Abstracting Interactions Based on Message Sets

Svend Frølund[1] and Gul Agha[2*]

[1] Hewlett-Packard Laboratories, 1501 Page Mill Road, Palo Alto, CA 94303
[2] University of Illinois, 1304 W. Springfield Avenue, Urbana, IL 61801

Abstract. An important requirement of programming languages for distributed systems is to provide abstractions for coordination. A common type of coordination requires reactivity in response to arbitrary communication patterns. We have developed a communication model in which concurrent objects can be activated by sets of messages. Specifically, our model allows direct and abstract expression of common interaction patterns found in concurrent systems. For example, the model captures multiple clients that collectively invoke shared servers as a single activation. Furthermore, it supports definition of individual clients that concurrently invoke multiple servers and wait for subsets of the returned reply messages. Message sets are dynamically defined using conjunctive and disjunctive combinators that may depend on the patterns of messages. The model subsumes existing models for multi-RPC and multi-party synchronization within a single, uniform activation framework.

1 Introduction

Distributed objects are often *reactive*, i.e. they carry out their actions in response to received response. Traditional object-oriented languages require one to one correspondence between response and a receive message: i.e. each response is caused by exactly one message. However, many coordination schemes involve object behaviors whose logical cause is a set of messages rather than a single message. For example, consider a transaction manager in a distributed database system. In order to commit a distributed transaction, the manager must coordinate the action taken at each site involved in the transaction. A two-phase commit protocol is a possible implementation of this coordination pattern. In carrying out a two-phase commit protocol, the manager first sends out a status inquiry to all the sites involved. In response to a status inquiry, each site sends a positive reply if it can commit the transaction; a site sends back a negative reply if it cannot commit the transaction. After sending out inquiries, the manager becomes a reactive object waiting for sites to reply. The logical structure of the manager is to react to a set of replies rather than a single reply: if a positive reply is received from *all* sites, the manager decides to commit the transaction; if a negative reply is received from *any* site, the manager must abort the transaction.

In traditional object-oriented languages, the programmer must implement a response to a set of messages in terms of sequences of responses to single messages.

[*] The reported work was carried out while the first author was affiliated with the University of Illinois. The current email addresses are `frolund@hpl.hp.com` and `agha@cs.uiuc.edu`

Such an implementation complicates the construction of distributed systems: in order to defer a response until a number of messages have been received, objects must maintain a number of temporary variables which reflect the contents and structure of messages received thus far. Hence, in the implementation of objects, programmers are forced to inter-mix two orthogonal design concerns: when to react and how to react. Separating these design concerns will enhance the modularity of programs and make it easier to reason about logically distinct design issues in isolation.

In order to support activation by sets of messages, we propose an object-oriented communication model based on *activators*. An activator is a command that waits for certain sets of messages before triggering its continuation. The sets are described using arbitrary patterns of disjunctive and conjunctive combinators that depend on message contents.

Activators wait for messages sent to special destinations called *receptionists*. A receptionist is a first class entity that can be created dynamically, communicated in messages, and stored in data structures. Receptionists provide a uniform activation model since they can be endpoints for replies as well as input messages.

Sending a message to a receptionist is an asynchronous operation: a receptionist has a buffer in which messages are stored. Triggering an activator causes the triggering messages to be removed from the buffers in which they are stored. In this way, a given message can only trigger a single activator. A receptionist can be shared between multiple objects, enabling multiple activators to wait on messages for the same receptionist. Consistency of the triggering scheme is ensured through atomic removal operations.

The remainder of this paper is organized as follows. In Section 2, we introduce a simple base language, and in Section 3 we extend it with activators and receptionists. The resulting language provides a concrete setting and allows us to give a number of examples in Section 4 and Section 5. Section 6 discusses of related work. In Section 7, we give our concluding remarks.

2 Base Language

Our aim is to provide general insights and not tie activators and receptionists to any specific language. However, we do need an example language in order to illustrate the expressive power of the suggested constructs. We integrate support for activators and receptionists in a simple "toy" concurrent object-oriented base language invented for the purpose.

Our constructs require few assumptions about the "host" language in which they are integrated: we believe that activators and receptionists can elegantly and efficiently supplement the interaction model in most existing object-oriented languages. The design of our base language is primarily dictated by pedagogical concerns. We have chosen an Algol-like syntax for commands and declarations. Furthermore, we have ignored many of the aspects that are normally considered essential to object-orientation such as inheritance, polymorphism, dynamic binding, dynamic object creation, etc.

The computational foundation of our work is the Actor model [Agh86]. Consequently, our base language provides asynchronous message passing as the only mechanism for objects to interact. For our purposes, an object is an actor: the message-passing interface of an object consists of a set of methods that are executed in response to the reception of messages. It is immaterial whether or not objects are internally concurrent. However, for simplicity we assume that there is only one thread per object. Concurrent execution is obtained through asynchronous message-passing where the sender and receiver may proceed concurrently.

An object may have a local state that can be manipulated by the methods of that object. Local state is described as a set of instance variables that can be mutated using assignment. Objects are instantiated from classes as part of declarations. Methods can declare local variables.

```
class account
  var balance : real := 0;

  method deposit(amount : real)
    balance := balance + amount;
  end deposit

  method withdraw(amount : real)
    if ((balance - amount) >= 0) then balance := balance - amount;
  end withdraw
end account
```

Fig. 1. The structure of a simple bank account class.

