RC23354 (W0410-011) October 1, 2004
Computer Science

IBM Research Report

Topology Changes in a Reliable Publish/Subscribe System

Sumeer Bhola
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Topology Changes in a Reliable Publish/Subscribe System

Sumeer Bhola
sbhola@us.ibm.com
IBM T.J. Watson Research Center

Abstract

Publish/Subscribe Middleware is an important infrastructure component in many enterprises, and must support both best-effort and reliable delivery of messages. For availability and scalability, such middleware should be deployed as a redundant overlay network of routing brokers. This paper examines the problem of changing the topology of such a network, without stopping delivery of messages, and without degrading the reliability guarantees.

Unlike the vast amount of recent research on self-organizing overlay networks, the focus of our work is on managed networks, which is how messaging middleware is typically deployed. A major drawback of self-organizing overlays is that they can only support reliable delivery protocols that are purely end-to-end. Such protocols do not work for content-based routing since intermediate brokers can filter messages not needed by downstream subscribers, and perform aggregation of acknowledgements.

We develop a formal and practical system model, which separates this problem into multiple layers: Administrators, the Topology Change Subsystem (TCS), the Topology Database and routing brokers. We describe topology change programs executed by the TCS on behalf of administrators, prove their correctness, and discuss how ordering between programs can be relaxed without impacting the predictability of the final topology. Our solution only requires eventual convergence of topology state cached by running brokers.

1 Introduction

Content-based publish/subscribe messaging (pub/sub) is a popular paradigm for building asynchronous distributed applications [8, 3, 1, 7]. The pub/sub system is responsible for routing published events to the set of interested subscribers. A scalable system is typically implemented using an overlay network of application-level routers, which we refer to as brokers.

The pub/sub model allows the publishers and subscribers to request reliability guarantees, such as exactly-once delivery (a best-effort service offers atmost-once delivery). Reliability guarantees need to be system supported due to (1) decoupling of publishers and subscribers, and (2) content-based filtering which prevents subscribers from using gaps in message sequence numbers to detect message loss. The popular Java Message Service (JMS) API for pub/sub supports exactly-once delivery, which we will refer to as reliable delivery.

The focus of this paper is on topology changes in a reliable, scalable pub/sub system. The problem is different and challenging, compared to classic one-to-many reliable delivery protocols built using a best-effort delivery service (like IP multicast). This is due to multiple factors such as (1) the *high-level subscription model*, where subscribers provide a subscription filter and do not care where the sources are located and what trees are used by the sources, (2) for efficiency, pub/sub systems *filter messages* and *aggregate acknowledgments* at intermediate brokers in the overlay network, and therefore ensuring reliability requires enforcing some consistency on the topology state across brokers.

Unlike the vast amount of recent research on *self-organizing overlay* networks for multicast, like Overcast [6] and SCRIBE [11], the focus of our work is on *managed overlay* networks, which is how messaging middleware has typically been deployed in commercial settings. Self-organization can cause inconsistencies in topology state across brokers that conflict with the requirements of content-based routing and reliability. For instance, consider the tree on the left in figure 1, and say C0 is publishing messages. If C2 believes C5 is not a child of C2, say due to a transient failure, it can update the filter on the edge (C1,C2) to filter out messages needed by C5. After C5 reconnects to C2, it will never know it missed some messages. Similarly, if C5 moves from parent C2 to parent C1, it is possible that there is a period of time when C5 is not connected to either, and both C2 and C1 acknowledge messages that have not been delivered to C5.

In a managed network, one or more administrators control the topology of the system. Such networks can scale to a wide-area environment and over a thousand nodes. The hard problem in such environments is the process of enacting a topology change. Instead of requiring administrators to do this manually, the administrators should only provide information regarding what changes need to be made, and the system should enact the change. More concretely, we consider a layered model for topology management and change. The top layer contains administrative entities, which may be human or automated agents, and are referred to as *administrators*. The next layer is the *Topology Change Subsystem* (TCS) which provides program templates to the administrators, and executes parameterized programs on behalf of the administrators. The TCS programs in turn perform updates to the Topology Database (TD), *and* instruct certain brokers to read the latest TD state.

This model separates the topology state of the system, into 2 parts, (1) the Topology Database (TD) which is persistent, and (2) the topology state in a running broker, which is mostly non-persistent. Such 'caching' of the state at running brokers is important for efficiency. Brokers read the part of the latest TD state that is relevant to them, either when recovering from a failure or when instructed by the TCS programs (referred to as programs in the rest of the paper). This read state is remembered and used till the next time the broker reads the TD state. A program that has instructed a broker to read the TD state has to wait till the broker *acknowledges* or is known to be currently in a crashed state. In a purely asynchronous system it is not possible to distinguish between a crashed and slow broker, but we make the practical assumption that the program can eventually distinguish between the two. However this can increase the execution time of a program.

The TD may be partitioned for scalability, and the partitions may be replicated. Replica consistency can be maintained using standard techniques like read/write quorums. We do not prescribe a specific implementation for the TD in this paper.

It is an open problem whether self-organizing networks can provide reliability guarantees. Our work could be seen as a first step in this direction by giving a solution for managed networks that does not require strong consistency in topology state across brokers, and permits concurrency in bringing about different topology changes.

Next, we discuss the overlay network model we consider in this paper.

1.1 Redundant Overlay Network Overview

The overlay network is a graph of *cells*, where each cell is a (usually) fully connected collection of brokers. Over this graph, one or more trees may be defined. Trees may overlap, so the same cell can be a member of multiple trees. There are 2 kinds of cells (1) *forwarding* cells (*f-cells*), which route messages on behalf of endpoints, and (2) *client* cells (*c-cells*), which contain brokers to which clients can connect. Brokers may do both forwarding and hosting of clients, so a broker can be in multiple cells. Figure 1 shows a tree with 6 cells, where C0, C3, C4 and C5 are c-cells, and C1 and C2 are f-cells. Broker b1 is in 2 cells, C1 and C3. An edge between cells consists of one or more inter-broker edges. Figure 1 shows that there are 3 inter-broker edges in the edge (C1,C2), i.e., (b1,b2), (b1,b3), (b2,b3). There is also an implicit edge since b2 is in both cells C1 and C2.

The brokers form an asynchronous distributed system in which messages between brokers can be delayed or lost, and brokers can crash and restart. Note that redundancy in this topology exists at 2 levels, the cells on the path between cells Ci and Cj can be different on different trees, and the path between Ci and Cj on a particular tree can consist of multiple inter-broker paths.

