On the design of multi-tagged events, event streams and fault tolerant event services [*]

Shrideep Pallickara[†] Geoffrey Fox

November 6, 2000

Abstract

In this paper we explore the design of events and the issues pertaining to delivery of these events in a fail safe and resilient manner. The clients we consider in this scheme also try to reflect the changes taking place in the pervasive computing area. Clients express their interest in certain events or event streams, these streams and events need to be delivered in the presence of failures and prolonged disconnects from the distributed network that would be in place to support such operations. This paper, is about events, the organization, retrieval and specification of attributes and constraints associated with that event. This paper is also about event streams and the construction of event queues comprising of the aforementioned events. We provide a motivation and a solution for how the queues can exist over the network. Issues pertaining to replication and consistency will also be discussed. Finally this paper is also about the design of an event service, specifying client and server rules, which would use these events and the event queues.

Contents

[*]Northeast Parallel Architectures Center, Syracuse University, Syracuse NY 13210.
[†]**Version: 1.7**

1 Introduction

Events are an indication of an interesting occurrence. Events point to nuggets of information which are related to the event itself, and help us understand the event completely. When we refer to an event we refer to the occurrence and the information it points to. The information contained in the event comprises of

- The occurrence which snapshots the context, priority and the application.
- Attribute information which constitutes the event type.
- Control information.
- Destination Lists

The attribute information comprises of tags which specify the attributes associated with the event type while the control information specifies the constraints associated with that event viz. ordering, stability. Thus say a person needs to sell stock A - the selling is the event, the general information is his account profile while the control information could be an indication that he wants guaranteed delivery of the event. Events trigger *actions*, through the state transitions induced in a delivering entity, which in turn can trigger events. The event and the associated actions taken by any part of the system share the *cause--effect* relationship. Actions are taken based on the event type and the information contained in the event. The action taken at any node could be influenced not only by different causes but by subsequent effects too.

The spectrum of relationships between events in traditional systems span from “unrelated” to where events are “related”. These events were related through different ordering permutations based on the *local order* imposed by the issuee, *total order* imposed by a deterministic algorithm hosted on multiple nodes and a system determined *causal order*. Events form the basis of our design and are the most fundamental units that entities need to communicate with each other. These events encapsulate expressiveness at various levels of abstractions - content, dependencies and routing. Where, when and how these events reveal their expressive power is what constitutes information flow within our system. The events that we consider exist within event streams and can specify and dictate resolution of complex spatial and chronological dependencies with other events in the system. Related events can be considered to be part of unique abstract merged stream. Clients can express an interest in receiving a merged stream or *bundles* within a stream. It should be noted however that a bundle or the complete merged stream being delivered at a client can have multiple stream sources. The organization, resolution, retrieval of the attributes and constraints associated with events, error checking and the reliable delivery to targeted clients is one of the most important goals of our work.

The clients we are considering for our system design try to address the enormous changes taking place in the area of pervasive computing and associated transport protocols. We make no assumptions regarding a client’s computing power or the reliability of the transport layer over which it communicates. Clients have *profiles* which indicate the kind of events, stream bundles and streams that it is interested in. The goal is to deliver the events reliably after satisifying any dependencies that may exist between the events, stream bundles and merged streams. We provide guarantees regarding the delivery of these events at the client.

One of the reasons why we use a distributed model is high availability. Having a centralized model would imply a single server hosting multiple clients. While, this is simple model, the inherent simplicity is more than offset by the fact that it constitutes a single point of failure. Thus all the clients present in the system would be unable to use any of the services provided by the system till a recovery mechanism kicks in. A highly available distributed solution would have data replication at various server nodes in the network. Solving issues of consistency while execution operations, in the presence of replication, leads to model where other server nodes can service a client despite certain server node failures. The underlying network that we consider for our problem are the nodes hooked onto the Internet or Intranets. We assume that the nodes which participate in the event delivery can crash or be slow. Similarly the links connecting them may fail or overload. These assumptions are based on the experiences we have drawn based on real situations. One of the immediate implications of our delivery guarantees and the system behavior is that profiles are what become persistent not the client connection or its active presence in the digital world at all times.

In the systems we are studying, unlike traditional group multicast systems, “groups” cannot be preallocated. Each message is sent to the system as a whole and then delivered to a subset of recepients. The problem of reliable delivery and ordering[1] in traditional group based systems with process crashes has been extensively studied [Bir85, HT94, Bir93]. The approaches normally have employed the “primary partition” model [RSB93], which allows the system to partition under the assumption that there would be a unique partition which could make decisions on behalf of the system as a whole, without risk of contradictions arising in the other partitions and also during partition mergers. However the delivery requirements are met only within the primary partition [GRVB97]. Recipients that are slow or temporarily disconnected may be treated as if they had left the group. This model works well for problems such as propagating updates to replicated sites. This approach doesn’t work well in situations where the client connectivity is intermittent, and where the clients can roam around the network. The main differences between the systems being discussed here and traditional group-based systems are:

1. We envision relatively large, widely distributed systems. A typical system would comprise of hundereds of server nodes, with thousands of clients.

2. Events are routed to clients based on their profiles, employing the group approach to routing the interesting events to the appropriate clients would entail an enormous number of groups - potentially 2^n groups for n clients. This number would be larger since a client profile comprises of interests in varying event foot prints.

3. The persistence of a clients profile is inconsistent with the approach of choosing a primary partition and dropping disconnected or slow clients.

The approach adopted by the OMG [OMG00b, OMG00a] is one of establishing channels and registering suppliers and consumers to those event channels. The event service [OMG00b] approach has a drawback in that it entails a large number of event channels which clients (consumers) need to be aware of. Also since all events sent to a specific event channel need to be routed to all consumers, a single client could register interest with multiple event channels. The aforementioned feature also forces a supplier to supply events to multiple event channels based on the routing needs of a certain event. On the fault tolerance aspect, there is a lack of transparency since channels could fail and issuing clients would receive exceptions. The most serious drawback in the event service is the lack of filtering mechanisms. These are sought to be addressed in the Notification Service [OMG00a] design. However the Notification service attempts to preserve all the semantics specified in the OMG event service, allowing for interoperability between Event service clients and Notification service clients. Thus even in this case the a client needs to subscribe to more than one event channel.

To truly compare the system we have designed with traditional approaches to delivering events the issues that need to be factored out include -

(a) Client connection semantics - where a client could communicate intermittently and can roam around the network. This would then rule out issues pertaining to storing and re-routing of missed events to its new location in the system.

(b) Client filtering behavior, where a client is not interested in all the events. This issue also factors out the calculation of destination lists for delivery of events.

(c) Event dependency resolution - resolving spatial and timing dependencies. Though there are schemes which relate events through causal order, fifo order and total order. The ability to specify interaction between events in multiple streams, detect cyclic dependencies and merge these streams (which may have their stream sources at disparate locations) is something very unique to our approach.

The issues we plan to investigate include scaling, fault tolerance, consistency, event delivery latencies in the presence of voluminous traffic, link failures and node failures. The other related issue is how these issues are affected by the number and the ratio of issuing clients to interested clients.

[1]The ordering issues addressed in these systems include FIFO, Total Order and Causal Order

2 Specifications

We now try to specify our problem. In section 2.2 we present our model of the system in which we intend to solve the problem. In section 2.3 we formally specify our problem. Sections 2.4 and 2.5 deal with the assumptions that we make in our formalism's and the properties that the system and it components must conform to during execution.

2.1 Events

An event is the most fundamental unit that entities need to communicate with each other. An event comprises of a set of properties and have one source and one or more destinations where it would be delivered. The properties could be boolean, or could take specific values within the range specified by the property. Events allow seperate entities to probe different set of properties, through accessor functions. The properties contained with an event could be mutable or immutable. The sub vector of this set of properties is what constitutes the type of the event. Events also possess a set of destinations that comprise the clients which are targeted by the event. This destination list could be explicitly contained within the event itself, or could be computed dynamically as a function of properties list contained within the event. Events induce state transitions in the entities that deliver the event. The state transition is followed by a set of actions. The event and the associated actions taken by any part of the system share the *cause-effect* relationship. These induced state transitions and associated actions are based on the values the properties can take.

Events can also exist within the context of an earlier event, we refer to such events as chasing events. Chasing events contain both spatial and chronological constraints pertaining to delivery at a node, subsequent to the delivery of the chased event. Events encapsulate information at three different levels - application specific, dependency in relation to chased events and routing information. The information encapsulated within an event defines the scope of its expressive power. Where, when and how these events reveal their expressive power is what constitutes information flow.

2.2 System Model

The system comprises of a finite (possibly unbounded) set of *server nodes*, which are strongly connected (via some inter-connection network). Special nodes called *client nodes*, can be attached to any of the server nodes in the network. Client nodes can never be attached to each other, thus they never communicate directly with each other. Let $\mathbf{C}$ denote the set of client nodes present in the system. The nodes, servers and clients, communicate by sending events through the network. This communication is *asynchronous* i.e. there is no bound on communication delays. Also the events can be lost or delayed. A server node execution comprises of a sequence of actions, each action corresponding to the execution of a step as defined by the automaton associated with the server node. We denote the action of a client node sending an event e as $send(e)$. At the client node the action of consuming an event e is $deliver(e)$. Server nodes relay the events to the client nodes, en route to destination client nodes, we denote this action $relay(e)$. For increased availability and reduced latency, some of the server nodes have access to a *persistent store* where they partially or fully replicate events and states of the nodes.

The failures we are presently looking into are node failures (client and server nodes) and link failures. The server node failures have crash-failure semantics and could be one of the following:

(a) Crash - A faulty node stops prematurely and does nothing from that point on.

(b) Send Omission - A faulty node stops prematurely, or intermittently omits to send messages it was supposed to send, or both.

(c) Receive Omission - A faulty node stops prematurely, or intermittently omits messages sent to it, or both.

(d) General Omission - A faulty node is subject to send and receive omission failures.

Link Failures are of two types:

(a) Crash - A faulty link stops transporting messages. Before stopping however it behaves correctly.

(b) Omission - A faulty link intermittently omits transporting messages sent through it.

As a result of these failures the communication network may *partition.* Similarly *virtual* partitions may stem from an inability to distinguish slow nodes or links from failed ones. Crashed nodes may rejoin the system after recovery and partitions (real and virtual) may heal after repairs.

2.3 The event service problem

Client nodes can issue and deliver events. Any arbitrary event e contains implicit or explicit information regarding the client nodes which should deliver the event. We denote by $L_e \subseteq \mathbf{C}$ this destination list of client nodes associated with an event e. The dissemination of events can be one-to-one or one-to-many. Client nodes have intermittent connection semantics. Clients are allowed to *leave* the system for prolonged durations of time, and still expect to receive all the events that it missed, in the interim period, along with real time events on a subsequent re-*join.* Consistency checks need to be performed before the delivery of real time events to eliminate problems arising from out of order delivery of certain events.

The system places no restriction on the server node a client node can attach to, at any time, during an execution trace σ of the system. We term this behavior of the client as *roam.* Clients could also initiate a roam if it suspects, irrespective of whether the suspicion is correct or not, a failure of the server node it is attached to. The choice of the server node to attach to, during a roam or a join, is a function of

- Preferences - Clients can specify which node they wish to connect to.
- Response Times - This is determined by the system based on geographical proximity and related issues of latency and bandwidth.

