# $\delta$-Fault-Tolerant Publish/Subscribe Systems

## Technical Report CSRG-570

**Middleware Systems Research Group**
**University of Toronto**

*Authors:*

Reza Sherafat Kazemzadeh
*reza@eecg.utoronto.ca*

Hans-Arno Jacobsen
*jacobsen@eecg.utoronto.ca*

UNIVERSITY OF TORONTO
Department of Electrical and Computer Engineering

November 20, 2007
Last revised: November 27, 2007

©2007 All rights reserved

## Abstract

*In this paper, we study reliable distributed publish/subscribe (P/S) systems that can "tolerate" multiple simultaneous node crash failures. We formally define a routing consistency property, and propose scalable algorithms that establish and maintain consistency in order to guarantee reliable, in-order, and duplicate-free delivery of messages. Furthermore, we introduce a system configuration parameter, $\delta$, that corresponds to the maximum number of simultaneous node failures that do not compromise P/S reliability guarantees or prevent system operations. This is achieved via replication of routing information in a fully decentralized manner compatible with the multicast nature of distributed P/S systems. Moreover, we assume node failures are transient and devise algorithms that allow failed nodes to "recover" by synchronizing with other nodes. A recovered node acts as if it has never failed before, and can fully participate in message forwarding.*

# 1 Introduction

*Publish/subscribe* (P/S) is a message-oriented communication paradigm that deals with selective delivery of *publication* messages from data sources, i.e., *publishers* to interested consumers, i.e., *subscribers*. Distributed P/S systems have gained much attention in both industry and academia mainly due to their high scalability to serve a large number of clients (publishers and subscribers) [9, 7, 8, 2]. However, the reliability aspect of this model which is crucial for widespread adoption in critical application areas (e.g., financial markets, military, and the health sector) needs further investigation. In fact, we show later in this paper that in absence of effective reliability mechanisms, the routing strategies (content-based and topic-based routing) employed in standard distributed P/S systems can lead to loss of publication messages even in failure-free scenarios. This problem is further exacerbated under crash failures of multiple nodes in the P/S system. Previous attempts to deal with failures are either unable to prevent publication loss [8, 6], or need special deployment procedure, to set up dedicated backup nodes for each primary node [2]. In contrast, our approach relies on a transparent replication scheme that can reliably tolerate and recover from multiple node failures.

We refer to this feature as the $\delta$-fault-tolerance, and guarantee that within the scope of the reliability specification, all operations of a $\delta$-fault-tolerant P/S system can be performed successfully provided that the number of simultaneous node failures do not exceed $\delta$.

A general definition of reliability is given in the IEEE Standard Computer Dictionary as "the ability of a system or component to perform its required functions under stated conditions for a specified period of time"[1]. We tailor this definition to our context, and characterize the reliable behavior of a distributed P/S system as duplicate-free and per-source in-order delivery of publication messages to *all* matching subscribers. To designate the exact moment in time from which the system is able to provide this guarantee, we rely on an explicit *confirmation* mechanism. More specifically, for each message $m$ issued by a client to perform an operation, e.g., subscribe, the system replies with a *confirmation* message, $Conf^m$ asserting successful completion of the requested operation.

We consider three main categories of operations: join, publish, and subscribe. To construct a P/S network and set up topology links, brokers and clients perform a join operation upon arrival, which updates the *topology routing information* of all the nodes that are within distance $\delta+1$ of the joining node. The subscribe operation is carried out by subscribers via issuing a subscription. Consequently, receipt of the corresponding confirmation message at the client signals start of the reliable system guarantee. This approach effectively prevents the anomalous execution scenario illustrated in Section 3. Finally, publishers carry out the publish operation by issuing a publication message which is forwarded through the network towards the *matching* subscribers. Confirmation message for a given publication indicates successful delivery of the publication to all non-faulty matching subscribers according to the reliability specification.

Since failures are a reality in distributed systems, we devise a *fault-tolerant forwarding* algorithm in order to ensure reliability in the case of node failures. Our approach can effectively bypass failed nodes in the topology, allowing all system operations to be performed successfully in presence of up to $\delta$ simultaneous failures. More specifically, each piece of routing information is replicated transparently on $\delta + 1$ nodes along the message forwarding path. The non-faulty copies are used if the other copies are not available due to failures. Furthermore, if failures are transient, we allow failed nodes to recover by receiving missed messages, and synchronizing the routing information. We only assume that a recover-

ing node is able to identify[1] nodes previously within distance $\delta + 1$. Upon completion of the recovery procedure, the recovering node is able to fully participate in message forwarding as if it had never failed before.

The contributions of this paper are as follows: (*i*) we formally define a two-level consistency property of the routing information among nodes; (*ii*) we propose a generic forwarding algorithm that is applicable to all types of multicast messages, and further refine it to individual types; (*iii*) we present a fault-tolerant forwarding algorithm that uses the consistent routing information to bypass any chain of up to $\delta$ consecutive failed nodes; (*iv*) we propose a recovery algorithm in order to bring the routing information of a failed node back into consistency as it recovers; (*v*) we devise a scalable duplicate detection algorithm that only uses partial information about a message's forwarding path.

The rest of the paper is organized as follows: Section 2 gives a short background overview; in Sections 3 we demonstrate an unreliable scenario in a typical P/S system that leads to incorrect loss of some publications; in Section 4 we describe our model, precisely specify reliable behavior of P/S system, and formally define the consistency of routing information. Furthermore, we propose the forwarding algorithms for different types of multicast messages. The fault-tolerance forwarding, and duplicate detection algorithms are presented in Section 5. Section 6 covers the recovery procedure. Section 7 gives an overview of the related work, and Section 8 presents the conclusions. The proof of correctness is given in Appendix A.

## 2 Background

A distributed publish/subscribe system is composed of an interconnected network of brokers, and a set of publishing and subscribing clients that connect to this network. Publishers issue publication messages, and send them to a network broker. The P/S system is responsible to deliver the publications to those subscribers that have issued a *matching* subscription. A publication *matches* a subscription when the attribute-value pairs in the publication evaluate all predicates of the subscription to *true*. Based on the expressiveness of the subscription language, P/S systems are typically broken into two types: those that perform *topic-based* matching and those that perform *content-based* matching. The former restricts the subscriptions to specify a *topic* as the only predicate; whereas the latter allows subscriptions to contain a conjunction of multiple predicates with a wide range of operators, such as $<, \leq, >, \geq$, or even string operators.

Since publishers and subscribers are likely to be attached to different brokers, delivery of publication messages requires routing through the broker topology. Furthermore, it is the responsibility of the P/S middleware to ensure that publications are delivered to all subscribers with matching subscriptions, and there is no delivery of publications to a subscriber unless it has a matching subscription. Without getting into the implementation details, a typical P/S system would layout the brokers in a tree-based network and propagate individual subscription messages throughout the network as they are issued. Any broker forwards the subscription to farther nodes downstream and also stores the id of the broker that it received the message from. This effectively constructs a routing path, between the subscriber and any other node in the network. Using this information, publication messages are *forwarded* towards their matching subscribers hop-by-hop in the reverse direction.

[1] This is done via restoring data from persistent storage or by contacting a topology directory repository, and is out of the scope of this paper.

# 3 Unreliable System Example

In this section, we present runtime execution of two distributed P/S systems that depict anomalous and *unreliable* behaviors.

## 3.1 Content-Based P/S System

Figure 1(a) illustrates the network topology of the system where there is only *one* path between each pair of nodes. In this system, $S1$ is a subscriber issuing subscription $s1$ that matches publications $p1$, $p2$, and $p3$ from $P$, and $S2$ issues subscription $s2$ which only matches $p1$. In this scenario, broker $B2$ falsely suppresses $p2$ and only forwards $p1$, and $p3$ towards $S1$. The details are illustrated in Figure 1(b), where each axis corresponds to one message over time. The filled circles indicate initial generation of the message from a client, and subsequent crosses on the axes represent arrival of the messages at other network nodes. False suppressions of $p2$ is shown by an empty circle on its corresponding axis.

As it can be seen in the diagram, broker $B1$ forwards $p1$ while it is unaware of the subscription $s1$. In fact, $p1$ is forwarded due to a side effect of $s2$ which was present at the time. On the other hand, $p2$ does not match $s1$ and since $B2$ is still unaware of $s1$, it is suppressed. Finally, $s1$ arrives at $B2$, and the following publications are not lost.

## 3.2 Topic-Based P/S System

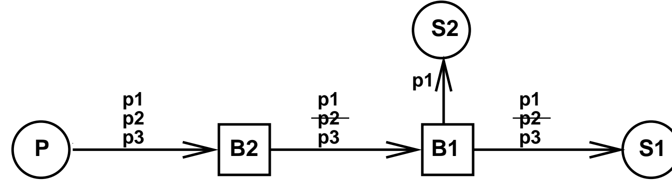
We use the same network topology illustrated in Fig 1(a), and present the execution trace of the system in Fig 1(c). Subscribers $S1$ and $S2$ both subscribe to the same topic, and hence presence of $S2$ at broker $B2$ prevents suppression of $P1$. Since $S2$ is removed shortly afterwards by an unsubscription message, $US$, $P2$ will not match any subscriptions and is dropped at $B2$. Finally, $S1$ arrives at $B2$ and the third publication is correctly forwarded to $B1$.

We identify the cause of this problem as inconsistent routing information corresponding to subscription from $S1$. While broker $B1$ was aware of this subscription, broker $B2$ had not received it yet. We hence argue that arrival of any number of publications at $S1$ must not be interpreted as the *start* of the reliable P/S service. We further propose that the client (subscriber $S1$) must be notified explicitly by the system on when the reliability guarantees can be provided. For this purpose, we rely on confirmation messages in our approach. In the following sections, we formally define the consistency property pertaining to forwarding of different types of messages, and present algorithms that ensure correct delivery of messages even in presence of node failures.

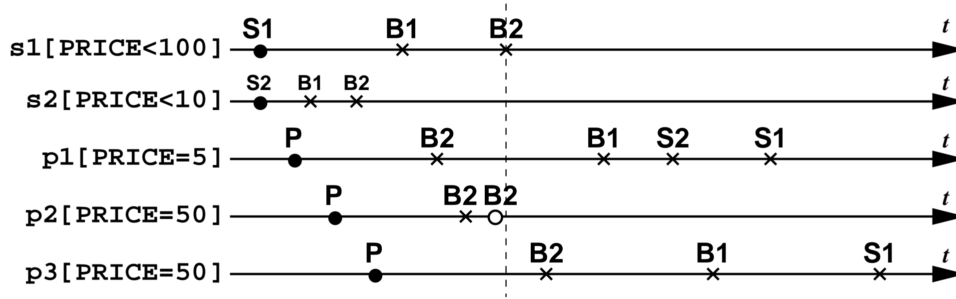
# 4 Reliability

## 4.1 Model and Notation

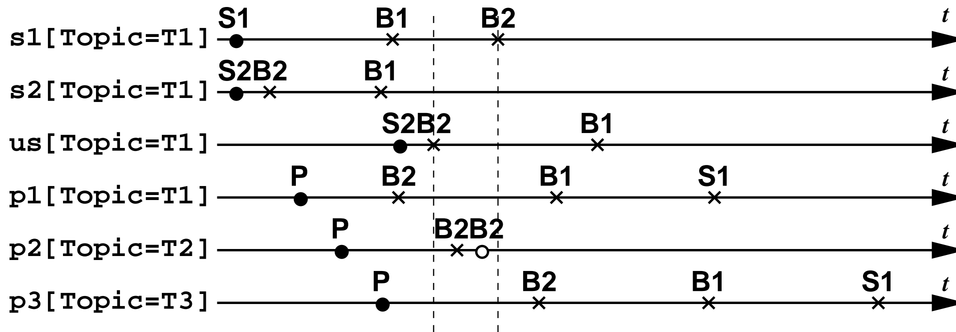
To simplify presentation, we initially ignore the distinction between clients and brokers, and refer to them as *nodes*. Since clients do not participate in routing the same way the brokers do, we will later elaborate on how to refine clients to operate with only a subset of this functionality. We model each node as a process, a communication layer, and a failure detector module. The process executes one of the presented algorithms, and uses the communication layer to send/receive messages to/from other nodes over a network. We assume each node is assigned a globally unique identifier, $nodeId$, which




(a) Network topology




(b) Content-based P/S system




(c) Topic-based P/S system.

**Figure 1. Sample runtime traces of a typical content-based P/S system, where intermediate publications are lost.**

also serves as an address to communicate with it[2]. We consider *transient* node failures, i.e., nodes may *fail* at some point in time, and then *recover* at a later time. The process of a failed node stops executing any instructions, and its communication layer is unable to send/receive any messages. Furthermore, all the data internal to the node's process and all messages in the communication layer are lost. If a node recovers from a failure it starts to execute a *recovery procedure*. We use the terms faulty node and failed node interchangably, and assume that any node may crash an unlimitted number of times.

[2]For example, in an IP network, it is reasonable to use $nodeId = \langle IP_address, Port_number\rangle$.

