Distributed Programming with Associative Broadcast

Bryan Bayerdorffer*
Department of Computer Sciences
The University of Texas at Austin
Austin, TX 78712

Abstract

We propose a new model for distributed programming, called Associative Broadcast, that achieves a high level of communication abstraction by using as its fundamental mode of communication a form of logical broadcast. Each message contains a predicate that specifies the local states of the objects that are to receive the message, rather than their identifiers. Message delivery is asynchronous and requires no knowledge of the global state. Associative Broadcast exploits the broadcasting capabilities of distributed systems, and enables straightforward specification of state-based communication. We use state-based communication to obtain a new solution to the partitioned-network consistency problem in distributed databases.
Keywords: distributed systems, broadcasting, naming mechanisms, communication abstraction.

1 Introduction

Many distributed systems support asynchronous message broadcast at a cost that is close to that of sending a message to a single recipient, e.g. using Ethernet [12], cellular radio, or satellite networks. Furthermore, there exists a large class of problems that give rise to concurrent algorithms in which messages must be delivered to multiple recipients, and in which the membership of this *target set* of a message is dynamic and determined by the local states of the potential recipients. Examples of domains in which such communications arise include distributed simulation (e.g. n-body simulations), distributed constraint satisfaction (parallel execution of logic programs [14]), and consistency protocols for replicated databases (recovery from network partition failures [5]). The communication primitives of most distributed programming models are not sufficiently abstract to allow distributed, symmetric solutions to these problems to be specified simply and concisely.

We propose a new model for distributed programming, called *Associative Broadcast* (because its operation is reminiscent of the content-addressability of an associative memory) in which broadcasting is the fundamental mode of communication. The model allows 1–N communication to be specified at a high level of abstraction, and enables programs to exploit the broadcasting capabilities of distributed hardware. Associative Broadcast is based on a scheme for naming the communicating objects in a computation (e.g. processes) in a way that allows the target set of a broadcast message to be defined in terms of the local states of the potential recipients.

Because the specification of the target set travels with each message, the sender need not know *a priori* the identities of the objects that will eventually receive the message. The cardinality of the set of objects that receive a given message is transparent to the sender, even when that set is empty. This enables straightforward specification of distributed, symmetric, and asynchronous algorithms that require communications in which the sender knows what the local states of the target objects should be, but does not know which potential recipients are in that state.

We describe the Associative Broadcast model in section 2, and summarize related work in section 3. In section 4, we use Associative Broadcast to specify a symmetric and fully distributed solution to the partitioned-network consistency problem in replicated databases. This solution is based upon Davidson's Optimistic Protocol [4]. Unlike the original protocol, it does not require transaction processing to be halted globally during the recovery phase, nor must transactions that are in progress at the time of recovery be aborted. Instead of collecting and analyzing transaction logs at a central controller, the detection of cycles in the transaction precedence graph is performed in place, using probe propagation. New vertices and edges may be added to the graph by ongoing trans-

*Author's current address: Computer Science Department, U.S. Naval Academy, Annapolis, MD.

actions while probes are in transit. Probe visitations form a consistent view of the distributed graph, in the spirit of Chandy and Lamport's Distributed Snapshots [3]. Since the graph is dynamic, and its edges are partitioned among many objects, it is not trivial for the sender of a given probe to determine which objects should receive the probe. By broadcasting a probe to a group of objects of indeterminate (including possibly zero) size that have the required attributes, we ensure that the probe is received where necessary without sending it to every object in the system. Additional applications of Associative Broadcast are described in [2].

2 Associative Broadcast

Associative Broadcast associates with each object a program-specific, mutable set of *attributes*, called a *profile*. All communication is carried out by executing the **broadcast** primitive, which includes in the outgoing message a sender-specified *selector*. A selector is a propositional expression over attributes, and specifies the profiles of those objects that are to receive the message. An attribute is either a *symbol* (a *simple* attribute) or a *(symbol, value)* pair (a *compound* attribute). An attribute may encode the *type* of an object (denoting operations it supports), an abstraction of the object's state, or it may be a unique identifier. The symbols may be treated as data values and passed among objects in messages.

The conventional notion of a static object or group name is subsumed by that of a selector. Selectors descriptively names dynamic sets of objects of arbitrary cardinality, while conventional names are nondescriptive identifiers that are often statically bound to objects. Matching of the selector against object profiles is performed locally and asynchronously as the message arrives at the sites where objects reside. Objects may be created on the fly. Each object controls its own profile and may add, remove, or alter the attributes in its profile at any time, although the matching of a selector against an individual profile must be atomic with respect to the modification of that profile. The only global knowledge[1] upon which communication depends is the program-specific meanings of the attribute symbols. This information is inherently static and known to all objects.

[1] We use "global knowledge" informally here to mean information that is known to at least two objects.