For an execution σ of the system, we denote by E_σ the set of all events that were issued by the client nodes. Let $E^i_\sigma \subseteq E_\sigma$ be the set of events e^i_σ that should be relayed by the network and delivered by client node c_i in the execution σ. During an execution trace σ client node c_i can *join* and *leave* the system. Node c_i could *recover* from *failures* which were listed in Section 2.2. Besides this, as mentioned earlier client nodes can roam (a combination of leave from an existing location and join at another location) over the network. A combination of join-leave, join-crash, recover-leave and recover-crash constitutes an *incarnation* of c_i within execution trace σ. We refer to these different incarnations, $x \in X = 1, 2, 3...$, of c_i in execution trace σ as $c_i(x, \sigma)$.

The problem pertains to ensuring the delivery of all the events in E^i_σ during σ irrespective of node failures and location transience of the client node c_i across $c_i(x, \sigma)$. In more formal terms if node c_i has n incarnations in execution σ then

$$\sum_{x=1}^{n} c_i(x, \sigma).deliveredEvents = E^i_\sigma.$$

All delivered events $e^i_\sigma \in E^i_\sigma$ must of course satisfy the causal constraints that exist between them prior to delivery.

2.4 Assumptions

(a) Every event e is unique.

(b) The links connecting the nodes do not create events.

(c) A client node has to accept every message, events and control information routed to it.

(d) Not all events can have zero targeted clients.

(e) If a client issues an event e infinitely often, eventually the event would be disseminated within the system.

Items (d) and (e) constitute the liveness property eliminating trivial implementations in which all events are lost or all events have no targeted clients.

2.5 Properties

(a) A client node can deliver e, only if e was previously issued.

(b) A client node delivers an event e only if that event satisfies the constraints specified in its control information.

(c) If an event e is to be delivered by client nodes $c, c' \in L_e$, then if c delivers e then c' will deliver event e.

(d) For two events e and e' issued by the same client node c, if a client node delivers e before e', then no client node delivers e' before e.

(e) For two events e and e' issued by nodes c and c' respectively, if a node delivers e before e', then no node delivers e' before e.

Properties (d) and (e) pertain to the causal precedence relation $\rightarrow$ between two events e, e', and can be stated as follows $\forall c_i \in L_e \bigcap L_{e'}$ if $e \rightarrow e'$ then $e.deliver() \rightarrow e'.deliver()$. $\rightarrow$ is transitive i.e. if $e \rightarrow e'$ and $e' \rightarrow e''$ then $e' \rightarrow e''$. The precise instant of time, from which point on, all these properties hold true are addressed in section 8.1.1.

3 Event Streams and events

An event stream denoted E is a stream of events $\{e_0, e_1, \cdots, e_n\}$ that are logically related to each other. Events within an event stream, $E.e_i$ are related to each other. This relationship is usually the precedence relationship $\rightsquigarrow$ shared by events within a event stream i.e. $e_0 \rightsquigarrow e_1 \rightsquigarrow \cdots e_n$. The precedence relationship $\rightsquigarrow$ is transitive, if $e_i \rightsquigarrow e_j$ and $e_j \rightsquigarrow e_k$ then $e_i \rightsquigarrow e_k$. Besides this individual events with an event stream could contain dependencies to one or more events in one or more other event streams. This dependency could be a direct association with events in other streams viz. one to one mapping. This dependency could also be a logical mapping, thus resulting in a mapping which is not exactly a one-to-one correspondence between the events in the event streams. It is conceivable that the information contained in events from multiple event streams are necessary to describe an event. In such cases the event in question, $E.e_i$, could be a container for the information contained within events in other event streams.

Events within an event stream could depend[2] on events from multiple event streams. Thus hypothetically we can assume that these related event streams merge. Consider three event streams E^A, E^B, E^C which merge to form an event stream E^D as depicted in Fig 3.1. Every event within the event streams contain information which describe the event. This information could pointers to events contained in other event streams, in which case we say that the event *encapsulates* events from other event streams. Thus if $E^A.e_i$ encapsulates $E^B.e_j, E^C.e_k$ besides containing information pertaining to $E^A.e_i$ we say that E^A is a *container* for streams E^A, E^B and E^C. Clients need not be aware of the existence of streams E^B, E^C or E^D. The information contained within $E^A.e_i$ determines the streams that need to merged. Besides this there should also be a precise indication of the events within other streams (the streams

[2]The scenario I am looking at is where a lecture is in progress, and the main stream is the lecture stream which contain the foils in text, however the events within this stream could point to information contained in the audio stream, video stream, images stream. These streams could be issued by streaming servers hosted at different locations. The video feed could be from Houston, audio feeds from Boston, Foils from Syracuse. The streams could have an independent stream created, which could be questions, questions may or may not arise for certain foils (*thus correlation between events in different streams could get arbitrarily complex*). The chat stream could originate from Jackson state while the responses could originate from Tallahassee. What we are looking at could be converted into a 24x7x365 education portal. Where chat streams and responses could be used to build a FAQ stream.

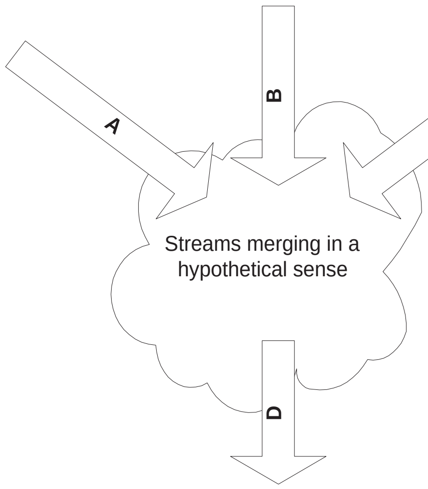

Figure 3.1: Existence of multiple event streams.

need to be identified unambiguously first of course) that are needed to describe an event completely. This indication could be a -

(a) A one-to-one mapping among events in all the streams. In our example this would be $E^A.e_i$ encapsulating $E^B.e_i$, $E^C.e_i$. The corresponding event in the merged event stream being $E^D.e_i$.

(b) Based on the information contained in individual events of the streams. This could be dependent on the tags contained in the events and the values that these tags could take.

(c) The dependency specification could take complex forms in which the information pointed to need not be a unique one and there could be several such events in the co-event streams which match the specification. In this case the dependency could take forms like

  (c.1) The first event which matches the constraint.

  (c.2) If there is an event which matches the constraint.

  (c.3) All the events that match this constraint.

(d) In addition to this, the dependency specification could also include timing constraints on the delivery of dependent events. This timing constraint specifies the time after the delivery of an event, that the dependent event should be delivered.

3.1 Event Stream Specifications

In this section we formally specify the streams, and the dependencies that exist between the events in one stream to the events within other streams. The dependencies are specified by the stream interaction rules within the event streams and controlled by the occurrence vector which dictates the number of events from a specific stream that an event can have a dependency on. We also formulate the resolution of these dependencies and how this subsequently leads to the creation of merged event streams. The event streaming problem is one of routing these merged event streams to clients.

Equation (3.1) specifies the relationships that exist within the events of an event stream. These relationships exist within the context of space and time. In the spatial domain the events within an event stream could be *precedence related* ($\leadsto$) or could have a simple logical relationship with each other. In the former case the event stream is an ordered set of events, while in the second case the stream is an unordered set which could be logically ordered based on the relationship that events would share with each other. In addition to the logical or precedence relationship existing between events within an event stream, events could be constrained by time's arrow. This arrow is a relative notion of time and always points in the same direction.

The timing constraint could be specified in terms of the time following the issue of the first event e_0 or the timing between successive events e_i, e_{i+1}. In either case the constraint we choose should be consistent throughout the event stream. Successive events within the stream can be spatially related in an arbitrary fashion, however the timing constraints follow the additional constraint imposed by time's arrow i.e. it should be monotonically increasing. The $\overset{t}{\rightsquigarrow}$ operator completes the spatial precedence relationship in the time domain.

$$E = \overbrace{\{e_0 \overset{t}{\rightsquigarrow} e_1 \overset{t}{\rightsquigarrow} \cdots\}}^{\text{Ordered Set}} \mid \overbrace{\{e_0 \overset{t}{;} e_1 \overset{t}{;} e_2 \cdots\}}^{\text{Unordered Set}} \tag{3.1}$$

In equation (3.2), $\hookrightarrow$ is the dependency operator, if $E \hookrightarrow E^j$ we say that E has a dependency on E^j. The dependency, $\hookrightarrow$ of a stream E on multiple streams is determined by the dependency of every event e within the stream. The set Π contains all the streams that events in E could possibly be interested in. As an aside, E would be the stream that clients would express their interest in and not $E^j \in \Pi$.

$$E \hookrightarrow \Pi = \{E^1, E^2, E^3, \cdots, E^N\} \tag{3.2}$$

The dependency relation $\hookrightarrow$ is the product of the spatial dependency relation $\overset{s}{\hookrightarrow}$ and the associated chronological dependency $\overset{t}{\hookrightarrow}$ that exist within the events in streams. Even though there may be no timing constraints imposed on successive events, they are still time constrained, in that they would be delivered only after $\overset{s}{\hookrightarrow}$ is resolved. The passage of time in the direction of time's arrow is marked by a succession of significant events which have been $\overset{s}{\hookrightarrow}$ and $\overset{t}{\hookrightarrow}$ resolved.

$$\hookrightarrow \; = \; \overset{s}{\hookrightarrow} \times \overset{t}{\hookrightarrow} \tag{3.3}$$

The occurrence vector $\mathcal{O}$ is used to determine the number of events within other individual streams in Π that an event e in E is interested in. In equation (3.4) we define the values which elements in the occurrence vector can take. This value specified could be one of ? (once or not at all), + (at least once), $*$(zero or more) and $\star$ (one and only one).

$$\text{Occurence Vector } \mathcal{O} = \{?, +, *, \star\} \tag{3.4}$$

Events within an event stream could have a simple mapping which snapshots their dependencies on events within other streams. This mapping $\leftrightarrow$ could be a simple one to one mapping, or a pre defined mapping which is consistent for all events within an event stream. Equation (3.5) is one of the forms that *stream interaction rules* could take. The $\overset{s}{\hookrightarrow}$ specifies the spatial dependency that exist between events in streams.

$$E \leftrightarrow E^j \Rightarrow E.e_i \overset{s}{\hookrightarrow} E_j.e^j_i \mid E.e_i \overset{s}{\hookrightarrow} E^j.e^j_{i\pm N} \text{ where } \leftrightarrow \text{ specifies the mapping rule} \tag{3.5}$$

Equation (3.6) specifies one of the more complex forms that stream interaction rules can take. The function e^{func} could specify either a *constraint* or a more complex *rule* which needs to be satisfied by the events within other event streams. The equation 3.6 snapshots the second half of the stream interaction rules that could exist between different streams and which is used as the basis for the resolution of dependencies that exist within streams.

$$E^j(e^{func}) = \sum e^j \in E^j \ni e^j \text{ satisfies } e^{func}_i \tag{3.6}$$