The `account` class in Figure 1 introduces our syntax for class definitions. The `account` class can be used to instantiate bank account objects. Bank account objects can be manipulated through two exported methods: `deposit` and `withdraw`. The declaration part of `account` introduces an instance variable called `balance`. The name `balance` is bound to a `real` object that is initialized with the value `0`.

Message-passing is described using traditional dot (".") notation:

```
myAccount.deposit(300)
```

The above command will asynchronously invoke the `deposit` method on the object bound to the name `myAccount`. It is important to note that message-passing described using the "." operator is one-way and asynchronous which means that message-passing commands are non-blocking. Moreover, objects are autonomous – the model does not support class variables.

3 Activators and Receptionists

In this section we integrate support for activators and receptionists in our base language. Receptionists are created as part of declarations in a manner similar to ordinary objects. The following declaration binds the name `int-rec` to a receptionist that can hold integer-valued messages:

```
var int-rec : recep( integer)
```

Like ordinary objects, receptionists can be declared as instance variables or be local to methods. Receptionists which are declared as part of a method are created dynamically when that method is invoked. Receptionists are first class entities that can be communicated in messages.

Depositing a message at a receptionist is accomplished using syntax that is similar to the method invocation syntax introduced in Section 2. Assuming the above declaration of `int-rec`, the following command deposits a message with contents 37 at the receptionist bound to the name `int-rec`.

```
int-rec(37)
```

Deposit operations are asynchronous and non-blocking. Each receptionist has a queue in which deposited messages are stored.

```
activator         ::= receptionist ? name
                    | activator where expression
                    | activator_1 or ... or activator_n
                    | activator_1 and ... and activator_n

activated-command ::= activator → command
                    | [ activated-command □ ... □ activated-command ]
```

Fig. 2. Abstract syntax for activators.

An activator specifies a set of messages to be retrieved from a number of receptionists. Furthermore, an activator has an associated command which is *triggered* when the specified set of messages is retrieved. An activator along with its associated command is itself a command called and *activated command* in our toy language. An abstract syntax for activators and activated commands is given in Figure 2. The syntax for activators and activated commands is inspired by input guard in CSP [Hoa78] and Dijkstra's guarded commands [Dij75].

An activator waits for a set of messages to be available at a number of receptionists. When the needed messages are available, the activator is said to be enabled. The following rules define the sets of messages for which activators wait:

- $\boxed{\textit{receptionist ? name}}$ waits for a singleton set containing a message that is available from *receptionist*. The name *name* is bound to the contents of an enabling message.
- $\boxed{\textit{activator}\ \textbf{where}\ \textit{expression}}$ waits for a set of messages that enables *activator*. The message set must also satisfy the **where** clause: *expression* must evaluate to **true** in a context that includes the names bound by ? in *activator*. As syntactic sugar, we use *receptionist ? value* as a shorthand for the activator

receptionist ? x **where** x = *value*. This shorthand comes in handy when the contents of messages is compared to simple values such as integer and boolean.

- $\boxed{activator_1 \textbf{ or}\ldots\textbf{or } activator_n}$ describes a non-deterministic choice: the activator waits for any set of messages that enables at least one of $activator_1, \ldots, activator_n$.
- $\boxed{activator_1 \textbf{ and}\ldots\textbf{and } activator_n}$ waits for a set of messages that can be partitioned into n disjoint subsets where each subset must enable at least one of the activators and all of the activators must be enabled by at least one subset. The **and** combinator captures activation by multiple messages.

An enabled activator may trigger its associated command, thereby removing the enabling set of messages from their respective receptionists. Because messages are removed, the same message can only trigger an activator once.[3] A triggered command is executed in a context that includes the names bound by the ? operator. In this way, the triggered command can depend on the values contained in messages triggering the command.

The [...] construct represents a disjunctive composition of activated commands. The semantics of executing a [...] construct is to wait until at least one of the constituent activators is enabled. Then, one of the enabled activators is chosen for triggering, and the associated command is executed. The choice of activator is non-deterministic. Notice that both □ and **or** provide non-deterministic choice. The □ operator applies to activated commands and the **or** operator applies to activators.

The act of triggering an activator may involve a number of choices to be made by the underlying implementation. For example, two messages may each enable the same activator and a choice must be made between the two messages. The notion of *fairness* arises in connection with these choices. In the following, we consider three important kinds of choices and discuss the associated notion of fairness.

The scenario in Figure 3 illustrates a situation in which two messages at a receptionist each enable the same activator. A choice must be made between the two messages. The message which is ignored, i.e. chosen, may be subject to a similar choice in the future. With this potential for repeated choice, it is important to ensure that the same message is not ignored forever. We can implement this notion of fairness by considering the order in which messages arrive at a receptionist. If a choice must be made between two or more messages, we can simply choose the "oldest" message. In this way, messages cannot be ignored forever: a message which is repeatedly ignored will eventually become the oldest. However, note that the pattern of the messages that is required to enable a particular activator which would consume a given message may never occur.

