Coordinating Multi-Transaction Activities

Hector Garcia-Molina

Dept. of Computer Science
Princeton University
Princeton, NJ 08544

Dieter Gawlick
Johannes Klein
Karl Kleissner

Digital Equipment Corporation
Mountain View, CA 94040

Kenneth Salem

Dept. of Computer Science
University of Maryland
College Park, MD 20742

ABSTRACT

Data processing applications must often execute collections of related transactions. We propose a model for structuring and coordinating these multi-transaction activities. The model includes mechanisms for communication between transactions, for compensating transactions after an activity has failed, for dynamic creation and binding of activities, and for checkpointing the progress of an activity.

THIS IS A REVISED VERSION OF TECHNICAL REPORT CS-TR-297-90, DEPARTMENT OF COMPUTER SCIENCE, PRINCETON UNIVERSITY, FEBRUARY 1990. A SHORTER VERSION OF THIS PAPER APPEARED AS "Modeling Long-Running Activities as Nested Sagas," IN *Database Engineering*, Vol. 14, No. 1, March, 1991.

Coordinating Multi-Transaction Activities

Hector Garcia-Molina

Dept. of Computer Science
Princeton University
Princeton, NJ 08544

Dieter Gawlick
Johannes Klein
Karl Kleissner

Digital Equipment Corporation
Mountain View, CA 94040

Kenneth Salem

Dept. of Computer Science
University of Maryland
College Park, MD 20742

1. Introduction

In this paper we study so called "data processing applications." These applications contain programs that perform computing functions necessary for running a business or enterprise. The functions may include, for instance, inventory control, accounting, market projections, and so on. The non-volatile data shared by the programs is typically stored in a *database*. The programs execute *transactions* that are the elementary or atomic units of work (e.g., record a new purchase order). However, there is more to an application than simply a set of independent transactions. There is also code that controls the calls to transactions, creates processes, and manages the overall operation.

To illustrate what we mean, consider the processing of a purchase order at some company. This *activity* can be represented as a collection of *steps*, as shown in Figure 1. This is a very simplified version of what may occur. The activity starts when a phone call is received to place a new order (step "phone call"). A clerk enters the required information into an electronic form. The next step is to run a transaction (step "enter order") to record the new order in the database. Note that the first step was not a transaction: there is no need to save the data permanently until the entire request has been made. After the

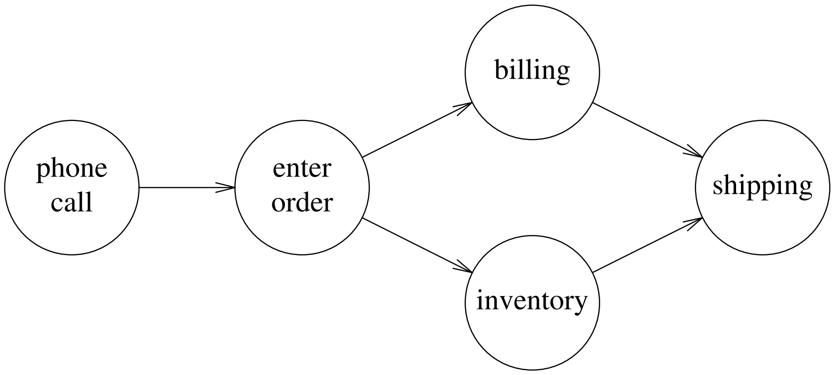

Figure 1: A Purchase Order Activity

order is entered, two parallel steps are executed. The "billing" step charges the customer and the "inventory" step updates the inventory. Each of these steps involves a transaction that reads and updates the database. After these transactions complete, a final "shipping" step is executed to generate the appropriate shipping orders.

Activities sometimes have to be aborted. For example, if the "billing" step discovers that the customer's credit is not good, then we may want to back out the entire activity. The simplest way to achieve this is to use *nested transactions* [Moss85]. That is, the activity can be encapsulated within a higher level transaction. When a step like "enter order" commits, it does so within the context of the higher level transaction and its changes are not visible outside the activity. Later, if "billing" aborts, the effects of "enter order" can be easily rolled back.

Unfortunately, using nested transactions may introduce performance problems. Since activities may be long running, accessed resources must be held (e.g., locked) for long periods of time. So in practice, activities are run as *sagas* [Garc87]. As each transaction commits, its resources are released. If the activity must be aborted, *compensating steps* are run for each committed transaction. For instance, to compensate for the "enter order" step, the "delete order" step can be run. Sagas provide a weaker notion of atomicity than nested transactions.

One goal of this paper is to generalize the notion of sagas as defined in [Garc87]. In particular, we need to provide for *nested sagas*, analogous to nested transactions. For example, it should be possible to let the "billing" step in reality be another activity, with its own sequence of steps and compensations. The interactions between the parent and child sagas must be defined, e.g., what happens to the parent saga if a step in a child saga aborts.

A second goal of this paper is to specify an environment in which activities and nested sagas can be specified and executed. The environment is essentially a *model* of the underlying system for use by application programs. The model can be thought of as providing a collection of services for application activities, much as an operating system provides a set of services for processes.

Within this model, it should be possible to specify how steps are linked together into activities, how activities are nested, how committed steps are compensated for, and how steps communicate with each other and access the database. We would like these specifications to be as dynamic as possible. For example, the "shipping" step from the Purchase Order activity may also be useful in another activity which accomplishes a transfer of stock between two company warehouses. Since a given step will be connected to different steps within the two activities, the connections, or *bindings*, between steps should not be part of a step's static definition. Other properties of steps, such as atomicity, should also be specified dynamically, since they may vary from activity to activity.

Our model should provide for *forward progress* of activities. Since activities may be long running and may consume substantial system resources, aborting an activity due to a system failure will be costly. Instead, we would prefer to restart where we left off before the failure, i.e., simply re-do (or complete) the missing steps.