2.1 Elements

An object consists of a set of local data structures, a set of *operations* and a *profile*. The profile is a mutable set of *attributes*, and an attribute is either a singleton, s, or a pair, (s, v), where s is a *symbol* and v is a *value*. For each operation of the object, the profile contains a corresponding attribute, called a *type attribute*. By convention we assign identical names to an operation and its type attribute. An operation has the form $\texttt{name}(y_0, ..., y_n)\{\mathit{body}\}$, where the y_i are formal parameters and *body* is a program. The body may contain one or more instances of the following statements:

broadcast m
$\texttt{profile} := \mathit{pexpr}$

where m is a message, and *pexpr* is and expression that evaluates to a valid profile. The former invokes the Associative Broadcast primitive, and the latter modifies the object's profile.

A *message* is a tuple, *([selector], arg, ...)*, where *[selector]* is a propositional expression over attributes, and each *arg* is an actual parameter. A message is *well-formed* relative to a profile **iff** exactly one type attribute of the profile appears as a singleton conjunct of the selector, and the actual parameters of the message agree in type and number with the formal parameters of the operation indicated by the type attribute.

A *selector* is an expression $[t_0 \diamond \cdots t_k]$, where $\diamond$ is one of $\{\vee, \wedge\}$, and each t_i is a *term*, s or $s\phi v$, where ϕ is a binary relation on the set of attribute values. The term $s\phi v$ evaluates to *true* with respect to a profile, p, **iff** p contains an attribute, (s, x), and $x\phi v$ holds. The term s evaluates to *true* **iff** p contains the simple attribute s. For example, the profile

$$\{\texttt{INSERT}, \texttt{DELETE}, (\texttt{ITEMTYPE}, \mathit{blue}), \texttt{NOTFULL}\}$$

satisfies the selector

$$[(\texttt{ITEMTYPE} = \mathit{red} \vee \texttt{ITEMTYPE} = \mathit{blue}) \\ \wedge \texttt{INSERT} \wedge \texttt{NOTFULL}]$$

While the specification language for selectors is conceptually that of first-order logic, we have not found it necessary to make use of quantification in any of the problems to which Associative Broadcast has been applied. Restricting selectors to propositional expressions improves the efficiency of the matching operation. It is also desirable to disallow negation in selectors to guard against inadvertent communications. We wish to disallow selectors of the form $[\neg x]$, where x is a simple attribute that does not appear in any profile. Such a selector allows the sender to send a message to all objects in the system, even though it may not know any of the attributes appearing in the profiles of any given object. Note that we can always replace the negation $\neg x$ with $notx$, and include the attribute $notx$ in a profile whenever x is removed from the profile. Similarly, the term $\neg(s\phi v)$ may be replaced by $(s\phi' v)$, assuming every ϕ has an inverse ϕ'.

2.2 Computations

For the sake of simplicity, we adopt a single-threaded task model of execution for objects. A formal specification of object behavior and the profile/selector matching mechanism, omitted here to conserve space, may be found in [2]. An object begins executing an operation when it receives a message, and receives no subsequent messages until execution of that operation terminates. Each object has an initial state that is determined by initial values of its data structures and an initial profile. A *behavior* of an object is a sequence of events. Each event is one of *broadcast, arrival, receipt, profile*, or *internal*. *broadcast* occurs when an object executes the statement **broadcast** m, and *profile* occurs when an object modifies its profile by executing `profile` := *pexpr*.

The *broadcast* of m is followed in finite time by the *logical arrival*of m at every object[2] (including the sender), with events at distinct objects being partially ordered according to the usual notion of causality [11]. Let $m = ([r], d \ldots)$. The *arrival* of m at object o is followed immediately (i.e. without intervening events at o) by the *receipt* of m at o **iff** the profile of o satisfies r at the occurrence of the arrival event. Upon receiving message m, an object executes the operation specified by the type attribute appearing in the selector of m.

The *profile* event is atomic with respect to both *arrival* and *receipt*. Note that *arrival* and *receipt* are events that are initiated by the environment of an object, rather than the object itself. Thus an implementation is free to define the *arrival* (i.e. profile-matching) event as occurring when the object is ready to receive a message. Objects may modify their profiles while messages are in transit, and it is possible for such profile modification events to cause objects to join or leave the target set of a message in transit. In such cases the ordering of the respective *arrival* and *profile* events is nondeterministic: Receipt of the message is possible but not guaranteed.

[2] In an implementation, the physical arrival of a message at a site corresponds to the logical arrival of the message at all objects residing at that site.

We assume no ordering of messages whatsoever: The order in which messages arrive at objects is arbitrary, provided that causality is preserved. However, our model does not prevent an implementation from using appropriate protocols to guarantee global ordering of messages.