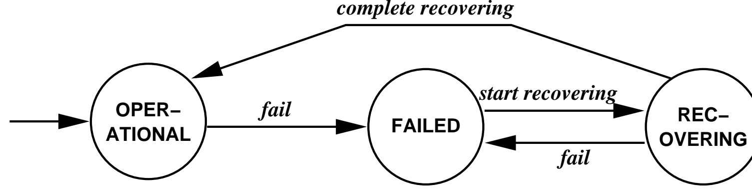



**Figure 2. Node operational states.**

Fig 2 illustrates different operational states of a node at any point in time. The *OPERATIONAL* state corresponds to the state of a non-faulty node that operates normally and executes the *reliable forwarding* algorithm (Section 4), whereas, the *FAILED* models the state of a crashed node that does not execute any algorihtms. The *RECOVERING* state on the other hand, is an interim state during which the node executes the *recovery* procedure. The node may fail while recovering. However, upon completion of the recovery procedure, the state transition to *OPERATIONAL* takes place.

We use the $data_X$, and $procedure_X()$ notations to respectively refer to a data structure, and a piece of algorithm executed by the process of node $X$. For any pair of nodes, $N$ and $M$, we use $\alpha_N^M$ to refer to $N$'s view of the operational state at $M$. Due to network delays, this view may not be accurate at all times. Furthermore, $\alpha_N^M = FAILED$ if $N$ is unaware of $M$'s state. To maintain consistency in notation, we use $\alpha_N^N$ to allow algorithms executed at node $N$ to access or update its own operational state.

The communication layer of a node is composed of a FIFO input queue to receive messages, and zero or more FIFO output queues used to send messages. $Connections_N$ is a set of $nodeId$s representing nodes that $N$ maintains a connection to. For each $X$, $X \in Connections_N$, there is an associated output queue in the communication layer of $N$ that is solely used to send messages to $X$. As $nodeId$s are added/removed from $Connections_N$, corresponding output queues are creaded/destroyed. Messages are *transmitted* by being dequeued from an output queue of the sending node and enqueued at the input queue of the receiving node. We assume that the communication links retain the message order, and are also reliable, i.e., unless the receiving node fails, no message is lost while in transit between queues. The process of a node, *send*s/*receive*s messages, by simply performing an equeue/dequeue operation on the corresponding output/input queue of its communication layer. Furthermore, messages may be tagged as *low* priority; the dequeue operation on all queues ensures that the highest priority message closest to the head of the queue is always removed first.

We assume that the process at $N$ can execute the instruction "`Start failure detector with` $M$", to monitor $M$'s operational state. If the failure detector detects failure of $M$, it will notify the process at $N$, by executing $\alpha_N^M \leftarrow FAILED$. Other than what has been stated so far, and the requirements of the particuar failure detector implementation used, e.g., link delays, we do not make any further assumptions about the network communication links. In the rest of this section, we will introduce the notation that we adopt in the subsequent sections.

We explicitly distinguish between two types of links: a *communication link* corresponds to a directed edge in the topology, and is used to send/receive messages between its endpoint nodes. A communication link from node $X$ to $Y$ is created when $X$ adds $Y$ to $Connections_X$. On the other hand, a *topology link* between node $X$ and $Y$ is created during a *join* operation and represented as $\overleftrightarrow{XY}$. Topology links are undirected, and we have $\overleftrightarrow{XY} = \overleftrightarrow{YX}$. Furthermore, the set of all topology links construct a spanning tree in the graph of all successfully joined nodes. $\overleftrightarrow{XY}^*$ is a *marked* copy of $\overleftrightarrow{XY}$ and represents a

communication link created while a join operation is in progress. By the time, the join operation is complete $\overleftrightarrow{XY}^*$ is replaced by $\overleftrightarrow{XY}$.

For any pair of nodes, $A$ and $B$ in this graph, the *topology path*, $P(A, B)$, or simply $P$ is an ordered non-empty list of node identifiers that fall along the path of topology links connecting $A$ to $B$. $d(A, B)$ represents the length of this path, and we refer to the $i^{th}$ node on this path by $P[i]$. We have $X \in P(A, B) \Leftrightarrow (\exists i, 0 \leq i \leq d(A, B) \wedge P(A, B)[i] = X)$. Finally, $P(A, B)$ is a $\delta$-path *if and only if* $d(A, B) \leq \delta$. The $append^{\delta}(item, l)$ operation, adds $item$ to the end of list $l$, possibly removing the item at the head of the list if its length exceeds $\delta$.

Each message $m$ generated at a source node $S$ is uniquely identifiable by its ID, $m_{id} = \langle S, sourceMsgId \rangle$, where $sourceMsgId$ is a locally unique identifier from a totally ordered set[3]. If $m$ and $m'$ are messages from the same source, and $m$ preceeds $m'$, we have:

$$m_{id}.sourceMsgId \prec m'_{id}.sourceMsgId$$

Duplicate messages are two copies of the same message, and we have $d$ and $d'$ are duplicates *if and only if* $d_{id} = d'_{id}$. The sender of a message $m$ can be identified by $m.sender$ which is updated automatically at the time $m$ is being enqueued in the output queue. Furthermore, any message $m$ has a type $T(m)$, where $T(m) \in \{J, SUB, PUB, ConnectionRequest, SynRequest,$ $JoinRequest, CloseConnection, Confirmation, Status,$ $Guided, REJOIN\}$. We refer to $J$, $SUB$, $PUB$ as *multicast* messages, i.e., a receiving node may decide to forward them to some other nodes. All other types of messages are meant to be sent and recieved between individual pairs of nodes, and are hence non-multicast (unicast). For any multicast message $m$, the *recipient set* of $m$ at node $N$, $\Gamma_N^m$, is a set of node identifiers that $N$ wishes to deliver the message to. This set is typically computed by a function, $RCPT_N^{T(m)}(m)$, based on the type of message, and $N$'s internal routing information. To make an analogy, in a traditional P/S systems, if $m$ is a subscription message, then the computed recipient set at broker $B$ would contain all the neighboring brokers excluding the sender of $m$. Likewise, if $m$ is a publication message, then the recipient set would contain all local clients and neighboring brokers of $B$ that had previously forwarded a matching subscription message. Furthermore, $N$ is responsible to either directly or indirectly deliver the message to all members of this recipeint set. For any multicast message $m$ at node $N$, a $Confirmation$ message, $Conf^m$, is generated at $N$ indicating successful delivery of $m$ to all non-faulty members of $\Gamma_N^m$.

If $N$ is a successfully joined node in the topology, and $N \in \Gamma_N^m$, we say that $N$ is a *final destination* of $m$, i.e., a node that $m$ must be delivered to. The rationale behind this definition is that in a distributed system that deals with message forwarding, all nodes that need to receive a message must at least be able to identify themselves as such. We formalize the goal of forwarding to deliver any message to all of its non-faulty *final destinations*. For instance, if $p$ is a publication message, any matching *VALID_SUBSCRIBED* subscriber $S$ will compute (so $S$ is non-faulty) its own node identifier in $\Gamma_S^p$, and hence is a final destination of $p$.

### 4.2 Reliability Specification

$J$, $SUB$, and $PUB$ are the message types associated with *join*, *subscribe*, and *publish* operations respectively. Upon arrival to the system, all nodes have to successfully perform a join operation and

[3] This is different from totally ordered multicast.

become a *VALID_JOINED* node. *VALID_JOINED* nodes can then participate in message forwarding, accept other joining nodes, and issue subscriptions/publications. Subscribing clients are mainly concerned with the *subscribe* operation. This is carried out by issuing a $SUB$ message, $s$, to the connected broker. After $s$ is forwarded to all its final destinations the broker will send confirmation message, $Conf^s$, back to the client, indicating successful updates to subscription routing information in the network. At this point, the operation is complete and $s$ becomes a *VALID_SUBSCRIBED* subscription.

The reliable P/S system specification is as follows:

> *For each publication message, $p$, issued at time $t$, the reliable P/S system guarantees duplicate-free delivery of $p$ to all non-faulty subscribers with a matching subscription confirmed before $t$. Furthermore, delivery of any publication $p'$ published after $p$ by the same source can only follow delivery of $p$.*

In the following sections we formally define the consistency properties that lead to realization of the above reliability specification.

### 4.3 Consistency

In a $\delta$-failure-tolerant P/S system, we consider two levels of consistency. $\delta$-Level-1 consistency (or simply Level-1 for short) requires that for any final destination $F$ of a message $m$, all nodes within distance $\delta + 1$ of $F$ directly identify $F$ as a recipient of $m$[4].

$$\delta\text{-Level-1 consistency holds} \Leftrightarrow \left(\forall m, F, F \text{ is a non-faulty } \textit{final destination} \text{ of } m \Rightarrow \begin{pmatrix}\forall N, N \text{ is non-faulty } \wedge d(N,F) \leq \delta + 1 \Rightarrow \\ F \in RCPT_N^{T(m)}(m)\end{pmatrix}\right)$$

On the other hand, $\delta$-Level-2 consistency (or simply Level-2) extends this notion to nodes farther in the network. $\delta$-Level-2 requires that in addition to $\delta$-Level-1 consistency, nodes farther than $\delta + 1$ from all final destinations, $F$, of any message $m$, recursively identify a node on the connecting path to $F$, which is $\delta + 1$ hops closer to $F$. Fig 3 illustrates a final destination $F$ of a message $m$ which is forwarded based on $\delta$-Level-2 consistent routing information.

$$\delta\text{-Level-2 consistency holds} \Leftrightarrow \delta\text{-Level-1 consistency holds} \wedge \left(\forall m, F, F \text{ is a non-faulty } \textit{final destination} \text{ of } m \Rightarrow \begin{pmatrix}\forall N, N \text{ is non-faulty } \wedge d(N,F) > \delta + 1 \Rightarrow \\ P(N,F)[\delta+1] \in RCPT_N^{T(m)}(m)\end{pmatrix}\right)$$

Inherent to these definitions, is that no node in the network is required to have global information about the final destinations of any type of message. In fact, this is an important feature and accompanied with advanced optimization techniques, can improve the scalability of the system. Furthermore, as we see later, some types of multicast messages need only to reach nodes within distance $\delta + 1$ of their sources and hence rely on Level-1 consistency. Other types of multicast messages, need to reach the entire network, and hence rely on the Level-2 consistency property.

[4] This also includes $F$, since $d(F, F) \leq \delta + 1$.

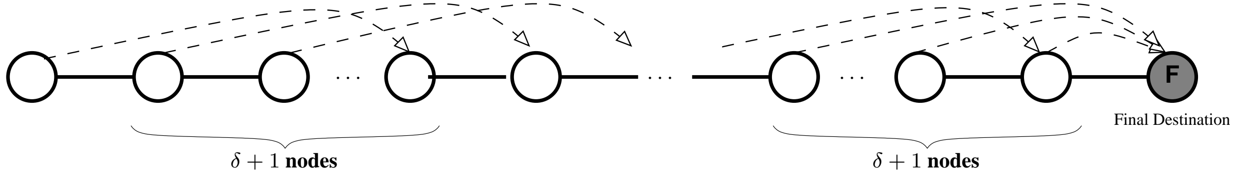



**Figure 3. The requirement of $\delta$-Level-2 consistency for a final destination $F$ (shaded node) of a message $m$. Arrows from node $A$ to $B$, implies that $B \in RCPT_A^{T(m)}$.**

### 4.4 Reliable Forwarding Protocol

The reliable forwarding protocol is executed at *OPERATIONAL* nodes and deals with delivery of multicast message types to non-faulty final destinations. In this section, we describe the algorithm with generality in mind, such that it applies to all types of multicast messages. In subsequent sections, we will elaborate on the details with regard to specific message types, e.g., how the recipient sets are computed, and how forwarding leads to updating nodes' routing information.

Algorithm 1 illustrates the generic reliable forwarding algorithm. In failure-free executions of the system, all forwarding nodes execute this algorithm and messages are always transmitted along the edges of a spanning tree that correspond to the topology links. However, in case of node failures, other algorithms presented in subsequent sections may try to resend some messages over links that are not part of the topology tree. We avoid more details here, and only note that this may lead to arrival of duplicate messages at receiving nodes.

In the generic reliable forwarding algorithm, node $N$, receiving a message $m$, will need to make sure that it has not previously forwarded another copy of $m$. We use a local timestamp object for this purpose, and use a duplicate detection algorithm that is presented in Section 5. If $m$ is a duplicate, there are two possibilities: (*i*) $N$ has previously forwarded the first copy, $m'$, but has not yet sent $Conf^{m'}$. In this case, the sender of the duplicate message must receive a confirmation along with the sender of $m'$ (the first copy). This is taken care of at Line 3 by adding $m.sender$ to a set $Sender_N^m$ which already contains the sender of $m'$. (*ii*) $N$ has already sent confirmation of the first copy, and hence needs to immediately send $Conf^m$ to $m.sender$ (Line 5). To distinguish these two cases, $N$ uses $ProcessingList_N$, which is a FIFO list of previously sent but non-confirmed multicast messages.