The model we propose here is similar in many ways to exiting solutions for specifying office procedures [e.g., Elli83], for progamming-in-the-large [Dere76], and for modular programming and reuse of code [e.g., Haye87, Purt88]. However, none of these proposals include facilities for transaction and saga management. Our work can be viewed as an extension of these earlier ideas to include these facilities. Other researchers have embedded in programming languages facilities for conventional (nested) transaction processing [e.g., Herl87, Lisk88]. Our focus is not on the linguistic aspects, but rather on the system support that is necessary. Also, as we have stated, we work with nested sagas, as opposed to nested transactions. Interconnected sets of persistent transactions are presented in [Giff85]. These are similar to sagas, although nesting is not discussed. Our goals are also very similar to those of [Reut89]; our paper provides more details as to how these goals can be achieved. Finally, a recent paper [Daya90] also studies multi-transaction activities, relying on triggers to start up new steps.

We will proceed by first presenting our model and then (Section 4) discussing and justifying our major design choices. We will present the model as a collection of primitives or system calls that can be used to structure activities. For pedagogical reasons, we divide the system calls into two layers. The first layer (Section 2) provides a minimal set of constructs, while the second layer (Section 3) provides facilities for nested sagas.

2. Layer 0 of the Model

The layer 0 model provides four services to data processing applications:

1) dynamic creation and binding of "steps"

2) atomic execution of "steps"

3) notification on "step" termination

4) persistent messages

At this point, we have not really defined what a "step" is. Intuitively, a step is a program that accomplishes some part of a data processing activity (e.g., billing). In our model, a step is an instance of a *module*. Modules, in turn, are code fragments. A module contains the following:

(1) A header giving the module *name* and a set of *ports* (to be described shortly). The name is simply a string of characters, but should be unique across all modules. The ports are also strings, unique within the module.

(2) The body, written in any programming language. The body may define and use local variables. There are no global variables across modules.

The execution of a module, then, is a step. A module can have many instances active at one time, each one created explicitly by the Create command described below. A step can also access the database and and create other steps.

Embedded in the module programming language are special communication and step control commands (such as Create). We refer to these primitives as the *system interface*. The *system* is a control program that manages step execution and shared resources. Logically, the system is a centralized entity, although in reality it may be implemented as a collection of distributed sub-systems.

Steps communicate with each other via *mailboxes*. A mailbox is a system-managed queue of messages. Mailboxes are *persistent*, i.e., survive system crashes. Modules do not specify statically the mailboxes they will use. Instead they specify ports. A port is simply a local name for a mailbox that will be determined at run time. For example, a module may specify that it will use a port P and in its code will have send and receive commands to port P. At run time, the name P will be bound to a physical mailbox that will hold the messages produced or consumed when the step accesses P.

There are only six commands in the layer 0 system interface:

• *Create*: This command creates a new step using the module specified in the call. The new step will be executed atomically. An optional notification port can be specified in the call; the system delivers a message at this port when the new step commits or aborts. An *identifier* for the new step is returned from this call. The identifier can be used as an argument to subsequent system calls.

• *Commit:* Commit terminates and commits the results of the calling step. A step is implicitly committed after executing its last statement if it has not already committed or aborted.

• *Abort:* Abort takes a step identifier as an argument (if none is given, abort refers to calling step). It terminates the specified step and rolls back all actions performed by the step.

• *Bind:* Bind links two specified ports to a common mailbox. If neither port is already associated with a mailbox, then a new empty mailbox is created and both ports are linked to it. If one or the other port is linked to an existing mailbox, then the unbound port is linked to that mailbox. If *both* ports are bound to different mailboxes, an error occurs.

• *Send:* A message is delivered to a specified port. If the port is not bound to a mailbox, a new, empty mailbox is created.

• *Receive:* A message is read from a specified port. Again, if the port is unbound, a new, empty mailbox is created. (Although we do not discuss it here, Receive may be blocking or unblocking.)

To illustrate the use of the Bind command, consider the following example. Say we have four unbound ports, p1, p2, p3, p4, in different steps. Suppose that p1 and p2 are bound. This creates a new, empty mailbox, call it m1. Any messages sent to p1 or p2 will then be placed in m1. Any receives will read messages from m1. If p3 is later bound to p2, then p3 is simply linked to m1. Sends and receives on p3 will be directed to m1. If later on p4 is bound to p3, p4 will also be linked to m1. Considering a different sequence, say p1 is bound to p2 as before, and then p3 is bound to p4. This creates two separate mailboxes, m1, m2. A third bind of say p2 to p3 yields an error. (An alternative would be to merge the messages in m1 and m2 into a single mailbox.)

As we have stated, step execution is atomic. This means that the effects of any system primitives used by a step are not felt until that step commits. Specifically, newly created steps do not exist until their creator commits. A message cannot be received until its sender commits, and ports are not considered bound until the binding step commits. Should a step abort, the effects of any system primitives it has used are erased. The one exception to this rule is for self-Aborts. That is, a call to abort oneself takes effect immediately and obviously the effect of this call is not undone during the abort.

To enforce atomicity of steps the system has to control concurrent accesses to system structures and mailboxes. For instance, if one step is binding a port while another one is trying to send a message to it, the system will have to delay one of the steps until the other commits. For mailboxes, concurrent access to them can be improved by making them *sets* of messages rather than FIFO queues [Herl88, Schw84, Weih89]. That is, for a FIFO queue, if one step reads the first message, no other step can read from that queue until that first step commits. With a set mailbox, several steps can concurrently read. Our model includes facilities for different types of mailboxes, but they are not discussed here. A recent paper [Bern90] discusses the implementation of message queues for transaction processing systems.