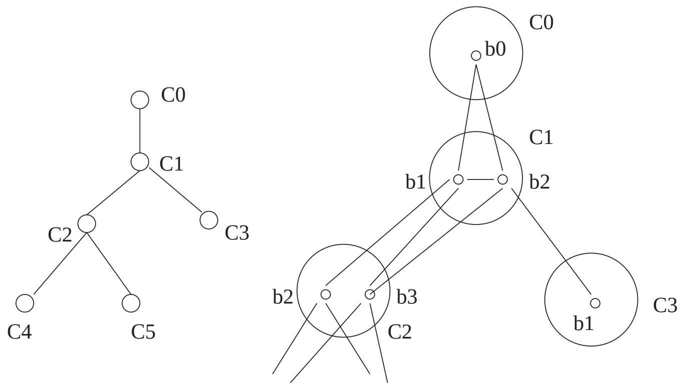

Figure 1. An Example Routing Tree

Forwarding Cells A forwarding cell can have one or more brokers. Multiple brokers in a f-cell increases availability and allows the routing load to be split among the available brokers. Brokers in a f-cell can route messages without having to communicate with other brokers in the same cell. Strong state consistency techniques such as read/write quorums are not used in a f-cell. Therefore a f-cell with n brokers can continue routing messages even if only 1 broker is up.

Client Cells Each client cell has exactly one broker. Availability of c-cells could be increased with multiple brokers and using techniques like primary-backup replication. Our topology change solution can be easily generalized to handle that, but we do not discuss it in this paper. Some c-cells may not host publishers since that requires substantial persistent storage for storing messages.

1.2 Topology Changes

The kinds of topology changes we are interested in are (1) adding/removing brokers from a cell, (2) creating/deleting cells, (3) creating/deleting trees, (4) mutating the topology of a tree being used by a c-cell for messages published at that c-cell, or switching the c-cell to use a different tree.

Changes to topology should not compromise the *safety* of the reliable delivery protocol. Also, we need to ensure that the trees used by different c-cells for published messages share the same set of c-cells. This is necessary since all subscribers, regardless of their c-cell, should receive all messages that are relevant to their subscription. This requirement, along with allowing concurrency of programs, makes the problem challenging.

We present our work in the context of a general stream protocol for reliable delivery [2]. This general protocol can be used to generate a suite of different protocols with different liveness and message caching approaches. The stream protocol assumed a fixed topology, so we also describe extensions to this protocol for handling topology changes.

We have prototyped our solution in the Gryphon system. Our current prototype uses a centralized Topology Database.

There are two primary contributions of this paper.

- We develop a system model for topology changes, and define correctness criteria in this model. We describe programs to bring about topology changes and prove they meet the correctness criteria.

  These programs do not require atomic change in topology state across multiple brokers in the system, only eventual convergence, and do not require the system to stop routing messages while changing the topology.

- We describe which programs need to be ordered, for correctness and predictability of the final result. For two programs P1 and P2 that need to be ordered in a manner $P1 \rightarrow P2$, we consider two ways of ordering them,

(1) by starting the execution of P2 after P1 has terminated, or (2) by utilizing the monotonicity of their behavior and input parameters. The latter allows program execution to be commutative, which increases concurrency.

We also discuss an important special case of $P1 \rightarrow P2$, in which P2 masks all the changes made by P1. This can be used to handle premature termination of P1.

Section 2 describes the system model for a fixed topology. Section 3 discusses topology changes, including TCS programs and broker behavior. Section 4 discusses related work and we conclude in section 5. The appendices contain proofs and some additional details.

2 System Model - Fixed Topology

In this section we describe the topology database and the broker state for a fixed topology. We also describe a general stream model for reliable delivery that is used later to prove the correctness of the topology change solution.

2.1 Topology Database (TD)

Each routing-tree has a unique name, and routing-tree names are totally ordered using a known function, such as lexicographic ordering. Cells, and cell membership, are independent of a particular routing-tree, i.e., the same cell can be in multiple routing-trees. A broker can be in zero or more forwarding cells, but at most one client cell.

Formally, a routing-tree in the system, with name T, has a set of vertices $v(T)$, that are cells, and a set of undirected edges, $e(T)$. The sets $v(T)$ and $e(T)$ have two special states, *undefined*, representing a tree that has not yet been created, and *final*, representing a tree that has been deleted. Both *undefined* and *final* sets have cardinality 0, i.e., they are *empty*.

Definition 1 *A routing-tree, T, is* well-formed *iff all the following conditions are true: (1) $v(T)$ and $e(T)$ are not empty, (2) the edges $e(T)$ on the vertices $v(T)$ form a tree, (3) all vertices in $v(T)$ that are leaves of the tree (i.e., they only have one edge), are c-cells, (4) each c-cell in the tree has exactly one broker in it.*

Since our solution preserves the well-formed property of routing-trees, we use the term tree and routing-tree interchangeably from now on. Note that definition 1 did not place any condition on the number of brokers in an f-cell. This is for two reasons: (1) we want to give administrators flexibility in ordering programs that change f-cell membership, and (2) a f-cell with no brokers does not compromise safety of the reliable delivery protocol.

A system topology is a tuple <*fcellSet, ccellSet, treeSet*>, where:

- *fcellSet* is a table keyed by f-cell name, C, and the column *brokerSet(C)* specifies the set of brokers that are members of C. The table contains a row for all possible f-cell names that can exist in the system. In appendix C we discuss concise encoding of such tables.
- *ccellSet* is a table keyed by c-cell name[1], C, and has 2 columns, *brokerSet(C)*, and *currentTree(C)*. The *currentTree(C)* is the tree name used by c-cell C to route messages that originate here (i.e. published by clients connected to C). It can be empty, meaning that there is no tree to route messages that originate here. The brokerSet(C) can have 2 special states, undefined, and final, as described earlier.

  The names of c-cells are totally ordered using a known function. The table contains a row for all possible c-cell names.
- *treeSet* is a table keyed by tree name T, and has columns *v(T)* and *e(T)*, as defined earlier. The table contains a tuple for every possible tree name. For notational convenience, we use *c-cell(T)* and *f-cell(T)* to refer to the subsets of *v(T)* that are c-cells and f-cells respectively.

[1]Assume that c-cell and f-cell name spaces are disjoint

Let *activeTreeSet* be defined as the currentTree values of all c-cells that have cardinality 1, i.e.,

$$\cup_{C \in ccellSet \,\wedge\, |brokerSet(C)|=1} currentTree(C)$$

Definition 2 *A system is* well-formed *iff all the following conditions are true:*

1. *Any non-empty c-cell C is also a member of* currentTree(C)*, i.e.,* $\forall C \in ccellSet : |brokerSet(C)| = 1 \Rightarrow currentTree(C) \neq empty \wedge C \in v(currentTree(C))$.
2. *A broker is not in two c-cells. Formally,* $\forall$ *c-cell C1, C2 :* $brokerSet(C1) \cap brokerSet(C2)$*=*$empty$*.*
3. *All trees in* $activeTreeSet$ *are well-formed.*
4. $\forall T1, T2 \in activeTreeSet$ *: c-cell(T1)=c-cell(T2). Let this set of c-cells be labelled* $activeCCellSet$*.*

Lemma 1 *In a well-formed system, for all c-cells C,* $C \in activeCCellSet \Leftrightarrow |brokerSet(C)| = 1$

All proofs are in Appendix A.

2.2 Broker Topology State

Each broker has a system-wide unique name, that also identifies the machine it will run on. This name to machine mapping does not change.