Equation (3.7) specifies the resolution of an events dependency in the spatial domain. A specific event within an event stream E has a dependency to events within streams in Π or a subset of the streams contained in Π, denoted Π'. The # operator is the cardinality of a set. The operator $\odot$ is the *refinement* of the stream interaction rules with an element of the occurrence vector $\mathcal{O}$. This refinement pin points the precise event/events in $E^j \in \Pi$ that an event in E is dependent on. As is clear, the result of this dependency resolution is either a Null (if $e_i \hookrightarrow \Pi'$ and $\#\Pi' = 0$) or either an event or an array of events as determined by $\#\Pi'$ and the occurrence vector. The array of events could comprises of zero or single or multiple events from each of the event streams in Π.

$$\forall e_i \in E, e_i \overset{s}{\hookrightarrow} \Pi' \subseteq \Pi \equiv \overbrace{e_i(data)}^{\text{Implied}} \cup \sum_{j=1}^{\#\Pi'} \overbrace{\{E \leftrightarrow E^j \mid E^j(e_i^{rule}) \mid E^j(e_i^{tags})\}}^{\text{Stream Interaction Rules}} \odot \overbrace{o_i \in \mathcal{O}}^{\text{Occurrance}} \tag{3.7}$$

$$\equiv Null \mid e \mid e[\,] \tag{3.8}$$

In addition to this, the dependency specification also includes timing constraints on the delivery of dependent events. This timing constraint specifies the time after the delivery of an event, that the dependent events should be delivered. This timing constraint between events in E and Π, is in addition to the timing constraints that exist between the events of a stream. Equation (3.9) follows from equation (3.3) where the product of the spatial resolution and the imposed chronological dependency between events of related streams,specifies the complete dependency resolution.

$$\forall e_i \in E, e_i \hookrightarrow \Pi' \subseteq \Pi \equiv \left(e_i \overset{s}{\hookrightarrow} \Pi' \subseteq \Pi\right) \times \overbrace{0 \mid t_i \mid t_i[\,]}^{\text{Timing Constraints}} \quad . \tag{3.9}$$

Equation (3.10) details the creation of a merged event stream after the resolution of dependencies within Π of every event e_i within an event stream E as specified by the event dependency resolution in equation (3.9). The event dependency resolution of every event within E results in the creation of the merged event stream.

$$\sum_{i=0}^{\#E} e_i \hookrightarrow \Pi' \subseteq \Pi = E^{MergedStream} \tag{3.10}$$

3.2 Stream Properties

(a) For an event stream $E = \{e_0 \rightsquigarrow e_1 \rightsquigarrow \cdots\}$ and $e_i, e_j \in E$, if $e_i \rightsquigarrow e_j$ then no client can deliver e_j before e_i. Also clients cannot deliver e_j unless the dependencies of e_i are resolved.

(b) If $E \hookrightarrow E^j$ and $E.e_i \hookrightarrow E^j.e^j$ based on the stream interaction rules and the occurrence vector then no client delivers e^j before e_i.

(c) For a client interested in an event stream E and $E \hookrightarrow \Pi$ then every such client eventually delivers the merged event stream $\sum_{i=0}^{\#E}(e_i \hookrightarrow \Pi' \subseteq \Pi)$.

4 The Anatomy of an Event

When we refer to an event we refer to the occurrence and the information it points to. The information contained in the event comprises of

- The occurrence which snapshots the context, priority and the application.
- Attribute information which constitutes the event type.
- Control information.
- Destination Lists

The attribute information comprises of tags which specify the attributes associated with the event type while the control information specifies the constraints associated with that event viz. ordering, stability.

4.1 The Occurrence

The *occurrence* relates to the cause which evinces an action or a series of actions. Thus for a person Bob, who would like to check mail, the occurrence is

```
‘‘Bob wants to check his mail’’
```

4.1.1 The event context

The event context pertains to whether the event is a normal, playback or recovery event. Also events could be a response to some other event and associated actions.

4.1.2 Application Type

This pertains to the application which has issued a particular event. This information could be used be used by message transformation switches to render it useful/readable by other applications.

4.1.3 Priority

Events can be prioritized, the information regarding the priority can be encoded within the event itself. The service model for prioritized events differs from events with a normal priority. Some of the prioritized events can be preemptive i.e. the processing of a normal event could be suspended to service the priority event.

4.2 Attribute Information

The attribute information comprises of information which describe the event uniquely and completely (tagged information).

4.2.1 Tagged Information & the event type

The tagged information contains values for the tags which describe the event and also for the tags which would be needed to process the event. The tags also allow for various extraction operations to be performed on an event. The tags specify the type of the event. Events with identical tags but different values for one or more of these tags are all events of the same event type.

4.2.2 Unique Events - Generation of unique identifiers

Associated with every event e sent by client nodes in the system is an event-ID, denoted $e.id$, which uniquely determines the event e, from any other event e' in the system. These ID's thus have the requirement that they be unique in both space and time. Clients in the system are assigned Ids, ClientID, based on the type of information issued and other factors such as location, application domain etc. To sum it up client's use pre-assigned Ids while sending events. This reduces the uniqueness problem, alluded earlier to a point in space. The discussion further down implies that the problem has been reduced to this point in space.

Associating a timestamp, $e.timeStamp$, with every event e issued restricts the rate (for uniquely identifiable[3] events) of events sent by the client to one event per granularity of the clock of the underlying system. Resorting to sending events without a timestamp, but with increasing sequence numbers, $e.sequenceNumber$, being assigned to every sent event results in the ability to send events at a rate independent of the underlying clock. However, such an approach results in the following drawbacks

a) If the client node issues an infinite number of events, and also since the sequence numbers are monotonically increasing, the sequence number assigned to events could get arbitrarily large i.e. $e.sequenceNumber \rightarrow \infty$.

b) Also, if the client node were to recover from a crash failure it would need to issue events starting from the sequence number of the last event prior to the failure, since the event would be deemed a duplicate otherwise.

A combination of timestamp and sequence numbers solves these problems. The timestamp is calculated the first time a client node starts up, and is also calculated after sending a certain number of events $sequenceNumber.MAX$. In this case the maximum sending rate is related to both $sequenceNumber.MAX$ and the granularity of the clock of the underlying system. Thus the event ID comprises of a tuple of the following named data fields : $e.PubID$, $e.timeStamp$ and $e.sequenceNumber$. Events issued with different times t1 and t2 indicate which event was issued earlier, for events with the same timestamp the greater the timestamp the later the event was issued.

4.3 Control Information

The control information specifies the delivery constraints that the system should impose on the event. This control information is specified either implicitly or explicitly by the client. Each of these specifiers have a default value which would be over-ridden by any value specified by the client. Control Information is an agreement between the issuer, the system and the intended recipients on the constraints that should be met prior to delivery at any client.

4.3.1 Time-To-Live (TTL)

The TTL identifier specifies the maximum number of server hops that are allowed before the event is discarded by the system.

4.3.2 Correlation Identifiers

Correlation identifiers help impose the causal delivery constraints on the request→reply events.

4.3.3 Qualities of Service Specifiers

QoS specifiers pertains to the ordering and delivery constraints that events should satisfy prior to delivery by clients.

[3]When events are published at a rate higher than the granularity of the underlying system clock, its possible for events e and e' to be published with the same timestamp. Thus, one of these events e or e' would be garbage collected as a duplicate message.

4.4 Destination Lists

A particular event may be consumed by zero or more clients registered with the system. Events have implicit or explicit information pertaining to the clients which are interested in the event. In the former case we say that the destination list is *internal* to the event, while in the latter case the destination list is *external* to the event.

An example of an internal destination list is "Mail" where the recipients are clearly stated. Examples of external destination lists include sports score, stock quotes etc. where there is no way for the issuing client to be aware of the destination lists. External destination lists are a function of the system and the types of events that the clients, of the system, have registered their interest in.

4.5 Derived events

The notion of derived events exists to provide means to express hierarchical relationships. These derived events add more attributes to the base event attribute information discussed in Section 4.2.1. Derived events can be processed as base events and not vice versa.

4.6 The constraint relation

In addition to derived events, clients could specify *matching constraints* on some of the event attribute information. A constraint specifies the values which some of the attributes, within an event type, can take to be considered an *interesting event*. Constraints on the same event type t can vary, depending on the different values each attribute can take and also depending on the attributes included within the constraint. A constraint $g(t)$ on an event type t could be stronger, denoted $>$ than another constraint $f(t)$ on the same event type i.e. $g(t) > f(t)$. The constraint relation $>^*$ denotes the transitive closure of $>$.

Consider an event type with attributes a, b, c, d. Consider a constraint g which specifies values for attributes a, b and a constraint f which specifies values for attributes a, b, c then $f > g$. However no relation exists between 2 constraints f and g if

- They specify constraints on different event types i.e. $f(t), g(t')$
- They specify constraints on identical attributes
- They specify constraints on attributes within the same event type which do not share a subset/superset relationship.
  Formally $f(t).attributes \supset g(t).attributes \bigcap f(t).attributes \subset g(t).attributes$

4.7 Specifying the anatomy of an event

These sets of equations follow from our discussions in section 4 and section 3.1. Equation (4.1) follows from our discussions in section 4.2.2 regarding the generation of unique identifiers. This tuple is created by the issuing clients.

$$eventId =< clientId, timeStamp, seqNumber, incarnation > \tag{4.1}$$

The tuple in 4.2 discriminates between live events and recovery events (which occur due to failures or prolong disconnects).

$$liveness =< live|recovery > \tag{4.2}$$

The type of an event is dictated by the event *signature*. These signatures could change, to accommodate these changes we include the concept of versioning in our event signatures. This along with *liveness* (equation 4.2) describe the event type completely.

$$eventType =< signature, versionNum, liveness > \tag{4.3}$$

Destination lists within an event could be internal to the event in which case it would be explicitly provided or it could be external to the event in which the destination lists would be computed by the system.

$$destinationLists =< \overbrace{Implied}^{\text{External}} \mid \overbrace{Explicit}^{\text{Internal}} > \tag{4.4}$$

The dependency indicator follows from our discussions in section 3.1 and equations (3.4) through (3.9).

$$spatialDependency =<? \mid * \mid + \mid \star > \odot < mapping \mid rules \mid constraints > \tag{4.5}$$

The data within the event is contained within the values which different attributes in the *attributesList* can take.

$$\begin{aligned} event \quad = \quad & < eventId, eventType, attributesList, spatialDependency, timingDependency \\ & stream, applicationType, destinationLists > \end{aligned} \tag{4.6}$$

5 The Rationale for a Distributed Model

One of the reasons why one would use a distributed model is high availability. Having a centralized model would imply a single server hosting multiple clients. While, this is simple model, the inherent simplicity is more than offset by the fact that it constitutes a single point of failure. Thus all the clients present in the system would be unable to use any of the services provided by the system till a recovery mechanism kicks in.

A highly available distributed solution would have data replication at various server nodes in the network. Solving issues of consistency while execution operations, in the presence of replication, leads to model where other server nodes can service a client despite certain server node failures.

5.1 Scalability

We envision the system comprising of thousands of clients. Having all these clients being serviced by one central server raises a lot of issues in scalability and associated problems like average response times and latencies.

5.2 Dissemination Issues

Clients of the system could be scattered across wide geographical locations. Having a distributed model distributed model enables the client to connect to server nodes with better response times and lower communication latencies.

5.3 Redundancy Models

To ensure guaranteed services for clients, a distributed model lends itself very easily for the construction of redundancy levels. This redundancy can be achieved through replication, multiple levels of connectivity and ensuring consistency.