2.3 Associative Broadcast Characteristics

By providing a mechanism for *senders* to specify the set of objects that will receive a message and implementing this mechanism at a low level, the profile/selector naming scheme yields programs that are clearer and more concise, easier to analyze, and more efficient to execute than those using a more naive approach in which individual recipients simply filter messages from system-wide broadcasts.

It is much easier to specify the intended membership of a target set as a single formula in first-order logic at the originator of a broadcast than it is to repeat the specification at each potential recipient, which is the technique suggested by the BSP model in [8]. The code that an object executes to alter its profile is typically significantly simpler than any message-filtering code would be. Placing the specification logic in the domain of the sender also makes it easier to understand existing programs.

Associative Broadcast also offers advantages for implementation. The matching of selectors to profiles can be implemented once at a low level in the communication subsystem, rather than repeatedly at the object level as is the case for arbitrary message-filtering code. This is particularly important where several unrelated computations may coexist: A system in which each object has to be written to handle every possible message type is clearly non-modular.

Associative Broadcast extends to communication the principle of abstraction that is commonly applied to data types, synchronization, and control sequencing [6]. Associative Broadcast provides three forms of communication abstraction:

- *Description* allows the names bound to an object (the selectors that its profile satisfies) to encode information about the state of the object.
- *Grouping* allows such names to be concurrently bound to multiple objects.
- *Indirection* allows the set of objects to which a name is bound to vary.

A formal treatment of these properties appears in [1].

2.4 Implementation strategy

We assume that each object resides at one of a set of host *sites* that are interconnected by a communication network that supports asynchronous, unordered broadcast. Such networks include broadcast-bus (Ethernet), n-cube, ring, and shared-memory architectures. The implementation of Associative Broadcast is divided between the compiler, the *send* runtime system (send RTS), and the *receive* runtime system (receive RTS). The send RTS and receive RTS at each site form the interface between objects and the network hardware. They maintain the data structures that implement profiles, provide the *broadcast* primitive and the profile-modification mechanism, and perform selector/profile matching.

To allow the matching operation to be performed efficiently, the selectors should be converted to a normal form for which the matching algorithm can be optimized. This conversion may be performed by the compiler in cases where selectors are constant expressions, but may have to be left to the send RTS if selectors are generated by objects at runtime (i.e. if we allow variables of type `selector`). Selectors should be converted to conjunctive normal form expressions in which the conjuncts are arranged in order of increasing complexity of evaluation. That is, singleton conjuncts that refer to simple attributes should appear first. Since the matching of a selector against a particular profile terminates as soon as any conjunct is not satisfied, simpler conjuncts should be evaluated first to reduce the average time for failed matches. E.g.,

$$[(\mathtt{ITEMTYPE} = \mathit{red} \vee \mathtt{ITEMTYPE} = \mathit{blue}) \\ \wedge \mathtt{INSERT} \wedge \mathtt{NOTFULL}]$$

should be converted to

$$[\mathtt{INSERT} \wedge \mathtt{NOTFULL} \wedge \\ (\mathtt{ITEMTYPE} = \mathit{red} \vee \mathtt{ITEMTYPE} = \mathit{blue})]$$

The send RTS implements the **broadcast** primitive, possibly performs selector normalization, and initiates message transmission on the network. Most of the functionality of Associative Broadcast is contained in the receive RTS. The receive RTS at each site maintains the profiles of the local objects ("local profiles"), and implements the profile-modification mechanism. It matches the selectors of incoming messages, and delivers messages to objects upon successful matches. The matching operation is performed in two stages, called the *primary filter* and the *secondary filter*. This division is based upon the assumption that sites will receive many messages whose selectors are not satisfied by *any* local profiles. Such "sparse broadcasts" occur when the average size of a target set is small compared to the number of sites, or when objects are grouped so that those that are likely to need to communicate frequently are collocated to minimize network traffic.

The purpose of the primary filter is to quickly discard those messages that have no possibility of matching a local profile. This is done by maintaining a hash table of the *union* of the sets of attribute symbols appearing in the local profiles. When a message arrives at a site, the primary filter scans the selector, and attempts to find each attribute symbol appearing in the selector in the hash table. If none of the attribute symbols in the selector are found, then no local profiles satisfy the selector, and the message can be discarded. Furthermore, if any simple attributes that are singleton conjuncts in the selector are not hashed successfully, the message may be discarded as well.

Once a message has passed the primary filter, it must be matched against each local profile by the secondary filter. This operation also takes advantage of the selector's normal form, terminating as soon as an unsatisfiable term is encountered. Several profiles can be matched concurrently with associative or parallel hardware support.

2.5 Matching time complexity

A selector can be scanned by the primary filter in $O(1)$ time per attribute symbol appearing in the selector, and a hash-table lookup takes $O(1)$ time. Since the lookup of the first conjunct may cause the message to be discarded, the best-case time complexity for the primary filter is $O(1)$ (no profile at the site contains the attribute symbol of the first conjunct), and the worst case is $O(n)$ (each attribute symbol in the selector appears in at least one profile at the site), where n is the (typically small) number of attribute symbols appearing in the selector. An implementation of Associative Broadcast is likely to require that n be bounded. Average case times depend upon the distribution of profiles and selectors. Where the average size of a target set is small compared to the number of sites, the primary filters at most sites will immediately discard the message.

