COMPUTING SCIENCE

Structuring Call Control Software Using Distributed Objects

H. Blair, S.J. Caughey, H. Green and S.K. Shrivastava

TECHNICAL REPORT SERIES

No. 558 November, 1996
Structuring Call Control Software Using Distributed Objects

H. Blair, S.J. Caughey, H. Green and S.K. Shrivastava

Abstract

Present day telecommunications systems make use of computing technology that places excessive reliance on specialist equipment and techniques for delivering core services of switching and call management. It is becoming increasingly difficult to maintain and enhance these systems to incorporate new services and functionalities. Continuing advances in distributed computing technology hold the promise of a way out of this difficulty. The paper analyses the problems facing telecommunications software and describes how it can be restructured using object-oriented techniques. The approach presented opens up a way of structuring telecommunications applications using CORBA technology.
Bibliographical details

STRUCTURING CALL CONTROL SOFTWARE USING DISTRIBUTED OBJECTS

Structuring Call Control Software Using Distributed Objects
[By] H. Blair [and others].


(University of Newcastle upon Tyne, Computing Science, Technical Report Series, no. 558)

Added entries

UNIVERSITY OF NEWCASTLE UPON TYNE.
BLAIR, H.

Abstract

Present day telecommunications systems make use of computing technology that places excessive reliance on specialist equipment and techniques for delivering core services of switching and call management. It is becoming increasingly difficult to maintain and enhance these systems to incorporate new services and functionalities. Continuing advances in distributed computing technology hold the promise of a way out of this difficulty. The paper analyses the problems facing telecommunications software and describes how it can be restructured using object-oriented techniques. The approach presented opens up a way of structuring telecommunications applications using CORBA technology.

About the author

Mr. Blair has been with GPT since 1989 where he has been working on call control, database and software architecture issues. Prior to that he was working for a company in Germany on a number of process control projects.

Steve Caughey joined the Department of Computing Science in October 1989 after he completed his CSSD course. He is a research associate currently working on fault tolerance for distributed memory multiprocessors.

Dr. Howard I. Green has been in the telecommunications industry as a system designer and architect all his working life. He is currently System Design Manager for the next-generation evolution of GPT's System X public switching system.

S.K. Shrivastava joined the Department of Computing Science in August 1975, where he is a Professor.

Suggested keywords

CORBA
DISTRIBUTED COMPUTING
FAULT-TOLERANCE

OBJECT-ORIENTED TECHNIQUES
TELECOMMUNICATIONS SOFTWARE
TELEPHONY

Suggested classmarks (primary classmark underlined)
Dewey (18th): 001.64404 001.6425 621.385
U.D.C. 519.687 621.322.06 621.395
Structuring Call Control Software Using Distributed Objects

H. Blair†, S. J. Caughey‡, H. Green‡ and S. K. Shrivastava‡

†GPT Ltd.,
New Century Park, Coventry, CV3 1HJ, England.

‡Department of Computing Science,
University of Newcastle,
Newcastle upon Tyne, NE1 7RU, England.

Abstract

Present day telecommunications systems make use of computing technology that places excessive reliance on specialist equipment and techniques for delivering core services of switching and call management. It is becoming increasingly difficult to maintain and enhance these systems to incorporate new services and functionalities. Continuing advances in distributed computing technology hold the promise of a way out of this difficulty. The paper analyses the problems facing telecommunications software and describes how it can be restructured using object-oriented techniques. The approach presented opens up a way of structuring telecommunications applications using CORBA technology.

1. Introduction

Telephony systems have two very onerous, and potentially conflicting requirements. Firstly, as such systems are largely concerned with protecting the ability to communicate, they have very strict requirements for availability (downtime in current systems: < 10 minutes / year, typical next generation system requirements: < 3 minutes / year). Secondly, they require high performance.