The situation in Figure 4 illustrates two activators in different objects which are both enabled by the same message. In this situation only *one* of the activators can be triggered by the message, and a choice must be made between the two activators. Following the same principle as above, the ignored activator may be subject to a

[3] There is nothing in our framework that prevents the introduction of an operation similar to **read** in Linda [CGL86] that inquires about the availability of messages without removing them.

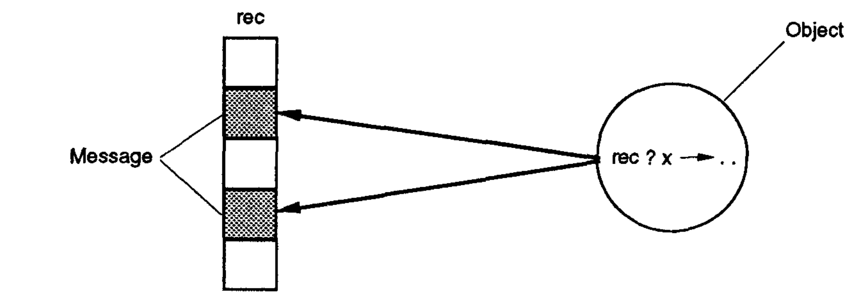

Fig. 3. Choice between two messages.

similar choice in the future, and there is a potential for repeatedly ignoring the same activator. If an activator is ignored forever, the enclosing object will remain blocked. Hence, the choice between activators should be fair and it should not be possible for the implementation to ignore an activator forever. The are many possible ways of implementing this notion of fairness. In the following we briefly outline one possible implementation.

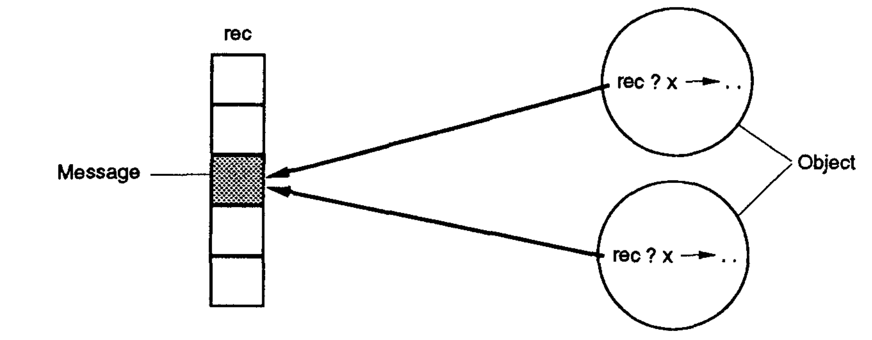

Fig. 4. Choice between two activators.

Fairness in the choice between activators in different objects can be implemented using time-stamps. When an object starts to execute an activator, a time-stamp is generated to uniquely identify the logical time at which the execution was initiated. When choosing between two activators, the activator with the "oldest" time-stamp is chosen. As above, the linear order prevents starvation. Unique time-stamps can be created on a per-object basis with little overhead. Each object maintains a counter which is incremented each time an activator in that object is executed. Time-stamps are then constructed by pairing an object's identity with the value of this local counter. The notion of a time-stamp is a standard concept in distributed systems. For example, time-stamps are used for deadlock prevention in [DJRI78].

The third kind of choice is the non-deterministic choice described by $\Box$ and **or**. These operators describe a choice between activators and activated commands in the same object. We do not believe fairness should be provided for in this scenario. First, it is possible for programmers to describe activators that never give rise to this choice. For example, programmers can use boolean conditions to ensure that two activators composed by **or** are never enabled at the same time. Second, it would be expensive for the implementation to provide fairness in general for this kind of choice: it would be necessary to maintain representation of the choices made in the past.

Having introduced activators and receptionists, the following two sections give a number of examples illustrating the use of these constructs. In the examples, we distinguish between *input synchronization* and *reply synchronization*. Input synchronization is concerned with producer-consumer style interaction where a number of consumers receive input from a number of producers. Activators and receptionists can be used to synchronize the arrival of input from multiple sources at consumers. Reply synchronization is concerned with client-server style interaction where a client sends a request to a number of servers and waits for the replies from the servers. Activators and receptionists can be used to synchronize the arrival of replies from multiple servers. In Section 4, we describe examples of input synchronization and in Section 5, we outline a number of examples involving reply synchronization.

Both input and reply synchronization involve activation of objects by a set of messages where the elements of the set come from different sources. Unlike most traditional languages which address *either* input *or* reply synchronization, activators and receptionist provide a common representation for describing both kinds of coordination.

4 Input Synchronization

In this section we outline a number of scenarios where objects synchronize the arrival of input from multiple sources.

Example 1. Assume that two sensors are measuring the condition of a critical system. The observations made by the sensors must be processed by a monitor object in order for the operators of the critical system to determine its logical status. An example system could be a nuclear power plant where the sensors measure the temperature of the nuclear reaction at two points in the reactor. In order to determine whether or not the plant is in a safe state, some equation containing the measured temperatures must be solved.

The logical structure of the sketched computer system involves a two-party invocation where the two sensors collectively invoke the monitor object. Since data may be produced faster than it can be processed, it is desirable to have multiple monitor objects that can concurrently process different data sets. It is necessary to de-couple the sensors and the monitors so that the sensors do not invoke a specific monitor object.