Similarly, a secondary-filter match may fail in $O(1)$ time, yielding a best-case complexity of $O(p)$ (all matches fail) and worst case of $O(pn)$ (all matches succeed), when there are p profiles that are matched sequentially.

Previous work on "packet filters"has shown that low-level per-object filtering is practical, and may in fact improve efficiency by reducing the number of context switches needed to process an incoming message [13]. Since packet filters consist of arbitrary user-supplied code, we expect that on average the matching of a selector to a single profile is considerably more efficient than the execution of a single packet filter. In the algorithms we have developed using Associative Broadcast, selectors contain a small, bounded number of attribute symbols. Finally, it is possible for a sophisticated compiler or send RTS to identify those conjuncts that have a low probability of being satisfied, and rearrange selectors accordingly to optimize the matching operation.

3 Related work

There are several models in the literature that share some of the properties of Associative Broadcast. Perhaps the closest in its ability to express communication at a high level of abstraction is Linda [9], in that it makes use of the concept of pattern matching. Linda differs from Associative Broadcast mainly in its communication semantics and its implementation requirements. The same Linda "message" may be received multiple times by the same process, if the process executed multiple *read*s with the same pattern. Another difference is that there is no way to specify in a pattern that an element of a matching tuple should satisfy some relation with a corresponding value in the pattern. Thus the Associative Broadcast naming system is slightly more expressive in this respect. Linda does not have the semantics of message passing, relying instead upon a logically-shared data structure (the *tuple space* (*TS*)). While distributed implementations of *TS* have been proposed, these require complex and expensive protocols to maintain the consistency of the tuple space. Linda is therefore better suited to a tightly-coupled multiprocessor with shared storage than to a distributed system. Associative Broadcast retains the expressiveness of Linda, but requires only unordered, asynchronous broadcast, efficiently obtainable on most distributed architectures.

Our concept of a target set is also related to that of a process group. However, a target set is an implicitly-defined grouping of objects that is an artifact of the logic used to specify selectors, while process groups are typically explicitly maintained. When an Associative Broadcast object alters its profile, it may be joining or leaving infinitely many target sets. Since a process joining or leaving a process group typically involves at least one operation per group, emulating target sets using process groups is potentially very expensive.

Gehani's paper on Broadcasting Sequential Processes (BSP) [8] contains persuasive arguments in favor of using broadcast primitives to structure concurrent computations. However, the BSP model allows only system-wide broadcasts, and leaves it to individual recipients to filter out unwanted messages. Similar message filtering is performed by the *selective accept* statement of Concurrent C [7], although that language has no built-in broadcast primitives.

4 Example

In a replicated database, there exist several copies of each data item, with individual copies residing at distinct sites throughout the system. Correct operation of the database requires all *transactions* (which read and update data items) to be executed in *serializable* order.

When the interconnection network fails in such a way as to prevent communication between some subsets of sites (a "partition failure"), then it may be impossible to both maintain serializability and allow all transactions to proceed while the network remains partitioned. This is because copies of a data item may reside in distinct partitions, and there is no way to bring these copies into agreement if a transaction originating in one partition attempts to write a new value to the data item.

Solutions to this problem take one of two forms: either transactions that attempt to update data items, some of whose copies are inaccessible, are prevented, thus maintaining consistency (pessimistic protocols), or such transactions are permitted, thus potentially creating inconsistencies that must later be detected and eliminated (optimistic protocols). In section 4.3 we present a distributed version of the well-known optimistic protocol of Davidson [4] that relies upon Associative Broadcast to eliminate the centralized control and global freezing of transactions of the original algorithm.

4.1 Original optimistic protocol

The Optimistic Protocol is based upon the idea of constructing a precedence graph that captures the partial temporal order of all the transactions in the system between the occurrence of a partition failure and the reconnection of the network. The database is consistent **iff** this precedence graph contains no cycles. The graph is constructed as follows: a log is maintained in each partition that records the order of reads and writes on the data items, and the transaction to which each operation belongs. The set of all transactions is the set of vertices of the precedence graph. An edge $T_{xi} \rightarrow T_{yj}$ indicates that transaction T_{xi} occurred before transaction T_{yj}, where x and y indicate the partition in which the transactions were executed. Such an edge exists **iff** one of the following holds:

- *Dependency edge*: T_{yj} read a data item that was previously written by T_{xi}, and $x = y$.
- *Precedence edge*: T_{xi} read a data item that was later written by T_{yj}, and $x = y$.
- *Interference edge*: T_{xi} read a data item that was written byT_{yj}, and $x \neq y$.

