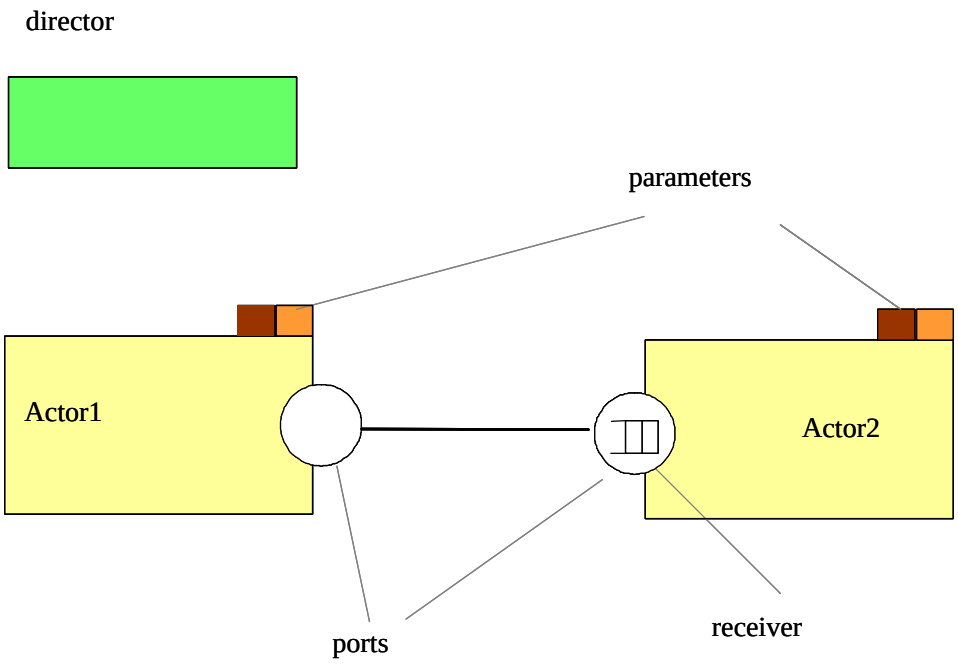


FIGURE 2.1 Components in a Ptolemy model

2.2 Basic Concepts

Now we define a set of concepts that are used in the push-pull model.

2.2.1 Push port vs. pull port

Each port contains a attribute named *push/pull*, which can be configured with three possible values: `push`, `pull` and `agnostic`. A port is a push port if the attribute is set to `push`, and is a pull port if the attribute is set to `pull`. An agnostic port can be either push

or pull. The exact type is resolved according to its connected ports with the assumption that connected ports have to be the same type.

2.2.2 Push connection vs. pull connection

The type of a connection is defined by the two ports at its endpoints. A connection is push type if it connects two push ports; a connection is pull type if it connects two pull ports.

We restrict the push-pull model to not allow mixed port type combinations for a connection, such as connecting a push output port to a pull input port, and a pull output port to a push input port. Intuitively, a connection that connects a push output port to a pull input port imports an implicit queue on this connection. This is not allowed since this model is intended to let the designer to model and manage the storage property explicitly due to the asynchronous interaction between push processing and pull processing. If a designer does want to link a component with a push output to a component with a pull input, he/she can use a *queue* actor (with push input and pull output) as a mediator. The other kind of connection, connecting a pull output port to a push input port, is not allowed because it is a lifeless connection, i.e. no data can pass through it. In case one needs to link a component with pull output to a component with push input, a mediating actor with a pull input and a push output can be used, and such an actor is called a *schedule* actor. Details about the queue actor and the schedule actor are discussed in section 2.3.

2.2.3 Passive vs. active actors

The push-pull model classifies actors as two kinds, passive actors and active actors, as shown in figure 2.2, according to their ports' configuration.

Passive actors include:

- Source actors (actors with no input) with pull outputs. As an example, this can be used to model a server that only responds to pull requests from clients.
- Sink actors (actors with no output) with push inputs. As an example, this can be used for a data consumer that only reacts to input events.
- Actors with pull inputs and pull outputs. This kind of actor reacts to pull requests from its downstream actors and propagates the requests to its upstream actor if it dose not have the required data to satisfy the request.
- Actors with push inputs and push outputs. This kind of actor reacts to data received at its input port, processing the data and outputing the processed data to its downstream actors.
- Actors with push inputs and pull outputs (we call them queue actors).