The topology information that a broker needs is divided into 2 categories, (1) *authoritative* information, retrieved from the Topology Database (TD), and (2) *discovered* information, which may be retrieved from the TD or disseminated through message exchange among brokers. In the absence of any topology changes, authoritative information is retrieved from the TD only at broker startup.

The authoritative information retrieved by broker B is:

- The set of c-cell names of which it is a member, $\{C \in ccellSet : B \in brokerSet(C)\}$, and for each element C in this set, currentTree(C). Note that the cardinality of this set is ≤ 1, for a well-formed system. Similarly, the set of f-cell names of which it is a member. Let in_cell_set be all the cells of which it is a member.
- A set of tree names defined in the system that are relevant to B, tree_set $= \{\ T \in treeSet : in\_cell\_set \cap v(T) \neq empty\ \}$.
- For each tree T in tree_set, and each cell C in in_cell_set, the set of all edges containing C in e(T).

Note that authoritative information does not contain *global* knowledge, such as the currentTree values for all other c-cells in the system. This is important for scalability. The discovered information includes other brokers that are members of the same cell, brokers in adjacent cells etc.

2.3 Reliable Delivery Protocol

This section describes a general stream model [2] that can be used to generate a suite of reliable delivery protocols. It is used for proving correctness of our topology change solution. The protocol operates independently for each c-cell that is publishing messages, say c-cell C0.

Let broker B0 be the member of c-cell C0. Broker B0 maintains a publishing endpoint (*Pubend*), to which messages published at this broker are persistently logged. The pubend contains a persistent knowledge stream, called *client input stream* or *c-istream,* which conceptually contains knowledge for each tick in time. The c-istream is the root of a tree of streams, with the leaf streams at c-cells.

For each incoming edge on a tree into a cell, a broker in that cell maintains a *forwarding-istream* (f-istream), and for each outgoing edge, it maintains a *forwarding output stream* (f-ostream). A broker in a c-cell also maintains an *client-ostream* (c-ostream) for delivering messages to subscribers. All streams other than the c-istream are non-persistent. For example, let the tree in figure 1 be named T0, and broker b0 have a pubend named p. Then, broker b0 has an istream, c-istream(p,C0), and two ostreams: f-ostream(p,T0,C0,C1) and c-ostream(p,C0). A c-ostream or c-istream is specific to the pubend and c-cell, and independent of the tree. In contrast, a f-istream or f-ostream is specific to the pubend, tree and edge of that tree. Similarly, broker b1 has two f-istreams, since it could receive a message from pubend p on the edge (C0,C1) or (C1,C3).

A tick in a stream can be in one of three states: Q, F and D. A D tick represents a data message, and a Q (question) tick represents lack of any knowledge. An F (final) tick represents one of 3 things: (1) there is never a data message at this tick at the pubend (c-istream), (2) there is a data message but was filtered enroute to this stream, (3) there is a data message but all downstream c-cells that are interested in it have received and acknowledged it. In general, ticks start in the Q state and either go directly to F state or go to D and then to F. At the c-istream, the only Q ticks are the ones representing future time. F and D ticks flow downstream from the c-istream. A D tick in an istream is transformed into an F tick in the ostream if the filter on the outgoing edge is not interested in that tick. The filter on an outgoing edge is provided by the Subscription Propagation Subsystem (section 3.1.2). For instance, a D tick in c-istream(p,C0) can change to F in ostream(p,T0,C0,C1) if the filter on edge (C0,C1) is not interested in this message. Acknowledgement (ack) messages flowing upstream change Q and D ticks in an ostream into F state. For an istream, when the corresponding tick in all its ostreams is F, the istream tick also changes state to F.

A broker does not know apriori on which edge it will receive a message from p. Therefore f-istreams and their corresponding f-ostreams are created on-demand, and all ticks are initialized to Q. Any protocol message for pubend p is marked with the tree on which it is flowing and the edge of the tree it is traversing. Since we are considering a fixed topology for now, this tree will be currentTree(C0). The pubend only sends and receives messages that are on currentTree(C0). Since messages contain the edge information, they also unambiguously define which f-istream or f-ostream they are meant for (f-istream for downstream messages and f-ostream for upstream messages).

At any c-cell C, the c-ostream for pubend p maintains a tuple in persistent storage *(p, latestDelivered(p))*. The c-ostream will resume from *latestDelivered(p)* when it crashes and recovers, i.e., it will set all ticks up to latestDelivered(p) to F, and all other ticks to Q. The first time the broker in c-cell C sees a message from p, it initializes latestDelivered(p) to 0.

2.3.1 Safety Criterion

To simplify our discussion of the subscription propagation subsystem, which is not the subject of this paper, and to simplify the statement of the safety theorem, we assume the following: a broker B in c-cell C knows apriori all the subscribers that it will serve, and these subscribers are created when broker B first starts up in c-cell C. Let the union of the filters for all subscribers at broker B in c-cell C be defined as $filter(C)$. This simplification does not affect the validity of the topology change solution, which is valid in the general case.

Consider a pubend p, hosted at c-cell C0, and any other c-cell C.

Definition 3 *The reliable delivery protocol is* safe *if for any F tick in c-ostream(p,C) that is greater than latestDelivered(p), the tick was either never D at the pubend, or was D at the pubend but the D tick does not match filter(C).*

It is easy to see that the protocol is safe when the topology is fixed and the filter on the path from C0 to C is always *wider* than filter(C). Informally, a filter $f1$ is wider than $f2$ if all messages matching $f2$ also match $f1$.

3 System Model - Topology Changes

In section 3.1 we describe how a broker's internal topology state is changed. Section 3.2 describes how TCS programs bring about changes in the TD and broker topology state, and section 3.3 deals with ordering and masking of programs.

3.1 Changes to Broker Topology State

A broker may process multiple messages concurrently. However, a change to internal topology state is not concurrent with message processing. The broker is said to have *applied* a topology retrieved from the TD when all subsequent messages will see this topology state until the next change is applied.

A broker B retrieves the current authoritative topology state from the TD when it starts up and in two other cases:

- It is instructed to do so by a TCS program. The broker acknowledges to the TCS program once it has applied the state.
- Broker B has no knowledge of the existence of tree T, and receives a message on tree T. The motivation for this is to simplify the TCS program that creates a tree.

A broker that is down is considered to have applied the latest TD state since it will read the latest TD state at startup, before processing any messages.

We now discuss how the broker behaves when it learns about topology changes.

3.1.1 Change to f-cell membership or edge

As discussed in the previous section, f-istreams and f-ostreams contain no persistent state and are created on demand. When an f-istream is created for a particular p, T, and edge, all corresponding f-ostreams are also created (if they did not already exist).

When the f-cell membership of a broker changes, or an edge between cells is added or removed on a tree, some of the current f-istreams, f-ostreams may no longer be valid. These invalid streams are deleted. If necessary, new f-ostreams are created for existing f-istreams, for instance if a new edge has been added to a tree.