What follows after Line 9 deals with the first copy of any message. $N$ initializes $MsgId_N^m$ by a call to $timeStamp_N()$ which returns a locally unique identifier from a totally ordered set. It is used to maintain a local order between arrived messages at node $N$. Furthermore, the input timestamp object of $N$ is updated by the information in $m$ (details in Section 5); $Sender_N^m$ is initialized; and $m$ is added to $ProcessingList_N$. In Line 14, $N$ initializes the recipient set of $m$, $\Gamma_N^m$, by computing $RCPT_N^{T(m)}(m)$. Line 15 corresponds to initialization of $Out_N^m$ by the set of closest non-faulty nodes $M$, that $N$ maintains a connection with ($M \in Connections_N$) and are on the paths to members of $\Gamma_N^m$. In Line 17, $N$ uses a message-type-dependent function to make copies of $m$, and send them to all members of $Out_N^m$. We will elaborate later on $RCPT_N^{T(m)}(m)$ and $make_copy_send_N^{T(m)}(m)$ for each multicast message type.

In the *while* loop of Lines 19–24, $N$ waits to receive $Conf^m$ from all members of $Out_N^m$. If $N$ is a final destination of $m$, it delivers the message, and updates its local routing information in Line

**Algorithm 1**: Generic forwarding algorithm

**Input**: Message $m$ being forwarded at *OPERATIONAL* node $N$

```
1  if ts_obj_N.isDuplicate_N(m) then
2      if m ∈ ProcessingList_N then
3          Sender_N^m ← Sender_N^m ∪ {m.sender}
4      else
5          Send Conf^m to m.sender
6      end
7      Discard m //last arrived copy of m
8      return
9  end
10 MsgId_N^m ← timeStamp_N()
11 ts_obj_N.update(m)
12 Sender_N^m ← {m.sender}
13 ProcessingList_N.append(m)
14 Γ_N^m ← RCPT_N^{T(m)}(m)
15 Out_N^m ← { X | X ∈ Connections_N ∧ (∃F ∈ Γ_N^m, X ∈ P(N,F))∧
                   (∀X' ∈ Connections_N, X' ∈ P(N,X) ⇒ d(N,X) ≤ d(N,X')) }
16 forall X ∈ Out_N^m − {N} do
17     make_copy_send_N^{T(m)}(m, X)
18 end
19 while Out_N^m − {N} ≠ ∅ do
20     forall X ∈ Out_N^m − {N} do
21         Receive Conf^m from X
22         Out_N^m ← Out_N^m − {X}
23     end
24 end
25 if N ∈ Out_N^m then
26     update_routing_information_N^{T(m)}(m)
27 forall X ∈ Sender_N^m do
28     Send Conf^m to X
29 end
30 ProcessingList_N.remove(m)
31 Discard m, msgId_N^m, Out_N^m, Sender_N^m, and Γ_N^m
```

26. In Lines 27–29, $N$ sends $Conf^m$ to the senders of $m$ (including senders of duplicates). Finally, $N$ performs a clean up by removing $m$ from the $ProcessingList_N$, and discarding it along with its associated temporary data structures.

When there are no failed nodes in the system, each node $N$ is connected to all its neighbors in the topology, i.e., $\forall X, d(N,X) = 1 \Rightarrow X \in Connections_N$. Thus, Line 15 of the forwarding algorithm can be rewritten as follows:

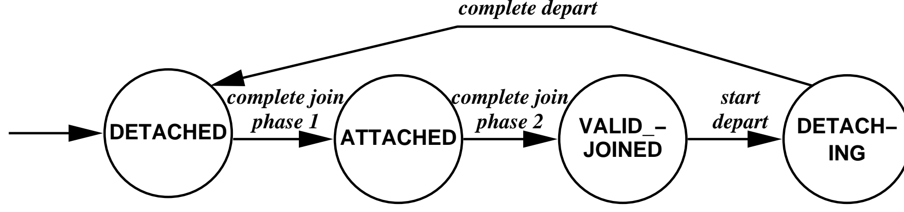



**Figure 4. Join-state transitions of nodes.**

$$Out_N^m \leftarrow \{X | d(X, N) = 1 \wedge (\exists M \in \Gamma_N^m \wedge X \in P(N, M))\}$$

### 4.5 Multicast Messages

Although the forwarding protocol of Section 1 provides a generic algorithm for all multicast messages, there are differences between message types in the computation of the recipient sets, and how routing information is updated. In this section, we elaborate on these details by going through the operations that respectively use $J$, $SUB$, and $PUB$ messages.

### 4.6 The join operation

Nodes that are not part of the topology are initially in the *DETACHED* state. These nodes perform a join operation on a *join point* which is an already joined node in the network[5]. Nodes that successfully complete this operation become *VALID_JOINED*, and we say that $\overleftrightarrow{XY}$ is a *valid* topology link *if and only if* either $X$ or $Y$ has successfully completed a join operation on the other node. The *topology routing information*, $TOP_N$, is a set of topology links, that represents $N$'s view of a portion of the topology falling within distance $\delta + 1$ of $N$. More formally, if $\overleftrightarrow{XY}$ is a valid topology link, and $N$ is a *VALID_JOINED* node, we have:

$$d(X, N) \leq \delta + 1 \wedge d(Y, N) \leq \delta + 1 \Leftrightarrow \overleftrightarrow{XY} \in TOP_N \quad (1)$$

Furthermore, $N$ uses $TOP_N$ to compute topology paths between *VALID_JOINED* nodes at distance $\delta+1$ from itself.

The join operation ensures that Eq.1 holds for all *VALID_JOINED* nodes, and is organized in two phases. Figure 4 illustrates the nodes state transitions during join, and Algorithm 2, illustrates the protocol of the first phase. A *DETACHED* node $N$ initiates the join operation by sending a $JoinRequest$ message, $jReq$, to a join point $M$[6]. $jReq.link$ contains the information about the new link, $\overleftrightarrow{NM}$. The join point $M$, receives $jReq$, and retrieves the topology link. $M$ adds $\overleftrightarrow{NM}$ to $TOP_M$ as a *marked* link to $TOP_M$. Finally, $M$ computes a partial set of its own topology routing information, $PAR_TOP_M^N$, and sends it to back to $N$. $PAR_TOP_M^N$ contains topology links that have endpoints at distance $\delta$ of $M$ (these nodes are hence within distance $\delta + 1$ of $N$). $N$ receives the partial topology routing information and concludes the first phase of join by assigning it to $TOP_N$. At this point $N$ becomes an *ATTACHED* node in the system.

[5]Join points can be discovered by means of a naming service. This discussion is outside the scope of this paper.

[6]In this paper we do not address the discovery of join points.

**Algorithm 2**: *Join* operation – Phase 1

| **Joining node** $N$ | **Join point** $M$ |
|---|---|
| $//N$ is $DETATCHED$ | |
| $//T(jReq) = JoinRequest$ | |
| $jReq.link \leftarrow \overleftrightarrow{NM}^*$ | |
| $\alpha_N^M \leftarrow OPERATIONAL$ | |
| Start fault detection with $M$ | |
| $Connections_N \leftarrow \{M\}$ | |
| Send $jReq$ to $M$ | |
| | Receive $jReq$ from $N$ |
| | $link \leftarrow jReq.link$ |
| | $TOP_M \leftarrow TOP_M \cup \{link\}$ |
| | $PAR_TOP_M^N \leftarrow \left\{ \overleftrightarrow{XY} \middle\vert \begin{array}{l} \overleftrightarrow{XY} \in TOP_M \wedge \\ d(M,X) \leq \delta \wedge d(M,Y) \leq \delta \end{array} \right\}$ |
| | $\alpha_M^N \leftarrow OPERATIONAL$ |
| | Start fault detection with $M$ |
| | $Connections_M \leftarrow Connections_M \cup \{N\}$ |
| | Send $PAR_TOP_M^N$ to $N$ |
| Receive $PAR_TOP_M^N$ from $M$ | |
| $TOP_N \leftarrow PAR_TOP_M^N$ | |
| $//N$ becomes $ATTACHED$ | |

In the second phase of the join operation, $N$ uses the reliable forwarding algorithm (Algorithm 1) to deliver a $J$ message, $j$, to topology nodes within distance $\delta + 1$. This effectively enables these nodes to know about the new joined node by locally storing the newly created topology link. Furthermore, $N$ receives the subscription routing information currently present in the network via invoking a synchronization procedure with the join point $M$. This step is only needed if $N$ is a broker node, and we will describe it in detail in Section 6. Upon completion of these operations, $N$ becomes a *VALID_JOINED* node and can issue subscriptions, publications, and accept new joining nodes.

The $J$ message, $j$, issued as part of join operation is initialized and forwarded as follows: $j.operation = JOIN$, and $j.link = \overleftrightarrow{NM}$. The final destination of $j$ corresponding to joining of $N$ to $M$, are all nodes within distance $\delta$ of the join point, and we have:

$$X \text{ is a final destination of } j \Leftrightarrow d(X, M) \leq \delta$$

The recipient sets at forwarding nodes $B$ are computed as follows:

$RCPT_B^J(j)$ :
**begin**
$TOP_B \leftarrow TOP_B \cup \{j.link^*\}$
**return** $\left\{ X \middle\vert \begin{array}{l} d(X,M) \leq \delta \wedge B \in P(X,M) \wedge \\ (\exists Z, \overleftrightarrow{XZ} \in TOP_B) \end{array} \right\}$
**end**

In other words, the recipient set of $j$ ($j.link = \overleftrightarrow{NM}$) at node $B$ consists of all nodes that are covered by $B$'s topological routing information, and have a $\delta$-topology-path to $M$ that goes through $B$. Furthermore, the $\overleftrightarrow{NM}^*$ is added to $TOP_B$ (as a marked link).

A call to $make_copy_send^J_B(j, M)$ simply sends $j$ to $M$, and is as follows:

$make_copy_send^J_N(m, M)$ :
**begin**
  Send $j$ to $M$
**end**

Finally, $update_routing_information^J_B(j)$ is called by all final destinations, $B$, to update the local topology routing information:

$update_routing_information^J_B(j)$ :
**begin**
  **if** $j.operation = JOIN$ **then**
    $TOP_B \leftarrow TOP_B - \{j.link^*\} \cup \{j.link\}$
**end**

In effect, this function adds the topology link, $j.link$, to $TOP_B$ replacing any marked copy of it.

### 4.7 The subscribe operation

The subscribing clients are the only nodes in the P/S system that perform this operation by generating a subscription message $s$ of type SUB. The message contains $s.from$ and $s.predicates$ fields: the former is initialized with the source identifier, and the latter specifies the subscribing source's interest in receiving publications. All nodes in the network must be aware of this subscription in order to (prevent the anomalous scenario presented in Section 3) and be able to forward matching publications towards the subscriber. Thus, the final destination of any subscription $s$ is as follows:

$$\forall N, s, N \text{ is } \textit{VALID_JOINED} \Leftrightarrow N \text{ is a final destination of } s$$

At node $N$, the recipient set of $s$ is computed as follows:

$RCPT^{SUB}_N(s)$ :
**begin**
  **return** $\left\{X \middle| \; N \in P(X, s.from) \wedge (\exists Z, \overleftrightarrow{(s.from)Z} \in TOP_N)\right\} \cup \{N\}$
**end**

In other words, the recipient set of $s$ at node $N$ contains any node in the portion of the topology that is covered by $TOP_N$, and connects to $s.from$ via a path through $N$. These nodes are in the subtrees rooted at $N$, other than the subtree that send $s$ to $N$. Node $N$ forwarding $s$ calls $make_copy_send^{SUB}_N(s, M)$ to

send a copy of $s$ to $M$, and is as follows:

$$
\begin{array}{l}
make_copy_send_N^{SUB}(s, M): \\
\textbf{begin} \\
\quad s' \leftarrow s.clone() \\
\quad \textbf{if } d(N, M) + d(N, s.from) \leq \delta + 1 \textbf{ then} \\
\quad\quad s'.from \leftarrow s.from \\
\quad \textbf{elseif } d(N, M) + d(N, s.from) > \delta + 1 \textbf{ then} \\
\quad\quad s'.from \leftarrow P(N, s.from)[d(N, M) - (\delta + 1)] \\
\quad \textbf{end} \\
\quad \text{Send } s' \text{ to } M \\
\textbf{end}
\end{array}
$$

Intuitively, if $M$ is more than $\delta + 1$ hops away from $s.from$, it will receive a copy $s'$, such that $s'.from$ is a node on the connecting path between $s.from$ and $M$, and at distance $\delta + 1$ of $M$. On the other hand, if $s.from$ is closer to $M$ than $\delta + 1$, then the message that is sent to $M$ is exactly similar to $s$ (except for $s.sender$).