The required interaction between sensors and monitors can be elegantly described using activators and receptionists. The sensors and monitors communicate via two shared receptionists. Each sensor has an associated receptionist to which it sends messages that represent observations. A monitor object has an activator that waits for two messages: one from each receptionist. Since multiple monitor objects share the receptionists, it is crucial that the removal of messages is an atomic action.

```
method survey(sens-1,sens-2 : recep( real);)
  var r1,r2 : real;
  while true
   sens-1 ? r1  and sens-2 ? r2  where non-safe(r1,r2)
     → process-data(r1,r2);
end survey
```

Fig. 5. Survey method that is executed by monitor objects.

The **survey** method in Figure 5 is a part of monitor objects. A monitor object is put into effect by invoking the **survey** method with two receptionists that receive observation messages sent by the two sensors. The activator in the **survey** method waits for one observation to be available from each sensor. For a pair of observations to trigger the activator, the observations must be outside a safety window determined by the function **non-safe**. For simplicity we have ignored the issue of guaranteeing that two messages denote observations made at the "same" time. That issue can be addressed by using some kind of tagging. □

The use of receptionists in Example 1 de-couples sensors and monitors. The de-coupling is scalable: more monitors can be added transparently to the configuration if the rate of observations increases. Because the sensors do not invoke any particular monitor, it is possible to dynamically vary the number of monitors without modifying the sensors.

Achieving the same degree of de-coupling without receptionists would require introducing an "administrator" object that collects data produced by sensors and distributes the data among the monitors. The centralization caused by using an administrator creates a potential bottleneck in the system. Moreover, it makes the system architecture less fault-tolerant: if the administrator fails, the system fails. Receptionists provide a decentralized and open system structure without bottlenecks. Using receptionists, the system configuration is inherently fault-tolerant: it is possible to duplicate the sensors and have the monitors retrieve messages in a non-deterministic fashion from multiple sets of receptionists.

Example 2. The following example is adopted from [Cha87]. A moving point in a three dimensional space is to be plotted on a graphics device. Each of the **x**, **y** and **z** coordinates are to be computed as a function of the previous coordinates in discrete time. The computation of coordinates are performed by three concurrent "worker" objects.

The three worker objects collectively invoke the graphics device with a set of coordinates to be displayed – each worker object provides a dimension of the displayed point. The graphics device communicates the previous set of coordinates to the workers, thereby initiating computation of the next set of coordinates.

```
class graphics-device
   var x-rec, y-rec, z-rec :  recep( real);
   method plot()
     var x,y,z :  real;
     x-rec ? x  and y-rec ? y  and z-rec ? z
       → display(x,y,z);
     ...
   end plot
  ...
end graphics-device
```

Fig. 6. A graphics device displaying a point whose coordinates are concurrently computed by worker processes.

Part of the graphics device is sketched in Figure 6. We have left out the description of the worker objects and the communication of the previous coordinates from the graphics device to the worker objects. The figure focuses on the synchronization of the "streams" of newly computed coordinates sent from the worker objects to the graphics device. Each stream is represented by a receptionist, and the synchronization is described by an activator which only triggers when an element can be extracted from all three streams. Such an element set denotes a point in three dimensional space.

In Figure 6, the receptionists are called `x-rec`, `y-rec` and `z-rec` respectively. A worker object knows one of these receptionists. For example, the worker object which computes coordinates on the x axis knows the `x-rec` receptionist. A worker object computes the next coordinate value and deposits that value in the receptionist that it knows about. The `plot` method of the graphics device has an activator which awaits the deposit of a message at each of the receptionists. The logical interaction pattern is that the three worker objects collectively and atomically invoke the graphics device when a new set of coordinates is ready. □

Example 2 illustrates that activators can be used as a flexible tool for describing barrier synchronization.

5 Reply Synchronization

In this section we model with client-server style interaction where client objects and server objects communicate using request-reply protocols. A client sends a request to a number of servers and waits for the servers to send back a reply in response to the request. We can use receptionists and activators to describe request-reply protocols where clients wait for certain *subsets* of the replies.

In describing request-reply protocols, receptionists are used as *reply destinations*: when sending a request, a client creates a receptionist to which the server should send the reply. The name of the receptionist is then passed to the server as part of the request. After sending the request, the client executes an activator which waits for the reply to be sent to the receptionist.

Although it is possible to describe request-reply protocols by explicitly communicating reply destinations, it is useful to provide syntactic sugar which makes reply destinations implicit rather than explicit arguments of a request. The following command introduces this kind of syntactic sugar:

```
o.m(...) @ rec;
```

This command sends a request to the object `o` for invocation of the method `m`. The receptionist `rec` is used as the reply destination for this request. The interpretation of the `@` operator is "deliver at." The syntax for specifying reply destinations is inspired by ABCL/1 [Yon90].

Since a reply destination is an implicit parameter of a request, we need a way for the receiver of the request to refer to this implicit argument. During method invocations, the name **reply** is bound to the receptionist which servers as reply destination for the invocation.[4] The receiver of a request can reply to the request using the already introduced syntax for receptionist manipulation.