All subsequent messages received on edges that are either (1) no longer on the tree, (2) or this broker does not implement the f-cell on the incoming edge, or (3) the tree no longer exists, are discarded without processing.

3.1.2 Change to c-cell membership

For any c-cell C, brokerSet(C) is allowed to have only two state changes, from *undefined* to having a broker in it, and then to *final*. In the previous section we had mentioned that a broker B in c-cell C keeps a persistent value *latestDelivered(p)*, for pubend p. There is some additional persistent state needed to handle topology changes:

- Broker B persists the c-cell it is in. We call this state *c-cell(B)*, which is null if B is not in any c-cell.
- For each pubend p it is delivering messages from, it maintains two persistent values, *latestDelivered(p)*, as described earlier, and *oldestTree(p)*, which is the oldest named tree it will accept messages on, from p. Messages on a tree T<*oldestTree(p)* (< represents the ordering of tree names) will be ignored. The *latestDelivered(p)* and *oldestTree(p)* values are specific to B's membership in c-cell(B), i.e., they are reinitialized if c-cell(B) changes.

When c-cells on a tree are changed, the filters on edges in the tree are also adjusted. This is the responsibility of the Subscription Propagation Subsystem (SPS).

Subscription Propagation Subsystem We discuss the guarantees provided by SPS, which are used to prove correctness of our topology change programs. The current implementation in the Gryphon system [13] provides these guarantees.

Each edge in a tree has two filters, one in each direction. For instance, the edge (C0,C1), has different filters for messages being routed from C0 to C1, and C1 to C0. We use *filter(C0,C1)* to represent routing from C0 to C1.

1. When a tree T is created, the filter on any edge in e(T) is true in both directions, i.e., it matches all messages.

2. Let c-cell C0 and c-cell C1 be both in v(T), since the creation of T. Then the filter on any edge on the path from C0 to C1 matches all messages that match $filter(C1)$.
3. Let c-cell C0 be in v(T) and subsequently c-cell C1 is added to v(T). C1 can execute the operation `propagateSub(f,T)`. When this operation completes, any filter on the path from C0 to C1 matches all messages matching f. This operation is idempotent.

Behavior of broker when c-cell changed When the c-cell that B is a member of is changed, B performs the following steps to apply the change:
(1) set *c-cell(B)* to null, and discard all *latestDelivered(p)* and *oldestTree(p)* values,
(2) disconnect all existing publishers and subscribers, and discard all c-ostreams,
(3) destroy any hosted pubend and associated messages [2],
(4) set c-cell(B) to the value read from the TD. If it is not null, a new pubend can be created.

Initialization of latestDelivered(p), oldestTree(p) Let *c-cell(B)* be equal to C. The first time broker B receives a message for c-cell C, from p, it has no *latestDelivered(p)* and *oldestTree(p)* values. Also, since streams are created on demand, this means it does not have a c-ostream(p,C). Broker B performs the following initialization sequence:

1. Send a request to the broker hosting p, say B', asking for the name of the tree it is currently publishing on. Let the response be T.
2. Execute the operation `propagateSub(filter(C), T)`. If the operation completes, set *oldestTree(p)*$=T$. If this operation is executing but B receives a message from p on a tree T', where $T' > T$, abort the operation and set *oldestTree(p)*$=T'$.
3. Perform another request-response interaction with p to find out the lowest Q tick at the pubend. Let this be t.
4. Set *latestDelivered(p)*$=t-1$, and create c-ostream(p,C), with all ticks $< t$ set to F and other ticks set to Q.

We explain the intuition behind the steps in this initialization sequence. In step 1 we explicitly ask B' for the current tree instead of using the tree on which the message was received. This is because it is possible that the tree on which the message was received is an old tree on which p no longer publishes, and an old message on this tree is received by C. We do not want to propagate the subscriptions on such a tree. Regarding step 2, by the constraints described in section 3.2 for TCS programs, C must be a member of $v(T')$ from the time of creation of T', so the filters on the path from B' to B are already wide enough.

It is possible that B blocks in a step of this initialization because of some reason, such as B' is down. Also, `propagateSub` may return an error if T no longer exists. It is safe for B to abort the initialization sequence, and not create a c-ostream for p. It can repeat the complete initialization sequence, if and when it chooses.

3.1.3 Change to currentTree(C)

Let *c-cell(B)* be equal to C, and let the old value of currentTree(C) be equal to T. Once this change is applied, all subsequent messages sent by the pubend hosted by B will not be on T, and all upstream messages received by B on T will be ignored.

3.2 TCS Programs

Parameterized instances of TCS programs are executed by the Topology Change Subsystem (TCS) on behalf of administrators. A program consists of two kinds of *operations* (1) reading and writing the TD, (2) instructing a broker B to read and apply the latest TD state (represented as `readLatest(B)`). All these operations are synchronous,

[2] To avoid losing published messages, an administrator should not change the c-cell of B until the pubend at B has no unacknowledged messages.

and we specify when a program may concurrently execute some operations. The `readLatest(B)` operation completes when B has *applied* the latest TD state.

All *operations* that read and write the TD are *atomic*, so we do not explicitly specify locks to handle concurrent reads and writes on the TD. However, it is sometimes important to atomically execute a read/write of the TD with one or more readLatest(B) operations. For this, we use explicit locks provided by the TD. Locks can be acquired non-exclusively or exclusively, analogous to read and write locks, a well-known concurrency control technique.

We start by assuming that programs are totally ordered, and the previous one terminates before the next one begins. This assumption is relaxed in section 3.3.

A program has *input parameters*, provided by an administrator. The TD state before execution of the program is the *starting state* and at termination of the program is the *ending state*. Given a *well-formed* starting state, we restrict the input parameters to a set which we will call *well-formed* parameters.

The programs that create and delete trees, and add brokers to an f-cell are very simple and described in appendix B for completeness. We describe the interesting ones next.

3.2.1 RemoveFromFCell(C, bSet)

The parameter C is a f-cell name, and bSet is the set of brokers to remove from C. The program performs the following operations in sequence:
(1) acquire non-exclusive lock on C.
(2) set $brokerSet(C) = brokerSet(C) - bSet$.
(3) for all brokers B $\in bSet$, execute readLatest(B).
(4) release non-exclusive lock on C.

The lock coordinates with programs that change an edge incident on C (section 3.2.3). Any input parameter is well-formed. As f-cell membership is irrelevant for a well-formed TD, it is obvious that if the starting state is well-formed, then the ending state is well-formed.

3.2.2 SwitchTree(C, T)

C is a c-cell name, and T is a tree name. The program performs the following operations in sequence:
(1) if $T > currentTree(C)$, set currentTree(C)=T
(2) for all brokers $B \in C$, execute readLatest(B).

We ensure monotonically increasing tree names for currentTree(C) in operation (1), since we later relax the total ordering between programs while ensuring a predictable result in the presence of concurrent execution.

The input parameters are well-formed if either of the following two conditions is true:

- $C \notin$ *activeCCellSet1*. (see def. 2. We use 1 to represent a value computed on the starting TD state).
- $C \in$ *activeCCellSet1* and all the following are true: (a) c-cell(T) = activeCCellSet1, (b) T is well-formed, (c) T has not been mutated by a preceding program.