The first two types of edges relate transactions within a common partition, while the third captures a conflict between transactions that access copies of a data item in distinct partitions. Since the read transaction in one partition did not read the value written by the write transaction in the other partition, the only way that the database can possibly be consistent is to consider the read to have occurred before the write. It is assumed that set of data items written by a transaction is contained in the set of items read by the transaction. Therefore, a write-write conflict across partitions is manifested as a pair of interference edges. The existence of interference edges does not imply that the database is inconsistent. However, in order for the precedence graph to contain a cycle, there must exist at least two interference edges, since it is assumed that the commit protocol prevents cycles within a partition (figure 1).

4.2 Disadvantages

The original Optimistic Protocol is centralized, relying upon a *coordinator* to collect the transaction logs upon reconnection. This has several disadvantages:

- *Asymmetry*: A coordinator must be chosen to construct the graph. This may require an additional leader-election protocol, create communication bottlenecks, make the system less fault tolerant, and make the algorithm more difficult to specify.
- *Log transfer*: Either during the execution of transactions or at the time the algorithm is executed, log information for each data item must be transferred to the coordinator. This transfer may be expensive. If log information is kept only at the coordinator, it may be lost due to further network partitioning or failure of the coordinator.
- *Global freezing*: While the precedence graph is being constructed, no additional transactions may be executed, since these might add cycles that are not reflected in the logs. Transaction processing must therefore be frozen at the time network reconnection is detected, and persists while the log information is first collected and then analyzed.

4.3 Distributed optimistic protocol

The above disadvantages might be avoided by distributing the graph construction algorithm. Specifically, the problems of asymmetry and expensive log transfers would not be present if the log information were stored with the data items and evaluated in place. Furthermore, freezing transactions globally while the logs are evaluated runs counter to the basic optimistic assumption that inconsistencies are rare. A truly optimistic protocol would allow transactions to proceed concurrently, so that, if no inconsistencies are eventually discovered, transaction processing is never suspended.

We propose a new protocol based on the following observation: If each copy of a data item is maintained by a distinct object, and each such object logs only the transactions that access the object, then the set of dependency edges, as well as the set of precedence edges, is partitioned among the set of all objects. That is, each object is aware of some subset of these edges (figure 2). There is also sufficient information to determine the set of interference edges. We call the objects that maintain the individual copies of a particular data item *cohorts*, and the partition in which an object resides its *local partition*.

Cycles in the graph are discovered by passing tokens, or "probes," along the edges of the graph, and thus from object to object. If the graph contains a cycle, then at least one probe will reach some object twice. When a probe reaches a particular vertex at an object, it must be sent to all other objects that also contain that vertex, i.e. that were touched by the corresponding transaction. Since transactions may be in progress as the protocol executes, the set of objects that must receive the probe may be growing concurrently. We propagate probes using Associative Broadcast by adding a vertex (transaction ID) to the profile of an object when the object is touched by a transaction, and specifying the destination vertex in the selector of an outgoing probe message.

The protocol effectively operates on a set of mutually consistent "distributed snapshots" of the system state, even as the global state is changing, in a manner similar to the general algorithm described in [3].