These are "traditional" telecommunications system concerns, which have been the daily fare of switch designers for many years. Our work is motivated by the observation that in present day systems, the requirements stated above have been met by making use of computing technology that places reliance on specialist equipment and techniques for delivering core services of switching and call management. It is becoming increasingly difficult to maintain and enhance these systems to incorporate new services and functionalities, and customers increasingly wish to take advantage of the rapid improvement in price/performance offered by industry standard hardware. Continuing advances in distributed computing technology hold the promise of a way out of this difficulty. However, the ideas developed in the area of distributed computing (objects interacting via invocations, with dependability achieved by utilising techniques such as atomic transactions and object replication) will need considerable refinement before they can be adapted to deliver the availability as well as the timeliness requirements demanded by next generation telecommunications services. In particular, the
asynchronous nature of telecommunications signalling protocols and the resulting state machine specifications needs to be taken into account.

This paper analyses the problems facing telecommunications software and describes how it can be restructured using object-oriented techniques. The approach presented opens up a way of structuring telecommunications applications using CORBA technology by making use of a suitably enhanced ORB [1] and CORBA services [2].

2. Structure of Telecommunications Applications

The idea of a client-server interaction, in which a user “asks the system a question” and the system responds, is deeply embedded in the way the data processing industry thinks about what it does. Telecommunications, by contrast, has always required at least two users and a system, or a network of systems, for a meaningful interaction. The system becomes an intermediary between the users, forming connections (or more abstractly bindings) on their behalf.

Current telecommunications applications are still essentially bound by the “step-by-step” setup of circuits. In this context, we are creating a bearer model and then establishing what the users want. In the terms above, the binding to a service is implicit (since basic telephony is the only service on offer). When we move to a network providing variable bandwidth, one-to-many communications and many qualities of service, this model is clearly inadequate.

TINA has promoted the notion of a “separated” service layer, in which services are provided by a network-wide distributed processing infrastructure, which makes requests of a lower bearer layer to create connections [3]. The service layer is responsible for coordinating the creation, modification and destruction of service agreements amongst users. For a given call, this leads to the identification of the required stream model, which identifies for each participant the information streams sent and received, with the appropriate information characteristics. The bearer layer receives requests from the service layer and is responsible for setting up the stream model connecting a set of endpoints.

Within the service layer, the original “step-by-step” principle for call control is still maintained, but in the more abstract context of piecewise binding creation. At each step, starting from the originating user agent, we can create a new binding segment by binding to an entity (e.g. a network interconnection point or node) “in the direction of” the terminating user agent. Each of these binding segments may possess state. In this context, the invocation of a telecommunications feature (e.g. call waiting), or the traversing of a transparency constraint (e.g. network boundary) may be modelled by the creation of a new binding segment (i.e. adding an independent state machine representing the feature to the total relationship state). Below we discuss this aspect in detail.

2.1. State Machines

Within telephony services, call functionality is traditionally modelled as a state machine in which external events are received from the users active in a particular call (or the system administering the call) and each event drives the machine into some new state. This model is particularly suited to the world of call processing where, due to the limited user interface (traditionally a small number of buttons), complex call features
may only be initiated by sequences of user actions, each driving the state of the call forward. In any real-world telephony system the total state diagram, which specifies the system's responses to all possible combination of events is hugely complex, describing a myriad of interactions between users and the system. In order to reduce this complexity, the state machine is structured as a number of smaller state machines, each dedicated to handling some particular portion of the state diagram (often relating to a particular call feature). These state machines consume external events but additionally may produce internal signals which can be communicated between state machines to drive them forward. The total state of a call is therefore described by the total state of all its individual state machines.

Structuring in this manner has two advantages. Firstly, as the number of states which can be represented by two state machines is the product of their individual number of states, by creating smaller state machines the total number of states to be represented can be greatly reduced. Secondly, improved modularity means that changes in functionality can be limited to some (hopefully small) subset of state machines and the introduction of new functionality can be expressed, largely, by creating a new state machine. These changes could be introduced without effecting other state machines. As telephony functionality tends to grow by the steady introduction of new and increasingly complex features, reached by extending sequences of user actions, this design approach has proved popular.