2.1 Examples for Layer 0

To illustrate the model, let us return to the purchase order example of Figure 1 and see how it can be implemented. We can start by writing a module for each of the five steps of Figure 1. For example, the "billing" module is shown in Figure 2, and its graphical representation in Figure 3. A commit operation is implicit at the end of the module. The remaining modules are similar.

To run the purchase order activity we need to write a module that creates the necessary steps and bindings. Such a module is shown in Figure 4, and its graphical representation in Figure 5. From the outside, the purchase order activity is simply a module. We

```
Module Billing;
 [     port: input, status
       ...
       Receive:: port: input  buffer: ddd
       get order number from ddd
       access database to read purchase order
       do billing
       prepare results in rrr
       Send:: port: status  buffer: rrr
 ]
```

Figure 2: Billing Module

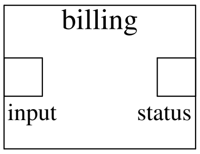

Figure 3: A Pictorial Representation of Billing Module

have added a port "trigger" simply to show that activities can have inputs (and although not shown, outputs).

```
Module PO;
 [     port: trigger
       ...
       pc ← Create::   module: "phone call"
       eo ← Create::   module: "enter order"
       bi ← Create::   module: "billing"
       in ← Create::   module: "inventory"
       sh ← Create::   module: "shipping"
       Bind::   port1:  trigger    port2:  pc.in
       Bind::   port1:  pc.out     port2:  eo.in
       Bind::   port1:  eo.tobill     port2:  bi.input
       Bind::   port1:  eo.toinv      port2:  in.input
       Bind::   port1:  bi.status     port2:  sh.in1
       Bind::   port1:  in.out        port2:  sh.in2
 ]
```

Figure 4: Purchase Order Module

Let us now walk through the execution of this activity. First, some other step creates an instance of module PO. The step that creates PO also binds port "trigger" to some port, associating "trigger" with a mailbox, say m1. When PO executes, it creates the five steps necessary for the activity. The "in" port of the "phone call" step is bound to "trigger", which associates "in" with m1. Thus, any messages that were written in m1 for PO

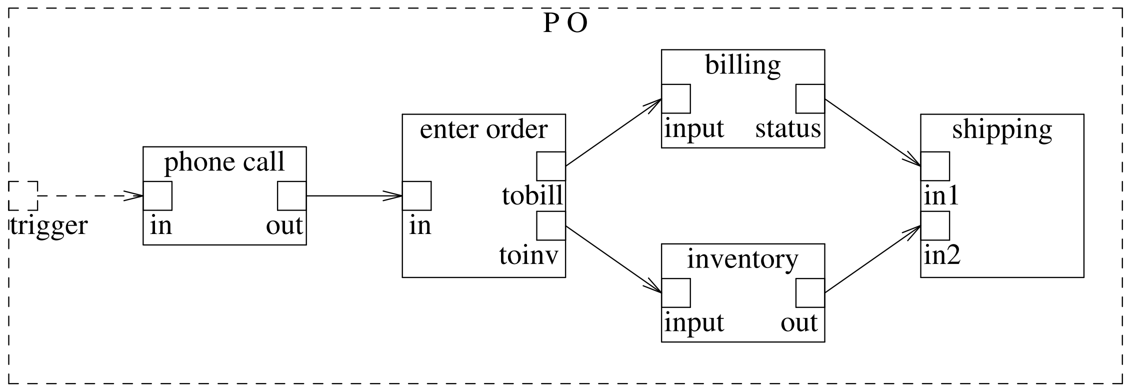

Figure 5: A Pictorial Representation of PO Module

can actually be read by the "phone call" step. However, the fact that the "phone call" step (rather than the PO step itself) will ultimately read from m1 is transparent to the PO's creator. The rest of the binds in PO will create new mailboxes. For example, the output of the "phone call" step will be written into one of these mailboxes, to be read later as input for "enter order".

The steps created by PO do not start until PO commits. At this point the "phone call" step reads from m1 and starts processing. The rest of the steps block waiting for input. When "phone call" commits, its output becomes visible and "enter order" starts processing it. (Like all steps, "phone call" is executed atomically. In our initial description we noted that this is not necessary.) When "enter order" commits, both the "billing" and "inventory" steps proceed in parallel, and so on.

When a system failure occurs, the system aborts and restarts running steps. For instance, suppose that the system fails after the "billing" and "inventory" steps have read their input and are processing it. These steps are aborted, returning the read messages to their mailboxes. (Since the steps never committed, the messages were never actually received, and therefore are still available in their mailboxes.) Both steps are then restarted. Each will read the messages it had read before the crash. Hence we achieve forward progress, as discussed in Section 1.

Although our focus here is not on implementation issues, we should note that the system must maintain persistent data structures to achieve this behavior. It must record the creation, commit, or abort of each step, as well as other activity such as the binding of ports to mailboxes and message passing.

One of the major strengths of our model, in our opinion, is that it lets us view a composite activity such as PO as a single module. The structure that is created within PO is not visible from the outside. To the creator of PO it appears that a single step is processing the order.

However, there is a subtle difference between calling a module like PO and calling another module, call it Z, that performs all the necessary actions within a single step. The difference is that Z is truly an atomic action. If PO is used, partial results may be visible before the purchase order is fully processed. For example, after "enter order" commits, the order becomes visible in the database to any other transaction. The step PO is atomic, but unlike Z, it only generates other steps that actually do the processing. From the point of view of performance, it is usually better to break up the activity into separate transactions, especially if the activity is long lived. This makes it possible to release resources earlier and reduces contention. However, if one wishes to abort an activity, it is easier if it is a single transaction like Z. In Section 3 we return to this issue and show how a multi-step activity can be effectively aborted.