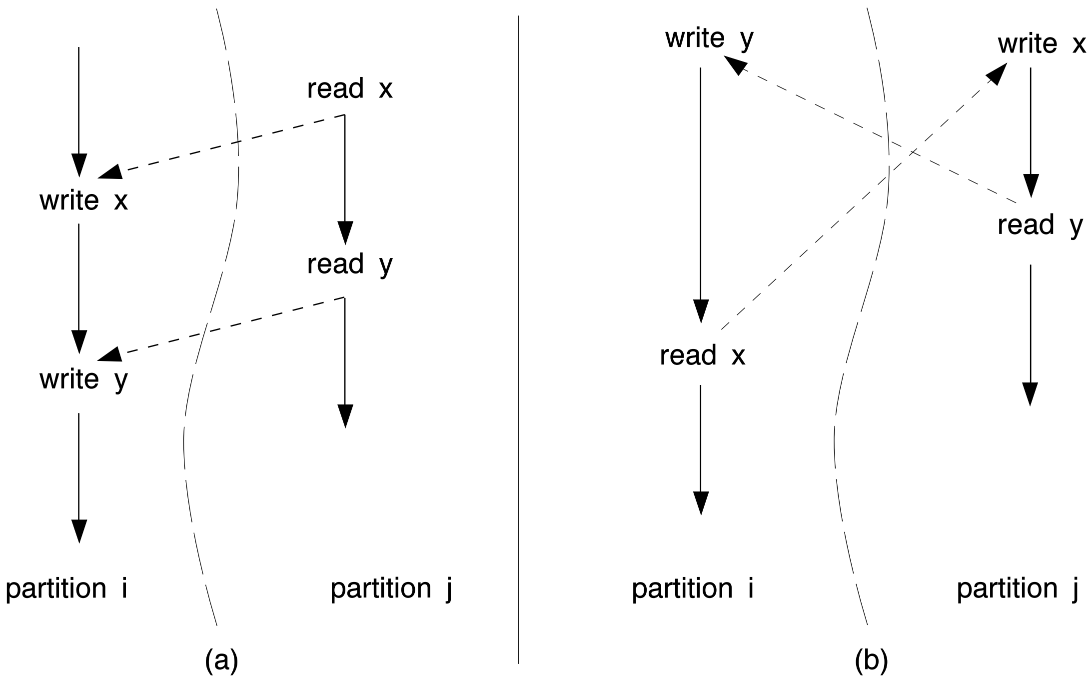

Figure 1: (a) Interference edges. (b) A cycle.

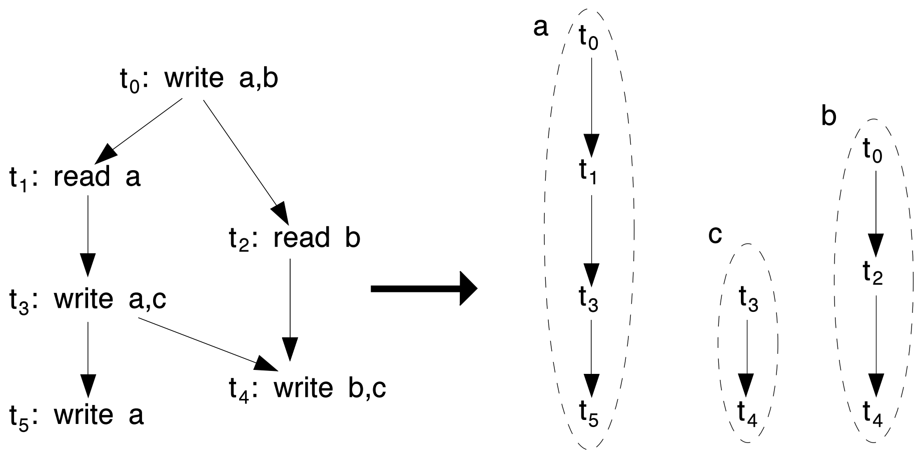

Figure 2: Example partitioning of edges among objects.

The global state examined by the protocol consists of the local states of the objects at the time each first receives a probe. This global state is consistent[3], but need not exist at any instant. It is this characteristic that allows transaction processing to proceed concurrently with construction of the precedence graph.

For the expository purposes of this example, we make the following assumptions about objects, transactions, and network partition failures (figure 3).

- Both partition failure and network reconnection are instantaneous events. Let I_f be the time at which a failure occurs, and I_r the time of the earliest reconnection, such that $I_f < I_r$. Then there exists no distinct failure at time I_x such that $I_f < I_x \leq I_r$.
- Let I_c be the earliest time at which the database is consistent, such that $I_r \leq I_c$. Then there exists no failure at time I_x such that $I_r < I_x < I_c$.
- The failure at time I_f is detected by each object in the system during the interval $I_f \ldots I_r$.
- The reconnection at time I_r is detected by each object in the system during the interval $I_r \ldots I_d$, where $I_d - I_r$ is positive and finite.
- Each object knows the total number of its cohorts.
- During the interval $I_f \ldots I_r$, all objects in a partition agree upon a globally unique *partition ID*, and each object determines the number of its cohorts in the local partition.
- Transactions and objects have globally unique IDs.

4.4 Object specification

The initial profile of each object is

$$\{(\texttt{ITEM}, i), (\texttt{READS}, \emptyset), \texttt{PROPAGATE}, \texttt{MERGE}\}$$

where i identifies the data item contained in the object, and `PROPAGATE` and `MERGE` are type attributes bound to the corresponding methods defined below. The value of `READS` is the set of IDs of transactions that have read or written[4] data item i, initially the empty set. The simple attribute `WRITTEN` is added to the profile when data item i is modified. The `PARTITION` attribute identifies the partition in which the object resided prior to reconnection.

[3] The notion of consistency referred to here is that defined in [3], and is distinct from database consistency.

[4] Recall that the readset of a transaction contains its writeset.