![Diagram of call state machines](image)

**Fig. 1.**

As the specification is structured in terms of state machines, reflecting each of these as a software entity (a *call segment* in our terminology) is the natural, and traditional, mechanism for implementing call processing software. Calls progress through a series of state transitions towards a specific goal, generally communication with some party, and so call segments tend to be bound together in *chains* connecting the parties (these chains may bifurcate i.e. split, in the case of multi-party calls) with messages passing along the chain in all directions. We have used this chaining as an organising principle for our current call control software implementation [4,5]. Call segments all support a common vocabulary of telephony events (essentially, a “superset” of network signalling protocol). For the reasons above, telephony features are represented by separate call segments, which may dynamically break in and drop out of the chain. The order in which these segments appear in the chain allows for precedence in the handing
of events, and may be used as a basis for the management of “feature interaction”. Fig. 1 shows three users connected together by a chain of call segments. We discuss problems with the above way of structuring call control software.

In most implementations, call segments are contained within ‘heavyweight’ communicating processes, where each process manages large numbers of calls, and keeps track of the progress of each call through the segment by maintaining call records. Events are implemented as asynchronous messages. Each process waits for messages and on receiving one directs it to the appropriate call record. A key feature of this model is that a process handles only one message at a time. This avoids any concurrency control problems. As these processes are single threaded, it becomes necessary to make all inter-process communication asynchronous (synchronous, request-response, communication will block a process, thereby making it unresponsive to incoming messages). Unfortunately, this makes the programming of a given state machine more complex than need be. Call segments may have to access persistent data services (or interact with hardware devices) which are structured as separate entities. Such accesses are ‘request-response’ interactions, that are best structured as (synchronous) remote procedure calls. However, call segments have no provision for making synchronous calls, so need to explicitly program using messages. This introduces complexity, since code and states must be written to deal with incoming events happening before the response. Moreover, it leads to a conflict of objectives between efficiency and modularity. Efficiency considerations demand that the call processing software be broken into only a small number of processes, so as to minimise the number of inter-process messages (because sending/receiving of a message requires expensive kernel calls), but software engineering concerns of modularity and extensibility demand that the software be broken into larger number of processes.

The present way of restructuring a single state machine (say X) into a number of smaller state machines (say a,b,c,d) has another problem. A state machine selects a message from its queue of input messages, performs some action and, optionally, outputs one or more messages. This entire process is atomic (free from interference), in the sense that the processing of the action is not interrupted to service other input messages. A single state transition of X could represent a sequence of transitions of a...d. Although each individual state transition of this sequence is atomic, no attempt is made to make the entire sequence atomic. This can lead to situations where different state machines have conflicting views of the state of the call or of resources attached to it. Some of these situations are inherent to the telecomms problem, and must be catered for in any case. For example:

1. User A calls User B. At some point during the setup, User B picks up his phone. (In Fig 1 above, suppose the state machine d represents the agent of User B.) If the event ‘call seize’ reaches d before the event ‘off-hook’ from User B, then d must deal with the problem. If d has got so far as to ring B’s telephone, then d assumes B is answering A’s call. If the ‘off-hook’ from B precedes the ‘seize’ from A at d, then B is treated as busy.

2. It is possible for network circuits connecting exchanges A and B to be selected from both ends. This creates the possibility that A may select a particular circuit x for a new call using the route A->B, since x is free in its busy-free tables. At the same time, B may already have selected x for a call using B->A. This will be discovered when A
signals to B. The network protocol resolves this problem by getting one end or the other to “back off”.

However, where state machine decomposition is being used to simplify the behaviour of one particular actor in a call, extra complexity can arise from differences in the ordering of events. This is true, for example, of the user agent and user features (i.e. features modifying the behaviour of the user agent). Where the features are represented as separate state machines from the user agent, new possibilities for interleaving appear (particularly as the feature state machines may be dynamically inserted and removed from the call chains). The approach developed in the next section explores transactional solutions to this problem.

2.2. Fault-tolerance

Switches have traditionally been designed with extensive hardware level redundancy to provide a highly available hardware platform. All major telecommunications manufacturers have spent much time and money in developing and improving their own processor technology, because until now there has been no commercial platform available that could deliver the availability required of a switching system. Typical architectures involve microprocessors with proprietary instruction sets targeted at efficient bit operations and indexing, with all elements (processor, memory I/O, power supply etc.) replicated with no single points of failure.

Despite the fact that call control software executes on highly reliable hardware platforms, application software is known to get into inconsistent state. A major cause of unreliability of software must therefore be due to faults (bugs) in software itself. There is much evidence to suggest that many of these faults are of a transient nature (e.g., [6]). For example, interference during call processing, as discussed previously, may cause seemingly random faults which are difficult to diagnose or prevent. Additionally, programming errors which have remained undetected through the development phase may be present, manifesting themselves only within certain, rare, states.