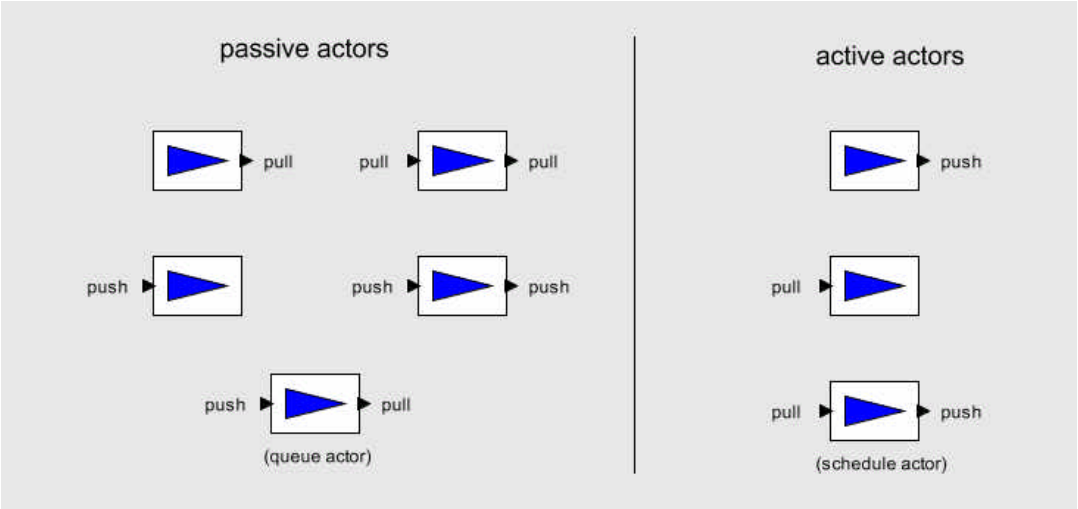


FIGURE 2.2 Passive actors and active actors in the CI domain

Active actors include those that can either initiate production of data or can pull requests:

- Source actors with push outputs, such as an interrupt source.
- Sink actors with pull inputs, such as an output interface of a router that pulls the next packet to send.
- Actors with pull inputs and push outputs (we call them schedule actors)

2.3 Queue actor and schedule actor

2.3.4 Queue actor

A queue actor has a push input port and a pull output port. It responds to a pushed token at its input by enqueueing it, and to a pulled request at its output port by dequeueing a token from it and returning the token. This is to say that a queue actor only reacts to events or requests from other actors, i.e., it never passes tokens or requests actively to other actors.

A queue actor can act as a mediator between an actor with a push output port and an actor with a pull input port.

We can implement different queue actors using different storage policies. For example, besides simple FIFO queue actor, one can implement a queue actor that drops token from its head (FrontDropQueue) or a queue actor that drops token from its tail (BackDrop-Queue) when the queue is full. User can decide where to use a queue actor or which queueing mechanism to use. Such a decision may have important effects on a system's properties.

2.3.5 Schedule actor

An actor with pull input ports and push output ports is also called a schedul actor. Such an actor is active; it actively pulls data from its upstream actors according to its local schedule, processes the data if necessary and pushes the data to its downstream actors. A scheduler actor models some component that can control the timing of requesting and distributing data. It is for this reason that we call it a schedule actor.

A schedule actor can act as a mediator between an actor with a pull output port and an actor with a push input port.

Schedule actors can implement different scheduling algorithms, such as round-robin scheduling, priority scheduling. And the designer may compose different schedule actors and queue actors for more complex operations.

2.4 Operational Semantics

The execution of the push-pull model is multi-threaded. However, unlike the process network models, CSP models or DDE models, not every actor has its own thread. In this model, only active actors possesses their own thread of execution. All the passive actors share a single thread managed by the director which schedules them to execute. From this perspective, we say that the execution of the push-pull model uses both a process-based approach and scheduler-based approach.

The execution of each active actor is controlled by a separate thread. The semantics for the three kinds of active actors are described as follows:

- An active source actor may fire with some frequency or according to some interrupt or event arrivals that it listens to. When it fires, it pushes data to its connected downstream actor, which in turn causes the downstream actor to be enqueued into the director's *AsynchronousPushedActors* queue. The term asynchronous here simply means that the downstream passive actor can be added to the director's queue at any time.
- An active sink actor may fire when it is ready to process the data. When fire, it issues a pull request to its connected upstream actor, which in turn causes the upstream actor be enqueued to the director's *AsynchronousPulledActors* queue. It waits until the request is satisfied and then continues to process it.
- A schedule actor with pull inputs and push outputs models a component that can control when and where to request and distribute data. When a schedule actor fire, it requests data from some of its connected upstream actors and wait until the data is ready, and then pushes the token to its connected downstream actors. The first step causes the upstream actors be added to the director's *AsynchronousPulledActors* queue, and the second step causes the downstream actors be added to the *AsynchronousPushedActors* queue.