6 Client

A Client is a user of the system. Client's can generate and consume events in the system. The three issues which describe a client are

- Connection Semantics
- Client Profile
- Logical Addressing

6.1 Connection Semantics

Events in the system have an underlying continuity associated with them. Events are continuously generated and consumed within the system. Clients on the other hand have an inherently discrete connection semantics. Clients can be present in the system for a certain duration of time and can be disconnected later on. Clients reconnect at a later time and receive events which it was supposed to receive as well as events that it is supposed to receive during its present incarnation. Clients can issue/create events while in disconnected mode, which would be held in a local queue to be released to the system during a reconnect.

6.2 Client Profile

A client profile keeps track of information pertinent to the client. This includes

(a) The application type.

(b) The events the client is interested in.

(c) The server node it was attached to in its previous incarnation, and its logical address (discussed in Section 6.3) in that incarnation.

(d) Its current IP address and its IP address in its previous incarnation.

6.3 Logical Addressing

Given its connection semantics (Section 6.1), a client at the epoch of its present incarnation needs to -

- Receive events intended for it from earlier incarnations.
- Issue events which it created while in disconnected mode
- Receive any event currently being issued within the system

The dissemination of this information needs to be done in a *timely* (real time for events currently being published) and *efficient* (minimum number of hops or some function of bandwidth, speed and hops) manner. The issue of logical addressing pertains to this problem of event delivery. At the epoch of the new incarnation there should be a *logical address* associated with the client which would help specify the fastest routing of events to the client.

7 The Server Node Topology

The smallest unit of the system is a *server node* and constitutes level-0 of the system. Server nodes grouped together from a *cluster* and level-1 of the system. Clusters could be clusters in the traditional sense, groups of server nodes connected together by high speed links. A single server node could also decide to be part of such traditional clusters, or along with other such server nodes form a cluster connected together by geographical proximity but not necessarily high speed links. The only requirement that a cluster must satisfy is that at least one node should have access to stable storage. This is to aid the recovery process in case of unit and gateway failures (both transient and permanent) which may take place.

Several such clusters grouped together as an entity comprises the level-2 of our network and are referred to as *super-cluster*, shown in Fig. 7.1. Clusters within a super-cluster have one or more links with at least one of the other clusters within that super-cluster. When we refer to the links between two clusters, we are referring to the links connecting the nodes in those individual clusters. Referring to Figure 7.1 Cluster-A has links to Clusters B, C and D while Cluster-B has links to Clusters A and C. For two clusters with at least one link between them, any node in either of the clusters can communicate with any other node of the other cluster. In general there would be multiple links connecting a single cluster to several other clusters. This approach provides us with a greater degree of fault-tolerance, by providing us with multiple *routes* to reach nodes within other clusters.

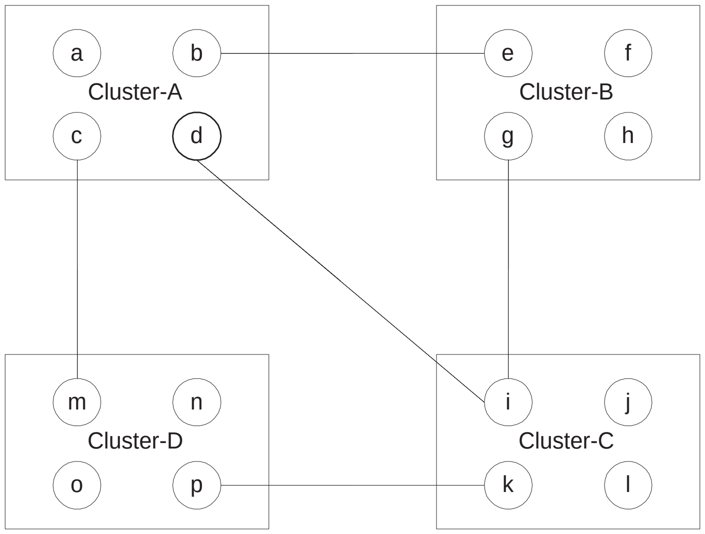

Figure 7.1: A Super Cluster - Cluster Connections

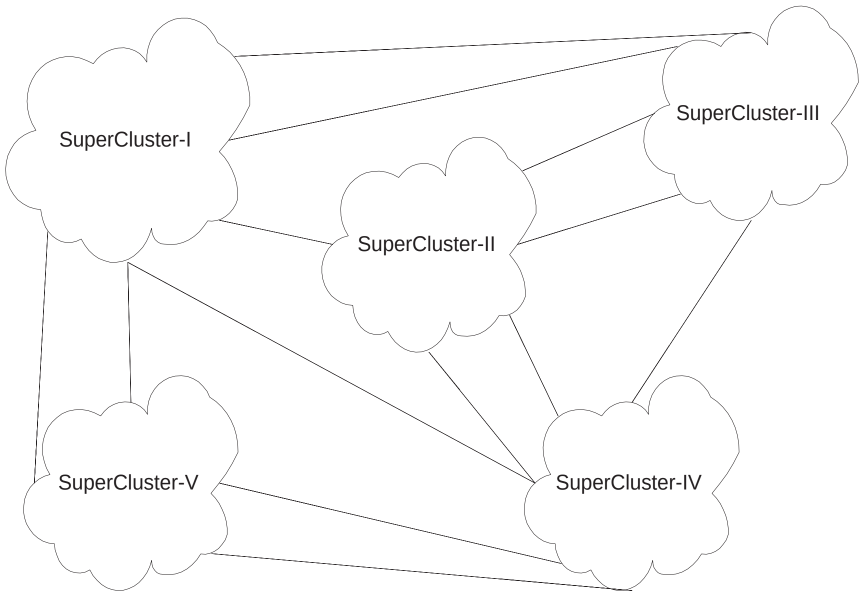

Figure 7.2: A Super-Super-Cluster - Super Cluster Connections

This topology could be extended in a similar fashion to comprise of *super-super-clusters* (level-3) as shown in Fig. 7.2, *super-super-super-clusters* (level-4) and so on. A Client thus connects to a server node, which is part of a cluster, which in turn is part of a super-cluster and so on and so forth. We limit the number of super-clusters within a super-super-cluster, the number of clusters within a super cluster and the number of nodes within a cluster viz. the *block-limit* to 64. In a N-level system this scheme allows for $2^6_{N-1} \times 2^6_{N-2} \times \cdots 2^6_0$ i.e 2^{6*N} server nodes to be present in the system comprising of one super-super-super-cluster which encompasses all the nodes within the system.

7.1 Contexts

Every unit within the system, has a unique context associated with it. In an N-level system, a server exists within the context C^1_i of a cluster, which in turn exists within the context C^2_j of a super-cluster and so on. In general a context C^ℓ_i at level l exists within the context $C^{\ell+1}_j$ of a level $(\ell+1)$. In a N-level system the following hold -

$$C^0_i = (C^1_j, i) \tag{7.1}$$
$$C^1_j = (C^2_k, j) \tag{7.2}$$
$$\vdots$$
$$C^{N-2}_p = (C^{N-1}, p) \tag{7.3}$$
$$C^{N-1}_q = q \tag{7.4}$$

In an N-level system, a unit at level ℓ can be uniquely identified by $(N-\ell)$context identifiers of each of the higher levels. Of course, the units at any level l within a context $C^{\ell+1}_i$ should be able to reach any other unit within that same level. If this condition is not satisfied we have a *network partition.*

7.2 Gatekeepers

Within the context C^2_i of a super-cluster, clusters have server nodes at least one of which is connected to at least one of the nodes existing within some other cluster. In some case cases there would be multiple links from a cluster to some other cluster within the same super-cluster C^2_i. These nodes thus provide a gateway to the other cluster. This architecture provides for a higher degree of fault tolerance by providing multiple routes to reach the same cluster. We refer to such nodes as the *gatekeepers*. Similarly, we would have gateways existing between different super-clusters within a super-super-cluster context C^3_i. In a $N-level$ system similar such gateways would exist at every level within a higher context. A gateway at level ℓ within a higher context $C^{\ell+1}_j$ denoted $g^\ell_i(C^{\ell+1}_j)$ comprises of -

- The higher level Context $C^{\ell+1}_j$
- The Gateway identifier i
- The list of gateways in level ℓ that it is connected to within the context $C^{\ell+1}_j$.

It should be noted that a gatekeeper at level l need not be a gatekeeper at level $(\ell+1)$ and vice-versa. Fig 7.3 shows a system of 78 nodes organized into a system of 4 super-super-clusters, 11 super-clusters and 26 clusters. When a node establishes a link to another node in some other cluster, it provides a gateway for dissemination of events. If the node it connects to is within the same super-cluster context C^2_i both the nodes are designated cluster gateways. In general if a node connects to another node, and the nodes are such that they share the same context $C^{\ell+1}_i$ but have differing contexts C^ℓ_j, C^ℓ_k, the nodes are designated gateways at $level-\ell$ i.e. $g^\ell(C^{\ell+1})$. Thus in Fig 7.3 we have 12 super-super-cluster gateways, 8 super-cluster gateways (6 each in SSC-A and SSC-C, 4 in SSC-B and 2 in SSC-D) and 4 cluster-gateways in super-cluster SC-1.

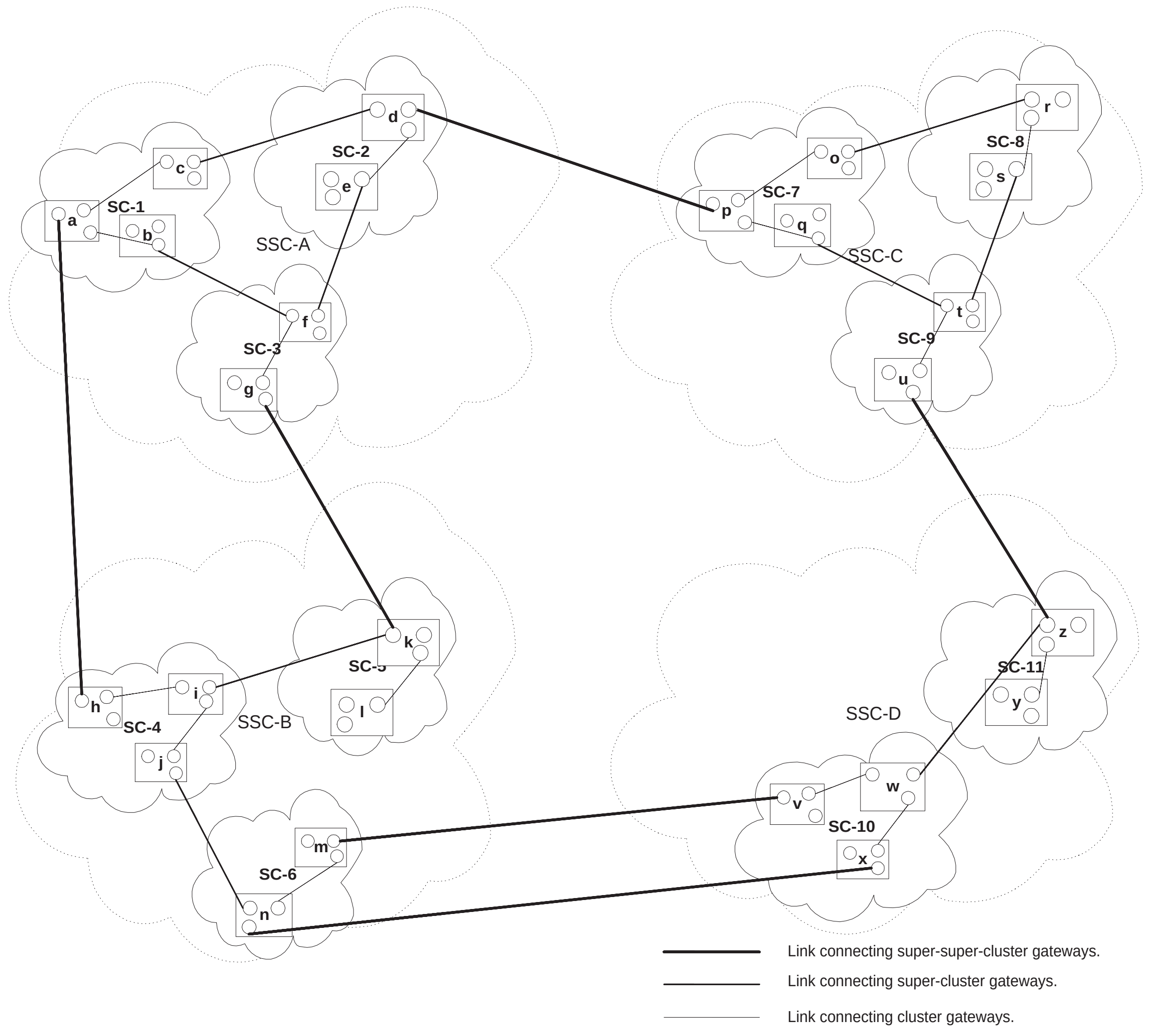
SSC-A
SC-1
SC-2
SC-3
a
b
c
d
e
f
g
SSC-B
SC-4
SC-5
SC-6
h
i
j
k
l
m
n
SSC-C
SC-7
SC-8
SC-9
o
p
q
r
s
t
u
SSC-D
SC-10
SC-11
v
w
x
y
z
Link connecting super-super-cluster gateways.
Link connecting super-cluster gateways.
Link connecting cluster gateways.