For this reason, call processing relies upon the use of backward recovery, i.e. the ability to reset the system to some predefined or earlier state, following error detection. Code is inserted by programmers to check the validity of messages received and consistency of internal data. Errors detected are reported to a distributed rollback service which analyses reports and decides upon appropriate recovery action based upon the severity and frequency of the reports. Recovery actions range from reinitialising particular pages of data belonging to a suspected process, through process and processor reset, with the ultimate level being a complete restart of the system.

Switching systems need to maintain persistent data, e.g., subscriber data, configuration information statistics etc. (data/resource entities shown in fig. 1). This data is shared between many calls and call segments and frequently held within a number of databases which offer simple transactional support to control concurrent access. In general this data consists of a working copy (often held in main memory for efficiency considerations) and one or more backup copies. The higher levels of recovery recover portions, or all, of these backup copies.

The fault-tolerance approach discussed above suffers from drawbacks both at the hardware as well as at the software level.
At the hardware level, reliance on the use of specialist fault-tolerant hardware is becoming increasingly unattractive. There is intense pressure to reduce costs and at the same time, meet increased performance requirements. It would be cheaper to use processor types that are widely available commercially - but one of the most difficult technologies to master is that of hardware-based replication, especially as processor busses become wider, and clock rates become faster. Further, every new microprocessor architecture requires considerable re-design effort. Software-based replication (whereby replica synchronism is maintained at the process level by making use of appropriate software implemented protocols) is therefore an attractive technology [7, 8]. The telecomms applications are, however, particularly demanding for such a strategy, since they have very high rates of interaction with the environment, and each such interaction must be agreed between the processors. In terms of performance, a hardware-implemented platform will always out perform its software equivalent. In the medium term therefore, switching systems are expected to make use of commercially available microprocessors and use hardware-based replication techniques for fault-tolerance. In the long run, the situation is likely to change in favour of software-implemented approaches.

At the software level, structuring call processing as communicating state machines concentrates upon the encapsulation of functionality but fails to encapsulate data. This is unfortunate as it is data which is of primary interest within software error recovery and so using this approach makes it difficult to contain errors and prevent their effects spreading into the system. Recovery action is taken upon the units of encapsulation familiar to the rollback service i.e. pages, processes and processors, rather than the units of concern to the application programmer e.g. call processing activities. In such an approach the rollback service must often take a conservative approach and recover parts of the system which may be unaffected by a fault, in order to ensure recovery. Indeed as an individual call's data can span pages, processes, processors and even exchanges, backward recovery of these units can place calls into inconsistent states which then need to initiate further recovery.

3. Structuring Using Objects

In this section we discuss how structuring of call control software using distributed objects can help reduce software level problems mentioned above. The main problems with the call segment based software structuring is that it lacks a conceptually simple way of dealing with consistency problems caused by processing of multiple events (discussed in section 2.1) and further, recovery action is taken upon the units of encapsulation familiar to the rollback service rather than the units of concern to the application programmer. We remedy this by introducing the notion of an activity. Activities are units of recovery and deadlock resolution. An attractive feature of the model that we present here is that it can be implemented by making use of a suitably enhanced ORB and CORBA services.

3.1. Distributed Objects

We now develop an object oriented approach for restructuring call processing software in a way that masters complexity without compromising efficiency. Segments are objects (instances of appropriate classes) and inter-segment communication is via method invocation. Multi-threaded processes are used, where each process is capable of managing several objects. This means that breaking the call control software into a
number of segments need not lead to corresponding increase in inter-process communication, as objects within a process communicate using procedure calls. Objects may request the creation and deletion of other objects. (Note that our model does not assume dynamic creation/deletion but allows for objects to be obtained/returned from some free pool).