In the following code, the variables *mcount*, t_0, t_w, t_n, D, and P represent the number of `MERGE` messages received by the object, the first logged transaction, the first logged write transaction, the last logged transaction, the local copy of the object's data item, and the object's unique ID, respectively.

initially
    $mcount = 0$;
    $merged = \text{FALSE}$;

`Upon_partitioning`() {
    $x :=$ local_partition_id;
    $profile := profile \cup \{(\texttt{PARTITION}, x)\}$;
}

`Upon_committing_transaction`(t) {
    log t;
    $\texttt{READS} := \texttt{READS} \cup \{t\}$;
    if t modifies D **then**
        $profile := profile \cup \{\texttt{WRITTEN}\}$;
}

`Upon_detecting_reconnection`() {
    $m := (\texttt{WRITTEN} \in profile)$
    broadcast($[\texttt{MERGE} \wedge \texttt{ITEM} = i$
    $\wedge \texttt{PARTITION} \neq x]$, D, m);
    foreach $t \in \texttt{READS}$ {
        label t with P;
        broadcast($[\texttt{PROPAGATE} \wedge \texttt{ITEM} \neq i$
        $\wedge \texttt{PARTITION} = x \wedge t \in \texttt{READS}], t, P$); }
    if $(\texttt{READS} \neq \emptyset \wedge \neg merged)$ **then** {
        broadcast($[\texttt{PROPAGATE} \wedge \texttt{ITEM} = i$
        $\wedge \texttt{PARTITION} \neq x \wedge \texttt{WRITTEN}], t_n, P$); }
    if $(\neg merged)$ **then**
        delay subsequent transactions;
}

`propagate`(r,p) {
    if ($\exists\ u \in \texttt{READS}$ labeled with $p \wedge (r \rightarrow u \vee r = u$
        $\vee (r \notin \texttt{READS} \wedge (t_w \rightarrow u \vee u = t_w))))$ **then**
        declare inconsistency;
    else if ($\nexists\ u \in \texttt{READS}$ labeled with p) **then** {
        foreach $t' :: t' \in \texttt{READS} \wedge (r \rightarrow t' \vee t' = r$
            $\vee (r \notin \texttt{READS} \wedge (t' = t_w \vee t_w \rightarrow t')))$ {
            label t' with i;
            broadcast($[\texttt{PROPAGATE} \wedge \texttt{ITEM} \neq i$
            $\wedge \texttt{PARTITION} = x \wedge t' \in \texttt{READS}], t', p$); }
        if $(\neg merged)$ **then** {
            broadcast($[\texttt{PROPAGATE} \wedge \texttt{ITEM} = i$
            $\wedge \texttt{PARTITION} \neq x \wedge \texttt{WRITTEN}], t_n, p$);
}}}

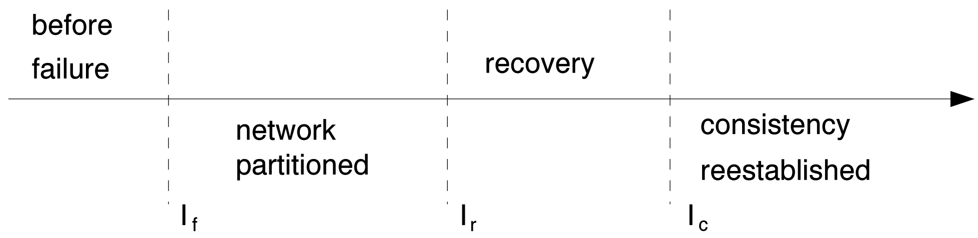

Figure 3: Time intervals bounded by partition failure, reconnection, and termination of recovery.

```
merge(val, mod) {
    if (mod) then
        if (WRITTEN ∈ profile ∨
            currently locked for write) then
            declare inconsistency;
        else D := val;
    mcount := mcount + 1;
    if (mcount = total cohorts −
        cohorts in local partition) then {
        ensure all cohorts participate in
        subsequent transactions;
        allow delayed transactions to proceed;
        merged := TRUE;
}
```

The computation proceeds as follows. When an object is notified that a partition failure has occurred, it adds an attribute to its profile whose value identifies the partition in which it resides. When the object subsequently commits a transaction, it logs the transaction and adds the transaction ID to its `READS` attribute. If the transaction modifies the data item, the attribute `WRITTEN` is added to the object's profile.

When an object detects that the network has been reconnected, it broadcasts the value of its data item copy to all other copies in partitions other than its own, and begins an instance of the graph construction algorithm. For each local transaction, the object labels the transaction with the unique ID of the object, and traverses the precedence edges reachable from the transaction as follows: It broadcasts the transaction ID and the unique identifier of the object (the probe). This probe is received at each object in the local partition that contains an edge originating at that transaction. The traversal of interference edges is effected by broadcasting a probe to all objects that contain copies of the same data item as the sender, reside in partitions that differ from that of the sender, and whose copies have been modified.

Subsequent transactions involving the object's copy of the data item are delayed until the object has received a `MERGE` message from all other copies of its data item that reside in partitions other than its own. This is necessary to prevent subsequent transactions from creating cycles. Such cycles might not be detected, since the object might not receive any more probes.

Upon receiving a probe message, an object declares a database inconsistency if it has propagated the probe before, and the transaction at which the probe enters precedes or equals that at which it previously exited. The declaration of an inconsistency causes all transaction processing to stop and a separate recovery algorithm to be run. Otherwise, if the probe has not been propagated before, the object labels all reachable local transactions with the probe, and propagates it along all those edges reachable from the vertex (transaction ID) at which the probe entered, as above.