Figure 7.3: Gatekeepers and the organization of the system

7.3 The addressing scheme

The addressing scheme provides us with a way to uniquely identify each server node within the system. This scheme plays a crucial role in the delivery and dissemination of events to nodes in the system(discussed in Section 8.9). As discussed earlier units at each level are defined within the context of a unit at the next higher level. In a N-level system the context C^{ℓ}_{j} is $C^{\ell}_{i} = \overbrace{C^{N}_{j}(C^{N-1}_{k}(\cdots(C^{\ell+1}_{m}(C^{\ell}_{i}))\cdots))}^{N-l}$. Thus in a 4-level system, to identify a server node, the addressing scheme specifies the super-super-cluster C^{3}_{i}, super-cluster C^{2}_{j} and cluster C^{1}_{k} that the node is a part of along with the node-identifier within C^{1}_{k}. Thus for server node a, within cluster B, within super-cluster C and super-super-cluster D the logical address within the system is D.C.B.a.

8 The problem of event delivery

The problem of event delivery pertains to the efficient and reliable delivery of events to the destinations which could be internal or external to the event. In the latter case the system needs to compute the destination lists pertaining to the event. The system merely acts as a conduit to reliably deliver the events from the issuing client to the interested clients. To snapshot the event constraints that need to be satisfied by an event prior to dissemination within a unit we use the Profile Propagation Protocol (PPP) discussed in Section 8.1. Providing precise information for routing of these events, and the updating of this information in response to the addition, recovery and failure of gateways is in the purview of the Gateway Propagation Protocol (GPP) discussed in Section 8.3. In Section 8.6 we present the Event Routing Protocol (ERP) which uses the information provided by GPP to efficiently disseminate events. The problem of routing events is a three pronged problem, which needs to address the basic routing scheme, the routing of real-time events (section 8.7) and events to a newly reconnected client (section 8.9). To ensure the fastest dissemination of events the following are the desirable objectives -

(a) We need to route the event to the highest order gateway first or as soon as possible. In the case of an $N - level$ system we are of course referring to the $g^{N-1}(C^N)$. What this provides us, is the optimum amount of concurrency in the dissemination of events.

(b) It is possible that we may encounter lower-level gateways en route. The dissemination of events can proceed once the event has been routed on its way to the highest order gateway.

(c) The nodes must be fairly smart enough to decide which is the next best node to route this event to. Of course we will be using gateways to get across to nodes within a different context.

(d) A gateway g^ℓ could use the $g^{\ell-1}$ information within the same context $C_i^{\ell+1}$ to ensure delivery to other gateways $g^\ell(C_i^{\ell+1})$.

8.1 The profile propagation protocol - PPP

A gatekeeper at $level - \ell$ needs to snapshot the behavior of all the client nodes that exist in the unit within the context C. This information could be used by the gatekeepers to decide if the event needs to be routed within the unit.

Clients specify the constraints (Section 4.6 on page 11) on the kinds of events that they are interested in. The organization of our system needs to be exploited so that units not interested in the event are *not* flooded by events that do not satisfy the constraints specified by any of the client attached to server nodes within that unit. Each server node, which could be considered as a gatekeeper at level-0 for the client nodes attached to it, keeps track of these constraints for every client node attached to it. The server node profile is the profile of all the client nodes attached to it.

Gatekeepers at $level - \ell$ snapshot the profile of the system from $level - 0$ to $level - l$ within the context $C_i^{\ell+1}$. We refer to this snapshot as the *range* of the gatekeeper. Thus, the range of a cluster gatekeeper is the server nodes within that cluster while that of a super-cluster is all the clusters within that super-cluster. Gatekeeper profiles keep track of the weakest constraints that exists within its system range. The gatekeeper profile should be able to -

- Capture the commonalities of the server nodes (albeit expressed by the client nodes that are attached to that node) within its range
- Expressive enough to snapshot the profile of each and every client connected to every server node within its system range.

When a new client joins in, or just changes its profile, the server checks to see if this new profile change would result in a change in the server's profile. If this does result in a change all the gatekeepers with the server's cluster are notified about this change. The same procedure is followed at the higher levels of the system, causing super-cluster gatekeepers to be notified when any of the cluster gatekeepers within its context report a profile change. The range at gatekeeper g_i^ℓ, denoted ω_i^ℓ, snapshots the profile of all clients attached with the level-ℓ unit. Profile changes are initiated by clients, this profile change could be stronger ($+\delta\omega$) or weaker ($-\delta\omega$) than the original one. As we discussed earlier the node to which the client is attached can be considered a level-0 gatekeeper. At g^ℓ for a profile change $+\delta\omega/+\delta\omega$ the following cases are possible.

$$\omega^\ell_{old} \cup +\delta\omega \longrightarrow \omega^\ell_{new} \tag{8.1}$$

$$\omega^\ell_{old} \cup +\delta\omega \longrightarrow \omega^\ell_{old} \tag{8.2}$$

$$\omega^\ell_{old} \cup -\delta\omega \longrightarrow \omega^\ell_{new} \tag{8.3}$$

$$\omega^\ell_{old} \cup -\delta\omega \longrightarrow \omega^\ell_{old} \tag{8.4}$$

Equations (8.1) and (8.3) result in profile change propagation while equations (8.2) and (8.4) do not need any profile change propagation. In general if there's a change in the profile of a gatekeeper g^ℓ_i within the context $C^{\ell+1}_j$, all the level-$(\ell+1)$ gatekeepers, $g^{\ell+1}$, at context $C^{\ell+1}_j$ within $C^{\ell+2}_k$ are notified about this change. In an $N - level$ system this process of updating the 'higher level' gatekeeper profile stops if the profile update occurs at a level $\ell = N - 1$.

When a profile change just adds/deletes the number of units interested in the event type - not much of updating needs to be done. A server node keeps track of the number of clients interested in a specific event constraint. While a cluster gateway keeps track of the server nodes interested in the event constraint. It is quite possible that a profile change at a client could change the profile of the server node, which in turn could cause a change in the profile of cluster gateway.

8.1.1 Active profiles

The profile propagation protocol aids in the creation of destination lists at units within different levels. These destination lists are then employed at each level for finer grained disseminations. Since the profile add/change propagates through the system to higher level gateways, it is possible that a gateway at a higher level hasn't yet been notified about the profile add/change. Thus though it may receive an event which would match the profile change, the destination list may not include the lower level unit. It is possible that a client may receive events issued by clients within a certain unit, though it may not receive similar events from clients published by units within a different context.

What interests us is the precise instant of time from which point on we can say that all events that satisfy the client's profile will be delivered to the client. To address this issue we introduce the concept of *active profiles*, which provides guarantees in the routing of events within a unit. The active profile approach provides us with a unit-based incremental approach towards achieving system guarantees during a profile add/change. If a profile is *super-cluster active* all events issued by clients attached to any of the server nodes within a super-cluster C^2_i will be routed to the interested client. Thus the first event that is received by the client is an indication that all subsequent events routed to that unit, matching the same profile would also be delivered by the client. When we say that a profile is *unit-active*[4] what we mean is that for every event that is being routed within that unit the destination lists calculated would include information to facilitate routing to the client. Since a client profile is unit active all events, issued within the unit, will be routed to the client if it satisfies the client profile.

Theorem 8.1 *For a change $+\delta\omega$ in a client's profile, if this client delivers an event e corresponding to the $+\delta\omega$ and if the routing information contained in the client differs with the contextual information of the client, and if the difference (bottom-up) is at level-ℓ then this client will deliver all events issued by clients within the same context (till level-ℓ) from that point on.*

When a client (after it has initiated a profile add/change) receives an event with just the cluster routing information, it is cluster active. Every node is part of a cluster, which in turn is part of a super-cluster and so on. Similarly if it receives an event with routing information (discussed in section 8.6) pertaining to contexts different from the node it is attached to, the highest such unit where this

[4]The unit we are referring to in this case are the clusters, super-clusters, super-super-clusters etc. Of course these units are assumed to be within some higher level context of the server node to which the interested client is attached to or was last attached to

difference occurs forms part of the profiles active unit. Reception of events with differing contextual information also implies that the profile propagation has been successfully completed. This is because if the profile propagation hadn't been completed the event wouldn't be routed to that gateway in the first place. Its quite possible that the profile propagation can continue to the highest level. If the contextual information which differs is that at level-2 the client's profile is said to be super-cluster active.

Of course if a client profile change doesn't result in profile change propagation beyond the cluster, the changed profile is *system active*. A system active profile ensure system wide guarantees for the properties listed in section 2.5 The profile remains active from then on till a change is made to the profile. Also, it is quite possible that no events (matching the profile change) are being issued by any client throughout the system. But the client needs to be aware of its system guarantees as the profile propagation process is taking place. To accommodate such a scenario we also require the gateways to route these system guarantee events once the profile change has effected that unit.

Theorem 8.2 *For an N-level system, for a profile change $\pm\delta\omega$ initiated by a client eventually the client profile change $\pm\delta\omega$ is $N-1$ active.*

8.1.2 Profile changes which are weaker than the original one ($-\delta\omega$)

The question we are trying to address here concerns a profile change which introduces a weaker constraint than the original one. This results in a client with this weaker constraint receiving some of the events which have been routed due to the existing stronger constraint. The client would thus assume that its profile is system active, which is not the case since there would be events arriving at the higher level gateways[5] (which haven't been notified about this weaker constraint by the PPP) which wouldn't be disseminated within the unit.