Fig. 2 illustrates the basic architecture. External events e.g. events generated by subscriber telephones, are transformed into software messages by signalling specific hardware e.g. a Line Handler Unit, and delivered at a software message port. A thread, dedicated to listening at the port, allocates each incoming message to a thread which is then responsible for processing that particular event. This thread performs an invocation on some relevant object, passing it the message contents as parameters to the invocation. Objects process events by carrying out invocations on other objects. If an invocation is upon a remote object then a message is passed to the relevant process (which may be on a different processor) via its message port and the invocation is performed there. Invocation is distribution transparent from the perspective of the invoking and invoked objects.

An object directly represents a state machine: input messages represent requests for object method invocations, a state transition represents a method execution and output messages represent asynchronous method invocations on other objects (an asynchronous method invocation is non-blocking; the caller invokes the method but does not wait for results). In addition, we also permit an object to make synchronous (blocking) invocations. When an object invokes another object and expects results from that invocation, then such invocations are made using blocking calls. Generalising, a method of an object may contain both asynchronous (non-blocking) as
well as synchronous (blocking) calls. To ensure interference free execution of object methods, each object enforces concurrency control permitting multiple (simultaneous) execution of its read-only methods but exclusive execution of its methods involving state changes. Concurrency control raises the possibility of deadlocks. In the next subsections we discuss how we deal with deadlocks, consistency problems caused by processing of multiple events and call recovery.

Examples of simple 2-party and 3-party calls using our object infrastructure are illustrated below. We omit the mechanism by which telephony events are delivered to objects by modelling the process as a direct invocation performed by the subscriber’s telephone upon its Line object, where a Line object is the software representation of the telephone, holding line specific information such as whether incoming or outgoing calls are barred.

![Fig. 3 (a)](image1)

With reference to Fig. 3(a), when a phone goes offhook (1) an ‘offhook’ operation is invoked on the telephone’s Line object, L1, (2) and, if outgoing calls are allowed, L1 creates a Call object, C1, to handle the resulting call (3).

![Fig. 3 (b)](image2)

With reference to Fig. 3(b), digits which are entered (1) cause a digit operation to be invoked on L1 (2), which propagates the digit by invoking the digit operation on C1 (3).

![Fig. 3 (c)](image3)

With reference to Fig. 3(c), digits continue to be accepted until C1 ascertains that sufficient have been received and invokes the ‘translate’ operation on the Digit Translation object (1), the response (2) indicating the Line object, L2, associated with the called party. (Note that this call/response can be implemented as a synchronous
call). An 'incoming call' operation is invoked on L2 (3) and L2 causes the phone to ring (4).

Fig. 3 (d)

With reference to Fig. 3(d), whenever the called party answers (1) the 'answer' operation is invoked on L2 (2) which then invokes the same operation on C1 (3). C1 then connects the two parties in speech (by a mechanism not shown here).

Fig. 3 (e)

With reference to Fig. 3(e), the calling party might at some time within the speech phase press the 'recall' key (1) in order to initiate a 3-party call. This causes the operation 'recall' to be invoked on L1, which invokes the same operation on C1 (2). C1 connects the 'held' tone to the called party (not shown) and then creates a 3-Party object (3) and informs L1 that it has done so (4). The 3-Party object meanwhile creates another Call object, C2, (5) which will be responsible for the call set-up to the third party. As a result of this process the 3-Party object has been inserted into the call chain.

Fig. 3 (f)
With reference to Fig. 3(f), digits may now be entered (1) as in the 2-party case, except that in this situation they go via the 3-Party object, until sufficient digits are received by C2. The Digit Translation Table is then accessed (not shown) and the called phone is rung (2) as in the 2-party case. When the third party answers, they and the controlling party are connected in speech. Finally, when the controlling party presses recall, all 3 parties are connected in 3-party speech.

3.2. Activities and References

Objects involved within call processing activities require firewalls which prevent errors spreading, through interaction, to other objects. We describe later how object references can be used for this purpose. Some form of recovery is also required behind such firewalls to ensure that, having detected erroneous behaviour, some (hopefully small) set of objects may be identified as being responsible and recovery action taken upon those objects to return them a consistent state. We accomplish this by introducing the notion of an activity. An activity represents the computation carried out as a result of processing an external event. Processing of an external event starts a new activity and all the processing related to that event is executed as part of this activity. Activities are units of recovery and deadlock resolution.