With our approach, reply destinations are first class objects, and it is possible to describe *reply delegation* and *reply propagation* without introducing additional language constructs. With reply delegation, the receiver of a request delegates the responsibility of responding to the request to another object. Reply delegation can be accomplished by communicating the receptionist bound to the name **reply** to another object. With reply propagation, the sender of a request re-directs the reply to other objects. In our language, reply propagation is possible since a client can communicate the reply destination to other objects which can then wait for the reply.

The following examples illustrate situations which require pattern-based waiting on replies as well as the first class status of reply destinations.

Example 3. Inspired by Cooper [Coo90] we use a two-phase commit protocol as an example of pattern-based waiting on replies. A coordinator process initiates a two-phase commit with three participant objects `a`, `b` and `c`.

In Figure 7, the method `two-phase-commit` concurrently sends a request to `a`, `b` and `c`. The replies are directed to three receptionists `res-a`, `res-b` and `res-c`. If all replies are positive a `commit` message is broadcast. If one of the replies is negative, an `abort` message is broadcast. The pattern-based triggering implies that `abort` can be broadcast to the participants as soon as one negative reply has been received. □

Example 4. Consider a situation where some critical data is stored on a disk. Access to the data is only granted to clients who have certain capabilities, e.g. know a password. The interface to the data is a server that, given a password and an

[4] For simplicity, we do not consider the situation where a server sends a reply and no reply destination is specified by the client.

```
class coordinator
  var a, b, c : participant;
  method twoPhaseCommit()
    var res-a,res-b,res-c : recep( boolean);
    a.ready() @ res-a; b.ready() @ res-b; c.ready() @ res-c;
    [
    res-a ? true and res-b ? true and res-c ? true
      → a.commit(); b.commit(); c.commit();
    □
    res-a ? false or res-b ? false or res-c ? false
      → a.abort(); b.abort(); c.abort();
    ];
  end twoPhaseCommit
   ...
end coordinator
```

Fig. 7. Coordinator for two phase commit.

identification of some data, returns the data if the password is valid. In the following example we demonstrate how to realize this functionality by composing generic servers.

We want to employ a generic disk server and a generic authentication server. We do *not* want to modify an existing disk server to incorporate authentication. First of all, it is desirable to reuse existing components. Secondly, the disk server may already be in operation and it may not be possible (or feasible) to halt the system and change the functionality of the disk server. In order to enforce proper access discipline on clients, the system is structured as a hierarchy of servers. The highest level is an "interface" server that delegates requests to the disk server and authentication server. Requests are only delegated to the disk server if the client was correctly authenticated.

The code in Figure 8 sketches the structure of the interface server. Clients invoke the method called **get-data** in order to access the protected data. A naive way of designing the interface server would be to first invoke the authentication server and, if the authentication is successful, invoke the disk server. This naive approach introduces unnecessary sequentiality. Instead, our interface server concurrently invoke the disk server and the authentication server. Concurrent invocation is facilitated by our asynchronous reply-request protocol. The requests executed in the body of **get-data** are asynchronous: the ; combinator of commands does not block the requests.

We do not want the interface object to wait for the replies from the two servers: this would block the interface object while waiting for the replies and limit the concurrency in the system. Instead, we introduce a continuation object which waits for the replies. The servers send their replies to two receptionists called **data** and **check**. These receptionists are communicated to the continuation object so that it can wait for the replies. Notice that the propagation of replies from the interface to the continuation is transparent to the servers. The resulting system structure is

```
class interface
  method get-data(passwd,name-of-data : string)
    var check : recep(boolean); data : recep(data-type);
        cont : continuation;
    cont.action-part(check,data) @ reply;
    authentication-server.authenticate(passwd) @ check;
    disk-server.read(name-of-data) @ data;
  end get-data
end interface

class continuation
  method action-part(check : recep(boolean); data : recep(data-type) )
    var x : data-type;
    [
      check ? true and data ? x → reply(x)
    □
      check ? false → reply( NIL)
    ]
  end action-part
end continuation
```

Fig. 8. Interface server that propagates replies from authentication server and disk server.

sketched in Figure 9. In the figure, requests are represented by solid lines and replies are represented by dashed lines.

Using activators, it is possible to wait for patterns of replies to be received: the continuation either waits for a negative reply from the authentication server or a positive reply combined with the returned data from the disk server. The thing to note is that the continuation does not wait for the disk if the access is illegal: the user is notified right away.

The example illustrates reply delegation: the interface delegates the reply responsibility to the continuation. This reply delegation is accomplished by the command

```
cont.action-part(check,data) @ reply;
```

The name **reply** refers to the reply destination of the interface. Because this reply destination is a receptionist, it can be used to specify the destination of another request giving rise to reply delegation. □

The structure in Example 4 illustrates how our constructs support interaction patterns in hierarchical systems while avoiding unnecessary bottlenecks. The example also demonstrates that flexible and uniform handling of reply messages promotes reuse since it is easier to inter-connect existing components in a transparent fashion.