  The intuition behind (c) is the following: all c-cells on tree T were part of T when it was created, hence the filters on the path from C to any c-cell C1, in T, are always wider than filter(C1). Hence subscribers at C1 are not disrupted by the switch to T.

Lemma 2 *If starting state and input parameters of SwitchTree are well-formed, the ending state is well-formed.*

3.2.3 ChangeActiveClientCells

This program, abbreviated as *CACC*, has four parameters, *addCSet, removeCSet, switchTSet* and *mutateTSet*. Informally, these represent the set of c-cells to be added or removed, the c-cells whose currentTree is being switched, and trees whose structure is being mutated.

The $addCSet$ is a table keyed by c-cells, C, that are to be added. The columns, $broker(C)$ and $tree(C)$ represent the broker to be added to this c-cell, and the value that currentTree(C) should be set to. The $removeCSet$ is a set of c-cells to be removed.

The $switchTSet$ is a table keyed by c-cells, C, and the column $tree(C)$ is defined like in $addCSet$. The $mutateTSet$ is keyed by tree names, T, and has columns *removeEdges, removeVertices, addEdges*, where *removeEdges(T), removeVertices(T)* specify the edges and vertices to remove from T and *addEdges(T)* specify the edges to be added. Adding an edge also adds the incident vertices, if they were not already on the tree.

We describe the operations performed by this program, in no particular order, and then describe the sequencing of these operations. Each operation is further described as a sequence of steps, executed in order. The steps are structured as reads and writes of the TD, followed by calls to `readLatest` for some brokers. All the steps that read and write the TD are atomic:

op1 For each c-cell C in $removeCSet$, do the following in sequence: (1) set tempSet = brokerSet(C), (2) set $brokerSet(C) = final$, (3) for each B in tempSet, execute readLatest(B).

op2 For each c-cell C in $addCSet$, do the following in sequence:

```
(1) if $brokerSet(C) = undefined$
      if ∃ C', $broker(C) \in brokerSet(C')$
        if $C' > C$, set $brokerSet(C) = final$
        else set $brokerSet(C) = broker(C)$,
                 $brokerSet(C') = final$
      else set $brokerSet(C) = broker(C)$
(2) if $currentTree(C) < tree(C)$
      set $currentTree(C) = tree(C)$
(3) execute readLatest(broker(C))
```

op3 For each (C, T) in switchTSet, do the following in sequence: (1) if $T >$ *currentTree(C)*, set *currentTree(C)* $= T$, (2) for all brokers $B \in C$, execute readLatest(B).

op4 For each (T, removeEdges, removeVertices, addEdges) in mutateTSet do the following in sequence:

(1) For all (C1, C2) $\in addEdges$, set $e(T) = e(T) \cup \{(C1, C2)\}$, $v(T) = v(T) \cup \{C1, C2\}$,

(2) For all (C1, C2) $\in removeEdges$, set $e(T) = e(T) - \{(C1, C2)\}$

(3) Set $v(T) = v(T) - removeVertices$.

(4) For all cells C such that C is on some edge in $addEdges$ or $removeEdges$, perform the following steps: (a) acquire exclusive lock on C, (b) for all brokers B $\in brokerSet(C)$, execute readLatest(B), (c) release exclusive lock on C ,

Steps (1), (2) in **op2** and step (1) in **op3** contain checks to ensure that they do not violate monotonicity. This allows us to subsequently relax the total ordering of programs.

The ordering constraint on these operations is: op1 is executed before op4, and op2 is executed after op4 and op3. Executing op1 before op4 ensures that all c-cells that are to be removed are 'shutdown' before tree mutation begins, and executing op2 as the last operation ensures that all mutation and switching of trees is done before 'startup' of added c-cells.

Before stating the conditions for the input to be well-formed, we define some terms.

We use 1 to represent a value computed on the starting TD state, and 2 for a value on the ending state. For instance, *activeTreeSet1* and *activeTreeSet2* are the values of the *activeTreeSet* at the start and end of the program respectively.

Definition 4 *Let,* addedT $=$ activeTreeSet2 $-$ activeTreeSet1, *and* existingT $=$ activeTreeSet2 $\cap$ activeTreeSet1. *Let* finalCCellSet *be a value computed on a TD state that represents the set of c-cells that are in final state.*

We split the well-formed constraints on the input parameters into two categories, *basic* constraints and *safety* constraints. The basic constraints, though fairly long, are straightforward and designed to ensure that well-formed parameters take the TD from a well-formed starting state to a well-formed ending state. The *safety* constraints are more subtle, and are designed to help satisfy the safety criterion (section 2.3.1).

Basic Constraints

1. *addCSet* $\cap$ *removeCSet* $=$ *empty* and *addCSet* $\cup$ *removeCSet* $\neq$ *empty*.
2. $\forall C \in$ *addCSet*, *brokerSet1(C)* $=$ *undefined*, i.e., it is undefined in the starting state.
3. $\forall C1, C2$ where $C1 \in addCSet, C2 \in activeCCellSet1, broker(C1) \notin brokerSet1(C2)$ or $C2 \in removeCSet$. Informally, a broker cannot be in two c-cells that will both be active when this program terminates.
4. $\forall T \in addedT$, T is well-formed when the program terminates, and *c-cells2*$(T) = (activeCCellSet1 \cup addCSet) - removeCSet$.
5. $\forall T \in existingT$, $T \in mutateTSet$. This is because it has to be mutated to add and remove c-cells.
6. $\forall C \in switchTSet$, $tree(C) \in addedT$. This means that existing c-cells cannot switch to a tree that is in $existingT$ and is therefore being mutated. This is done to simplify this program, since administrators can get around this restriction by executing a SwitchTree program prior to this.
7. $\forall T \in mutateTSet$, T is well-formed when the program terminates and *c-cells2*$(T) = (activeCCellSet1 \cup addCSet) - removeCSet$.

Safety Constraints

1. $\forall T \in addedT$, T should not be mutated by this program or any preceding program. This is to ensure that all edge filters are initialized to true.
2. $\forall T \in mutateTSet$, if C1 and C2 are both in *c-cells1(T)* and *c-cells2(T)*, then the path between them is unchanged. This restricts the kind of mutations that can be done to a tree, both to ensure that existing subscribers are not affected and to avoid routing cycles during mutation.
3. $\forall C1, C2$, if $C1 \in addCSet$ and $C2 \in activeCCellSet1 \cup finalCCellSet1$, then $C1 > C2$. The $>$ relation represents the total ordering on the names of c-cells. This monotonicity constraint was utilized in operation **op2** of the program.

Lemma 3 *If starting state and input parameters of CACC are well-formed, the ending state is well-formed.*

Theorem 4 *For any well-formed starting state, and any sequence of execution of RemoveFromFCell, SwitchTree and CACC, with well-formed parameters, the safety criteria is true.*

3.3 Ordering and Masking of TCS Programs

In the previous section, we assumed that all programs were totally ordered, and the previous one terminated successfully before the next one started.