$update_routing_information_N^{SUB}(s)$ is called by all final destinations, $N$, to update the local subscription routing information as follows:

$$
\begin{array}{l}
update_routing_information_N^{SUB}(s): \\
\textbf{begin} \\
\quad SUBS_SET_N \leftarrow SUBS_SET_N \cup \{\langle s.predicates, s.from \rangle\} \\
\textbf{end}
\end{array}
$$

### 4.8 The publish operation

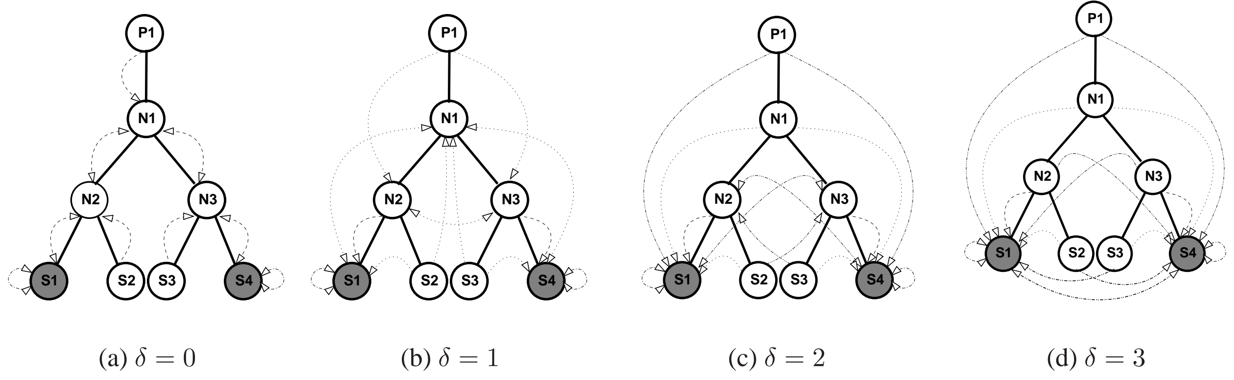
The publishing clients are the only nodes in the P/S system that perform this operation by generating $PUB$ messages. In contrast to $J$ and $SUB$ message types, $PUB$ messages are only forwarded. A $PUB$ message, $p$, contains $p.attrValSet$ which is a set of attribute-value pairs used to evaluate the predicates of subscriptions. Furthermore, the final destinations of $p$, is a dynamic subset of all ***VALID_SUBSCRIBED*** subscriptions that match $p$. For publication message $p$, and subscriber $S$, we formalize this as follows:

$$
\forall S, p, S \text{ is a final destination of } p \Leftrightarrow \begin{pmatrix} \exists s, s \text{ is VALID_SUBSCRIBED} \wedge s.source = S \wedge \\ matches(p.attrValSet, s.predicates) \end{pmatrix}
$$

Furthermore, at any node $N$, the recipient set of $p$ is computed based on the subscription routing information, $SUBS_SET_N$, as follows:

$$
\forall N, p, X \in RCPT_N^{PUB}(p) \Leftrightarrow \big(\exists \langle pred, X \rangle \in SUBS_SET_N \wedge matches(p.attrValSet, pred)\big)
$$

To give further intuition about the recipient set of a $PUB$ message, Figure 5 illustrates a simple network of one publisher, $P$, and four subscribers, $S1$, $S2$, $S3$, and $S4$. An arrow, from node $A$, to




(a) $\delta = 0$ (b) $\delta = 1$ (c) $\delta = 2$ (d) $\delta = 3$

**Figure 5. $RCPT_X^{PUB}(p)$ of a publication $p$ from $P1$ matching subscriptions from $S1$ and $S4$ for different values of $\delta$. An arrow from $A$ to $B$ implies $B \in RCPT_A^{PUB}(p)$.**

| Node | $RCPT_{node}^{PUB}(p)$ | | | |
|---|---|---|---|---|
| | $\delta = 0$ | $\delta = 1$ | $\delta = 2$ | $\delta = 3$ |
| $P1$ | $\{N1\}$ | $\{N2, N3\}$ | $\{S1, S4\}$ | $\{S1, S4\}$ |
| $N1$ | $\{N2, N3\}$ | $\{S1, S4\}$ | $\{S1, S4\}$ | $\{S1, S4\}$ |
| $N2$ | $\{S1, N1\}$ | $\{S1, N3\}$ | $\{S1, S4\}$ | $\{S1, S4\}$ |
| $N3$ | $\{S4, N1\}$ | $\{S4, N1\}$ | $\{S1, S4\}$ | $\{S1, S4\}$ |
| $S1$ | $\{S1, N2\}$ | $\{S1, N1\}$ | $\{S1, N3\}$ | $\{S1, S4\}$ |
| $S2$ | $\{N1\}$ | $\{S1, N1\}$ | $\{S1, N3\}$ | $\{S1, S4\}$ |
| $S3$ | $\{N3\}$ | $\{S4, N1\}$ | $\{S4, N2\}$ | $\{S1, S4\}$ |
| $S4$ | $\{S4, N3\}$ | $\{S4, N1\}$ | $\{S4, N2\}$ | $\{S1, S4\}$ |

**Table 1. Computed recipients sets of all nodes in Figure 5 for different values of $\delta$.**

$B$ illustrates the fact that for a publication $p$ that is being handled at $A$, $B \in RCPT_A^{T(p)}(p)$. In this particular example, the publication being handled only matches subscriptions from $S1$ and $S4$. The Figure illustrates the situation under various values of $\delta$. Table 1 summarizes the computed recipient sets at different nodes.

$PUB$ messages do not contribute to the routing information at nodes, and $update_routing_information_N^{PUB}(p)$ simply delivers the publication message $p$ to the subscriber (since $N$ is a final destination of $p$).

## 5 Fault-Tolerance

We now consider the possibility of node failures. We define the $\delta$-*fault-tolerance* property of a P/S system, as its ability to provide the P/S reliability guarantees (Section 4.2), and perform all operations in presence of up to $\delta$ non-*OPERATIONAL*[7] nodes. We configure the system to establish $\delta$-Level-1 consistent topology routing information and $\delta$-Level-2 consistent subscription routing information. Using

[7] *FAILED* or *RECOVERING*.

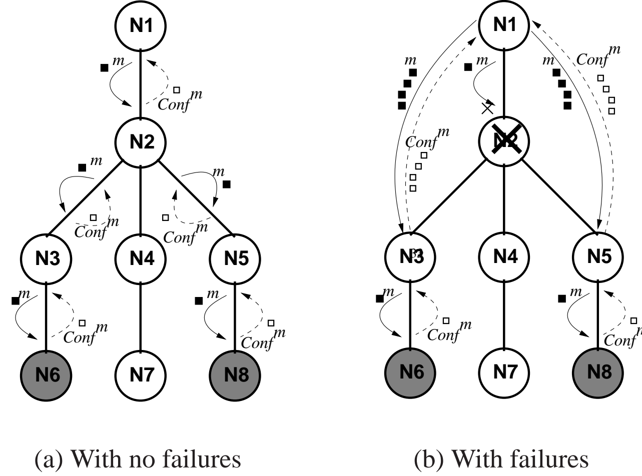



**Figure 6. Fault-tolerance forwarding bypasses faulty nodes (filled circles represent final destinations).**

the replicated nature of these consistency properties, nodes can identify and connect to their closest non-faulty nodes by skipping any chain of at most $\delta$ consecutive failed nodes. The (non-confirmed) messages are then (re-)send towards final destinations via the new routes that bypass failed nodes. Figure 6 illustrates the general idea. As we prove in Appendix A, using this approach all system operations (including subscribe and publish) can be successfully carried out with up to $\delta$ simultaneous failures and the P/S reliability requirements are met.

If there are no failures, *VALID_JOINED* nodes maintain communication links to their immediate neighbors in the topology. Furthermore, communication links are monitored by failure detectors, and failure of any node $M$ detected by any connecting peer $N$ invokes $failureDetected_N(M)$. This algorithm is illustrated in Algorithm 3. In this section, we assume $N$ is in the *OPERATIONAL* state, and postpone the case in which $N$ is a *RECOVERING* node to Section 6. $failureDetected_N$ handles the failure of $M$ in two stages. In the first stage, Lines 1–12, $N$ updates its communication link information, and further computes a set of nodes, $NewPeers$, to connect to. Each node $X \in NewPeers$ is an immediate neighbor of $M$ that is farther away from $N$ than $M$. If there is no previous communication link to $X$, $N$ assumes that $X$ is *OPERATIONAL*, and establishes a new communication link to $X$ by sending a $ConnectionRequest$ message. At the receiver, a $ConnectionRequest$ message is handled according to Algorithm 4.

The next stage to handle failure of $M$ (Lines 17–27) deals with retransmission of any message, $m$, that was previously sent to $M$ but has not yet been confirmed, and we have: $M \in Out_N^m$, and $m \in ProcessingList_N$. $NewOut_N^m$ contains the new peers of $N$ that fall on the paths to members of $\Gamma_N^m$, and hence can be used to deliver $m$ to final destinations. For all these message in order, $N$ updates $Out_N^m$ and send the message to the nodes in $NewOut_N^m$.

**Duplicate Elimination:** Unconfirmed messages that are re-sent by node $N$ to bypass a failed node $M$ may arrive at the receiving peers as duplicate messages. This is caused by the fact that $M$ had the

**Algorithm 3**: $failureDetected_N(M)$

**Input**: Failed node $M$ (detected by node $N$)

1 **begin**
2 $Op_N^M \leftarrow FAILED$
3 Stop failure detection with $M$
4 $Connetions_N \leftarrow Connections_N - \{M\}$
5 $NewPeers \leftarrow \{X | \overleftrightarrow{XM} \in TOP_N \wedge M \in P(N,X)\}$
6 **if** $\alpha_N^N = OPERATIONAL$ **then**
7 **forall** $X \in NewPeers \wedge X \notin Connections_N$ **do**
8 $\alpha_N^X \leftarrow OPERATIONAL$
9 Start failure detection with $X$
10 $Connections_N \leftarrow Connections_N \cup \{X\}$
11 Send $ConnectionRequest$ to $X$
12 **end**
13 **else if** $\alpha_N^N = RECOVERING$ **then**
14 **forall** $X \in NewPeers$ **do**
15 $synchronizeTo_N(X)$
16 **end**
17 $Synch_N \leftarrow Synch_N - \{M\}$
18 **end**
19 **while** $(m \leftarrow ProcessingList_N.getNext())$ **do**
20 **if** $M \in Out_N^m$ **then**
21 $NewOut_N^m \leftarrow \left\{X \middle| X \in NewPeers_N \wedge (\exists Y \in \Gamma_N^m, X \in P(N,Y))\right\}$
22 $Out_N^m \leftarrow Out_N^m \cup NewOut_N^m - \{M\}$
23 **forall** $X \in NewOut_N^m$ **do**
24 $make_copy_send^{T(m)}(m, X)$
25 **end**
26 **end**
27 **end**
28 **end**

chance to forward the message to a number of nodes before its failure. As a result, duplicate detection and elimination is a key element in our approach, and in this section, we propose two different algorithms for this purpose.

Our first approach relies on the *globally* unique message identifiers assigned by the source. Since identifiers are totally ordered, each node $N$, uses a naïve timestamp object $nts_obj_N$, to keep track of the message identifier of the "most_recent" message received from *each individual* source. A message is a duplicate, if its $sourceMsgId$ does not succeed the timestamp associated with the source node. If the message is not a duplicate, the timestamp of the source node is updated by storing the new $sourceMsgId$.

However, this approach scales poorly, and may require each node to keep track of messages from an excessive number of sources. As a solution to this problem, we propose an enhanced approach in which

**Algorithm 4**: $handleConReq_N(ConnectionRequest\ c)$

**Input**: $ConnectionRequest$ message $c$
**begin**
  $M \leftarrow c.sender$
  **if** $M \notin Connections_N$ **then**
    Start failure detector with $M$
    $Connections_N \leftarrow Connections_N \cup \{M\}$
  **end**
**end**

the timestamp object only maintains information about the most recent message from nodes within distance $\delta + 1$.

Each multicast message $m$, has an additional field $m.visited$, which is a list of at most $\delta + 1$, $\langle nodeId, nodeMsgId \rangle$ pairs corresponding to the last $\delta + 1$ nodes that forwarded $m$. If $m$ bypasses failed nodes, $nil$ values are appended to the visited list. The algorithm to update this list is carried out as part of the "send" operation, and is illustrated below:

$i \leftarrow 1$
$append^{\delta+1}(\langle N, MsgId_N^m \rangle, m.visited)$
**while** $i < d(N, M)$ **do**
  $append^{\delta+1}(\langle P(N, M)[i], nil \rangle, m.visited)$
  $i \leftarrow i + 1$
**end**
$Send\, m\, to\, M$

Similar to the previous approach, receiving nodes maintain a timestamp object, $ts_obj$, and update it with the information in the $m.visited$ field of non-duplicate messages. $ts_obj_N.update(m)$ is as follows:

**forall** $\langle nodeId, msgId \rangle \in m.visited \wedge msgId \neq nil$ **do**
  $ts_obj_N.most_recent(nodeId) \leftarrow msgId$
**end**

Finally, $m$ received at node $N$, is not a duplicate *if and only if*, $m.visited$ contains pairs that succeed the pair with identical $nodeId$ in $ts_obj_N$. $ts_obj_N.isDuplicate(m)$ is as follows:

**forall** $\langle nodeId, msgId \rangle \in m.visited \wedge msgId \neq nil$ **do**
  **if** $msgId \not\succ ts_obj_N.most_recent(nodeId)$ **then**
    **Return** $True$
**end**
**Return** $False$

## 6 Recovery

Failed nodes are likely to miss messages that contain routing information. Thus, the recovery procedure typically involves synchronization with several neighboring nodes, and ensures that the routing

information of the recovered nodes is consistent. We assume that a recovering node $R$ retains the same node identifier (and address) it used prior to failure, and it can restore its topology routing information, $TOP_R$. While synchronization is in progress, the node participates in message forwarding by the use of $Guided$ messages and upon successful completion of the recovery procedure, the node becomes ***OPERATIONAL***. A $Guided$ message, $g^m$, encapsulates a multicast message $m$, such that all fields of $m$ are accessible in $g^m$. Furthermore, $g^m.to$ is a set of $(\delta - 1)$-paths that contain additional information on how to forward $m$. The routing information at a ***RECOVERING*** node is inconsistent, and may lead to message loss. On the other hand, if the synchronization point, $S$, is ***OPERATIONAL*** (we elaborate on the case where $S$ is itself ***RECOVERIN*** in Section 6.2), it has all the information about the final destinations of a message within distance $\delta + 1$ of $S$. Thus, for any multicast message being sent from a synchronization point, $S$, to a ***RECOVERING*** node, $R$, all calls to $make_copy_send_S^{T(m)}(m, R)$ are preceded by the following:

**if** $\alpha_S^R = RECOVERING$ **then**
  $g^m \leftarrow$Guided$(m)$
  $g^m.to \leftarrow \{P(R, X) | X \in \Gamma_S^m \wedge R \in P(S, X)\}$
**end**
$make_copy_send_S^{T(m)}(g^m, R)$

This ensures that $R$ will have enough information on how to further forward $m$. Handling of $Guided$ messages at ***RECOVERING*** nodes is presented in Section 6.1. *OPERATIONAL* nodes on the other hand, extract the encapsulated multicast message, and invoke the reliable forwarding algorithm as presented in Section 4.