6 Related Work

In general, existing languages address input synchronization and reply synchronization in fundamentally different ways. By contrast, our activation model supports

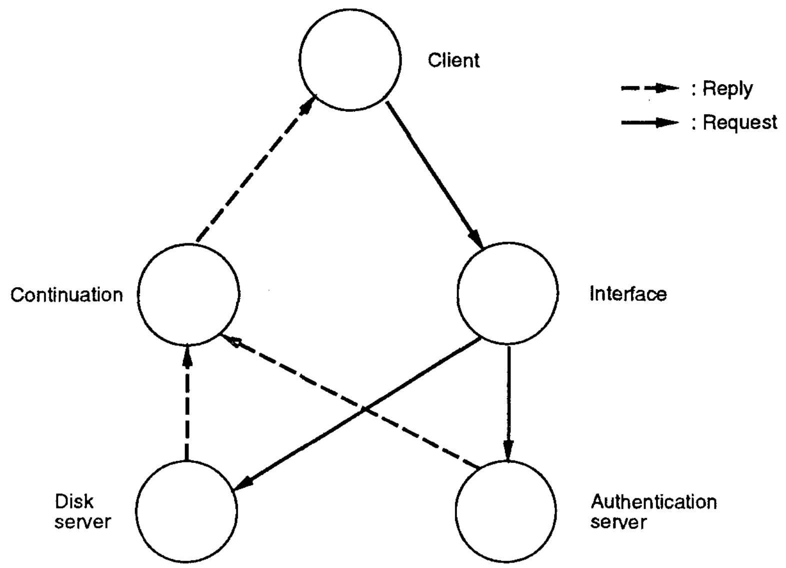

Fig. 9. Topology of the request-reply communication between the interface, continuation, authentication, and disk servers.

both kinds of activation using the same abstractions namely, activators and receptionists. The resulting model is conceptually simpler and more flexible than existing models. Our model allows inter-mixing of reply messages and input messages. It is, for example, possible to describe an activator that waits for input messages as well as reply messages.

Activators generalize the notion of input guard found in CSP [Hoa78]. Input guards cannot capture conjunctive waiting on multiple messages. Furthermore, it is not possible in CSP to describe activation conditions that depend on patterns over the values conveyed in messages.

Receptionists subsume ports [ABG$^+$92], mailboxes [AB83] and channels [Rep92, MSK86]. Receptionists are similar to mailboxes in that multiple objects can receive messages from a shared receptionist. Receptionists are more general since they can contain input messages as well as reply messages. Furthermore, activators are more powerful than the corresponding operations available for ports, mailboxes and channels.

The Tuple Space abstraction in Linda [CGL86] is in many ways similar to receptionists. Although retrieval in Linda is pattern-based, it is not possible to capture the functionality of activators since retrieval operations cannot operate on multiple tuples. In [MK88], Matsuoka developed tuple space as a communication facility for object-oriented languages. The proposed tuple space abstraction supports conjunctive and disjunctive composition of tuple operations. However, the communication model does not support set-based activation by reply messages.

One way to make existing communication facilities more powerful is customization through reflection. In [Jag91], Jagannathan introduces customizable tuple spaces. However, the customization policies do not support the definition of operations that remove multiple tuples. In some object-oriented reflective architectures it is possible to customize the behavior of message-reception [WY88, AFPS93]. This customization does not provide a general framework for describing reception of multiple messages. For example, it is not possible to describe generic meta-level abstractions to support composable abstractions that operate on sets of messages.

Guarded Horn clause languages (gHcl's) have an execution semantics that is analogous to that of actors [HA88]. Moreover, gHcl's support a form of input synchronization (for example, see Parlog [Gre87]). In such languages, patterns may be specified in guards which test conditions on asynchronous inputs; if a guard is satisfied, the relevant messages are removed atomically. However, gHcl's allow only conjunctive combinators for such patterns; thus their guards are more restrictive than in our model. Essentially, to mimic the behavior of a disjunctive pattern, different rules have to be used and the specification of any actions that are common between them has to be repeated. Moreover, reply synchronization is not directly supported. Finally, gHcl's have extraneous sources of complexity in them, e.g., they use unification, and they do not provide the modeling support afforded by an object centered view.

The concurrent constraint language Janus [KS90, SKL80] allows the description of agents that can wait conjunctively and disjunctively on input from bags of messages. The notion of a bag bears some resemblance to our notion of receptionist. However, in Janus it is not possible to describe retrieval of messages based on their contents. The only way to simulate such pattern-based retrieval is to inspect the contents of a message *after* it has been retrieved. With simulation, failure to match a pattern leads to explicit put-back of messages which can result in starvation. Another shortcoming of Janus is the inability to describe atomicity of conjunctive wait operations. With conjunctive waiting, an agent is blocked until it has retrieved a message from a number of bags. In Janus, there is no elegant way to ensure that none of the messages are retrieved until all messages can be retrieved. The lack of atomic retrieval can lead to deadlocks.

Andreoli and Pareschi proposed the language LO to extend the AND-concurrency of gHcl's with OR-concurrency which can be used to model the internal distribution of tasks in composite objects [AP90]. Specifically, LO allows an object to be modeled as a composition of sub-objects which may be independently accessed: a disjunctive pattern may be represented by propagation and global matching inference rules. Although it is quite similar, the model does not appear to be as efficient as that of activators.

The composition filter model provides a generic mechanism for abstracting object interactions [AWB$^+$93]. The model may be used to define objects which model complex interactions such as those modeled by activators. However, one consequence is that this fixes the implementation by directly programming the interaction pattern whereas activators provide an abstract specification.