In contrast to active actors, the execution of passive actors is controlled by the director. A passive actor that has push inputs push outputs will be fired by the director if it receives tokens at its input ports, and this may trigger its downstream actors with push inputs to fire; a passive actor that has pull inputs pull outputs will be fired if it receives a pull request at its output ports and will propagate the request to its upstream actors if it does not have the data pulled. In particular, when fired, a passive source actor will provide the

data (when it is available) and deliver it through the path that the pull request takes. A queue passive actor will fire when it receives a token, but it just saves the token in its local queue and does not trigger its downstream actors to fire; it is also fired when it receives a pull request at its output port, which returns a token in the queue but does not propagate the pull request any more.

Another possible approach of executing passive actors is to execute them in the calling thread of the active actor. But the calling thread can be blocked when executes the passive actors, and not be able to response to event arrivals or interrupts. We would like to explore this approach further by allowing preemptions in the execution of the passive actors.

Let's look at an example to understand the semantics. Figure 2.3 shows a simple router model that contains a FromDevice actor that listens to the input network interface and outputs the received data. The classifier actor identifies the data and puts them to three different queues, for example. The RoundRoubinSelector select data from the three queues according to the round-roubin algorithm when the ToDevice actor request data. The ToDevice actor requests data when it is notified to fill the output network interface.

According to the semantics we discussed before, FromDevice and ToDevice are the active actors in this model, and all others are passive actors.

At the beginning, assume FromDevice has no data to output and waits for input, and ToDevice requests data.

The RoundRoubinSelector is added to the *AsynchronousPulledActors* queue, and the director then wakes up to handle the request.

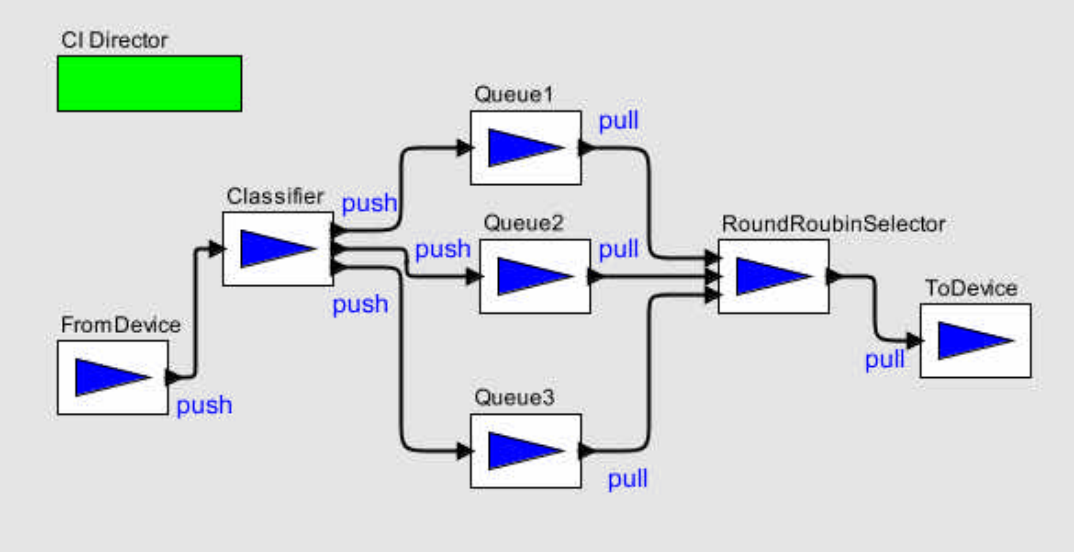


FIGURE 2.3 A simple router model

The RoundRoubinSelector first requires data from its first input. Since no data are available at this time, it then propagates to Queue1. No data are available at Queue1. Since Queue1 is a queue actor with push input, the propagation stops. And the director waits for firable actors.

Assume now the FromDevice receives packets and fires. This then adds the Classifier to the *AsynchronousPushedActors* queue.

The director wakes up and fires the Classifier actor. Let's simply assume that the data are sent to Queue1. This in turn triggers Queue1 to fire. It first enqueue the data, then, realizing that there is a request waiting for its output, also calls the director to fire it again to satisfy the request. (If the data are not sent to Queue1, then the director waits again.)

Now the RoundRobinSelector is enabled to fire and outputs the data to the ToDevice actor. The thread that manages the execution of this active actor then wake up to process the data.

The execution continues with new data arrivals and request issues.

Chapter 3

Implementation

The push-pull model is implemented as the Component Interaction (CI) domain. The kernel of the CI domain is realized in package `ptolemy.domains.ci.kernel`. The static structure diagram of the package is shown in figure 3.1. The important classes are `CIDirector, CIReceiver, and ActiveActorManager`.