So far we have not dealt with a voluntary abort by one of the steps. For example, "billing" may decide that the customer's credit is not good. One option is to compensate for the committed steps, but this will also be discussed in Section 3. Another option is to have "billing" terminate normally but encode in its output information for "shipping" that says that something went wrong. But let us say that "billing" was not written by us, and that it simply calls Abort (self) if it runs into trouble. If we do not do anything about this, "shipping" will hang waiting for its input.

The solution is to use the notification facilities provided by the model. When PO creates each step, it can use a command like:

bi ← Create:: module: "billing" notification_port: ppp

The notification port cannot be in PO itself. Recall that PO must terminate and commit for the "billing" step to execute. Hence, PO will not be around when "billing" aborts. The solution is for PO to create a "monitor" step. The notification step for "billing" above will be bound to a port in the "monitor". The same is done for other steps. The code of the "monitor" step checks at run time that all other steps commit, and takes appropriate action if not. Figure 6 is a graphical representation of the PO module with a monitor step. In the figure, the monitor is set up to receive termination notification for all other steps in PO through its port "ppp". Notification messages are produced automatically by the system. They need not be generated by the terminating step.

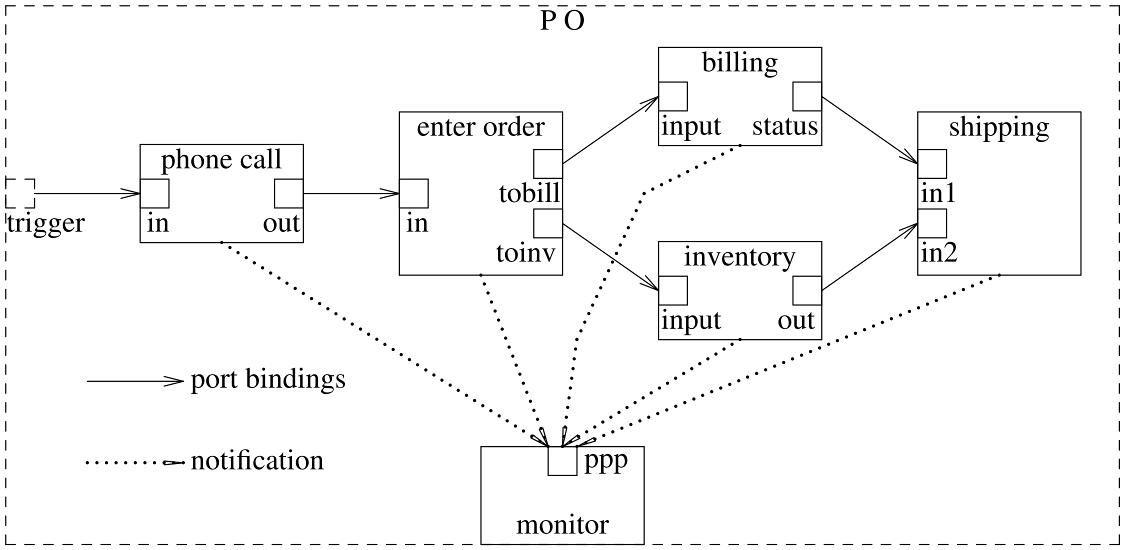

Figure 6: The Monitor and Notification in the PO Module

As we have stated, each step commits a single transaction. How can we then model a "server" task that must commit more than one transaction? The example in Figure 7 shows how this can be done. A certain mailbox m1 contains messages, where each message describes some work to do. Our server is supposed to take one message at a time, perform the work, and write an output message in a mailbox m2. When our server is initially created, its "inbox" port is bound to m1, its output port to m2. This binding is done by the creator of the server. This first instance of the server processes the first message. Before it terminates, it creates a clone of itself. At commit time, the first message disappears from m1, the first result appears in m2, and the clone is activated. The clone then does exactly the same thing, processing the second message and generating the next server.

3. Layer 1 of the Model

In our Purchase Order example we showed how an activity PO could be composed as a collection of steps. Initially, a step C creates an instance of the PO module to process a particular order. When PO is executed, it in turn initiates steps to handle billing, entry

```
Module Server;
 [      port: inbox,  outbox
        ...
        Receive::   port:  inbox   buffer:  ddd
        process data in ddd
        Send::   port:  outbox   buffer:  ddd
        now get ready for next task
        myclone ← Create::   module: "server"
        Bind::   port1:   inbox    port2:  myclone.inbox
        Bind::   port1:   outbox    port2:  myclone.outbox
        Commit::
 ]
```

Figure 7: Server Module

order, inventory, and other functions. From the point of view of C, PO appears to be a single step that is processing the order.

However, as we pointed out, the execution of the purchase order is not really atomic. This can present problems if the creator wishes to abort PO. To illustrate, suppose that the creator C has also created a monitor step CM, which is to be notified when PO completes. Suppose further that before the purchase order is fully processed, CM decides that the purchase order should be aborted. How is the purchase order's monitor, CM, to abort the purchase order?

The obvious answer is for CM to abort the step PO using the Abort command. However, some reflection reveals that this may not work! Recall that PO simply initiates a number of sub-steps to accomplish the task (and a monitor) and then commits. Thus, step PO may have committed long before the purchase order is fully processed. The difficulty here is that CM believes that PO is processing the purchase order, when in fact the processing is being done by PO's sub-steps.

A similar problem arises with the termination notification that CM is to receive when the purchase order has been processed. As we have described our system calls so far, CM will be notified that PO committed when that step commits. But as we know, PO commits at the beginning of the activity, once its sub-steps have been created. Instead, we would like to have the commit notification reach CM once all the associated sub-tasks have completed.

To handle these problems, the next level of our model introduces a generalized notion of *sagas* [Garc87]. Sagas are used to relate groups of steps, and to give them a common name. Sagas can be aborted and committed in a manner similar to steps. In short, a saga is aborted by aborting those of its steps that have not yet committed, and by *compensating* for committed steps. The sagas described here are more general than those in [Garc87] because they can be nested.