Activation by sets of input messages is supported by the numerous constructs that have been proposed for multi-party synchronization [ESFG88, FHT86, EFK89, JS91, AFL90, BKS88]. Multi-party synchronization involves a number of processes that synchronize by executing joint actions. In multi-party synchronization languages, processes refer directly to joint actions by name. In our model, we can describe joint actions as commands which are triggered by activators. Process objects can then synchronize by collectively providing the messages that trigger the joint action. Receptionists provide a level of indirection, and process objects do not refer directly to the joint actions in which they participate. This gives a more modular, flexible, and extensible structure because it is not necessary to modify existing process objects in order to synchronize them by new joint actions. Another difference is that our scheme supports collective activation without insisting on synchronizing the invoking objects.

Multi-functions [BBP86] extend the traditional method concept so that a message can supply *part* of the actual values required to invoke a method. A caller specifies one or more formal parameters by keyword and supplies actual values for the specified parameters. When all formal parameters are bound to actual values, a method execution starts. The parameter list of a multi-function describes a message set which can activate the multi-function. However, since *all* the parameters must be bound, message sets can only be described using conjunction. Furthermore, message sets cannot be described in terms of the actual values conveyed in messages.

A common notion of request-reply protocol is remote procedure call (RPC). Traditional RPC communication blocks the initiator until a reply is returned. Blocking the sender implies that RPC requests cannot be issued concurrently. In contrast, our scheme allows requests to be sent out asynchronously, and activators allow us to describe selective waiting on subsets of replies.

Futures [Hal90, YT86, YBS86], Multi-RPC [LS88, Coo90] and Join-Continuations [Agh90] all deal with reply synchronization. None of those constructs allow direct expression of reply synchronization that is based on the values conveyed in reply messages. In order to express such data-dependencies, the programmer must explicitly manipulate temporary variables that hold the values of previously returned replies.

7 Concluding Remarks

We have proposed activators and receptionists as constructs for describing activation by message sets in concurrent object-oriented languages. Activators provide a powerful synchronization and communication construct which operates on sets rather than single messages. A significant aspect of activators is that sets of messages are defined in a pattern-based manner that takes message contents into account. Receptionists give a unified approach to input messages and reply messages. In terms of reply handling, receptionists are first-class reply destinations supporting reply delegation and reply propagation.

Our analysis suggests that our model can be implemented efficiently. The "expensive" part of our constructs is the atomicity associated with removal of messages from receptionists. However, because our constructs abstract over everyday practice in the construction of concurrent systems, the involved consensus protocols will have the same complexity whether hand-coded explicitly or provided as part of a language implementation.

A topic for future work is further integration of receptionists and methods.

Both are communication endpoints. It is desirable to make it transparent to clients whether they are invoking a method or sending a message to a receptionist.

Acknowledgments

The first author was sponsored by a research fellowship from the Natural Science Faculty of Århus University in Denmark and generous support from the Danish Research Academy.

The research described in this paper was carried out at the University of Illinois Open Systems Laboratory (OSL). The work at OSL has been supported by the Office of Naval Research (ONR contract number N00014-90-J-1899 and N00014-93-1-0273), Digital Equipment Corporation, NEC, Hitachi, and by joint support from the Advanced Research Projects Agency and the National Science Foundation (NSF CCR 90-07195).

The authors wish to thank Carolyn Talcott, Nayeem Islam, Christian Callsen, Shingo Fukui, Chris Houck, Daniel Sturman, Mark Astley and other current and former members of OSL for numerous discussions as well as invaluable feedback on manuscript versions of this paper.

References

[AB83] S. Abramsky and R. Bornat. Pascal-M: A Language for Loosely Coupled Distributed Systems. In Y. Parker and J.-P. Verjus, editors, *Distributed Computing Systems*, pages 163 - 189. Academic Press, 1983.

[ABG+92] J. S. Auerbach, D. F. Bacon, A. P. Goldberg, G. S. Goldszmidt, A. S. Gopal, M. T. Kennedy, A. R. Lowry, J. R. Russell, W. Silverman, R. E. Strom, D. M. Yellin, and S. A. Yemini. High-Level Language Support for Programming Distributed Systems. In *In Proceedings of the 1992 International Conference on Computer Languages.* IEEE, April 1992.

[AFL90] P. C. Attie, I. R. Forman, and E. Levy. On Fairness as an Abstraction for the Design of Distributed Systems. In *Tenth International Conference on Distributed Computing Systems.* IEEE, 1990.

[AFPS93] G. Agha, S. Frølund, R. Panwar, and D. Sturman. A Linguistic Framework for Dynamic Composition of Dependability Protocols. In *Dependable Computing for Critical Applications III.* International Federation of Information Processing Societies (IFIP), Elsevier Scienc Publisher, 1993.

[Agh86] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, 1986.

[Agh90] G. Agha. Concurrent Object-Oriented Programming. *Communications of the ACM*, 33(9):125–141, September 1990.

[AP90] Jean-Marc Andreoli and Remo Pareschi. LO and Behold! Concurrent Structured Processes. In *Proceedings OOPSLA/ECOOP '90*, pages 44–56, October 1990. Published as ACM SIGPLAN Notices, volume 25, number 10.

[AWB+93] M. Aksit, K. Wakita, J. Bosch, L. Bergmans, and A. Yonezawa. Abstracting Object Interactions Using Composition Filters. In *ECOOP 1993, Lecture Notes in Computer Science.* Springer Verlag, 1993. LNCS 791.