An ideal way of making activities recoverable would be to make them transactional. Then, all activities will follow some protocol for concurrency control, such as the well-known two-phase locking protocol, to ensure interference free (serializable) execution: locks on objects are acquired as invocations are made and released only when the activity ends. Any erroneous situation would cause the relevant activity to be aborted. However, despite the desirability of making all activities transactional, in real-world telephony applications this may prove impractical. First, an activity can (and does) perform unrecoverable actions. Second, supporting serializable activities requires additional synchronisation between objects involved within the activity and this imposes communication overheads that may prove unacceptable for activities spanning multiple switching domains.

We propose a lightweight solution for constructing recoverable activities by permitting an activity’s computation to be a mixture of both transactional and non-transactional. In other words, objects may be transaction aware or transaction unaware.

As we mentioned earlier, telecommunications applications have to support a traditional data processing function in that large amount of shared persistent data (representing user data, hardware configurations, billing tariffs etc.) needs to be maintained; updates to this data must be carried out atomically. Hence, persistent objects would normally be transaction aware, whilst volatile objects would be transaction unaware. Naturally, only transaction aware objects will follow a concurrency control protocol to ensure serializability (e.g., two-phase locking), whereas other (volatile) objects will use only local concurrency control to ensure interference free execution of local object methods.

The recovery action undertaken for volatile objects involved in call processing e.g. C1, C2 and 3P in the previous diagrams, consists of simply deleting them. This is made possible through the novel use of object references to be described below. The action of deleting the objects of a call may be less drastic than it first appears as the state of a call in the speech phase (a high percentage of the calls in progress) may often be recovered by interrogation of the various hardware devices responsible for maintaining the speech path, and the user need not be aware that the recovery has occurred.
References provide the capability to communicate with a designated object, and in our model they may be freely held, copied and exchanged. References are created when an object is instantiated, being returned to the instantiator as a part of that process. Objects also obtain references by being passed them in messages from other objects, and this includes references returned as the results of invocations on objects acting as name services (e.g. traders). Communication by a particular object is therefore limited to those objects for which it holds references. Synchronous communication is a special case, in which the reference to the calling object is automatically passed in the request message (so that the results may be returned) but the called object may not retain the reference.

The underlying object support infrastructure is required to guarantee referential integrity, meaning that an object will continue to exist (will not be terminated) whilst any object retains a reference to it. This provides security against a faulty object wrongly attempting to terminate an object (the firewall referred to earlier); the object will continue to exist whilst any correctly functioning object retains a reference to it. The infrastructure must also provide distributed garbage collection of objects that is both safe and lively: objects with references are not garbage collected (safety) and objects without any reference are eventually garbage collected (liveness).

Fig. 4 shows the objects involved in 3 party speech and the references held by those objects. The references from each of the telephones to the Line objects are permanent (as illustrated by the earth symbols) and those objects will never be deleted. All persistent objects e.g. the Digit Translation object presented earlier, are permanently referenced and therefore cannot be incorrectly deleted by an erroneous object. Our model specifies that for every call the objects representing the communicating parties e.g. the Line objects in the example above, act as the roots of a directed, fully connected graph, i.e. every object in the call may be reached by following a path of references from any root. Note that the referencing graph shown above represents one example of the call segment chains mentioned previously.

As references provide the capability to access other objects then the controlled exchange of references prevents one call from interfering with another. Calls cannot erroneously delete objects which are correctly referenced, and although they do require access to certain persistent objects e.g. the Line and Digit Translation objects in the diagrams, updates to persistent objects may only occur transactionally and rigorous checks can be introduced as necessary in order to minimise the possibility of incorrect updates. Calls may therefore be implemented with an efficient firewall which prevents errors being propagated to other parts of the system.
Objects detecting inconsistencies within a transaction cause the transaction to abort (either by informing the transaction manager, by refusing to commit, or by raising exceptions which indirectly cause the abortion). A transaction manager might itself suspect an inconsistency and abort the transaction. Errors detected within some activity but outside of a transaction, for which no specific exception handler is available, cause the entire call to be terminated (this will also cause termination of any ongoing transactions within the activity). This is possible, as root objects have the ability to terminate a call by sending messages informing all root objects of the fact and then deleting all their references. Objects which hold a path to a root object may inform the root object and have them terminate the call. Unreferenced objects will automatically be garbage collected.