Before defining sagas more precisely, we will try to motivate them by showing our Purchase Order example as a saga. The new version of module PO is shown in Figure 8 (compare to Figure 4). The five steps that will process the order are created as before, except that an extra parameter "type: sub-saga" is added. This indicates that the created steps are to be child steps of PO. The parent PO and its children comprise a saga, which will be identified by the step id of the parent step, PO.

```
Module PO;
 [      port: trigger
        ...
        pc ← Create::   module: "phone call"  type: "sub-saga"
        eo ← Create::   module: "enter order" type: "sub-saga"
        bi ← Create::   module: "billing"     type: "sub-saga"
        in ← Create::   module: "inventory"   type: "sub-saga"
        sh ← Create::   module: "shipping"    type: "sub-saga"
        Bind::   port1:  trigger        port2:  pc.in
        Bind::   port1:  pc.out         port2:  eo.in
        Bind::   port1:  eo.tobill    port2:  bi.input
        Bind::   port1:  eo.toinv     port2:  in.input
        Bind::   port1:  bi.status    port2:  sh.in1
        Bind::   port1:  in.out       port2:  sh.in2
        eo' ← Create::   module: "delete order" type: "independent"
        bi' ← Create::   module: "crediting"    type: "independent"
        in' ← Create::   module: "add-stock"    type: "independent"
        Compensation_Bind:: forward_step: eo  comp_step: eo'
        Compensation_Bind:: forward_step: bi  comp_step: bi'
        Compensation_Bind:: forward_step: in  comp_step: in'
 ]
```

Figure 8: Purchase Order Module (as Saga)

The six Bind commands are as before and set up the communication structure for the activity. They are followed by three new Create commands. These are for *compensating steps*, which may be used in the event that the PO saga is aborted. Each compensating step is associated with a ‘‘forward’’ step via the Compensation_Bind command. We are assuming here that the compensating steps contain the application code necessary to compensate for their corresponding forward steps. For example, "delete order" would erase the customer order placed by the "enter order" step. Not all steps require compensation. In our example, we have not created compensating steps for the "phone call" and "shipping" sub-steps. Note that the compensating steps are created with ‘‘type: independent’’ to specify they are not actually part of the PO saga.

Let us now walk through an execution of the new PO. When C creates an instance of PO, it receives a step identifier, which it can then pass to the monitor step, CM. This same identifier also identifies the saga that has PO as a parent. Initially, this saga has no children. Step C can also specify a port in CM that should receive notification when the PO is completed.

After C commits, the created steps PO and CM are allowed to run. When the PO step is finished, it will commit, thereby activating each of the steps it created as part of the saga. (The compensating steps will only be activated if compensation is required.) Since the PO has active sub-steps, the CM is *not* notified of the commit of PO at this point. The system will notify CM only after the entire saga has committed or aborted.

The monitor can abort the PO saga at any time by using the Abort command, with the identifier for PO as an argument. However, since a saga is not atomic like a step, aborting a saga is not merely a matter of rolling back the effects of the component steps. Some of these steps may already have committed and their effects seen by other activities.

Let us suppose that CM issues an abort command while the "inventory" and "shipping" steps are still active, and that the others have committed. The command will cause the still-active "inventory" and "shipping" steps to be aborted and rolled back. The compensating actions "crediting" and "delete order" will be automatically initiated to compensate for the "billing" and "enter order" steps. (The compensating actions are initiated in the reverse of the commit order of their corresponding forward steps.) The monitor, CM, will then be notified of the abort of the saga PO.

In many cases a compensation step needs to receive information from its forward step. For example, "enter order" might pass an order number to "delete order." This can be accomplished by binding the appropriate ports together (using the Bind command). Within the "enter order" and "delete order" modules, Send and Receive would be used to transfer the information. Step "enter order" and its compensating step, "delete order", can exchange information even though they will not be executing concurrently. The message sent by "enter order" is simply stored in the mailbox created by Bind. In fact, "delete order" may never execute; it will only be executed if the "enter order" saga is aborted after committing. In effect, "enter order" is sending the compensation information *in case* compensation is eventually necessary.

Now let us briefly consider a different execution scenario. Say that CM does not abort PO, but instead step "billing" voluntarily aborts (e.g., customer credit was bad). Since "billing" is part of a saga, the PO saga itself is aborted. This proceeds as described above, with the abort of active steps and the compensation of committed ones. In some cases, it may be undesirable to have this automatic saga abort when an element aborts. A way of changing this default will be described in Section 3.2.

It is important to note that sagas can be nested. For example, what PO thinks is a simple "billing" step, may actually generate sub-steps of its own. Similarly, the creator step C may be a component step of a higher level saga. Thus, in general, a saga will contain a parent step, plus zero or more children, where the children can be steps or other sagas. Figure 9 illustrates the nesting of sagas. In the figure, the PO saga is shown as a sub-saga (We have used "C" to refer to the creator of PO and to the saga of which it is a member.) Similarly, the figure illustrates that the "billing" step of PO may itself be a saga, complete with a collection of steps (or sub-sagas) and a monitor.