Algorithm 5 illustrates the recovery algorithm executed by recovering node $R$. $TOP_R$ is initially used to identify a set of synchronization points, $Synch_R$, to receive the routing information from, and $TOP'_R$ and $SUBS_SET'_R$ temporarily contain the received routing information. Once recovery is complete, these sets along with the received $Guided$ messages during recovery are used to initialize $TOP_R$ and $SUBS_SET_R$. As the last step, $R$ notifies all its connected peers about its transition to the ***OPERATIONAL*** state. The synchronization algorithm, is presented in Algorithm 6. In this algorithm, the recovering node $R$ initializes a communication link with the synchronization point, $N$, and sends a $SynRequest$ message. A $Status$ reply message and the topology and subscription routing information are then received from $N$. Finally, if synchronization with $N$ is completed successfully, $TOP'_R$, $SUBS_SET'_R$, and $Synch_R$ are updated. The node also forwards a $REJOIN$ message that is forwarded similar to $J$ messages to distance $\delta + 1$ of the source. However, $REJOIN$ messages do not update the topology routing information, at receiving nodes, and its sole purpose is to make sure that all the messages that have been resent are received within distance $\delta + 1$ of $R$[8]. At the end, $R$'s transition to ***OPERATIONAL*** takes place.

Algorithm 7 illustrates the algorithm executed at a synchronization point, $N$, upon receiving a $SynRequest$ message, $s$, from a recovering node $R$. It is possible that $N$ was previously connected to $R$ and receives $m$ without having detected $R$'s failure. As a result, $N$ tears down the connection, and stops the failure detector. Furthermore, there might be messages, $m$, that were sent to $R$ and are not yet confirmed (i.e., $R \in Out_N^m$ and $m \in ProcessingList_N$). $N$ initializes the FIFO list $PrevOut_N^R$ with all such messages

[8] $REJOIN$ is used to deliberately delay the transition of $R$ to *OPERATIONAL*. This enables us to formalize the invariants and ensure guaranteed behavior. For more information refer to the correctness proof of the duplicate detection mechanism in Section 5.

**Algorithm 5**: Recovery algorithm at node $R$

**Input**: Restored topology routing information, $TOP_R$

1 $//\alpha_R^R = RECOVERING$
2 $Synch_R \leftarrow \{X|\overleftrightarrow{XR} \in TOP_R\}$
3 $TOP'_R \leftarrow \emptyset$
4 $SUBS_SET'_R \leftarrow \emptyset$
5 **forall** $N \in Synch_R$ **do**
6 $\quad synchronizeTo_R(N)$
7 **end**
8 **while** $Synch_R \neq \emptyset$ **do**
9 $\quad$ wait
10 **end**
11 $TOP_R \leftarrow TOP'_R$
12 $SUBS_SET_R \leftarrow SUBS_SET'_R$
13 Apply $Guided_TOP_R$ in-order to $TOP_R$
14 Apply $Guided_SUB_R$ in-order to $SUBS_SET_R$
15 $st.status \leftarrow OPERATIONAL$ //T(st)=$Status$
16 **forall** $X \in Connections_R$ **do**
17 $\quad$ Send $st$ to $X$
18 **end**
19 Reliably forward $rj$ //T(rj)=$REJOIN$
20 **while** $Out_R^{rj} \neq \emptyset$ **do**
21 $\quad$ wait
22 **end**
23 $\alpha_R^R \leftarrow OPERATIONAL$

while preserving the order (Lines 8–11). In Line 13, $PAR_TOP_N^R$ is a subset of $TOP_N$ and contains unmarked topology links with endpoints farther than $N$ from $R$ and within distance $\delta$ to $N$. These endpoints are hence within distance $\delta + 1$ to $R$. In Line 14, $PAR_SUB_N^R$ is a set of predicate-$nodeId$ pairs containing the subscription routing information that meets $\delta$-Level-2 consistency requirements from $R$'s point of view. In Lines 15–20, $N$ establishes a connection to $R$, and sends its operational state along with the routing information to $R$. A P/S system tends to have large number of clients, and hence $PAR_SUB_N^R$ is likely to be a very large set. In practice, transmission of this message may take a long time, and hence result in a slow down of all the communication between these nodes. As a result, $N$ sends $PAR_TOP_N^R$ and $PAR_SUB_N^R$ as low priority messages. All the normal priority messages are guaranteed to be transmitted first. Messages in $PrevOut_N^R$ are sent afterwards via $Guided$ messages. In Lines 24 and 25, $N$ awaits the status reply message from $R$ signifying completion of recovery, and updates $\alpha_N^R$ accordingly.

$PAR_TOP_S^R$ and $PAR_SUB_S^R$ computed at the synchronization point $S$, correspond to a snapshot of the routing information at $S$. Recovery may take a long time and hence there may be many messages arriving at $S$ that need to be forwarded towards subtrees of the topology rooted at $R$. The $Guided$ messages effectively allow $R$ to participate in message forwarding. In contrast to regular multicast

**Algorithm 6**: $synchronizeTo_R(N)$

**Input**: Node $N$ to synchronize with
1 **begin**
   // $\alpha_R^R = RECOVERING$
2 $Synch_R \leftarrow Synch_R \cup \{N\}$
3 **if** $N \notin Connections_R$ **then**
4 $\alpha_R^N \leftarrow OPERATIONAL$
5 Start failure detection with $N$
6 $Connections_R \leftarrow Connections_R \cup \{N\}$
7 **end**
8 Send $SynRequest$ to $N$
9 Receive $Status$ from $N$
10 $\alpha_R^N \leftarrow Status.status$
11 Receive $PAR_TOP_N^R$ from $N$
12 Receive $PAR_SUB_N^R$ from $N$
13 $TOP'_R \leftarrow TOP'_R \cup PAR_TOP_N^R$
14 $SUBS_SET'_R \leftarrow SUBS_SET'_R \cup PAR_SUB_N^R$
15 $Synch_R \leftarrow Synch_R - \{N\}$
16 **end**

messages, $R$ appends the routing information in $Guided$ messages to $Guided_TOP_R$ or $Guided_SUB_R$. Once synchronization ends, this information is applied to $TOP_R$ or $SUBS_SET_R$ accordingly. This ensures that the routing information at $R$ reflects not only the snapshots that were taken some while ago at the synchronization points, but also the most current information in the messages that were forwarded afterwards.

### 6.1 Handling $Guided$ Messages

A ***RECOVERING*** node $R$, discards all multicast messages and processes $Guided$ messages only. $Guided$ messages are used to convey the information about how to forward a message to a ***RECOVERING*** node that may potentially have inconsistent routing information. If $g^m$ is a $Guided$ message, $R$, executes a modified version of Algorithm 1 by first extracting m. The recipient set, and $Out_R^m$ are computed as follows (path information is extracted from $g^m.to$):

$$\Gamma_R^m \leftarrow \{X | \exists Y, P(R,Y) \in g^m.to \wedge X \in P(R,Y)\}$$
$$Out_R^m \leftarrow \left\{ X \middle| X \in Connections_R \wedge \begin{pmatrix} \exists Y \in \Gamma_R^m \wedge X \in P(R,Y) \wedge \\ (\forall Z \in Connections_R, Z \in P(R,Y) \Rightarrow d(R,X) \leq d(R,Z)) \end{pmatrix} \right\}$$

The rest of the generic forwarding algorithm remains unchanged, except for the fact that $update_routing_informat$ appends the routing information in $m$ to the FIFO lists $Guided_TOP_R$ and $Guided_SUB_R$. These will be applied after synchronization is complete (i.e., when $Synch_R = \emptyset$).

While a recovering node is synchronizing with a synchronization point, there are chances that either one fails before completion. Once detected by the other party, the $failureDetected$ (Algorithm 3) is invoked. We discussed this algorithm in Section 5 for the case where $N$ is ***OPERATIONAL***. In this

**Algorithm 7**: $handleSynReq_N(SynRequest\ s)$

**Input**: $SynRequest$ message $s$

```
1  begin
2    R ← s.sender
3    PrevOut_N^R ← ∅
4    if R ∈ Connections_N then
5     //Possibly failure of R was not detected
6       Stop fault detection with R
7       Connections_N ← Connections_N − {R}
8       while (m ← ProcessingList_N.getNext()) do
9          if R ∈ Out_N^m then
10            PrevOut_N^R ← PrevOut_N^R.append(m)
11      end
12   end
```

13 $PAR_TOP_N^R \leftarrow \left\{ \overleftrightarrow{XY} \middle| \begin{array}{l} \overleftrightarrow{XY} \in TOP_N \cup Guided_TOP_N \wedge \\ \overleftrightarrow{XY} \text{ not marked } \wedge d(X,N) < d(X,R) \wedge \\ d(X,R) \leq \delta+1 \wedge d(Y,R) \leq \delta+1 \end{array} \right\}$

14 $PAR_SUB_N^R \leftarrow \left\{ t = \langle p, Z \rangle \middle| \begin{pmatrix} t \in \begin{pmatrix} SUBS_SET_N \cup \\ Guided_SUB_N \end{pmatrix} \wedge \\ d(Z,N) < d(Z,R) \wedge \\ d(Z,R) \leq \delta+1) \end{pmatrix} \vee \begin{pmatrix} \exists t' = \langle p, Z' \rangle \wedge \\ t' \in \begin{pmatrix} SUBS_SET_N \cup \\ Guided_SUB_N \end{pmatrix} \\ \wedge d(R,Z') > \delta+1 \wedge \\ Z = P(R,Z')[\delta+1] \end{pmatrix} \right\}$

```
15   Op_N^R ← RECOVERING
16   Start fault detection with R
17   Connections_N ← Connections_N ∪ {R}
18   st.status ← Op_N^N //T(st)=Status
19   Send st to R
20   Send PAR_TOP_N^R and PAR_SUB_N^R as low priority to R
21   while (m ← PrevOut_N^R.getNext()) do
22      make_copy_send^{T(m)}(m, R)
23   end
24   Receive stRply from R //T(stRply)=Status
25   α_N^R ← stReply.status
26 end
```

section, we focus on the scenario where a ***RECOVERING*** node, $N$ detects failure of a synchronization point $M$. In this case, $N$ has to identify a set of new peers to connect to and synchronize with (Lines 13–18 of Algorithm 3). Furthermore, as we see later in this section, a recovering node participates in message forwarding by receiving $Guided$ messages. As a result, any non-confirmed messages that were previously forwarded to $M$ ($M$ is now failed), are re-sent to such that to bypass $M$ (Lines 19–27).

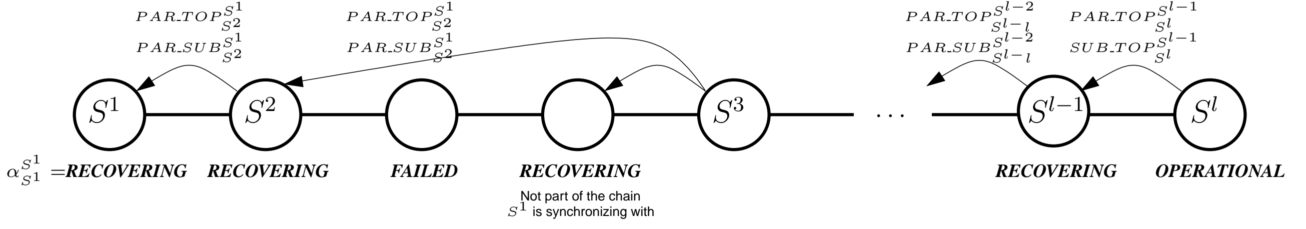



**Figure 7. Chain of *RECOVERING* node that all receive part of their routing information from *OPERATIONAL* node $S^l$.**

### 6.2 Recovery Through a Chain of *RECOVERING* Nodes

In this section, we consider the case where a node $S^1$ is synchronizing with a synchronization point $S^2$, while $S^2$ is itself a *RECOVERING* node. Furthermore, $S^2$ must have other synchronization point on each direction upstream. Considering $S^3$ to be $S^2$'s synchronization point on one direction, and further applying the same arguments. After at most $\delta - 1$ times application, we arrive at $S^l$ where $l \leq \delta$. We must either arrive at an *OPERATIONAL* synchronization point, or conclude that there is no farther node to proceed. This implies that in *each direction* $R$ is connected to an *OPERATIONAL* node (if it exists), $S^l$, through a chain of zero or more synchronizing *RECOVERING* nodes, $S^2 \cdots S^{l-1}$. Furthermore, the topology routing information that $R$ needs to receive from $O$ is only a subset of the toplogy routing infromation $TOP_O^{S^l}$ that $O$ sends to its closest *RECOVERING* node, $S^{l-1}$. Thus, $R$ can correctly receive this information through the chain of *RECOVERING* nodes. This is illustrated in Fig 7.