To account for such a scenario we augment our active profiles concept to also send notifications to a client informing the weaker constraint currently being added, and thus wait for system notifications regarding the active profile status instead of conjecturing based on the event routing information.

8.1.3 The propagation scheme

It should be understood that a client is attached to a server node which is part of a cluster, super-cluster and so on. Now this node itself could be a gatekeeper g^{ℓ} where $\ell = 1, 2, \cdots, N-1$ in a N-level system. It is also possible that the cluster that this node is a part of may possess gatekeepers g^{ℓ} where $\ell = 1, 2, \cdots, N-1$. The profile propagation scheme which proceeds in an incremental manner needs to be clearly defined to account for such scenarios.

8.2 The node organization protocol

Each node within a cluster has set of connection properties. These pertain to the rules of adding new nodes to the cluster, specifically some node may employ an IP-based discrimination scheme to add or accept new nodes within the cluster. In addition to this nodes also maintain a connection threshold vector which pertain to the number of gateways at each level that the node can maintain connections to, concurrently at any given time.

Nodes wishing to join the network, do so by issuing a connection set up request to one of the nodes in the existing network. The organization and logical addresses assigned are based relative to the existing logical address of the node to which this request was sent to. Nodes issuing such a set up request could be a single stand-alone node or part of an existing unit. New addresses are assigned based on the discrimination that the node is part of the existing system, or whether the node is part of a new unit being merged into the system. In the latter case no new logical address are assigned, while in the former case new logical addresses need to be assigned. Clients of the merged system need to renegotiate their new logical address using an address renegotiation protocol.

[5]These gateways share a higher level context with the node to which the client, initiating a profile change, is attached to.

8.2.1 Adding a new node to the system

Nodes which issue a connection setup request need to indicate the kind of gatekeeper that it seeks to be within the existing system. An indication of whether it seeks to be a level-0 system or not dictates the context the requesting node seeks to share with the node to which it has issued the request. If the node wishes to be a level-1 gatekeeper with the node in question, the two nodes would end up sharing a similar system context C_i^1. The level-1 indication establishes the *to* and *from* relationship between the requester and the addressee. The context varies depending on this relationship. In the event that the requester seeks to be a level-1 gatekeeper, the contextual information varies at the lowest level C_i^0. In the event that the requester seeks a *to* relationship with the addressee, the contextual information of the requester varies starting from the highest level-l gatekeeper that it seeks to be. Thus if the requester seeks to be a level-3, level-2 gatekeeper the contextual information vis-a-vis the addressee varies from level-3 onwards.

Nodes request the connection setup in a bit vector specifying the kind of gatekeeper it seeks to be. The position of 0's and 1's dictate the kind of gatekeeper a node seeks to be. The first position specifies the *to/from* characteristics of the node seeking to be a part of the system. A 0 signifies the *to* relationship while the 1 specifies the *from* relationship. A connection request $< 00000011 >$ from node **s** indicates that it wishes to be a configured as a cluster gatekeeper in cluster **n** to one of the clusters within super-cluster **SC-6**. Similarly a connection request $< 00000110 >$ from node **s** signifies that it wishes to configured as a level-3 gateway to supercluster **SC-6** and as a level-2 (cluster) gateway within the super-cluster (SC-4/SC-5) that it would be a part of.

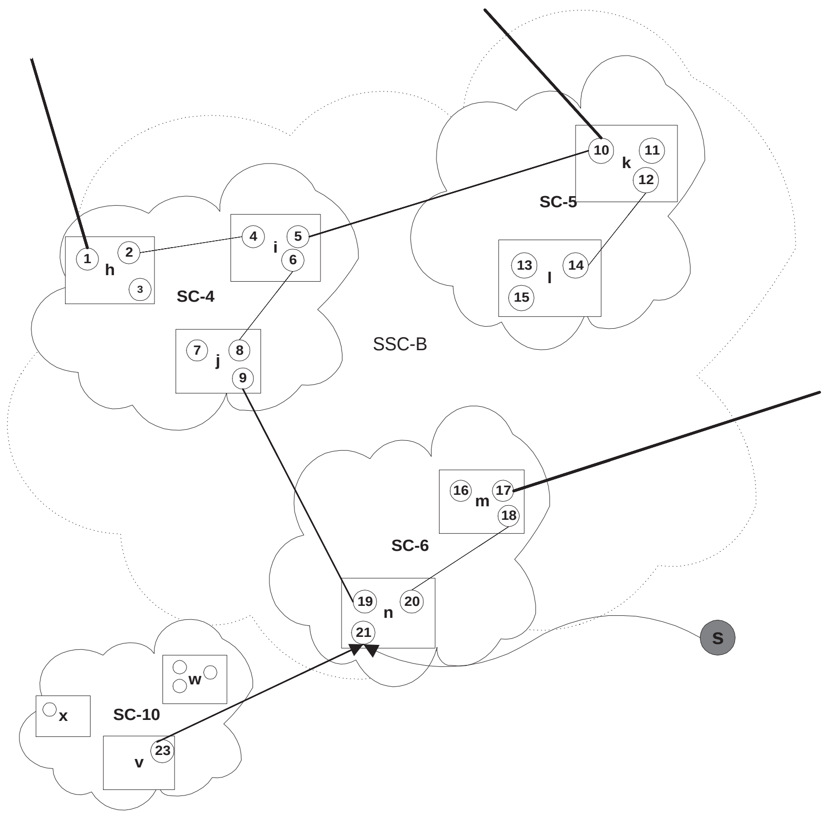

Figure 8.1: Adding nodes and units to an existing system

Figure 8.1 depicts a node **s** requesting a connection setup request. If **s** requests to be a level-1 node, then it needs to be part of the cluster **n**. Now, if node **n.21** hasn't exceeded the connection threshold limit for level-1 connections and also if the node **s** satisfies the IP-discrimination scheme for accepting nodes within the cluster then node **s** is configured as a level-1 node with a connection to node **n.21**. If however, node **n.21** has reached its connection threshold for level-1 connections but node **s** has satisfied the IP-discrimination requirements for cluster **n** then **n.21** forwards the request to other nodes within the cluster **n**. If there is a node within the cluster **n** which hasn't reached connection threshold limit, then node **s** is configured as a level-1 gateway to such a node in cluster **n**. If however, there are no nodes which have not reached their connection threshold limit, the node responds by providing a list of level-2 gatekeepers that are connected to cluster **n**. Node **s** then proceeds with the same process discussed earlier.

If node **s** doesn't seek to be a level-1 gatekeeper within cluster **n** but seeks to be a level-(l+1), $l > 0$, gateway to cluster **n** the procedure for setting up connections are different. Depending on the kind of gatekeeper that node **s** seeks to be, the location of suitable nodes which could satisfy the request varies. If the nodes seeks to be a level-2 gatekeeper *to* cluster **n**, then node **n.21** confirms the connection threshold vector. If there are no nodes that have not reached their connection threshold for level-2 gateways it returns a failed response. If however there is such a node in cluster **n** which hasn't reached its threshold for level-2 connections node **n.21** provides the address for such a node, and also the addresses of level-2 gatekeepers within supercluster **SC-6** to which it is connected. Node **s** then tries to be a level-1 gateway within cluster **m** which is also a level-2 gateway to the node in cluster **n**. If there are no clusters within super-cluster **SC-6** other than cluster **n** which can accept **s** as a level-1 gatekeeper, then the request fails.

8.2.2 Adding a new unit to the system

The unit that can be added to system could be a cluster, a super-cluster and so on. The process of adding a new unit to the system must follow rules which are consistent with the organization of the system. These rules are simple, a node can be a level-1 gatekeeper of only one cluster. Thus a node in an existing cluster cannot seek to be part of a cluster in another system. In general for a unit at level-l which is being added to the system, any node in such a system cannot seek to be a level-(n-1) gatekeeper to any sub-system of the existing system.

The process of adding a unit to the system, results in the update of the contextual information pertaining to every node within the added unit. This update is only for the highest level of the system. Lower level contextual information remains the same. Thus nodes within a cluster would have a context with respect to the cluster context C_i^1, when this cluster is added to the system, what changes is the context C_i^1 while the individual contexts of the nodes with respect to newly assigned cluster context C_j^1 is the same as it was before.

Figure 8.1 depicts the addition of a super cluster SC-10 to the system. Only one node within the unit that needs to be added can issue the connection setup request. The node which issues this request in Figure 8.1 is the node **SC-10.v.23**. Since this is a level-3 system that is unit added, node **23** or any other node with SC-10 can not be level-2 or level-1 gateway within the super-super-cluster SSC-B. Node 21 thus issues a request specifying that it seeks to be a level-3 gateway within super-super-cluster B. Upon a successful connection set up, a new address is assigned for SC-10 (say SSC-8) identifiers for clusters within SC-10 remain the same. However the complete address of these clusters change to **SSC-B.SC-8.w** and so on.

8.2.3 Handling requests for setting up higher level gateways

The approach varies depending on the *to* or *from* relationship with the addressee. If the node is going to be part of the cluster, the nodes need to contact one of the higher level nodes that it is connected to, and try to establish a connection with one of the other nodes within the unit l.

8.3 The gateway propagation protocol - GPP

The gateway propagation protocol (GPP) accounts for the process of the addition of a gateway. However, GPP should also account for failure suspicions/confirmations of nodes and links, and provide information for alternative routing schemes. Addition or deletion of links are events of relatively low occurrence. These operations could thus be allowed to be reasonably expensive. Routing should work fine if -

- All the nodes at level-ℓ within a context $C_i^{\ell+1}$ are aware of all the level-ℓ gateways $g^\ell(C_i^{\ell+1})$.
- The nodes are aware of the precise locations and connectivities of each of these gateways. In a N-level system, we could visualize each node having a stack comprising of $level - 0, 1, \cdots, N-1$ gateway information.

This scheme though it serves the purpose, is highly inefficient and would entail almost every node in the system being aware of almost[6] every other node in the system. However, if you optimize by having only gateways $g^\ell(C_i^{\ell+1})$ being aware of a newly added gateway $g_j^\ell(C_i^{\ell+1})$ then the objectives in the earlier section that we set out to meet are not met.

The first time a new gateway is added, the $level - \ell$ gateway $g^\ell(C_i^{\ell+1})$ sends a message to be routed to all the nodes within $C^{\ell+1}$. This message could take different routes to reach the other $level - \ell$ gateways within $C_i^{\ell+1}$. When the message is being disseminated the message keeps track of the hops it is taking. Now when a message is received by a gateway, the first such message provides the fastest route to reach the new gateway from the existing one and vice-versa. The message is routed in a similar way back to the new gateway. All the nodes en route to this new gateway keep track of the reachability vector for this gateway.

Every node is aware of -

- Every level-ℓ, where $0 < \ell < N$, gateway that exists within the cluster it belongs to.

Every gateway at level-ℓ is aware of all the other $g^\ell(C_j^{\ell+1})$ within the context $C_j^{\ell+1}$. Thus every cluster gateway maintains a list of all the cluster gateways within a specific super-cluster. When a new gateway is added at level-ℓ this information is disseminated within the context $C_j^{\ell+1}$. All other gateways $g^\ell(C_j^{\ell+1})$ then update their information to include this new gateway.