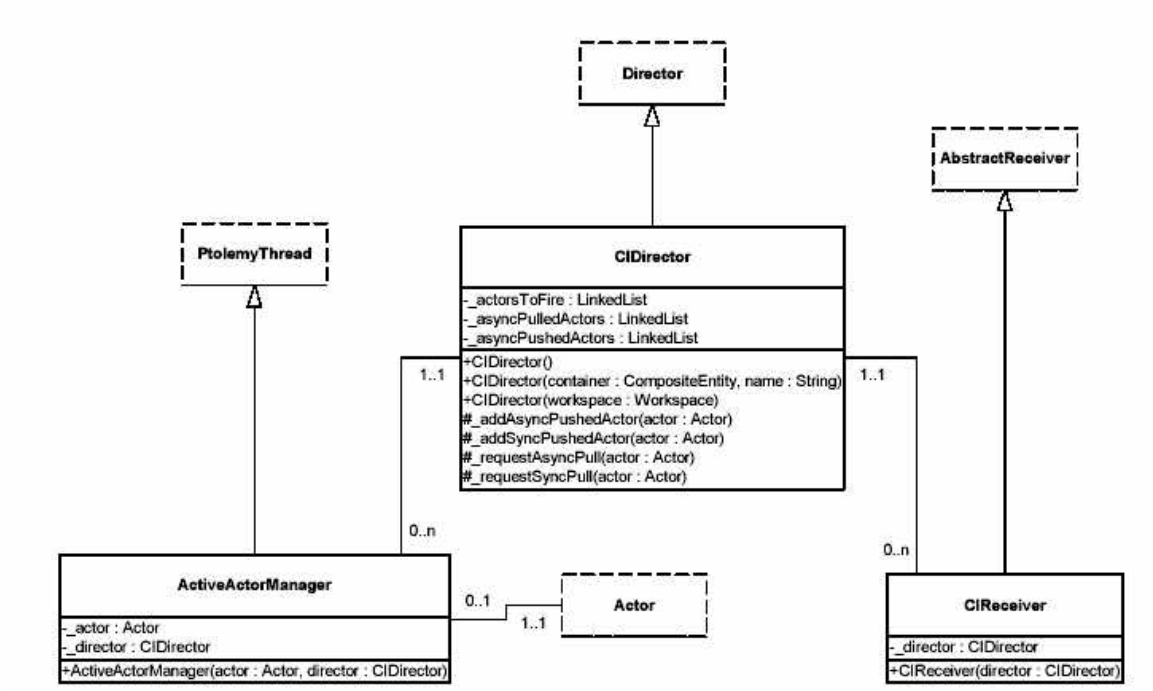


FIGURE 3.1 The class diagram of the CI domain kernel.

3.1 CIDirector

As mentioned above, there are passive and active actors in this domain. The *CIDirector* identifies whether there are active actors in its `initialize()` method. For each active actor, it creates a separate thread as an instance of *ActiveActorManager*. The execution of each active actor is managed by its own thread. The CIDirector is responsible for managing all the execution of passive actors. In the CIDirector, there is an `_actorsToFire` list, which saves actors that should be executed in one iteration. There is an `_asyncPushedActors` list, which saves actors that are triggered by some active actor while putting tokens in their receivers. Also there is an `_asyncPulledActors` list, which saves actors that are pulled by some active actor while issuing a pull request.

In each iteration, if the `_asyncPushedActors` list is not empty, The director will move an actor from the `_asyncPushedActors` list to the `_actorsToFire` list and fire the actor and other passive actors triggered by firing this actor. If the `_asyncPulledActors` list is not empty, then it moves an actor from the list and check whether it is ready to fire. If yes, it adds the actor to the `_actorsToFire` list and fire the actor and other passive actors triggered by firing this actor; otherwise it propagates the pull request to other passive actors by calling the `_requestSyncPull()` method: if some of those actors are ready to fire, the director adds them to the `_actorsToFire` list, or calls `_requestSyncPull()` recursively to propagate the request further. If all of these lists are empty, then the director waits.

At the very beginning, these three lists are all empty. The director will wait in its `fire()` method. At some point, some active actor pushes a token to its downstream actor, and this

in turn adds the downstream actor to the `_asyncPushedActors` list, and notifies the director; alternatively, some active actor pulls its upstream actor, adds that actor to the `_asyncPulledActors` list, and notifies the director. The director then wakes up and executes all the enabled actors.

3.2 CIReceiver

The *CIReceiver* class extends the AbstractReceiver class. Upon receiving a token, the receiver will check whether the port that contains it is push or pull, and whether the current thread that put the token to it is the director thread. (If the upstream actor is active, it has its own thread as an instance of ActiveActorManager). We refer to the port receiving the token as *portX*, and the actor containing portX as *actorX*. Possible cases are listed below:

- portX is a push input port. This requires that the upstream actor that sends token to it has a push output. The upstream actor can be active (with a pull input or no input) or passive (with push input) but actorX is always passive.

  If the upstream actor is active, then add actorX to the `_asyncPushedActors` queue in the CIDirector, and notify the director, which may be waiting for actors to fire.