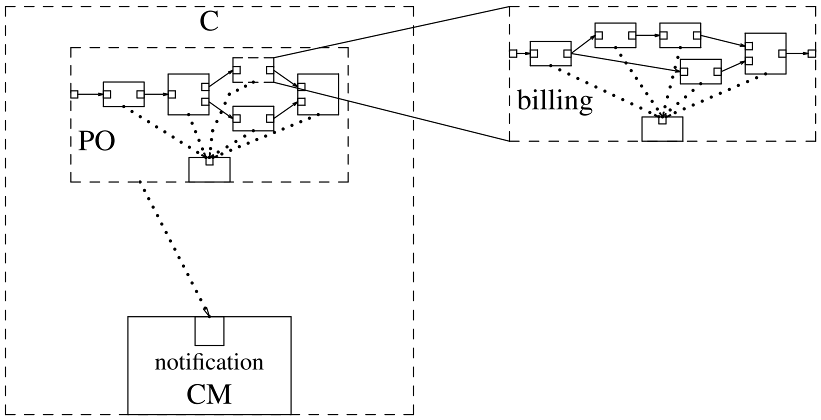

Figure 9: Nesting of Sagas

Aborts are propagated down and up the tree of nested sagas. To illustrate, suppose that PO is aborted and that the "billing" step has set up other sub-steps to do its work. If PO is aborted before the step "billing" commits, then the step is undone (none of its children were started). If "billing" had committed and its children were active, then the "billing" sub-saga would still be considered active. It would be aborted by aborting or compensating its children. If the saga "billing" had committed (i.e., its children completed), then the compensating step for "billing", "crediting", is executed. Going in the opposide direction now, if a child of "billing" aborts, then the "billing" saga would be aborted, which would then initiate the abort of the PO saga.

The definition of a nested saga can be simplified by treating steps as sagas with no children. Then we can say that sagas consist of a parent step plus zero or more child sagas. From the point of view of our commands, it is also possible and advantageous to treat a simple step as a saga (with the step as parent and no children). The id of the saga is the id of its parent which is the id of the step. Thus, instead of having in layer 1 commands (e.g., abort, create, ...) for steps *and* sagas, now we will only have commands for sagas.

3.1 Sagas

In this section we summarize the rules for sagas. As mentioned above, a saga consists of a parent step, plus zero or more sub-sagas, each created by the parent. We will refer to the parent step and any sub-sagas as the *elements* of the saga. Thus, any step can be a part of at most two sagas. It can be a parent in one, and a child in another. As we have stated, the parent's identifier serves as the saga's identifier as well.

A saga is committed implicitly when its parent step and each of its sub-sagas have committed. Similarly, a saga is aborted when any of its sub-sagas (or the parent step) abort. Only active (neither committed nor aborted) sagas can be aborted. A saga is aborted by aborting each of its active elements, and by *compensating* for each committed element. Compensation is accomplished by initiating a *compensating step*. A compensating step is simply a step that undoes, according to the semantics of the application, the effects of the step being compensated for [Garc87]. Note that the compensating step is itself a saga. Of course, it is up to the application to determine the appropriate compensation for each step. (Compensating steps should not be aborted. If they are, the system administrator is notified of an anomalous compensation.)

In order to incorporate sagas into our system interface, we need to add a few additional parameters to existing system calls. The Bind, Send, and Receive calls are unchanged. The remaining calls are modified as follows:

• *Create*: As already described, creating a new step implicitly creates a new saga. The creating step may specify whether the new step is to be a part of the parent's saga, or independent. This is accomplished by adding an additional parameter (type) to the call (the default is type: independent). If the new saga is independent, an abort of the creating saga will not effect the new one. The creating step can also specify a notification port, which will be notified at the commit or abort of the newly created *saga* (not the step itself). This issue is discussed further in Section 3.2.

• *Commit*: The commit call causes the effects of the calling step to be released, i.e., to become permanent, as in the Level 0 model. If the step's saga has no active sub-sagas, then the saga is also committed. The saga's commit may result in the commit of another saga, of which the just-committed saga was the last active sub-saga. Furthermore, if a step requested notification, then a message indicating that the saga has committed will be delivered.

• *Abort*: Takes as parameter a step/saga id. Abort is similar to commit, in that both the saga and the step are affected. (Note, however, that the step may have committed while the saga remains active, because of active sub-sagas.) If the step is active, it is undone, as in the level 0 model. Compensating steps are initiated for each of the saga's committed sub-sagas. Active sub-sagas are (recursively) aborted. The abort of a saga also propagates up the tree of sagas. That is, if the aborted saga was created with type: sub-saga, then its parent saga is aborted, and so on.

The new system call described below is used to specify compensation steps.

• *Compensation_Bind*: Two identifiers are given as parameters, one naming the saga to be compensated for (S), and the other the compensating saga/step (S'). Note that both S and S' must already exist when this call is made. This call makes S' the compensating step for step/saga S. It also has the two effects on S'. First, S' is turned off, so that it will not execute until initiated by a call to Abort. (Normally, a step will be executable as soon as its creator commits.) Second, S' is made an independent saga, regardless of whether or not it was created as an independent (see Create).

3.2 The Non_Vital Option

As we have described, there are two options for creating children. One is to use the "type: sub-saga" option. This makes the parent and the child sub-sagas a monolithic computation. Aborting the parent aborts all of its children recursively, and aborting any descendant aborts the parent (and therefore, all other children). Alternatively, children can be created as "independents". In this case, an abort of the parent has no effect on the child, and vice versa.

These options are sufficient for many situations. However, in some cases it will be desirable to permit some additional flexibility in the way aborts are cascaded. In some cases we would like to allow a child to abort without aborting the its parent, though the parent's abort should always result in the abortion of the child. Consider an instance of the PO activity in which the "inventory" step fails because sufficient parts are not available to satisfy the customer's order. In this case, it may be possible to substitute a similar part to complete the order. Thus, in the case of a failed inventory step, the corrective action desired by the application would be the invocation of an alternate instance of the "inventory" step, without affecting the other steps of the PO activity.

If the original "inventory" step is created as a sub-saga, then its Abort would cause the Abort of the entire PO activity, which is not desired in this case. If "inventory" is independent, the entire PO need not be Aborted. However, if another step (e.g., a step monitoring PO) should need to cancel the entire PO for some reason, an Abort of PO will leave the "inventory" step untouched!