Each node, $S^i$ on this chain is *RECOVERING* and receives $PAR_TOP_{S^{i+1}}^{S^i}$ from $S^{i+1}$. Furthermore, it computes $PAR_TOP_{S^i}^{S^{i-1}}$ as follows:

$$PAR_TOP_{S^i}^{S^{i-1}} \leftarrow \left\{ X \middle| \begin{array}{l} X \in \begin{pmatrix} PAR_TOP_{S^{i+1}}^{S^i} \cup \\ Guided_TOP_{S^{i+1}} \end{pmatrix} \wedge \\ d(X, S^i) \leq \delta \end{array} \right\}$$

Finally, $S^{i-1}$ receives this information as part of its synchronization with $S^i$.

The subscription routing information that is sent during synchronization is similar. Any node $S^i$ ($1 \leq i < l$) receives $PAR_SUB_{S^{i+1}}^{S^i}$ from $S^{i+1}$ and computes $SUBS_SET^{s^{i-1}} - S^i$ as follows:

$$SUBS_SET_{S^i}^{s^{i-1}} \leftarrow \left\{ t = \langle X, pred \rangle \middle| \begin{array}{l} \begin{pmatrix} t \in \begin{pmatrix} PAR_SUB_{S^{i+1}}^{S^i} \cup \\ Guided_SUB_{S^{i+1}} \end{pmatrix} \wedge \\ d(X, S^{i+1}) \leq \delta + 1 \end{pmatrix} \vee \\ \begin{pmatrix} \exists t' = \langle Y, pred \rangle \wedge, t' \in \begin{pmatrix} PAR_SUB_{S^{i+1}}^{S^i} \cup \\ Guided_SUB_{S^{i+1}} \end{pmatrix} \wedge \\ d(Y, S^i) > \delta + 1 \wedge X = P(S^{i+1}, Y)[\delta + 1] \end{pmatrix} \end{array} \right\}$$

Similar to the case where the synchronization point is *OPERATIONAL*, this information is sent with *low* priority. Furthermore, as illustrated in Fig 7, not all *RECOVERIN* nodes on $P(S^1, S^l)$ are necessarily part of this chain. This may occur when the skipped *RECOVERING* node starts its recovery procedure, after

the chain has formed. Finally, note that the $Guided$ messages, use communication links ($Connections$), and do not necessarily follow the synchronization chains.

# 7 Related Work

Related work falls into the following two categories, reliable group multicast, and reliable P/S systems. We discuss them in turn below. The problem of reliable publication delivery is to some extent relevant to the traditional reliable group multicast problems. However, in practical P/S systems with a high publication rate, it is very costly to enforce properties such as virtual synchrony [3], or total ordering of publication delivery. This is mainly due to the fact that each publication is delivered based on individual subscriber's interest, and maintenance of a shared *group view* for this level of dynamism among a large number of clients is infeasible. Thus, we believe that per-source in-order delivery, as required by our reliability specification (Section 4), provides a reasonable balance between application requirements and feasibility of the implementation.

In the reliable P/S literature, we give an overview of some of the most important related work. The Gryphon distributed P/S system [2] introduces the concept of virtual brokers to tolerate failures. Each virtual broker is a set of physical machines which act as identical replicas. Upon failure of a machine, its replicas (if any) take over and re-established the affected publication flows. For each propagation tree, this requires a special deployment procedure to map virtual brokers to physical machines. In contrast, our approach does not need special deployment procedure, and adopts a configurable and completely transparent replication scheme that takes place *along* the propagation tree.

Snoeren et al. [9] propose an approach to build a $\delta$-fault tolerant P/S system, by constructing $\delta+1$ disjoint forwarding paths between subscribers and publishers, and forwarding publications on all redundant paths. Thus, in presence of $\delta$ concurrent failures, at least one forwarding path is not affected. However, this approach incurs high bandwidth consumption. In contrast, when there are no failures our approach sends publications along *a single* shortest path towards each subscriber. Furthermore, messages are only re-sent when a failure is initially detected (and not afterwards).

In Hermes [8] the routing information at P/S brokers is *soft* state to deal with failures. This is done by having clients periodically renew subscriptions. However, unless the clients subscribe to *all* broker nodes simultaneously, this approach does not prevent loss of matching publications. XNET [4] proposes two schemes, *crash/recover* and *crash/failover*, to deal with broker failures. However, these approaches are unable to handle multiple simultaneous failures. Cugola et al. [6] also present an algorithm to improve the efficiency of reconfiguration in highly dynamic P/S networks. However, the proposed *reconfiguration path* approach only strives to minimize the publication loss during a reconfiguration process, rather than preventing it. Epidemic (gossip) algorithms have been applied to P/S systems [5], in an attempt to improve the reliability of highly dynamic systems. However, no strict guarantees in terms of publication delivery are provided. We believe that in a fairly stable environment, our algorithms enforce a much stricter reliability guarantee even in presence of failures.

# 8 Conclusions

In this paper, we studied the reliability and fault-tolerant aspects of distributed P/S systems. We formally defined a routing information consistency property and presented algorithms to ensure reliable delivery of different types of messages by establishing and maintaining consistency. Furthermore, in

order to deal with up to $\delta$ simultaneous node failures, we proposed an approach that uses replication of routing information to bypass failed nodes.

In our approach, the quality of service of the failure detectors plays a crucial role in ensuring fast reaction to failures. While this is a common requirement of almost all approaches, our system readily reacts to the detected failure by re-sending messages that are possibly lost due to the failure. Furthermore, confirmation mechanism prevents the anomalous scenarios presented in Section 3, and ensures that the reliability guarantees are met.

In this paper, we also theoretically proved that using the set of algorithms provided in this paper, all system operations can be carried out successfully and the reliability guarantees remain uncompromised, unless the number of non-*OPERATIONAL* nodes exceed $\delta$. Furthermore, we prove that the failure detector mechanism is able to detect duplicate arrival of messages using only information about the last $\delta + 1$ hops visited by a message, even in scenarios where nodes quickly oscillate between *FAILED*, *RECOVERING*, and *OPERATIONAL* states.

## References


[1] IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries, NY: 1990.

[2] S. Bhola, R. E. Strom, S. Bagchi, Y. Zhao, and J. S. Auerbach. Exactly-once delivery in a content-based publish-subscribe system. In *DSN*, 2002.

[3] K. P. Birman and T. A. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, 5(1):47–76, 1987.

[4] R. Chand and P. Felber. XNET: a reliable content-based publish/subscribe system. In *SRDS '04: Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems (SRDS'04)*. IEEE Computer Society, 2004.

[5] P. Costa, M. Migliavacca, G. P. Picco, and G. Cugola. Introducing reliability in content-based publish-subscribe through epidemic algorithms. In *DEBS*, 2003.

[6] G. Cugola, D. Frey, A. Murphy, and G. Picco. Minimizing the reconfiguration overhead in content-based publish-subscribe, 2003.

[7] E. Fidler, H.-A. Jacobsen, G. Li, and S. Mankovski. The PADRES distributed publish/subscribe system. *ICFI*, 2005.

[8] P. R. Pietzuch and J. Bacon. Hermes: A distributed event-based middleware architecture. In *ICDCS Workshops*, 2002.

[9] A. C. Snoeren, K. Conley, and D. K. Gifford. Mesh-based content routing using xml. In *SOSP*, 2001.

# A Correctness

In this section, we first present a Key Lemma which we use to prove some of the most important properties of our approach, e.g., in-order, duplicate-free delivery. At the end, we prove the correctness of the join, subscribe, and publish operations, and conlude that the reliability specification is met under up to $\delta$ node failures.

A node $N$ is skipped, *if and only if*, a node $A$ sends a message $m$ to node $B$, such that $N \in P(A, B) - \{A, B\}$. The *Key Lemma* and *Key Corollary* below prove that an *OPERATIONAL* node is never skipped. For the purpose of the proves *only* we refer to a recovering node that has successfully synchronized and is just about to send the $REJOIN$ message as *OPERATIONAL**. The scope of this naming is while the node is executing Lines 19–22 of Algorithm 5. This is internal to the node, and is merely a naming practice, with no algorithmic implications[9]. All properties that we prove and apply to *OPERATIONAL** nodes also apply to *OPERATIONAL* nodes.

**Lemma 1. (Key Lemma)** *For any pair of non-faulty connected nodes, $A$ and $B$, if $C$ is an* OPERATIONAL* *node on $P(A, B) - \{A, B\}$, then $C$ is connected to nodes on both $P(A, C)$ and $P(B, C)$. We formalize this as follows:*

$$\begin{array}{l} \forall A, B, C, \alpha_C^C = OPERATIONAL* \wedge \\ C \in P(A, B) - \{A, B\} \wedge \\ A \in Connections_B \wedge B \in Connections_A \end{array} \Rightarrow \begin{pmatrix} Connections_C \cap P(A, C) \neq \emptyset \wedge \\ Connections_C \cap P(B, C) \neq \emptyset \end{pmatrix}$$

*Proof.* Suppose the contrary becomes true for the first time at $t$. Since $d(A, C) < d(A, B)$, the fact that $B \in Connections_A$ implies that $C$ has failed at some point before $t$. Otherwise, an attempt to bypass $C$ would have never been made. There are further chances that $A$ and $B$ have had failures as well. Consider $t_A^r$, $t_B^r$, and $t_C^r$, as the last times prior to $t$, that either $A$, $B$, and $C$ were still faulty. For $A$ and $B$ assume $t_A^r$ and $t_B^r$ is 0, if they never failed. With regard to the precedence of $t_A^r$, $t_B^r$, and $t_C^r$, we have one of the following six possibilities:

i) $t_A^r \leq t_B^r < t_C^r < t$: At the time that $C$ started its recovery procedure, both $A$ and $B$ were non-faulty. Thus, $C$ synchronizes with the closest non-faulty node, on $P(C, A)$ and $P(C, B)$. If $C$ detects their failure at any point while synchronizing or afterwards, $C$ tries a farther node on the corresponding paths. Since by time $t$, $A$ and $B$ never fail again, $C$ never skips either $A$ or $B$, and the condition holds.

ii) $t_A^r < t_C^r < t_B^r < t$: Assume $k_1 = d(A, B)$. Since $C$ never sees the failure of $A$ it never skips $A$. This ensures that the first condition, $Connections_C \cap P(A, C) \neq \emptyset$, holds. Furthermore, since $A$ is connected to $B$, then all nodes between $C$ and $B$ have had failures too, and will recover at some point. If $B$ detects all these failures, it will connect to $C$, and the second condition holds. However, if $B$ connects to $X$ between $C$ and $B$, then either $t_X^r < t_C^r$ or $t_C^r < t_X^r$. The former is a case of (*i*) and $C$ connects to $P(C, X)$ which is a subset of $P(C, B)$. The latter, is another case of (*ii*), in which $A$, $B$, and $C$ are substituted by $A$, $C$, and $X$. Furthermore, we have $k_2 = d(A, X) < k_1$. In the worst case, since node distances are finite, we can apply this argument a limited number of times. At some point we get to $X_i = C$ and the condition holds.

[9] We do not make any use of this in the reliablilty and fault-tolerance algorithms and protocols.

iii) $t^r_C < t^r_A \leq t^r_B < t$: Since $A$ and $B$ never see failure of $C$, they will never skip $C$. Thus $A \in Connections_B$ and $B \in Connections_A$ cannot be true. The condition still holds.

iv) $t^r_C < t^r_B \leq t^r_A < t$: Similar to case (*iii*).

v) $t^r_B < t^r_C < t^r_A < t$: Similar to case (*ii*).

vi) $t^r_B \leq t^r_A < t^r_C < t$: Similar to case (*i*).

In all the cases above, the condition holds. This concludes the proof. □

**Corollary 1. (Key Corollary)** *No* OPERATIONAL* *node is ever skipped.*

*Proof.* Key Lemma shows that if there is an *OPERATIONAL** node, $C$, between any two communicating nodes, $A$, and $B$, then there is also a sequence of communication links that connect $A$ to $C$, and $B$ to $C$. Since messages are always sent to the nearest connected peers then messages from $A$ and $B$ to each other, always go through $C$. Thus, $C$ is never skipped. □

### A.1 Correctness of Forwarding

We now prove all *OPERATIONAL* final destinations of a message will receive a message, provided that the required level of consistency ($\delta$-Level-1 or $\delta$-Level-2) is already established and the number of non-*OPERATIONAL* nodes does not exceed $\delta$.