  If the upstream actor is passive, then add actorX to the `_actorsToFire` queue in the CIDirector and don't notify.
- portX is a pull input port. This requires that the upstream actor that sends the token to it has a pull output. The upstream actor cannot be active, but actorX can be passive

(with a pull output) or active (with a push output or without an output). Also notice that actorX must have issued a pull request to its upstream actor before it can receive the token.

If actorX is passive, and its `prefire()` method returns true, then ask the director to fire it by adding it to the `_actorsToFire` queue.

If actorX is passive, and its `prefire()` method returns false, which means it doesn't have all the data required, then ask the director to pull its upstream actor.

If actorX is active, do nothing to the director, but notify the thread managing it, which may be waiting for the data.

3.3 ActiveActorManager

The *ActiveActorManager* class extends the PtolemyThread class. It iterates the active actor through the execution cycle until stop is requested. If the active actor is ready to fire, then fire it. Otherwise, if the active actor has pull input ports, then try to pull data from the upstream actors by calling the `_requestASyncPull()` method of the CIDirector, and wait for the data; if the active actor is a source actor (like the DatagramReader), then keep on polling.

3.4 Domain Specific Actors

Queue actors are domain specific since they require the knowledge of whether there is demand on their output. The *CIActor* class provides the required information. The domain specific actors are implemented in package `ptolemy.domains.ci.lib` by subclass-

ing CIActor. Currently, three kinds of queue actors have been implemented: an basic queue actor the has an infinite FIFO queue, a FrontDropQueue and a BackDropQueue.

3.5 Example

This section discusses a model built in the CI domain to study storage management in distributed systems.

The Distributor actor as shown in figure 3.2 distributes its input data to different outputs. Its first and third output ports is connected to two queue actors. The lengthes of the two

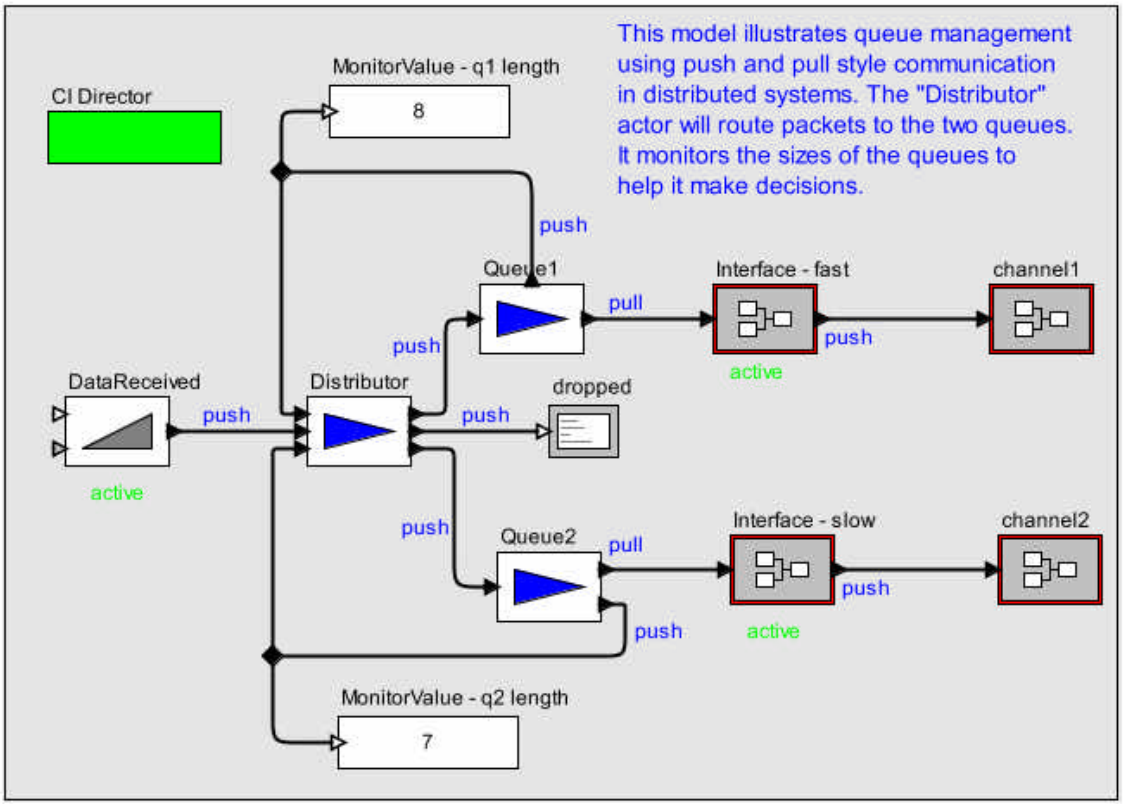


FIGURE 3.2 Distributed storage management demo

queue are fed back to the first and third input ports of the Distributor. The Distributor has two parameters: *minMark* and *maxMark*, that specify two thresholds for determining whether to drop a package. The input token at the second input port is distributed to its output ports according to the following policy:

- If the total length of Queue1 and Queue2 is less than the threshold1 specified by the minMark parameter, the input token is sent to the queue with the shorter length.
- If the total length of Queue1 and Queue2 is greater than threshold1 but less than threshold2 specified by the maxMark Parameter, the input token may be dropped randomly (with a probability proportional to the amount larger than threshold1) or send to the queue with shorter length.
- If the total length is greater than threshold2, then drop the input token.