We now introduce a third method of creating new sagas that allows a flexible response to the abortion of a sub-saga, while still allowing the entire activity to be treated as a saga "from above". For this we add a new argument to the Create command. It specifies whether the new sub-saga will be *vital* or *non_vital* to its parent saga (default is vital). Aborting a vital sub-saga causes its parent saga to be aborted as well, as described earlier. Aborting a non_vital sub-saga has no effect on the parent saga. Note that the distinction between vital and non-vital sagas is ignored in case the newly created saga is to be independent, rather than a sub-saga.

Note that with the non-vital option, if recovery from an element failure is desired, it must be undertaken manually. In our example, if the PO application requires that an alternative "inventory" step be initiated in case of the failure of the first one, the application must initiate the alternative step itself. A simple way to accomplish this is to create a monitor step for "inventory". The monitor can wait for notification of the termination (abort or commit) of the "inventory" step, and then take the appropriate action (e.g., Create another step).

In short, the system provides two "automatic" reactions to the abort of a saga, specified by the independent and vital sub-saga saga creation options. By using non-vital sub-sagas, an application can handle an abort "manually", taking whatever actions are deemed necessary.

4. Discussion

Our proposal can be thought of as an "operating system" for activities. A conventional operating system provides the notion of *processes* and inter-process communication facilities. Processes use system calls to start up other processes, to communicate, or to request other services. In our model, a step is analogous to a process. The calls we defined are used by steps to request services from the system.

Our major contribution is the addition of transactions, sagas, and forward progress to such an operating systems view. In our model, modules embody five concepts:

(1) Modules are the units of program composition. That is, we build complex programs (activities) by linking together modules.

(2) Modules are the units of execution. As processes in an operating system, module instances are created and executed. They have their own storage (local variables).

(3) Modules are the units of atomicity. That is, modules are executed as transactions.

(4) Modules are the units of compensation. When compensating for an aborted activity, we compensate for executed steps. The compensation itself is also a step.

(5) Modules are the units of checkpointing. At the end of each step, we essentially take a checkpoint (by making relevant system structures and mailboxes persistent), so that a completed step will never have to be rerun because of a system failure.

One can clearly argue whether combining all these concepts into a single one is the right thing to do. There are cases where this is not appropriate. For example, it may be desirable to execute several steps within a single transaction. However, we believe that this will be more of the exception. In a majority of the examples we have studied, the various concepts naturally overlap. For instance, consider the "billing" step in our running example. There is no need to share the local, volatile data used with other steps. If there is a problem, this is the natural unit of work to abort or to compensate, and so on. Thus, we opted for simple models, with few concepts and commands, that could be useful for describing the most common activities. (Note that Layer 2 of the model, not covered here, does allow one to run several steps within a single transaction, possibly as a nested transaction.)

Related to our decision to have a simple model is our decision *not* to have the notion of a context as suggested in [Reut89]. In other words, we do not have global variables that can be accessed by the steps of an activity. These variables must be non-volatile and have some form of concurrency control, so it is simpler to use the existing database facilities for this. Hence, our view is that if steps do have global data they want to share, they can store it in the database. If desired, the database access controls can be used to restrict access to this data only to the members of an activity.

Not having global variables may affect the programming style for steps. It could be argued that global variables simplify the programming task. For example, in our purchase order example, it would be convenient to keep the order number is a global variable, so that every step can refer to it simply by the name of this variable. In our model instead, the programmer is forced to either read the order number from the database, or to receive it in a message. However, with the appropriate facilities we feel that receiving the "global" data at the beginning of a step (and sending it at the end) should not be that different from using global variables.

Another area where our design philosophy is reflected is type checking. In particular, we have provided a very weak data type system. Essentially, we assume that data in messages is accompanied by its type. At run time, as data is transferred from the host language to a message (and vice versa), the types are checked.

An alternative would be to perform compile time checking. For this, we would need facilities for defining "message types". Send and Receive would then specify the message type, and at compile time the parameters used in the call would be compared to the pre-defined ones in the message type. While this may provide a more elegant solution, it does involve a more complex model. Such a model would also involve more work to implement: there would be more extensive modifications to existing compilers and a shared message-type repository would have to be supported.

As discussed in the introduction, one of our main goals was to facilitate the reuse of steps. We believe that our model successfully decouples the internals of a step from environment where the step will run. The environment is specified by the creator of the step, which defines the transaction and compensation rules for the step, as well as its binding to other steps. It should even be relatively simple to use software that was written without any knowledge of our model. For example, suppose we are handed a C procedure that does the "billing" step. This procedure as a set of input and output parameters. We can simply write a "shell" module around the given procedure. The shell will receive the necessary data, transferring it to local C variables. The shell then calls the C procedure, and on return, packages the results into an output message.

Another goal for our model was to provide a communication mechanism. We made the decision to use mailboxes, as opposed to more conventional message passing mechanisms where a connection is established between processes directly. Our belief is that mailboxes make it easier to decouple steps. A step can exchange data with steps that are not active concurrently (or may not even be created yet). Also, our automatic mailbox creation scheme makes it unnecessary for steps to create mailboxes or even know their names. As described in Section 2, the system creates mailboxes dynamically as they are needed. It also performs garbage collection, removing mailboxes that are not referenced by any existing step.

Ports (in modules) are bound to mailboxes at run time. This makes it possible to specify the communication paths between modules dynamically and outside the steps themselves. If this is a good idea, a natural question to ask is why it was not carried further. In particular, the same idea can be applied to module names (or database object names). As we have described it here, to create a step one needs to know the module name (see command Create_Step). For example, in Figure 4, module PO must know the name "enter order". Thus, there is a static association between the creator module PO and the created one, "enter order". As an alternative, we could define a "module_port" in PO, call it "ppp". The Create_Step call would refer to module ppp, but before this call was made, the name ppp would have to be bound to the real name "enter order". This gives us more flexibility is designing activities. For instance, we can now write module PO without knowing what the second step will be. This decision can be made by the creator of PO who binds the module_port ppp to the name of the module to actually use. This dynamic binding was not included in the model to keep our presentation simple, and because we are not sure how useful it would be in practice.