What a node also needs to decide is when it would be futile to try and find a higher order gateway, and also when all the higher level units that could possibly be covered have been covered. Of course it should also know if there's a higher order gateway that needs to be reached. This decision is based on the event routing information provided by the event routing protocol (discussed in section 8.6) and the information pertaining to gateways that's available at a node. If there are no such units that need to be reached, the event routing would proceed with lower order disseminations. However if there is a unit that needs to be reached, gateways would have to be employed to reach this unit as fast as possible.

At the same time we would need to understand and utilize the concept of proximity while routing over gateways. If the event routing information doesn't include a specific unit, it doesn't imply that the event hasn't reached that unit. The event routing information contained with an event simply indicates the units which were present en route to reception at the node. Now based on the information that an event was at a certain unit we can conjecture whether the event was routed to a particular unit. As discussed in Section 7.2 gatekeepers are employed across units within a given level ℓ or across levels $\cdots \ell-1, \ell, \ell+1 \cdots$. If the system is aware of the precise location of these gateways, the system can deduce if the event was delivered by any node within that unit.

We are of course assuming that the gateways across the units are fully functional. This scheme works fine since it is the responsibility of the unit to deal with failures to nodes, gatekeepers and gateways within that unit. Thus in a 4-level system if there's a level-3 gateway between super-super-clusters 1 and 2, and if the routing information for a event e is $e.\{1,3,4,9\}\{b,d,g\}\{C,E\}$ at a node $7.k.C$ we can decide that event was indeed routed to 2.

Thus the rationale for all gatekeepers being aware of all other gatekeepers does seem to hold water. But what we need most importantly as an optimization feature are the following -

- Gatekeepers should possess the minimal information to go about their routing schemes.
- How do they use this minimal information to extrapolate and arrive at conjectures based on the event routing information.
- What does it entail to have all gatekeepers being aware of each other, and how do we plan to relax this. How do we propagate these changes fast and reliably.

[6]This is possible in the case of a strongly connected network. In such a situation, the order of the nodes present in the system approaches that of the gatekeepers present in the system

8.4 Organization of gateways

The organization of gateways reflects the connectivities which exist between various units within the system. Using this information, a node should be able to communicate and take actions to for any other node within the system. Any given node within the system is connected to one or more other nodes within the system. We refer to these direct links from a given node to any other node as *hops*. The routing information associated with an event, specifies the units which should receive an event. At each $g^\ell(C_i^\ell)$ finer grained disseminations targetted for units u^ℓ within C_i^ℓ are computed. When presented with such a list of destinations, based on the gateway information the best hops to take to reach a certain destination needs to be computed. A node is required to route the event in such a way that it can service both the coarser grained disseminations and the finer grained ones. Thus a node should be able to compute the hops that need to be taken to reach units at different levels. A node is a level-0 unit, however it computes the hops to take to reach level-ℓ units within its context $C^{\ell+1}$ where $\ell = 0, 1, \cdots, N$ where $+1$ is the system level.

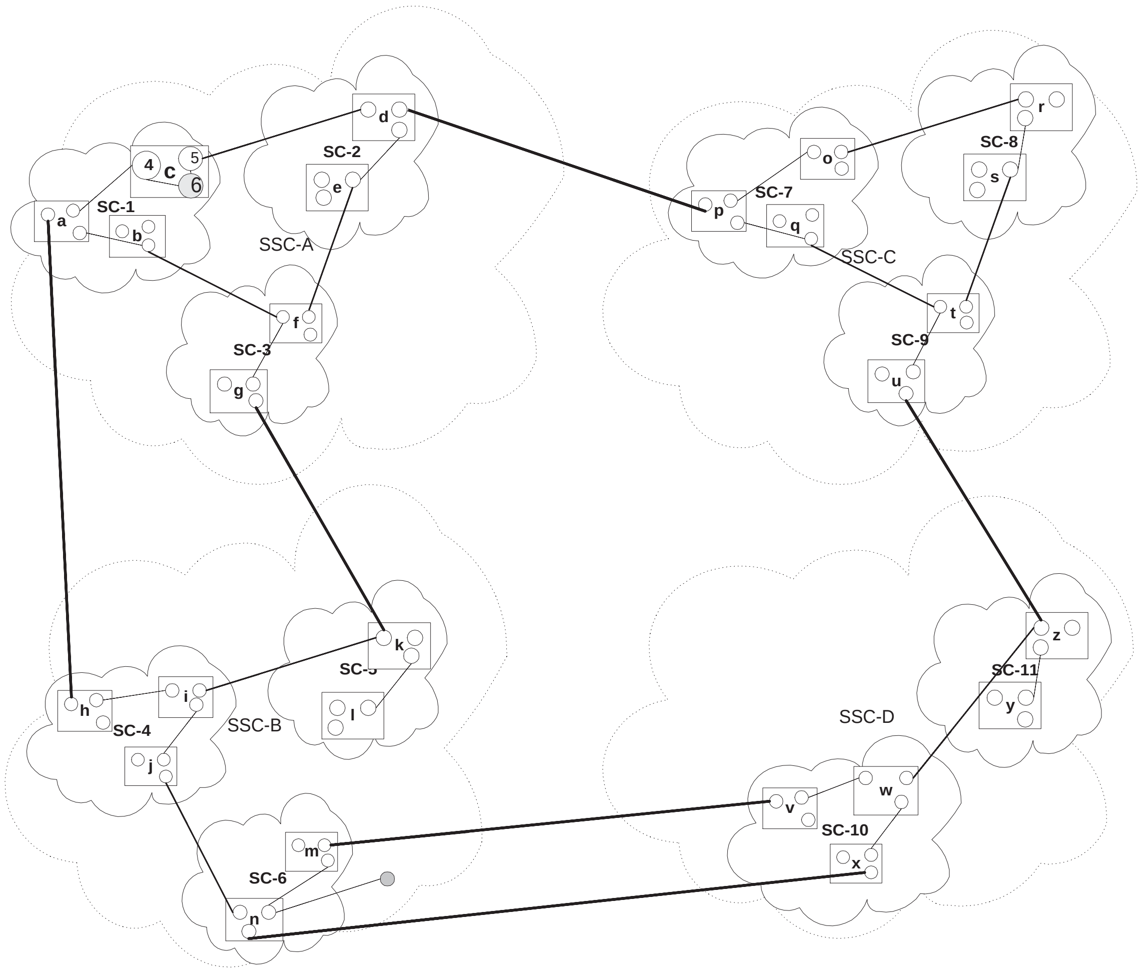

Figure 8.2: Connectivities between units

What is required is thus an abstract notion of the connectivities that exist between various units/super-units within the system. This constitutes the *connectivity graph* of the system. At each node the connectivity graph is different while providing a consistent overall view of the system. The view that is provided by the connectivity graph at a node should be of connectivities that are relevant to the node in question. Figure 8.2 depicts the connections that exist between various units of the 4 level system which we would use as an example in further discussions.

8.4.1 Constructing the connectivity graph

The organization of gateways should be on which provides an abstract notion of the connectivity between units u^{ℓ} within the context $C^{\ell+1}$ of the node. This interconnection can span multiple levels where if the gateway level is ℓ, a unit u_i^x $(x < \ell)$ within the context C^{x+1} is connected to u_j^{ℓ} within $C^{\ell+1}$. Units u_i^x and u_j^{ℓ} share the same $C^{\ell+1}$ context. For any given node within the system, the connectivity graph captures the connections that exist between units u^{ℓ}'s within the context C_i^{ℓ} that it is a part of. Thus every node is aware of all the connections that exist between the nodes within a cluster, and also of the connections that exist between clusters within a super cluster and so on. The connectivity graph is constructed based on the information routed by the system in response to the addition or removal of gateways within the system. This information is contained within the *connection*.

Not all gateway additions or removals/failures affect the connectivity graph at a given node. This is dictated by the restrictions imposed on the dissemination of connection information to specific sub-systems within the system, The connectivity graph should also provide us with information regarding the best hop to take to reach any unit within the system. The link cost matrix maintains the cost associated with traversal over any edge of the connectivity graph. The connectivity graph depicts the connections that exist between units at different levels. Depending on the node that serves as a level-ℓ gatekeeper, the cluster that the node is a part of is depicted as a level-1 unit having a level-ℓ connection to a level-ℓ unit, by all the clusters within the super cluster that the gatekeeper node is a part of.

8.4.2 The connection

A connection depicts the interconnection between units of the system, and defines an edge in the connectivity graph. Interconnections between the units snapshots the kind of gatekeepers that exist within that unit. A connection exists between two gatekeepers. A level-ℓ *node* denoted n_i^{ℓ} in the connectivity graph, is the level-ℓ context of the gatekeeper in question and is the tuple $< u_i^{\ell}, \ell >$.

A level$-\ell$ connection is the tuple $< n_i^x, n_j^y, \ell > \; where \; x \mid y = \ell \; and \; x, y \leq \ell$. Units u_i^x and u_j^y share the same level-ℓ context $C_k^{\ell+1}$. For any given node n_i^{ℓ} in the connectivity graph we are interested only in the level $\ell, \ell+1, \cdots, N$ gatekeepers that exist within the unit and not the $\ell-1, \ell-2, \cdots, 0$ gatekeepers that exist within that unit. Thus, if a level-ℓ connection is established, the connection information is disseminated only within the higher level context $C_i^{\ell+1}$ of the sub-system that the gatekeepers are a part of. This is ensured by never sending a level-ℓ gateway addition information across any gateway $g^{\ell+1}$. Thus, in Figure 8.2 for a super-cluster gateway established within **SSC-A**, the connection information is disseminated only within the units, and subsequently the nodes in SSC-A.

When a level-ℓ connection is established between between two units, the gatekeepers at each end create the connection information in the following manner.

(a) For the gatekeeper at the far end of the connection, the node information in the connection is constructed using its level-ℓ context

(b) The other node of the connection is constructed as level-0 node.

(c) The last element of the connection tuple, is the connection level ℓ_c.

When the connection information is being disseminated through the context $C_i^{\ell+1}$, it arrives at gatekeepers at various levels. Every gatekeeper $g^p \ni p \leq \ell_c$, at which it is received, checks to see if any of the node information depicts a node n^x where $x < \ell_c$ if this is the case the next check is if $p > x$. If $p > x$ the node information is updated to reflect the node as level-p node by including the level-p contextual information of g^p. If $p \not> x$ the connection information is disseminated *as is*. Thus, in Figure 8.2 the connection between **SC-2** and **SC-1** in **SSC-A**, is disseminated as one between node **5** and **SC-2**. When this information is received at **4**, it is sent over as a connection between the cluster **c** and **SC-2**. When the connection between cluster **c** and **SC-2** is sent over the cluster gateway to cluster **b**, the information is not updated. As was previously mentioned, the super cluster connection **(SC-1,SC-2)** information is disseminated only within the super-super-cluster **SSC-A** and is not sent over the super-super-cluster gateway available within the cluster **a** in **SC-1** and cluster **g** in **SC-3**.