If the input token is dropped, it is send to the second output so that it can be caught or monitored when necessary.

The Interface-Fast and Interface-Slow models two components that can transmit or process data with different speed. The numbers in figure 3.2 show a snapshot of the system where the data source has a production rate 5/*s*, the fast interface has a processing rate of 4/*s*, the slow interface has a production of 1/*s*, and the Distributor has minMark as 10 and maxMark as 20.

Chapter 4

Discussion

4.1 Relation to DDF and PN

An often asked question for the CI domain is what are the differences relative to Process Network (PN) models or Dynamic Data Flow (DDF) models? We discuss them here:

In a PN model, each component has its own thread, but in a dataflow model (DDF, BDF, SDF, etc.) a single thread is used to manage the execution of the entire model. CI only allows some components (active actors) to possesses a separate thread, and other components share a thread of execution.

CI allows the designer to specify which components use demand driven interaction and which components use data driven interaction. This can be viewed as some side control information provided to the model and can be used by the scheduler to decide the ordering of actors execution. Of course, the concepts of data driven and demand driven are not new, and algorithms using both data-driven and demand driven have been proved to be more promising than those only using data driven or demand driven [9]. But these algorithms require the scheduler to analyze the model to figure out whether an actor's firing will produce necessary data or not. In order to do so, the scheduler needs to have some informa-

tion about the model's state. By providing the control information in CI, the scheduler needs no such information. From this perspective, we argue that a CI model is easier to distribute over the network. A distributed example is discussed later.

Besides the above differences, the connections between them are also worth of exploring:

A CI model can contain composite actors which uses other dataflow MoCs, such as DDF or SDF. Currently, A CI model can't be embedded in other dataflow models due to the difficulty of defining an iteration.

A CI model can be partitioned according to its active sources and sinks. Each partition cluster may use an appropriate dataflow scheduling algorithm to improve efficiency.

4.2 Distributed CI models

As we discussed in last section, a CI model is more feasible for distributed execution than other dataflow models. Figure 4.1 shows a distributed system that was developed for the Mobies Ethereal String project.

We consider a distributed signal processing system that contains a group of users, a task manager and a set of computing resources. It works as follows: a user may ask the task manager to analyze a specific signal that is detected at his/her place, the task manager checks whether there are free computing resources and sends a model that should be used for signal analysis to the computing resource. Each computing resource runs a platform that allows the model to be dynamically constructed. When the computing resource is

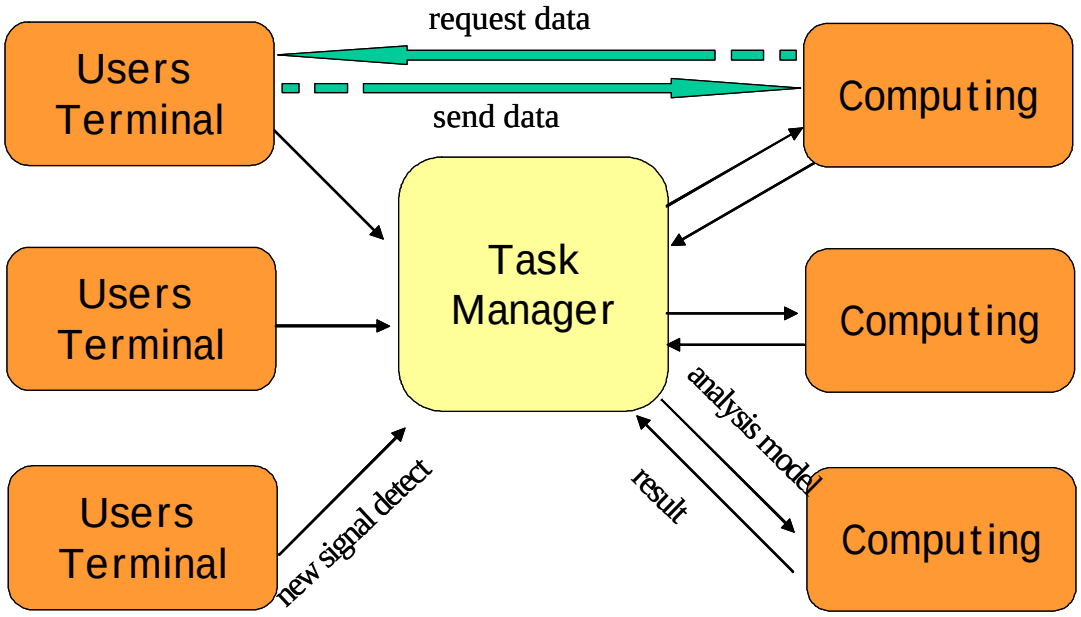


FIGURE 4.1 A distributed signal processing system

ready to receive data to analyze, it connects to the user model remotely and requests data to send. The bold arrows in the figure show the interactions for one user and one computing resource: the computing model requests data and the user sends data back upon request. After finishing executing the task, the computing model sends the result back to the task manager for further processing if neccessary.