An important question that must be addressed is that of performance. In particular, our model forces the system to make all messages and certain system data structures persistent. First, let us point out that the overhead may not be that great. We expect many steps to execute a transaction anyway, so the message and system data can be included in the context of these transactions. Second, we expect steps to do substantial amounts of computation, so the overhead in terms of percent increase in resource utilization may not be large. If steps were very small granules, then this would not be the case. Third, we can point out that we are getting something in return for the extra overhead. An activity is automatically checkpointed after each step; thus we need not worry about manual checkpoints or about losing long computations due to a system failure.

5. Conclusions

In this paper we have presented a simple model for coordinating multi-transaction activities, using a minimum of concepts. Let us briefly return to the goals that were set out for such a model.

Concurrency. Steps are executed as transactions, a well understood technique for coping with concurrent access to shared data.

Modularity. The use of ports, dynamic binding, and nesting makes it possible for a module to be used as part of other activities without modification. This capability is similar in philosophy to the standard-input/standard output facility provided for UNIX processes. In our model, collections of cooperating steps can be made to look like a single step, facilitating the construction of complex applications.

Step Failures. Facilities are provided for coping with failed steps. A monitor step can be notified of the failure, and it can manually correct the situation. Or with the saga facility, a collection of steps can be made into a monolithic computation, where the failure of one element causes the abortion of the entire activity.

System Failures. By making system structures persistent, forward progress is achieved. After a system failure, the system restarts activities where they were suspended.

Communication. Communication is achieved in a flexible way through the Bind, Send, and Receive commands. Steps can also communicate via the shared database.

We are currently working on Layer 2 of the model. As discussed in Section 4, this layer introduces nested transactions. It makes it possible to run several steps within the scope of a single transaction.

Another extension that is being studied is a graphical tool for defining activities. With this tool one would be able to describe the Purchase Order activity pictorially as in Figure 4, rather than with code, as in Figures 4 or 8. The tool would then produce the necessary code (Figures 4, 8) automatically. The tool could also make available commonly used structures, such as that of a server (Figure 7).

6. References

[Bern90] P.A. Bernstein, M. Hsu, B. Mann, "Recoverable Queues," *Proceedings ACM SIGMOD Conference*, Atlantic City, May 1990, to appear.

[Daya90] U. Dayal, M. Hsu, R. Ladin, "Organizing Long-Running Activities with Triggers and transactions," *Proceedings ACM SIGMOD Conference*, Atlantic City, May 1990, to appear.

[Dere76] F. DeRemer, H. H. Kron, "Programming-in-the-Large Versus Programming-in-the-Small," *IEEE Transactions on Software Engineering*, Vol. SE-2, No. 2, June 1976, pp. 80-86.

[Elli83] C. A. Ellis, "Formal and Informal Models of Office Activity," *Information Processing 83*, R.E.A. Mason, Editor, Elsevier Science Publishers (North Hollnad), 1983, pp. 11-22.

[Garc87] H. Garcia-Molina, K. Salem, "Sagas," *Proc. 1987 SIGMOD International Conference on Management of Data*, May 1987, pp. 249-259.

[Giff85] D. K. Gifford, J. E. Donahue, "Coordinating Independent Atomic Actions," *COMPCON85 Digest of Papers*, San Francisco, CA, IEEE Computer Society Press, February, 1985, pp. 92-95.

[Haye87] R. Hayes, R. Schlichting, "Facilitating Mixed Language Programming in Distributed Systems," *IEEE Transactions on Software Engineering*, Vol. SE-13, No. 12, December 1987, pp. 1254-1264.

[Herl87] M. P. Herlihy, J. Wing, "Avalon: Language Support for Reliable Distributed Systems," *Proceedings IEEE 17th International Symposium of Fault-Tolerant Computing*, Pittsburgh, July 1987.

[Herl88] M. P. Herlihy, W. E. Weihl, "Hybrid Concurrency Control for Abstract Data Types," Technical Report MIT/LCS/TM-368, Laboratory for Computer Science, Massachusetts Institute of Technology, August 1988.

[Lisk88] B. Liskov, "Distributed Programming in Argus," *Communications of the ACM*, Vol. 31, No. 3, March 1988, pp. 300-312.

[Moss85] J.E.B. Moss, *Nested Transactions: An Approach to Reliable Distributed Computing*, MIT Press, Cambridge, MA, 1985.

[Purt88] J. M. Purtilo, "A Software Interconnection Technology," Technical Report CS-TR-2139, Department of Computer Science, University of Maryland, November 1988.

[Reut89] A. Reuter, ‘‘Contracts: A Means for Extending Control Beyond Transaction Boundaries,’’ Presentation at *Third International Workshop on High Performance Systems*, Pacific Grove (Asilomar), California, September 1989.

[Schw84] P. M. Schwarz and A. Z. Spector, ‘‘Synchronizing Shared Abstract Types,’’ *ACM Transactions on Computer Systems*, Vol. 2, No. 3, August 1984, pp. 223-250.

[Spec87] A. Z. Spector et al, ‘‘Camelot: A Distributed Transaction Facility for Mach and the Internet — An Interim Report,’’ Technical Report CMU-CS-87-129, Department of Computer Science, Carnegie Mellon University, 1987.

[Weih89] W. E. Weihl, ‘‘Local Autonomy Properties: Modular Concurrency Control for Abstract Data Types,’’ *ACM Transactions on Programming Languages and Systems*, Vol. 11, No. 2, April 1989, pp. 249-282.

[Wipf89] A. J. Wipfler, *Distributed Processing in the CICS Environment*, McGraw Hill Book Company, 1989.