We relax this ordering requirement in section 3.3.1. In section 3.3.2, we discuss the cases where a program P2 masks the updates performed by an earlier program P1. Such masking programs are important to restore the system to a well-formed state when a previous program terminates prematurely due to a failure.

3.3.1 Ordering

For two programs P1 and P2, we say that they have a read-write conflict, if both write the same field in the TD, or one reads a field and the other writes the field. Evaluating the well-formedness of the input parameters, with respect to the current TD state, is also considered a read.

Let H be a program history, where the system was well-formed in its initial state, and the ordering between programs in H is represented using the $\rightarrow$ relation, which is transitive. We consider histories that satisfy the following two criteria (1) any two programs with a read-write conflict are ordered, (2) for any total ordering of H, and any program P_i in this history, the input of P_i is well-formed.

Programs that change f-cell membership are not ordered with SwitchTree or CACC. Also, two SwitchTree programs which switch the tree for different c-cells are not ordered. However, a SwitchTree program is ordered with a CACC program, since CACC is either mutating or switching the tree being used by a c-cell C. For the same reason, CACC programs are ordered with respect to each other.

For improving concurrency of programs that are ordered, we consider a special case of $\rightarrow$, $\rightarrow^i$. For two programs P1 and P2 such that P1 $\rightarrow^i$P2, the programs are such that the monontonicity of their input and behavior makes them commutative, and hence they can be concurrently executed. Given P1 $\rightarrow$P2, we define the conditions under which P1 $\rightarrow^i$P2.

I1 When both P1 and P2 are SwitchTree(C,*).

I2 When one of P1, P2 is SwitchTree and the other is CACC.

I3 When both P1, P2 are CACC programs, except when P1 and P2 are mutating the same tree.

Unlike the $\rightarrow$ relation, $\rightarrow^i$ is not transitive. The case that prevents transitivity is 3 programs, P1, P2, P3, such that $P1 \rightarrow P2 \rightarrow P3$ and P1, P3 are CACC and P2 is SwitchTree. Now, by condition I2, $P1 \rightarrow^i P2$ and $P2 \rightarrow^i P3$. However, P1, P3 may mutate the same tree, so $P1 \not\rightarrow^i P3$.

3.3.2 Masking Programs

We consider a special case of $\rightarrow^i$, $\rightarrow^m$, where P1 $\rightarrow^m$P2 means that P2 masks the effect of P1.

M1 The order defined by condition I1 is always $\rightarrow^m$.

M2 The order I2 is $\rightarrow^m$ if P1 is SwitchTree(C,*) and P2 is CACC, and either P2 is again switching the tree used by C or P2 is removing c-cell C.

M3 The order I3 is $\rightarrow^m$ when both the following conditions are satisfied: (1) P2 does not mutate any tree, (2) $removeCSet(P1) \cup addCSet(P1) \subseteq removeCSet(P2)$.

Theorem 5 *The relation $\rightarrow^m$ is transitive.*

4 Related Work

In addition to work in self-organizing overlays for multicast, which we mentioned earlier, there is much research on scalable publish/subscribe systems constructed as an overlay network of brokers [3, 7, 10, 12]. However these pub/sub systems only offer best-effort delivery.

The work by Cugola, Picco et al [9, 5] specifically deals with topology reconfiguration in a pub/sub system. They consider a single tree topology, where the tree nodes are single brokers, and do not consider node or link failures. Their work focuses mainly on how to minimize the repropagation of subscription information on a tree that has changed. This is complementary to our work, and could be used in the implementation of the Subscription Propagation Subsystem we assume in this paper. The algorithm described in [5] minimizes the events lost when reconfiguring the subscription filters.

Costa et al [4] describe epidemic algorithms for 'reliable' delivery in a pub/sub middleware. These algorithms deal with messages lost due to failures and topology reconfiguration. However, the reliability guarantees are probabilistic and their experiments show delivery probability of less than 90% in some scenarios. Also, their algorithms are based on sending data about which messages matched a randomly chosen filter f. When the number of unique filters in the system is large, this may not be scalable. In contrast, our topology reconfiguration solution is applicable to protocols that guarantee 100% reliability and the scalability is independent of the number of unique filters in the system.

Virgilitto [12] provides a formal model for reliability that is independent of any system implementation.

5 Conclusion

We have developed a formal and practical system model for topology changes in a publish/subscribe system that offers both best-effort and reliable delivery of published messages. The topology change programs we have developed are applicable to an overlay network with multiple paths between sources and sinks, do not degrade the reliability of message delivery, and do not stop delivery during the topology change. Many of these programs can also be concurrently executed.

References

[1] G. Banavar, T. Chandra, B. Mukherjee, J. Nagarajarao, R. E. Strom, and D. C. Sturman. An efficient multicast protocol for content-based publish-subscribe systems. In *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems, 1999*, pages 262–272, 1999.

[2] S. Bhola, R. Strom, S. Bagchi, Y. Zhao, and J. Auerbach. Exactly-once delivery in a content-based publish-subscribe system. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN'2002)*, pages 7–16, 2002.

[3] A. Carzaniga. *Architectures for an Event Notification Service Scalable to Wide-area Networks*. PhD thesis, Politecnico di Milano, December 1998.

[4] P. Costa, M. Migliavacca, G. P. Picco, and G. Cugola. Epidemic algorithms for reliable content-based publish-subscribe: An evaluation. In *To appear in Proceedings of ICDCS*, 2004.

[5] G. Cugola, D. Frey, A. L. Murphy, and G. P. Picco. Minimizing the reconfiguration overhead in content-based publish-subscribe. In *To appear in Proceedings of ACM Symposium on Applied Computing (SAC04)*, 2004.

[6] J. Jannotti, D. K. Gifford, K. L. Johnson, M. F. Kaashoek, and J. O. Jr. Overcast: Reliable multicasting with an overlay network. In *Proceedings of OSDI 2000*, pages 197–212, 2000.

[7] G. Mühl. *Large-Scale Content-Based Publish/Subscribe Systems*. PhD thesis, Darmstadt University of Technology, September 2002.

[8] B. Oki, M. Pfluegl, A. Siegel, and D. Skeen. The Information Bus - An Architecture for Extensible Distributed Systems. *Operating Systems Review*, 27(5), December 1993.

[9] G. P. Picco, G. Cugola, and A. L. Murphy. Efficient content-based event dispatching in the presence of topological reconfiguration. In *Proceedings of ICDCS*, pages 234–243, 2003.

[10] P. R. Pietzuch and J. Bacon. Peer-to-peer overlay broker networks in an event-based middleware. In *2nd International Workshop on Distributed Event-Based Systems (DEBS'03)*, 2003.

[11] A. Rowstron, A. Kermarrec, M. Castro, and P. Druschel. Scribe: The design of a large-scale event notification infrastructure. In *Proceedings of 3rd International Workshop on Networked Group Communication (NGC 2001), UCL, London, UK*, November 2001.