**Theorem 1. (Correctness Theorem)** *For any multicast message $m$ generated at time $t$ from source $S$, and forwarded based on $\delta$-Level-1 or $\delta$-Level-2 consistent routing information, a node $N$ downstream from $S$ sends $Conf^m$ at time $t_N$ towards $S$* if and only if all OPERATIONAL *final destinations $F$ of $m$ downstream from $N$ since $t$ have received and confirmed $m$, provided that no more than $\delta$ non-*OPERATIONAL *nodes are present in the system at any time.*

*Proof.* Suppose $F$ is a final destination of $m$ and let $N$ be the closest node to $F$ on $P(F, S)$, such that $N$ sends $Conf^m$ towards $S$, while $F$ has not yet received $m$.

- If $m$ is being forwarded based on $\delta$-Level-1: Since $\delta + 1 \geq d(F, S) \geq d(F, N)$, $\delta$-Level-1 consistency implies that (*i*) if $N$ is *OPERATIONAL*, it computes a recipient set for $m$ that contains $F$, i.e., $F \in \Gamma^m_N$. On the other hand, (*ii*) if $N$ is *RECOVERING*, $N$ has received $m$ as a $Guided$ message, $g^m$ from some node on $P(N, S) - \{N\}$ (possibly $F$). Through a chain of zero or more *RECOVERING* nodes, an *OPERATIONAL* node $M \in P(N, S) - \{N\}$ has incorporated path information into $g^m.to$. Since $\delta$-Level-1 consistency is establishes, and $d(M, F) \leq \delta + 1$, $g^m.to$ contains a path to $F$. From both (*i*) and (*ii*), it follows at $N$ we have $F \in \Gamma^m_N$, and $N$ is aware of a path towards $F$, $P(N, F)$. Thus, $F$ forwards $m$ to the first non-faulty node on $P(N, F) - \{N\}$. Furthermore, $N$ awaits $Conf^m$, from last node $X$ that it forwarded $m$ to, $X \in Out^m_N$. When $N$ itself sends $Conf^m$ upstream, it must have received $Conf^m$ from $X$, or conclude that all nodes on $P(N, F) - \{N\}$ are failed. The latter cannot be true since $F$ never fails in the interval $(t, t_N)$. Furthermore, since $F$ has not yet received $m$, $X$ will be the closest node to $F$ that sends $Conf^m$ before $F$ receives it. This is a contradiction, and $F$ must have received $m$ and confirmed it before and upstream node $N$, sends $Conf^m$ towards $S$.

- If $m$ is being forwarded based on $\delta$-Level-2: if $\delta + 1 \geq d(F, S) \geq d(F, N)$, then the proof is identical to the $\delta$-Level-1 consistency case. Now consider the case where $d(F, S) > \delta + 1$. Since $\delta$-Level-2 consistency is eatablished, (*i*) if $N$ is *OPERATIONAL*, it computes a recipient set for $m$ that contains either $F$, or a node $F'$ which is $\delta + 1$ hops closer to $F$, i.e., $F' = P(N, F)[\delta + 1]$ and $F' \in \Gamma_N^m$. If $N$ is *RECOVERING*, it has received $m$ through a chain of zero or more *RECOVERING* nodes from an upstream *OEPRATIONAL* node $M$. In this case, $N$ will again try to send $m$ towards $F'$ such that $F' = P(M, F)[\delta+1]$. Furthermore, in both cases, $N$ has the path information towards $F'$. Thus, $N$ will send $m$ towards $F'$ via a node $X \in P(N, F') - \{N\}$, and we have $X \in Out_N^m$. This implies that $N$ could not have send $Conf^m$ to its upstream without receiving $Conf^m$ from $X$; or otherwise, $X$ is a node closer than $N$ to $F$ that has sent $Conf^m$ to its upstream prior to arrival of $m$ at $F$. This is a contradiction, since $N$ was the closest node to $N$ that confirmed $m$ before $F$ has received $m$.

□

### A.2 Duplicate Detection

We say a message $m$ is "tagged by an *OPERATIONAL* node", *if and only if*, there is a pair $\langle X, msgId_X^m \rangle \in m.visited$, and $X$ was in *OPERATIONAL* state when processing $m$. We now prove that using a ($\delta + 1$)-list of visited nodes, the duplicate detection mechanism is able to track and identify duplicate arrivals of a message.

**Lemma 2. (Duplicate Detection Lemma)** *The duplicate detection mechanism is able to identify duplicate arrival of messages.*

*Proof.* Multicast messages are always sent in a direction that is away from their source. We use induction on the distance of a node $N$ from the source of any message $m$ and any of its duplicate copies $m'$ ($m_{id} = m'_{id}$). Let $S = m.source = m'.source$.
**Basis:** $\forall N, d(N, S) \leq \delta + 1$, $N$ is able to identify $m$ and $m'$ as duplicates.
Proof of induction basis: if $N$ is in distance $\delta + 1$ from $S$, both $m.visited$ and $m'.visited$ contain $\langle S, msgId_S^m \rangle$. Thus, $N$ can detect $m$ and $m'$ as duplicates by updating its input timestamp object, $ts_obj_N$, using the first arrived copy, and then calling $isDuplicate_N$ on the second arrived copy (this call returns $true$).
**Induction step:** If $\forall N, d(N, S) \leq k$, $N$ is able to detect $m$ and $m'$ as duplicates, then $\forall N, d(N, S) = k + 1$, $N$ detects $m$ and $m'$ as duplicates.
Proof of induction step: suppose the contrary is true, i.e., $d(N, S) = k + 1 > \delta + 1$ and $N$ cannot detect $m$ and $m'$ as duplicates. Let $P = P(P(N, S)[1], P(N, S)[\delta + 1])$; all members of $P$ are within distance $\delta + 1$ of $N$. Suppose $t$ is the *first* time a node sent either $m$ or $m'$ towards *any* node on $P$, or attempted to send $m$ or $m'$ via skipping any node on $P$. Furthermore, suppose $T$ is the time by which both $m$ and $m'$ arrive at $N$. We have the following facts:

i) From the way the algorithm works, every message being forwarded is tagged by at least one *OPERATIONAL* node (also applies to the encapsulated messages within a $Guided$ message).

ii) No node, $X$, $X \in P$, can be *OPERATIONAL* at all times in the interval $(t, T)$; otherwise, $X$ would not be skipped and would have tagged both $m$ and $m'$ (Key Corollary). Thus, $N$ could identify the messages as duplicates since they both have an identical entry corresponding to $X$.

iii) Since in a $\delta$-resilient system, at most $\delta$ non-*OPERATIONAL* nodes may exist, we conclude that there must be at least one node on $P$, that has completed its recovery procedure and became *OPERATIONAL* in the time interval $(t, T)$. Otherwise, either (i) or (ii) would be violated. Let $R$ be the first node that took the transition to *OPERATIONAL* at time $t^r$ after $t$.

iv) Not all nodes of $P$ that were initially *OPERATIONAL* at $t$ can fail before $R$ recovers; otherwise, (iii) is violated.

Consider all *OPERATIONAL* nodes on $P$ that fail were *OPERATIONAL* at $t$ and will fail before $T$. Let $L$ be the one that fails last. Now consider all the messages that were sent in the interval $(t, t^r)$, in which $L$ is always *OPERATIONAL* and $R$ becomes *OPERATIONAL* at $t^r$. Assume $r$ is the last message sent to $R$ at $t^{(r*)}$ from node $Y$ on $P(R, S)$ that would be enough to let $R$ become *OPERATIONAL** (had all the other needed messages for this transition arrive at $R$ before $r$ is processed). Since $Y$ is sending synchronization messages to $R$, it is either *OPERATIONAL* or is receiving synchronization messages from a node on $P(Y, S)$. In the former case, it is never skipped by nodes on $P(Y, S)$ since it is *OPERATIONAL*; and in the latter case, again it cannot be skipped by any node on $P(Y, S)$. This is because, $Y$ must be connected to a closest *OPERATIONAL* node, $O$ on $P(Y, S)$ to receive synchronization information from (this information is also supplied to $R$). Thus, a path of communication links exist between $R$, and $O$, and no message is ever sent from nodes on $P(R, S)$ that bypasses either $O$ or $R$. Now consider the following cases:

I – If either $m$ or $m'$ had been attempted to be sent to $R$, or bypass $R$ by any other node before $t^{(r*)}$: this implies, that during synchronization $R$ will receive $m$ or $m'$. If both are received, then $R$ detects them as duplicates (otherwise, $R$ would be a closer node to $S$ than $N$ that cannot detect duplicates). Furthermore, in terms of the relative location of $R$ and $L$, one of the following is possible:

a. $d(N, L) < d(N, R)$: $R$ forwards one of the received copies towards $L$. If $L$ has previously received a copy, it will discard the new one; otherwise, it will forward it towards $N$. This is also the first and only copy that is sent towards $N$ closer than $L$. Finally, $R$ reliably forwards a $REJOIN$ message, $rj$, and awaits receipt of $Conf^{(rj)}$. Since both $L$, and $N$ are non-faulty and within distance $\delta + 1$ of $R$, they both receive $rj$, and update their timestamp associated with $R$. By $t^r$, $R$ becomes *OPERATIONAL* and unless $R$ fails, no other copy of $m$ will arrive at $N$ without $R$'s initial timestamp, which precedes the current timestamp associated with $R$ at $N$.

b. $d(N, L) > d(N, R)$: $L$ has been *OPERATIONAL* since $t$, an attempt to send a copy of $m$ to $R$, or bypass it, implies that $L$, has already received the same copy and has tagged it. Thus, if due recovery of $R$, a duplicate copy of $m$ is sent to $N$, then both copies carry the same entry for associated with $L$. Thus, $N$ can detect the duplicates. Similar to the previous case, the propagation of the $REJOIN$ message from $R$ ensures that unless $R$ fails again, no duplicate copy of $m$ is sent to $N$ without having $R$'s associated timestamp, which precedes its timestamp updated by $rj$.

c. $d(N, L) = d(N, R)$: this implies $R = L$ and cannot be true.

II – If neither $m$ nor $m'$ had been attempted to be sent to $R$, or bypass $R$ by any other node before $t^{(r*)}$: unless $R$ fails again, it will receive a copy of $m$ and forward it towards $N$. Furthermore, any additional copies are discarded unless $R$ fails again (at some time beyond $t^r$).

We have shown that all copies of $m$ that are sent in the interval $(t, t^r)$ can be detected at $N$ as duplicates. Now consider the next recovered node $R'$ (if it exists) which took the transition to *OPERATIONAL* at time $t^{R'}$, and apply the same argument for the interval $(t^r, t^{r'})$. It further implies that by completion of recovery of no node on $P$, there is a chance that $N$ receives undetectable duplicates. Now let $Z$ to be the last node to recover (possibly $Z = R$) at $t^z$ in the interval $(t, T)$. For the interval $(t^z, T)$, we know that there is no recovery, and (*iii*) requires that there be a node that remains *OPERATIONAL*. This node, $H$, must have been recovered at least once since $t^z$. In the above discussion, we have also illustrated that undetectable duplicate arrivals cannot take place as long as the recovered node remains *OPERATIONAL*. This is also true for $H$. Thus, by time $T$, $N$ is able to detect all duplicate arrivals of $m$. □

### A.3 In-order Delivery

Now we prove that arrival of messages from the same source, follows the order they were generated from the source. In fact, we show that the order of messages along their path to any final destination never changes.

**Lemma 3. (Ordered Delivery Lemma)** *For all non-identical multicast messages $m$ and $m'$ from the same source that arrive at a final destination node $F$, the order of delivery is the same as the order of generation at the source.*

*Proof.* Let $S = m.sender = m'.sender$, $t$ and $t'$ respectively be the arrival times of $m$ and $m'$ at $F$. There are three algorithms that involve a multicast message $m$ being sent from $S$ to a final destination $F$.

1. Reliable forwarding algorithm: adds a message to a FIFO list $ProcessingList$, enqueues it on a FIFO output queue, sends it over a FIFO communication link, and dequeue it from a FIFO input queue at the receiver.

2. Fault-tolerant forwarding algorithm: considers messages in $ProcessingList$ in-order, enqueues the messages in a FIFO output queue while preserving the order, sends the messages in-order over a FIFO communication link, and dequeues the messages from a FIFO input queue at the receiver (if transmission was successful)

3. Synchronization algorithm (at the synchronization point): a combination of the reliable forwarding algorithm, and the fault-tolerant algorithm that both uses $Guided$ messages to transmit multicast messages. However, again messages in the $ProcessingList$ of the synchronization point are processed in-order. Furthermore, other messages that arrive afterwards are added to the $ProcessingList$ in-order, and follow the previous ones on the list.

Now suppose the contrary is true, and $F$ receives $m'$ first. Since $F$ is a final destination of $m$ and $m'$, the Forwarding Theorem requires that $F$ receives both $m$ and $m'$. Thus, we must have $t' < t$. This implies that by time $t$, no upstream node from $F$, will receive the confirmation of $m$. Thus, all these nodes, $X$, that have received $m$ and forwarded it downstream still have $m$ in their $ProcessingList_X$.