Figure 4.2 shows the model that runs at the user terminal, the task manager and the computing resource after the model has been constructed. A set of CORBA actors have been developed for distributed message passing. The PushSupplier transfers received data to its connected PushConsumer. This pair of actors is domain polymorphic. The PullSupplier waits for a request from its connected PullConsumer and asks for data from the model it belongs to. So it is a CI specific actor. The PullConsumer is designed to be domain poly-

morphic also rather than specific to CI. It encapsulates the pull interaction inside itself, and works as common Ptolemy actors from the outside perspective. This allows the use of an SDF Director in the computing model for efficiency. The unpaired PushSupplier in the TaskManager model is used to interact with the dynamic constructing platform running on each computing resource, which doesn't show here.

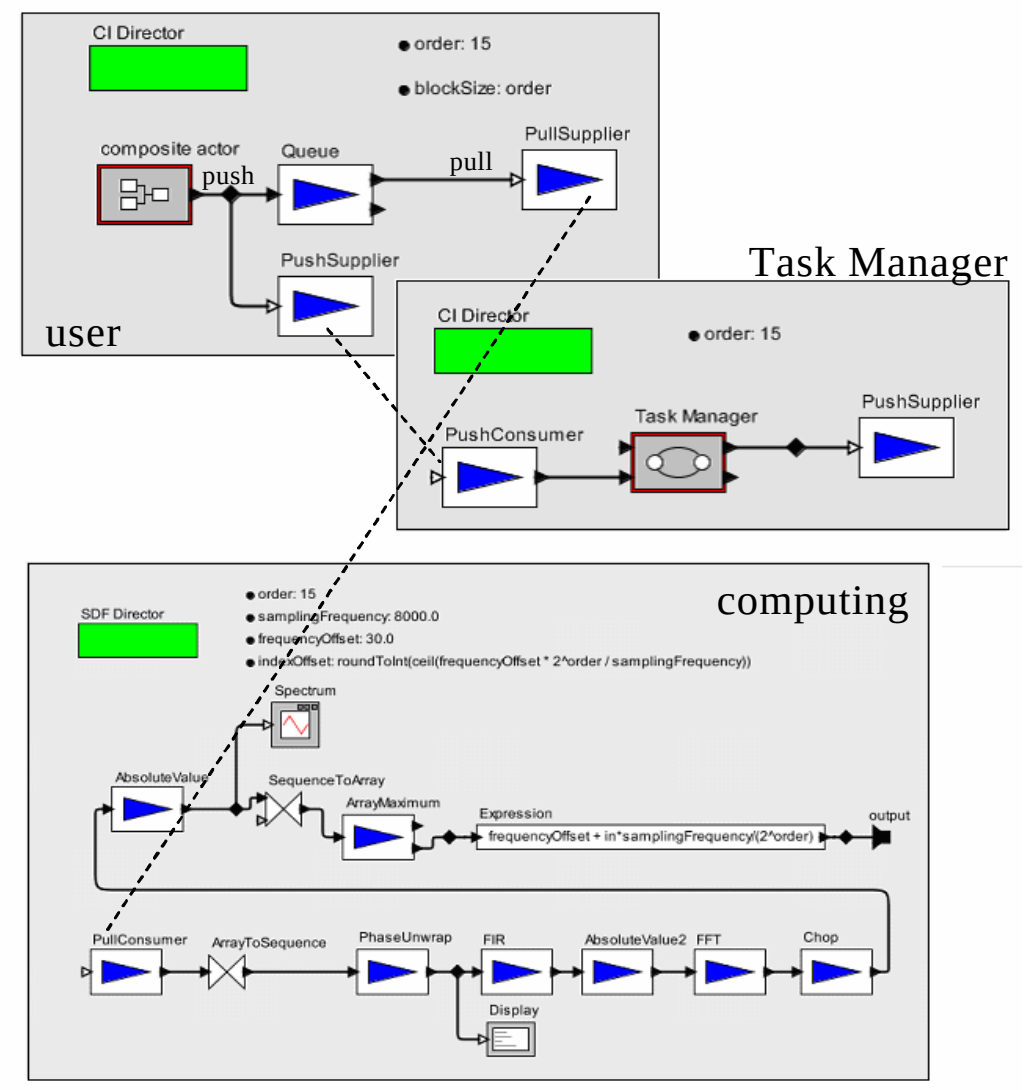


FIGURE 4.2 A distributed signal processing system

The whole model works as following:

- A user sends a signal detection message to the TaskManager via the PushSupplier and PushConsumer pair.
- The TaskManager chooses one free computing resource to analyze the signal, and sends an algorithm to the dynamic loading platform on the computing resource via another PushSupplier and PushConsumer pair.
- The algorithm is dynamically set up and uses the PushConsumer to contact the User's PullSupplier and asks for the signal data.
- The PullSupplier propagates the request to get the required data and sends them back to the PullConsumer.
- The signal is then analyzed on the computing resource.

This example is used to explain how can we use CI for distributed models with demand driven interaction over the network. Details about our work for the Ethereal String project can be found in the Ptolemy II package `ptII.ptolemy.apps.etherealSting`.

Future Work

This report describes a model of computation with both push and pull processing and discusses its preliminary implementation in Ptolemy II. The current implementation doesn't support an actor with mixed push - pull input ports, i.e. some of which are push and some of which are pull. It does allow some domain-specific actors to have mixed push-pull output ports, like the Queue actor used in Figure 3.2. We would like to study the demand for actors with mixed push-pull input ports or output ports and the execution semantics, such as whether such an actor should be active or passive. It would be interesting to use CAL [13] (an actor design language developed by Joern Janneck, et al. in the Ptolemy group) for these domain specific actors. The action concept in CAL may also be helpful for specifying and understanding the behavior of a actor with mixed input and output port types.

We also want to consider performance optimization for a push - pull model. A push - pull model can be partitioned according to the active threads, and it is possible to use different scheduling strategies that are appropriate for each cluster to improve efficiency.

Further exploration of the relation between this model and other dataflow models is also interesting. As we discussed before, this model can benefit from SDF for efficient scheduling in some clusters. We would also like to study whether the side control information provided explicitly can help the scheduling of other dataflow models.

References

[1] Robert Orfali, Dan Harke, “Client/Server Programming with Java and CORBA,” John Wiley and Sons,1997.

[2] Dirk Slama, Jason Garbis, Perry Russell, “Enterprise CORBA,” Publisher: Prentice-Hall, Inc. 1999.

[3] Paul Stephens, “Implementation of the CORBA Event Service in Java,” Master’s thesis, University of Dublin, Trinity College, 1999.

[4] Eddie Kohler “The Click Modular Router,” Ph.D. Dissertation, Dept. of EECS, MIT, 2001.

[5] Elaine Cheong, Judy Liebman, Jie Liu, Feng Zhao, “TinyGALS: A Programming Model for Event-Driven Embedded Systems,” Proc. ACM Symp. Applied Computing, Melbourne, FL, March 2003.

[6] Raymond Wong and Cho-Li Wang. “Push-Pull Messaging: A High-Performance Communication Mechanism for Commodity SMP Clusters,” Proc. of International Conference on Parallel Processing, Fukushima, Japan, September 1999, pp. 12-19.

[7] Mujtaba Khambatti, Kyung Ryu and Partha Dasgupta, “Push-Pull Gossiping for Information Sharing in Peer-to-Peer Communities,” The 2003 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA), June 2003.

[8] Shuvra S. Bhattacharyya, Elaine Cheong, John Davis II, Mudit Goel, Christopher Hylands, Bart Kienhuis, Edward A. Lee, Jie Liu, Xiaojun Liu, Lukito Muliadi, Steve Neuendorffer, John Reekie, Neil Smyth, Jeff Tsay, Brian Vogel, Winthrop Williams, Yuhong Xiong, Yang Zhao, Haiyang Zheng, “Heterogeneous Concurrent Modeling and Design in Java,” Memorandum UCB/ERL M03/27, July 2003.

[9] J. T. Buck, “Scheduling Dynamic Dataflow Graphs with Bounded Memory Using the Token Flow Model,” Tech. Report UCB/ERL 93/69, Ph.D. Dissertation, Dept. of EECS, University of California, Berkeley, CA 94720, 1993.

[10] Edward A. Lee, Stephen Neuendorffer and Michael J. Wirthlin,"Actor-Oriented Design of Embedded Hardware and Software Systems," Invited paper, Journal of Circuits, Systems, and Computers, Vol. 12, No. 3 pp. 231-260, 2003

[11] C. Hewitt, “Viewing Control Structures as Patterns of Passing Messages”, J. Artfical Intelligence 8, 3 (1977) 323 ~ 326.

[12] G. Agha, “Concurrent Object-Oriented Programming,” Commun. ACM 33, 9 (1990) 125 ~ 140.

[13] Johan Eker and Joern W. Janneck, “CAL Language Report,” Ptolemy Technical Memorandum, May 2002.