The treatment of deadlocks is straightforward given that activities are recoverable. One simple way of preventing deadlocks is to assign unique timestamps to activities and resolve lock conflicts uniformly in favour of say ‘older’ activities. All object invocations carry the identity of the activity. So if a lock conflict is detected, and the holder of the lock is ‘younger’ than the requestor, then the holding activity is aborted.

Currently available ORBs do not provide the necessary guarantees of referential integrity and garbage collection of unreferenced objects. A research object support system built by us [9] does provide these facilities. The techniques used there can be adapted easily for incorporation in an ORB.

3.3. Supporting Transaction Aware Objects

The Object Transaction Service (OTS) specification [2] describes the protocols and services necessary to enable distributed objects to become transaction aware. The transaction class library and related services of the distributed object system Arjuna, built by us [10], is in the process of being adapted to be OTS compliant and can be used to provide the necessary transactional support for persistent objects.

As stated before, all object invocations carry the identity of the activity. On receipt of an activity identity only a transaction aware object will register its participation in a transaction with some transaction manager. We illustrate our basic approach to transactional support with an imaginary call feature invoked from within 3-party speech. In the previous 3-party call illustration we described how Line objects hold state regarding some particular telephone e.g. whether incoming calls are barred or not. Let us imagine a call feature which, through the pressing of some sequence of keys, allows a party within a 3-party call to set a time at which all 3 phones will simultaneously ring and on answer the 3-party conference is re-instated. One implementation of such a feature has the 3-party object recognise the appropriate key sequence and then invoke operations (directly or indirectly) on the 3 Line objects informing them of the required time. Each Line object then holds the time (as persistent state which outlives the present call). This feature requires that the update of all the Line objects with the required time should occur atomically. Fig. 5 illustrates an implementation of the feature which makes use of transactions on persistent objects.

With reference to Fig. 5(a), in which all parties are in 3-party speech, the 3-Party object on receiving the appropriate key sequence (1) begins a transaction by creating a Transaction object (2).
With reference to Fig. 5(b), the 3-Party object now invokes synchronous 'update time' operations on all 3 Line objects, the identity of the Transaction object being passed with the message. In response each Line Object registers itself as belonging to the transaction (2) (via a synchronous 'register resource' invocation on the Transaction object) before returning a confirmation to the 3-Party object (3). (Only L1's response is shown in the diagram).

Fig. 5 (b)

Whenever all the Line objects have responded correctly the 3-Party object invokes 'commit transaction' on the Transaction object which then executes the commit protocol with the Line objects. Only when they receive the commit instruction will the Line objects commit the updated time to persistent store. A failure of any Line object to respond correctly to the 'update time' operation e.g. due to a communication failure or to a party having cleared down, causes the 3-Party object to invoke the 'rollback' on the Transaction object. The 3-Party object is informed whenever the commit or abort is completed and may then delete the Transaction object.

4. Concluding Remarks

We have presented a way of structuring telecommunications applications using CORBA technology. Our approach of structuring the software system as communicating objects allows better fault containment and provides localised recovery at the level of individual calls. For fault containment we require that objects involved within call processing activities be protected by firewalls which prevent errors
spreading through interaction to other objects. We achieve this by insisting that the underlying object support infrastructure provide referential integrity for objects. This provides security against a faulty object wrongly attempting to terminate an object. Although currently available ORBs do not provide the necessary guarantees of referential integrity, the techniques developed by us and described elsewhere [9] can be adapted in an ORB. To provide localised recovery at the level of individual calls, we have introduced the notion of an activity which represents the computation carried out as a result of processing an external event. A lightweight solution for constructing recoverable activities has been proposed by permitting an activity's computation to be a mixture of both transactional and non-transactional. We have described how Object Transaction Service can be used to provide the services necessary to enable distributed objects to become transaction aware. We are implementing key aspects of the model described here by developing a telecomms ORB. At the same time we are also adapting the transaction class library and related services of Arjuna [10] to be OTS compliant so that it can be used by this ORB in the manner described here.

Acknowledgements

The work of the University authors has been supported in part by a research grant from GPT Ltd. Discussions with G.D. Parrington clarified our ideas on Object Transaction Service (OTS) specification.

References