[BBP86] J. P. Banatre, M. Banatre, and F. Ployette. The Concept of Multi-Function: A General Structuring Tool for Distributed Operating Systems. In *Sixth International Conference on Distributed Computing Systems.* IEEE, 1986.

[BKS88] R. J. R. Back and R. Kurki-Suonio. Distributed Cooperation with Action Systems. *ACM Transactions on Programming Languages and Systems*, 10(4), 1988.

[CGL86] N. Carriero, D. Gelernter, and J. Leichter. Distributed Data Structures in Linda. In *POPL '86 Proceedings*. ACM, 1986.

[Cha87] A. Charlesworth. The Multiway Rendezvous. *ACM Transactions on Programming Languages and Systems*, 9(2), July 1987.

[Coo90] E. Cooper. Programming Language Support for Multicast Communication in Distributed Systems. In *Proceedings of the Tenth International Conference on Distributed Computing Systems*. IEEE, 1990.

[Dij75] E. W. Dijkstra. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *Communications of the ACM*, 18(8), August 1975.

[DJRI78] R. E. Stearns D. J. Rosenkrantz and P. M. Lewis II. System Level Concurrency Control for Distributed Database Systems. *ACM Transactions on Database Systems*, 3(2):178–198, June 1978.

[EFK89] M. Evangelist, N. Francez, and S. Katz. Multiparty Interactions for Interprocess Communication and Synchronization. *IEEE Transactions on Software Engineering*, 15(11), 1989.

[ESFG88] M. Evangelist, V. Y. Shen, I. R. Forman, and M. Graf. Using Raddle to Design Distributed Systems. In *Proceedings of the Tenth International Conference on Software Engineering, Singapore*. IEEE, 1988.

[FHT86] N. Francez, B. Hailpern, and G. Taubenfeld. Script: A Communication Abstraction Mechanism and its Verification. *Science of Computer Programming*, 6:35–88, 1986.

[Gre87] S. Gregory. *Parallel Logic Programming in PARLOG*. Addison-Wesley, first edition, 1987.

[HA88] C. Hewitt and G. Agha. Guarded Horn Clause Languages: Are they Deductive and Logical. In *Proceedings of Fifth Generation Computer Systems Conference*, Dec. 1988.

[Hal90] R. H. Halstead. New Ideas in Parallel Lisp: Language Design, Implementation, and Programming Tools. In *Parallel Lisp: Languages and Systems*. Springer-Verlag, 1990. LNCS 441.

[Hoa78] C. A. R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[Jag91] S. Jagannathan. Customization of First-Class Tuple-Spaces in a Higher-Order Language. In E. H. L. Aarts and J. van Leeuwen, editors, *Proceedings of PARLE '91 Parallel Architectures and Languages Europe*. Springer Verlag, June 1991. LNCS 506.

[JS91] Y. Joung and S. A. Smolka. Coordinating First-Order Multiparty Interactions. In *POPL '91 Proceedings*. ACM, 1991.

[KS90] Kenneth M. Kahn and Vijay A. Saraswat. Actors as a Special Case of Concurrent Constraint Programming. In *Proceedings OOPSLA/ECOOP '90*, pages 57–65, October 1990. Published as ACM SIGPLAN Notices, volume 25, number 10.

[LS88] B. Liskov and L. Shrira. Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, June 1988.

[MK88] Satoshi Matsuoka and Satoru Kawai. Using Tuple Space Communication in Distributed Object-Oriented Languages. In *Proceedings OOPSLA '88*, pages 276–284, November 1988. Published as ACM SIGPLAN Notices, volume 23, number 11.

[MSK86] D. May, R. Shepherd, and C. Keane. Communicating Process Architecture: Transputer and Occam. In P. Treleaven and M. Vanneschi, editors, *Future Parallel Architecture*, pages 35–81. Springer-Verlag, 1986. LNCS 272.

[Rep92] J. H. Reppy. *Higher-Order Concurrency*. PhD thesis, Cornell University, June 1992. Published as Technical Report 92-1285.

[SKL80] V. Saraswat, K. Kahn, and J. Levy. JANUS: A Step Towards Distributed Constraint Programming. Technical Report SSL-90-51, Xerox Palo Alto Reseach Center (PARC), 1980.

[WY88] Takuo Watanabe and Akinori Yonezawa. Reflection in an Object-Oriented Concurrent Language. In *Proceedings OOPSLA '88*, pages 306–315, November 1988. Published as ACM SIGPLAN Notices, volume 23, number 11.

[YBS86] Akinori Yonezawa, Jean-Pierre Briot, and Etsuya Shibayama. Object-Oriented Concurrent Programming in ABCL/1. In *Proceedings OOPSLA '86*, pages 258–268, November 1986. Published as ACM SIGPLAN Notices, volume 21, number 11.

[Yon90] A. Yonezawa, editor. *ABCL An Object-Oriented Concurrent System*. MIT Press, Cambridge, Mass., 1990.

[YT86] Yasuhiko Yokote and Mario Tokoro. The Design and Implementation of ConcurrentSmalltalk. In *Proceedings OOPSLA '86*, pages 331–340, November 1986. Published as ACM SIGPLAN Notices, volume 21, number 11.