[12] A. Virgilitto. *Publish/Subscribe communication systems: from models to applications*. PhD thesis, Universita "La Sapienza" Roma, 2003.

[13] Y. Zhao, D. Sturman, and S. Bhola. Subscription propogation in highly-available publish/subscribe middleware. In *ACM/IFIP/USENIX 5th International Middleware Conference (Middleware 2004) (To Appear)*.

A Proofs

A.1 Proof of Lemma 1

If C is in activeCCellSet, it must be on all the trees in activeTreeSet (by def. 2.4). Since these trees are well-formed (by def. 2.3), brokerSet(C) has cardinality 1 (by def 1.4).

If C has cardinality 1, it must have a non-empty currentTree(C) (by def. 2.1), say T. By the definition of activeTreeSet, T is in activeTreeSet. Also, by def. 2.1 $C \in v(T)$. Hence, by def. 2.4, C is in the activeCCellSet. □

A.2 Proof of Lemma 2

The well-formed conditions for a system are defined in def. 2, which contains 4 conditions.

Clearly, condition 2 is true in the ending state as it is true in the starting state, and the program does not change any c-cell membership.

For condition 1, we consider two cases:

1. $C \notin$ *activeCCellSet1*: By lemma 1, $|brokerSet(C)| \neq 1$, and therefore condition 1 is true.
2. $C \in$ *activeCCellSet1*: By the definition of well-formed input parameters, condition 1 is true.

Now we consider conditions 3 and 4. We again consider the 2 cases:

1. $C \notin$ *activeCCellSet1*: By lemma 1, $|brokerSet(C)| \neq 1$. There are 2 subcases (1) $T \in$ *activeTreeSet1*, and (2) $T \notin$ *activeTreeSet1*. In subcase (1), by the definition of *activeTreeSet*, there must be a c-cell $C' \neq C$, such that $|brokerSet(C')| = 1$ and *currentTree(C')* $= T$. Since this program makes no change to C', $T \in$ *activeTreeSet2* and *activeTreeSet1* $=$ *activeTreeSet2*. Therefore conditions 3 and 4 are true. In subcase (2), since $|brokerSet(C)| \neq 1$, $T \notin$ *activeTreeSet2* and so *activeTreeSet1* $=$ *activeTreeSet2*. Again, this means that conditions 3 and 4 are true.
2. $C \in$ *activeCCellSet1*: There are two subcases.

   (1) $T < currentTree1(C)$: In this subcase, *activeTreeSet1* $=$ *activeTreeSet2* and so conditions 3 and 4 are true.

   (2) $T \geq currentTree1(C)$: In this subcase, $T \in$ *activeTreeSet2* and is the only tree in *activeTreeSet2* that may have not been in *activeTreeSet1*. Since the input parameters are well-formed, T is well-formed and *c-cell(T)* $=$ *activeCCellSet1*. This means that *activeCCellSet1* $=$ *activeCCellSet2*. Therefore, conditions 3 and 4 are true.

□

A.3 Proof of Lemma 3

The well-formed conditions for a system are defined in def. 2, which contains 4 conditions.

Condition 2 is true because the starting state is well-formed and the input parameters obey basic constraint 3.

By lemma 1, we know that in the starting state

$$|brokerSet1(C)| = 1 \;\Leftrightarrow\; C \in activeCCellSet1$$

Due to basic constraint 1, 2 and 3, when the program terminates,

$$|brokerSet2(C)| = 1 \;\Leftrightarrow\; C \in$$

$$(activeCCellSet1 \cup addCSet) - removeCSet$$

We also know by construction of the program that

$$\forall C \in (activeCCellSet1 \cup addCSet) - removeCSet,$$

$$currentTree(C) \neq empty$$

This proves that part of condition 1 is true.

We also know that by definition,

$$activeTreeSet2 = addedT \cup existingT$$

Using basic constraint 4, 5 and 7, we infer that

$$\forall T \in activeTreeSet2 : c - cells2(T) =$$

$$(activeCCellSet1 \cup addCSet) - removeCSet$$

and T is well-formed when the program terminates. Clearly, condition 3 is true. And since every tree in activeTreeSet contains the same set of c-cells and these are precisely the c-cells that have cardinality 1, conditions 1 and 4 are true. □

A.4 Proof of Theorem 4

Informally, this theorem states that the safety criteria specified in definition 3 will not be violated when executing any sequence of programs with well-formed input parameters, starting with a well-formed system.

We prove this by contradiction. Let c-cell C observe an F value for tick t, where t is greater than the current value of $latestDelivered(p)$. Let this be a tick that was originally D at pubend p (hosted at C0), and this D tick matched $filter(C)$. This means that C received a downstream message that contained this F tick.

There are only 2 situations in which this can occur:

Filtering by some broker, say B1 on tree T: Let broker B1 have an f-istream(p,T,C1,C2) that contained D for this tick, and f-ostream(p,T,C3,C4) is a child of this istream that has F for this tick. Also, assume that the directed edge (C3,C4) in tree T, is on the path from C0 to C. Assuming correctness of the subscription propagation subsystem (SPS), the filter on (C3,C4) can only be narrower than filter(C) in one of the following cases (1) C was added to T after the creation of T, or (2) C is being removed from T by mutating T. Case (2) can only occur in an execution of CACC where C is in removeCSet. But by construction of CACC, C is 'shutdown' before mutating T, so C will not observe this F tick. Therefore, the only possibility is case (1). An execution of CACC must have added C to tree T. We enumerate the possible values of oldestTree(p) at c-cell C:

- $oldestTree(p) = T$: We enumerate the two possibilities:

  1. T was the response received from p, in the request-response initialization. Therefore C must have completed propagateSub(filter(C), T), before asking pubend p for its lowest Q tick. Therefore for all ticks > $latestDelivered(p)$, the filter on (C3,C4) must be wider than filter(C). So B1 could not have turned D into F. This contradicts our assumption.
  2. T' was the response received from p, but C aborted propagateSub(filter(C), T'), since it received a message on T, where $T > T'$. Let the CACC program that adds c-cell C terminate with the currentTree(C0)=T". Since op2 is the last operation in this program, we know that C knows that currentTree(C0)=T" before it receives the request from C. Therefore $T'' \leq T'$, and $T'' < T$. Since the input parameters of subsequent CACC and SwitchTree programs are well-formed, C must be a member of T from the time of creation of T. Hence the filter on (C3,C4) is always wider than filter(C). So B1 could not have turned D into F. This contradicts our assumption.

- $oldestTree(p) > T$: C-cell C will ignore all messages on T, so it cannot see the F tick generated by B1. This contradicts our assumption.

- $oldestTree(p) < T$: At some point in the execution history of these programs, currentTree(C0)=T. We also know that T was mutated to include c-cell C. By the rules of well-formed input to CACC and SwitchTree, T must be mutated to include C after currentTree(C0) was set to T. Therefore oldestTree(p) must be $\geq T$, which is a contradiction.