Receiving a `MERGE` message causes the object to declare an inconsistency if both its copy and the received copy of the data item have been modified. It is necessary to consider a write lock as a modification, because once the transaction commits, an inconsistency will exist[5]. If the local copy has not been modified and the received copy has been, then the local copy is updated to reflect the received value. When the required number of *merge* messages have been received, any delayed transactions, as well as subsequent transactions, are allowed to proceed. We say that a set of cohorts are *merged* when the variable *merged* has the value TRUE at each cohort. Since all cohorts now maintain identical copies and participate in subsequent transactions, these transactions do not create any new interference

[5] This is because the transaction may have already read the local copy of the data item at the time the `MERGE` message arrives. An optimization would be to merely abort the transaction instead of declaring an inconsistency.

edges between the cohorts.

In the absence of inconsistencies, the database is eventually restored to a state in which all cohorts corresponding to a given data item maintain identical copies. Since the database is consistent, there exists at most one modified copy of each data item. The unmodified copies are updated upon receipt of the `MERGE` message broadcast by the modified copy.

Unlike a centralized implementation of the Optimistic Protocol, our distributed protocol allows transaction processing to continue during the interval $I_r \ldots I_c$, so long as no inconsistencies are detected. In particular, transactions that are in progress at time I_r are not aborted, as they necessarily would be if the protocol were not distributed. In addition, because transaction logs are evaluated in place, and cycle detection runs concurrently with transaction processing, the overhead of log transfer is avoided, and transaction processing is never globally frozen.

5 Conclusions

Associative Broadcast is a concurrent programming model that allows distributed, symmetric and asynchronous solutions to problems requiring dynamic patterns of communication. In those cases where most programming effort is usually expended in specifying communication, Associative Broadcast significantly simplifies the programming task by making the specification of communication straightforward. It offers opportunities for efficient implementation of those algorithms that would normally require a complex object-level protocol, by pushing the functionality of such protocols down into the communication subsystem. Finally, Associative Broadcast simplifies the analysis of programs by using the familiar propositional calculus to specify target sets.

We have used Associative Broadcast to obtain solutions to complex problems from a range of domains, notably concurrent constraint satisfaction, and the specification of a new synchronization protocol based on virtual time [10]. An implementation of an Associative Broadcast prototype on an n-cube network is underway.

References

[1] B. Bayerdorffer, An analytical taxonomy of naming systems, *Univ. of Texas Dept. of CS Tech. Report TR-92-48*, Dec. 1992, submitted to *ACM Trans. on Comp. Sys.*

[2] B. Bayerdorffer, Associative Broadcast and the communication sematics of naming in concurrent systems, *PhD. dissertation, Dept. of Computer Sciences, Univ. of Texas at Austin*, Dec. 1993

[3] K. Chandy and L. Lamport, Distributed snapshots: Determining global states of distributed systems, *ACM Trans. on Comp. Sys.*, vol. 3, no. 1, Feb. 1985

[4] S. Davidson, Optimism and consistency in partitioned database systems, *ACM Trans. on Database Systems*, vol. 9, no. 3, Sept. 1984

[5] S. Davidson and H. Garcia-Molina, Consistency in partitioned networks, *ACM Computing Surveys*, vol. 17, no. 3, Sept. 1985

[6] N. Francez and B. Hailpern, Script: a communication abstraction mechanism, *Proc. ACM Symp. Principles Distributed Computing*, Aug. 1983

[7] N. Gehani and W. Roome, Concurrent C, *Software Pract. Exper.*, vol. 16, no. 9, Sep. 1986

[8] N. Gehani, Broadcasting sequential processes (BSP), *IEEE Trans. Software Eng.*, vol. SE-10, no. 4, July 1984

[9] D. Gelernter, Generative communication in linda, *ACM Trans. Prog. Languages and Systems*, vol. 7, no. 1, Jan. 1985

[10] D. Jefferson, Virtual time, *ACM Trans. Prog. Languages and Systems*, vol. 7, no. 3, July 1985

[11] L. Lamport, Time, clocks, and the ordering of events in a distributed system, *CACM*, vol. 21, no. 7, July 1978

[12] R. Metcalfe and D. Boggs, Ethernet: distributed packet switching for local computer networks, *Communications of the ACM*, vol. 19, no. 7, July 1976

[13] J. Mogul, et al., The packet filter: an efficient mechanism for user-level network code, *Proc. 11th ACM Symp. Operating Systems Principles*, Nov. 1987

[14] A Takeuchi and K. Furukawa, Parallel logic programming languages, *Proc. 3rd Int. Conf. Logic Prog.*, Springer-Verlag, Berlin, pp. 242–254, 1986