Now consider the messages $m$ and $m'$ that pass through a sequence of operations as mentioned above to travel to $F$. Furthermore, by time $t$ no upstream node from $F$ ever receives $Conf^m$ (since $m$ has not yet been received by $F$), nor removes $m$ from its $ProcessingList$. Thus, on all the upstream nodes that forward $m'$ towards $F$, $m$ has been received first. This is due to the fact that all operations listed above preserve the order of messages, and since $m$ is never removed from $ProcessingList$ of any node, $m'$ follows $m$ at all times. Since $F$ received $m'$ prior to $m$, this is a contradiction, and the proof is complete. □

### A.4 Recovery

In this section, we show that by the time a previously failed node recovers (becomes ***OPERATIONAL***) it has consistent topology and subscription routing information, as if it had never failed before.

**Lemma 4.** *The routing information, $TOP_R$ and $SUBS_SET_R$ at a recovered node $R$ is respectively $\delta$-Level-1 and $\delta$-Level-2 consistent. Furthermore, at the time $R$ takes the transition to* OPERATIONAL *no unconfirmed messages still in transit may cause the routing state at $R$ to violate the consistency requirements.*

*Proof.* Let $T^R$ be the time $R$ completes its recovery procedure, and consider the network in the interval $(0, T^R)$. Now consider the sequence of recovered nodes in this interval, $R_1, \cdots, R_l$, where $R_l = R$. Furthermore, let $T^{R_i}$ be the time node $i$ in this sequence recovers. We first prove that the routing state is consistent at $T^{R_1}$.

Node $R_1$ has been synchronizing with zero or more ***OPERATIONAL*** nodes in order to get the topology and subscription routing information. Consider the chain of ***RECOVERING*** nodes that lead to one of these ***OPERATIONAL*** nodes $O$, as illustrated in Fig 7. The multicast messages that have arrived at $O$ since time 0 are either confirmed, and thus updated in $O$'s routing information, or are not yet confirmed. Since there is no node that has recovered prior to $R_1$, thus $O$ has been ***OPERATIONAL*** ever since it joined the network. By the Key Lemma, no ***OPERATIONAL*** node is ever skipped. Thus, $O$ misses no messages, including the ones that must have been forwarded to $R_1$. Now let $t_O^W$ be the time when $O$ first received the $SynRequest$ from $W$, the closest node on the chain of recovering nodes of $R_1$. Let $ALL_{t_O^W}$ be all the $J$, $SUB$ messages distined to $R_1$, that arrived at $O$ prior to $t_O^W$. Furthermore, let $ALL_CONFIRMED_{t_O^W}$ be a subset of $ALL_{t_O^W}$ that has been confirmed by $t_O^W$. Since $R_1$ is a final destination of all $m \in ALL_CONFIRMED t_O^W$, $W$ is also a final destination of $m$. Thus, the recovering node $W$ receives $m$ from $O$ as part of its recovery as $PAR_TOP_O^W$ or $PAR_SUB_O^W$. Furthermore, each node on the chain, $S^i$ (where $W = S^{last-1}$ and $R_1 = S^1$) except for $O$, is also a final destination of the message that added corresponding entries to $PAR_TOP_{S^{i+1}}^{S^i}$ and $PAR_SUB_{S^{i+1}}^{S^i}$, and receives it from $S^{i+1}$. Thus, by $T^{R_1}$, $R_1$ has already received routing information on behalf of all messages in $ALL_CONFIRMED_{t_O^W}$.

Now consider the messages distined to $R_1$ that have arrived at $O$ by time $t_O^W$ but not yet confirmed, i.e., $ALL_{t_O^W} - ALL_CONFIRMED_{t_O^W}$. All these messages $m$ are still in $ProcessingList_O$, and as part of the sychronization procedure (Algorithm 7), they are sent as $Guided$ messages towards $R_1$ via $S^{last-1}$. Furthermore, at any node, $S^i$, on the chain this messages if added to the $ProcessingList_{S^i}$ and is sent towards $R_1$ via the next non-faulty topology node (perhaps $S^{i-1}$). However, if all these nodes including the rest of the chain ($S^{i-1}, \cdots, S^1 = R_1$) is detected to be faulty, we must have reached an edge of the network, thus the message is confirmed by $S^i$ and further added to its temporary routing information, $Guided_TOP_{S^i}$ and $Guided_SUB_{S^i}$. At some time later, the $SynRequest$ message is received from

$S^{i-1}$, this information is further sent towards $S^{i-1}$ as part of $PAR_TOP_{S^i}^{S^{i-1}}$ and $PAR_SUB_{S^i}^{S^{i-1}}$, thus $R_1$ eventually receives it, as $PAR_TOP_{S^2}^{R_1}$ and $PAR_SUB_{S^2}^{R_1}$.

Finally, consider all the rest of the *in transit* messages distined to $R_1$ that arrive at $O$ after $t_O^W$. These messages are again sent to $W$ as $Guided$ messages and arrive at $R_1$ eventually (but perhaps not by $T^R$). If $R$ does not receive the messages prior to $T^R$, it will receive it unless $R$ fails again. This is ensured since $R$ is an ***OPERATIONAL*** node and a final destination of the message.

We have showed, that the first recovered not in the chain $R_1 \cdots R$ can correctly recover, and establish consistent routing information. Application of the same argument for a finite number of times for $R_2$, $R_3, \cdots$, ensures that $R$ can recover without compromising the consistency of the system. □

### A.5 The Join Operation

The key in our approach, is the ability of *VALID_JOINED* nodes to compute a topology path between all nodes within distance $\delta+1$. This is ensured via the join operation, which updates the topology routing tables. We now show the correctness of the join operations by showing that $\delta$-Level-1 consistency is established.

**Lemma 5.** *Non-concurrent join operations establish Level-1 consistency among all* VALID_-JOINED *nodes in the network. Furthermore, any pair of these nodes,* $X$ *and* $Y$, $d(X,Y) \leq \delta + 1$ *can locally compute the path* $\delta$*-path* $P(X,Y)$.

*Proof.* To prove that $\delta$-Level-1 consistency property is established, we use induction on the size of the network, $size$.

If $size = 1$, $\delta$-Level-1 trivially holds. Now suppose that for all constructions of a network of size $k$, $\delta$-Level-1 holds. We prove that for all constructions of a network of size $k+1$, $\delta$-Level-1 is also achieved. For each such network, assume the last join operation carried out by $N$ on $M$, prior to which $size = k$. This operation took place when all other join operations were completed and all nodes were in *VALID_JOINED* state (non-concurrent). As a result, in the first phase of the last join, $N$ receives a $PAR_TOP_M^N$ that has all *VALID_JOINED* nodes with distance $\delta$ of $M$ ($\delta+1$ from $N$). This is in fact, all the information that $N$ needs to forward join messages based on $\delta$-Level-1 consistency. Furthermore, reliable forwarding of the join message $j$, takes place at each node, $X$, based on their local $TOP_X$, which is a superset of the computed recipient set at $N$, $RCPT_N^{T(j)}(j)$. This guarantees that $j$ will be forwarded to all members of this set. By the time $Conf^j$ propagates back towards $N$, $\overleftrightarrow{NM}$ is added to $TOP_X$, thus all nodes within distance $\delta+1$ of $N$, are also aware of $\overleftrightarrow{NM}$ and can compute a path to $N$. This ensures that by the time $N$ becomes a *VALID_JOINED* node, $\delta$-Level-1 consistency property holds for forwarding of $J$ messages. This concludes the proof. □

**Lemma 6.** *Concurrent join operations establish Level-1 consistency among all* VALID_-JOINED *nodes in the network. Furthermore, any pair of these nodes,* $X$ *and* $Y$, $d(X,Y) \leq \delta+1$ *can locally compute the* $\delta$*-path* $P(X,Y)$.

*Proof.* Suppose that there were a number of join operations taking place concurrently. Consider any pair of these operations that joining nodes $N_1$ and $N_2$ join, join points $A$ and $B$ repectively. If $d(A,B) > \delta$, there are no conflicts and they take place independently, as if they were taking place non-concurrently (Lemma 5). Thus, we only need to consider the cases where $0 \leq d(A,B) \leq \delta$. For $d(A,B) = 0$, there

are two nodes $N_1$ and $N_2$ that are joining the same node $A$, $(A = B)$. Without loss of generality, we can assume that $A$ initially received $N_1$'s join request. $A$ further adds $\overleftrightarrow{N_1A}^*$ to $TOP_A$, and replies to $N_1$ with $PAR_TOP_A^{N_1}$. At any time later, when $N_2$ requests a join, it will know about $\overleftrightarrow{N_1A}^*$ by receiving the updated $PAR_TOP_A^{N_2}$. On the other hand, when $N_2$ sends the join message to $A$ in its second phase of the join operation, $A$ will correctly route it to $N_1$. Furthermore, $N_2$ will not be able to get to *VALID_JOINED* state without $N_1$ having confirmed the join message of $N_2$. This ensures that by the time the two concurrent join operations end, Level-1 consistency is met for forwarding of $J$ messages. Furthermore, both $N_1$ and $N_2$ can locally compute $P(N_1, N_2)$.

Now consider the case, where $0 < d(A, B) \leq \delta$, and $A$ received the initial join request from $N_1$ at time $t_1$, and $B$ received the join request from $N_2$ at time $t_2$. Without the loss of generality suppose that $t_1 \leq t_2$. One possibility is that by time $t_2$, $N_1$ actually proceeds to phase 2 of join, and its join message $j_{N_1}$ has arrived at $B$. Thus, $B$ has added $\overleftrightarrow{N_1A}^*$ to $TOP_B$, and by time $t_2$, it will reply to $N_2$ with the updated $PAR_TOP_B^{N_2}$ which includes $\overleftrightarrow{N_1A}$. On the other hand, since all the nodes on $P(A, B)$ have received $j_{N_1}$ and have updated their local topology set, $N_2$'s join message, $j_{N_2}$ will also be routed to $N_1$. This ensures that by the time $N_1$ and $N_2$ receive $Conf^{j_{N_2}}$ and $Conf^{j_{N_1}}$ repectively, Level-1 consistency is met.

Now we consider the case where $A$ and $B$ reply to $N_1$ and $N_2$'s join request (in phase 1) without having received the join message of the $N_2$ and $N_1$, respectively. Thus, both $N_1$ and $N_2$ proceed to the second phase of join, without having any knowledge of the other party joining. Thus, by the time $A$ receives $j_{N_2}$ it already knows about $\overleftrightarrow{N_1A}$ and will forward the message to $N_1$ and wait for its confirmation. Hence, by the time $N_2$ receives $Conf^{j_{N_2}}$, $\overleftrightarrow{N_2B} \in TOP_{N_1}$. The same argument applies for $N_2$ and $j_{N_1}$. This concludes the proof. □

### A.6 The Subscribe Operations

Subscription forwarding establishes the routing state in the network that is used to forward publications. In this section, we prove that this process establishes $\delta$-Level-2 consistent subscription routing information used for forwarding of matching publications. (Note that the routing state stored in $SUBS_SET$ of any node is actually used for forwarding of publications, i.e., forwarding of publications use the $\delta$-Level-2 consistency established by the forwrading process of subscriptions).

**Lemma 7.** *Subscription forwarding algorithms establish $\delta$-Level-2 consistency.*

*Proof.* For any subscription $s$ from subscriber $S$, all nodes, $N$, that forward $s$, call the $make_co-py_send_N^{SUB}(s, M)$ procedure to send a copy of $s'$ (with a modified $from$ field) to all nodes $M$ downstream from $N$. Furthermore, by Lemma 5 and Lemma 6, $N$ is able to identify all downstream nodes, and compute paths towards them. Once $M$ receives this copy, it forwards $s'$ further downstream. Finally, it awaits $Conf^s$ messages from those nodes, and once all confirmation messages are received, $M$ updates its subscription routing information, with $\langle s'.predicate, s'.from\rangle$, such that $s'.from$ points to the subscriber $S$, if $d(S, M) \leq \delta + 1$, or else to a node $S' \in P(M, S)$, which is $\delta + 1$ nodes closer to $S$. Thus, by the time the $N$ receives $Conf^s$ from $M$, all ***OPERATIONAL*** nodes $X$, downstream in the subtree of $M$ have entries that can correctly identify a node $Y \in P(X, S)$ as mandated by the $\delta$-Level-2 consistency condition. This is also true when $S$ receives all pending $Conf^s$ messages. At this point, $S$ becomes *VALID_SUBSCRIBED* and the $\delta$-Level-2 consistency requirement is met. □

### A.7 The Publish Operation

We now prove that the publish operation is able to deliver the publications to matching subscribers, as required by the reliability specification (Section 4.2).

**Lemma 8.** *Unless there are more than $\delta$ simultaneous non-*OPERATIONAL *nodes in the system, using the $\delta$-Level-2 consistent routing information established by forwarding of subscriptions, all publications are delivered to matching subscribers as required by the reliability specification.*

*Proof.* Since subscription forwarding establishes $\delta$-Level-2 for *VALID_SUBSCRIBED* subscription $s$ from $S$, it directly follows from the Correctness Lemma that the publications matching $s$ will be delivered to $S$. Furthermore, the delivery order of publications from the same publisher to subscribers matches the order they were published from the source. Finally, since all duplicate messages are detected and eliminated using the duplicate detection mechanism of Section 5, no publication will be delivered more than once to a matching subscriber. This ensures that the reliability specification is met, under at most $\delta$ simultaneous non-*OPERATIONAL* nodes. □