Acking on tree T: Let this tick be acknowledged to the pubend on tree T, even though C has a latestDelivered(p) value less than this tick. This leads to 2 cases:

1. $oldestTree(p) > T$: Prior to C initializing oldestTree(p), pubend p must have stopped accepting acknowledgements on tree T. Therefore tick t must have been acknowledged prior to initialization of oldestTree(p). Since C initializes latestDelivered(p) after oldestTree(p), tick t must have been acknowledged prior to initialization of latestDelivered(p), which means latestDelivered(p) $\geq t$. This contradicts our assumption.

2. $oldestTree(p) \leq T$: Since we are assuming *latestDelivered(p)* $< t$, tick t in the knowledge stream at pubend p must be Q, at the time pubend p responded to the request from C for the value of latestDelivered(p). We consider 2 subcases (1) C is a member of T when T was created, and (2) C was not a member of T when T was created.

   In (1), there is only one reason that tick t was acknowledged though C had not delivered that D: T must be mutated to remove C. This can only occur in an execution of CACC where C is in removeCSet. By construction of CACC, C must be shutdown prior to mutation of T, hence C cannot ever observe tick t. This is a contradiction to our assumption that C observed an F for tick t.

   In (2), there are 2 further choices (A) T must be mutated to add C, or (B) T was never mutated to add C. Choice (A) can only occur in an execution of CACC where C is in addCSet. By construction of CACC, C is not started up until the mutation of T is complete. So no broker on T can acknowledge tick t without C acknowledging this tick. Which is a contradiction since we stated that C did not acknowledge this tick.

   Choice (B) can occur only if there was a $T' < T$ such that T' contained C, and there was a time when currentTree(C0)=T'. A subsequent execution of CACC that contained C in removeCSet must have caused C0 to switch to some other tree, and eventually to T (possibly after more executions of switchTree and CACC). Since a c-cell that has been removed is never added back (due to the well-formed input to CACC), C will not be on any tree such that $currentTree(C0) \geq T$. Therefore the acknowledgement of tick t, on tree T, cannot be observed by C. This contradicts our assumption.

A.5 Proof of Theorem 5

We are given P1, P2, P3, such that $P1 \rightarrow^i P2 \rightarrow^i P3$, $P1 \rightarrow^m P2$, and $P2 \rightarrow^m P3$. We want to prove $P1 \rightarrow^m P3$. There are 3 cases:

- *P1 and P2 are SwitchTree(C,*) programs.* There are 2 subcases (1) P3 is SwitchTree, (2) P3 is CACC. For subcase (1), by definition M1, $P1 \rightarrow^m P3$. For subcase (2), by definition M2, P3 must be switching the tree used by C or removing c-cell C. So condition M2 also applies to P1, P3, hence, $P1 \rightarrow^m P3$.

- *P1 and P2 are CACC programs.* Since P2 is CACC, and $P2 \rightarrow^m P3$, the only possibility is for P3 to be a CACC program (M3). By M3, P2 and P3 must not mutate any tree, and $removeCSet(P1) \cup addCSet(P1) \subseteq removeCSet(P2)$, and $removeCSet(P2) \cup addCSet(P2) \subseteq removeCSet(P3)$. Therefore $removeCSet(P1) \cup addCSet(P1) \subseteq removeCSet(P3)$. Using definition M3, $P1 \rightarrow^m P3$.

- *P1 is SwitchTree(C,*) and P2 is CACC*. By M2, P2 is either changing currentTree(C) or removing c-cell C. By M3, P3 must also be CACC, and P3 does not mutate any tree, and $removeCSet(P2) \cup addCSet(P2) \subseteq removeCSet(P3)$. There are 3 subcases:
  - P2 removes c-cell C: Since $removeCSet(P2) \subseteq removeCSet(P3)$, P3 must also be removing c-cell C. Therefore, $P1 \rightarrow^m P3$.
  - P2 does not remove c-cell C, and P3 removes c-cell C: Using M2, $P1 \rightarrow^m P3$.
  - Both P2 and P3 do not remove c-cell C: We know that P3 does not mutate any tree. If we can prove that P3 is changing currentTree(C), we can use M2 to show that $P1 \rightarrow^m P3$. We will prove that P3 is changing currentTree(C) by contradiction. Let it not be changing currentTree(C). Since P3 does not mutate any tree, C must not be in activeCCellSet of P3's starting state. Since P2 changes currentTree(C), there must be a CACC P4, such that $P2 \rightarrow^m P4 \rightarrow^m P3$, where P4 removed c-cell C. Since $P4 \rightarrow^m P3$, $removeCSet(P4) \subseteq removeCSet(P3)$, so P3 must remove c-cell C. This contradicts our assumption.
- *P1 is CACC and P2 is SwitchTree(C,*)*: This is not possible with any of M1, M2, M3.

□

B Other TCS programs

CreateTree(T, V, E) T is the tree name, V is the set of vertices and E the set of edges. This program has a single operation that reads and writes the TD.

1. if $e(T) = undefined$ and $v(T) = undefined$, set $e(T) = E$, $v(T) = V$.

Any value for the input parameters is well-formed.

DeleteTree(T) T is the tree name. This program has a single operation that reads and writes the TD.

1. set $e(T) = final$, $v(T) = final$.

The input parameter is *well-formed* with respect to the starting state if $T \notin activeTreeSet1$.

AddToFCell(C, bSet) C is a f-cell name, and bSet is the set of brokers to add to C. The program performs the following operations in sequence:

1. set $brokerSet(C) = brokerSet(C) \cup bSet$.
2. for all brokers B $\in bset$, execute readLatest(B).

Any value for the input parameters is well-formed.

C Concise Representation of TD Tables

The *fcellSet* can be concisely encoded by only keeping information for a cell C when *brokerSet(C)* has cardinality > 0. All others have *brokerSet(C)* equal to *empty*.

The *ccellSet* is not as simple, since it is important to distinguish between a *brokerSet(C)* that has cardinality zero because it is *undefined* or *final*. Due to the monotonicity of c-cells that are added in a CACC program (safety constraint 3), there is a c-cell C, such that all c-cells that are in added by future CACC programs are $> C$. In the terminology of safety constraint 3, this is the highest named c-cell in *activeCCellSet* $\cup$ *finalCCellSet*. Say we call this the *threshold* c-cell. The *ccellSet* can be concisely encoded by keeping the *threshold* c-cell, and rows for which *brokerSet(C)* has cardinality > 0. If a c-cell C is not in the table, then either it is $\leq$ or $>$ the *threshold* c-cell. In the former case, it is in the *final* state and in the latter case it is in the *undefined* state.

The *treeSet* does not have a monotonicity property like *ccellSet*, therefore it has to explicitly remember the rows for trees that have either been deleted ($v(T)$, $e(T)$ equal to *final*) or have been created but not deleted ($v(T)$, $e(T)$ cardinality > 0). In a real world setting, administrators will probably impose a monotonicity constraint on themselves, such that they will not create new trees with names less than some threshold value, say T_t. In that case, T_t can be made part of the *treeSet* state, and the row for any tree T that has been deleted and $T < T_t$ can be discarded.