8.4.3 The link cost matrix

The link cost matrix specifies the cost associated with traversing a link. The cost associated with traversing a level-ℓ link from a unit u^x increases with increasing values of both x and ℓ. Thus the cost of communication between nodes within a cluster is the cheapest, and progressively increases as the level of the unit that it is connected to increases. The cost associated with communication between units at different levels increases as the levels of the units increases. One of the reasons we have this cost scheme is that the dissemination scheme employed by the system is selective about the links employed for finer grained dissemination. In general a higher level gateway is more overloaded than a lower level gateway. Table 8.1 depicts the cost associated with communication between units at different levels.

level	**0**	**1**	**2**	**3**	$\ell_\mathbf{i}$	$\ell_\mathbf{j}$
0	0	1	2	3	ℓ_i	ℓ_j
1	1	2	3	4	$\ell_i + 1$	$\ell_j + 1$
2	2	3	4	5	$\ell_i + 2$	$\ell_j + 2$
3	3	4	5	6	$\ell_i + 3$	$\ell_j + 3$
$\ell_\mathbf{i}$	ℓ_i	$\ell_i + 1$	$\ell_i + 2$	$\ell_i + 3$	$2 \times \ell_i$	$\ell_i + \ell_j$
$\ell_\mathbf{j}$	ℓ_j	$\ell_j + 1$	$\ell_j + 2$	$\ell_j + 3$	$\ell_j + \ell_i$	$2 \times \ell_j$

Table 8.1: The Link Cost Matrix

8.4.4 Organizing the nodes

The connectivity graph is different at every node, while providing a consistent view of the connections that exist within the system. This section describes the organization of the information contained in connections (section 8.4.2) and super-imposing costs as specified by the link cost matrix (section 8.4.3) resulting in the creation of a weighted graph. The connectivity graph constructed at the node imposes directional constraints on *certain* edges in the graph.

The first node in the connectivity graph is the *vertex*, which is the level-0 server node hosting the connectivity graph. The nodes within the connectivity graph are organized as nodes at various levels. Associated with every level-ℓ node in the graph are two sets of links, the set L_{UL} which comprises connections to nodes $n_i^a \ni a \leq \ell$ and L_D with connections to nodes $n_i^b \ni b > \ell$. When a connection is received at a node, the node checks to see if either of the nodes is present in the connectivity graph. If any of the nodes within the connection is not present in the graph, they are added to the graph. For every connection, $< n_i^x, n_j^y, \ell > \ where \ x \mid y = \ell \ and \ x, y \leq \ell$, that is received if $y \leq x$ node

- n_j^y is added to the set L_{UL} associated with node n_i^x
- n_i^x is added to the set L_D associated with the node n_j^y.

The process is reversed if $x \leq y$. For the edge created between nodes n_i^x and n_j^y, the weight is given by the element (x, y) in the link cost matrix.

Figure 8.3 depicts the connectivity graph that is constructed at the node SSC-A.SC-1.c.6 in Figure 8.2. The set L_{UL} at the node **SC-3** in the figure comprises of node **SC-2** at level-2 and node **b** at level-1. The set L_D at **SC-3** comprises of the node **SSC-B** at level-3. The cost associated with traversal over a level-3 gateway between a level-2 unit **b** and a level-3 unit **SC-3** as computed from the linkcost matrix is 3, and is the weight of the connection edge. The directional issues associated with certain edges are imposed by the algorithm for computing the shortest path to reach a node.

8.4.5 Computing the shortest path

To reach the vertex from any given node, a set of links need to traversed. This set of links consitutes a *path* to the vertex node. In the connectivity graph, the best hop to take reach a certain unit is computed based on shortest path that exists between the unit and the vertex. This process of calculating the shortest path starts at the node in question. The directional arrows indicate the links which comprise

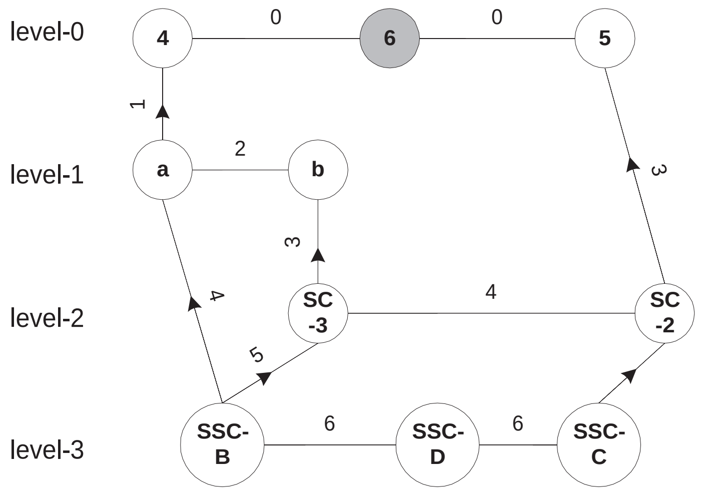

Figure 8.3: The connectivity graph at node 6.

a valid path from the node in question to the vertex node. Edges with no imposed directions are bi-directional. For any given node, the only links that come into the picture for computing the shortest path are those that are in the set L_{UL} associated with every node.

The algorithm proceeds by recursively computing the shortest paths to reach the vertex node, along every valid link (L_{UL}) originating at every node which falls within the valid path. Each fork of the recursion keeps track of the nodes that were visited and the total cost associated with the path traversed. This has two useful features -

(a) It allows us determine if a recursive fork needs to be sent along a certain edge. If this feature were missing, we could end up in an infinite recursion in some cases.

(b) It helps us decide on the best edge that could have been taken at the end of every recursive fork.

Thus say in the connectivity graph of Figure 8.3 we are interested in computing the shortest path to SSC-B from the vertex. This process would start at the node SSC-B. The set of valid links from SSC-B include edges to reach nodes **a**, **SC-3** and **SSC-D**. At each of these three recursions the paths are reflected to indicate the node traversed (SSC-B) and the cost so far i.e 4,5 and 6 to reach a, SC-3 and SSC-B respectively. Each recursion at every node returns with the shortest path to the vertex. Thus the recursions from a, SC-3 and SSC-D return with the shortest paths to the vertex. This added with the shortest path to reach those nodes, provides us with the means to decide on the shortest path to reach the vertex, which in this case happens to 5.

8.4.6 Building and updating the routing cache

The best hop to take to reach a certain unit is contained in the last node that was reached prior to reaching the vertex. This information is collected within the routing cache, so that messages can be disseminated faster throughout the system. The routing cache should be used in tandem with the routing information contained within a routed message to decide on the next best hop to take. Certain portions of the cache can be invalidated in response to the addition or failures of certain edges in the connectivity graph.

In general when a $level - l$ node is added to the connectivity graph, connectivities pertaining to units at level $\ell, \ell+1, \cdots, N$ are effected. For a $level - N$ system if a gateway g^{ℓ} within $u_i^{\ell+1}$ is established, the routing cache to reach units at level $\ell, \ell+1, \cdots N$ needs to be updated for all units within $u_i^{\ell+1}$. The case of gateway failures, detection of partitions and the updating of the cache is dealt with in a later section.

8.4.7 Exchanging information between super-units

When a subsystem u_i^ℓ is added to an existing system $u^{\ell+j+1}$ information regarding $g^{\ell+j}, g^{\ell+j-1}, \cdots, g^\ell$ is exchanged between the system and the sub system. The way the set of connections, comprising the connectivity graph, are sent over the newly established link is consistent with the rules we had set up for sending a connection information over a gateway as discussed in section 8.4.2. Thus if a new super cluster **SC-4** is added to the SSC-A sub-system and a super cluster gateway is established between SC-4 and node SC-1.6 the connectivity graphs at node 6 would be as depicted in Figure 8.4.(a) while the connectivity graph at the gatekeeper in SC-4 would comprise of the following connections sent over the newly established gateway.

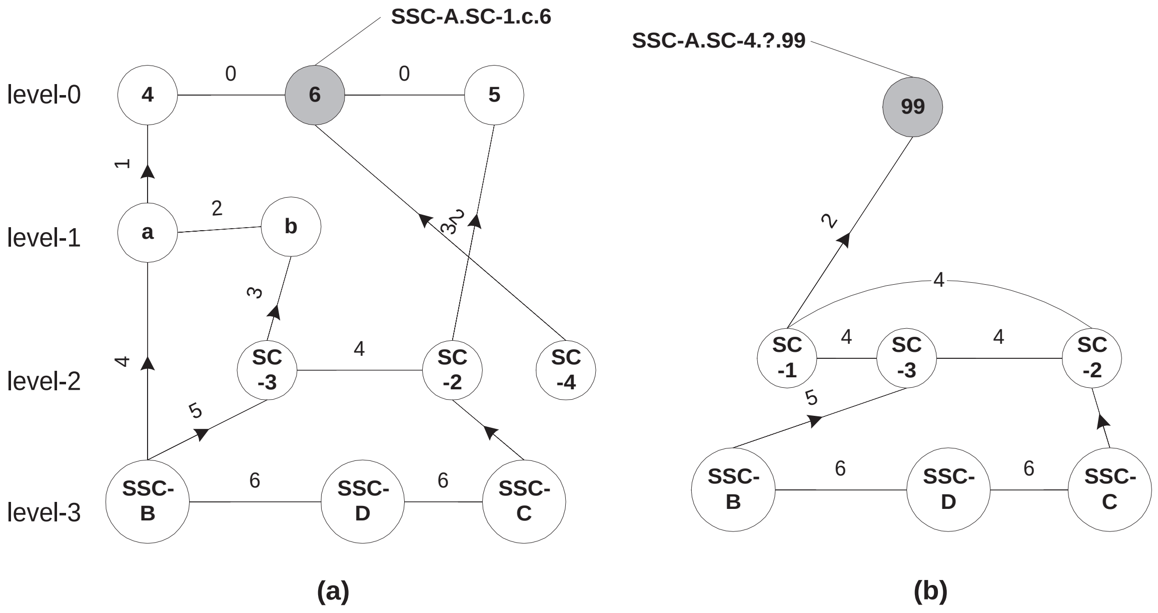

Figure 8.4: Connectivity graphs after the addition of a new super cluster SC-4.

Figure 8.4.(b) depicts only the connections which describe the connections involving level-2 gateways and upwards at node **99** in SC-4.

8.5 Detection of partitions

Partitions arise due to node failures or link failures. There are two different kinds of partitions that can arise in our system due to a connection failure - unit partitions and system partitions. The way the system deals with each case is different. Dealing with partitions is through *delegation* where, each super-unit of the system deals with partitions that could arise within its units. Detection of partitions is an extremely desirable feature since in our system a client can roam in response to the partition. Thus clients hosted within the units of a primary partition can roam to nodes which are in the primary partition. Healing of the partitions could result in the affected units being able to deal with clients in a consistent manner and share the client load of the system.

The information contained in the loss of connections is identical to that contained in the addition of a new connection. Also the dissemination of this loss of connection is dealt with in exactly the same way as additions are as described in section 8.4.2. Loss of connections result in the removal of link information from the sets L_{UL} and L_D associated with the node. If the connection that was lost is the connection $< n_i^x, n_j^y, \ell > \ where\ x \mid y = \ell\ and\ x, y \leq \ell$, if $y \leq x$ node n_j^y is removed from the set L_{UL} associated with node n_i^x and n_i^x is removed from the set L_D associated with the node n_j^y. The process is reversed if $x \leq y$. The detection of partitions is very simple. At the node whose L_{UL} is updated to reflect the connection loss. If $\#L_{UL} = 0$ the unit corresponding to the node is unit partitioned. If

$\#L_{UL} = 0 \bigcap \#L_D = 0$ the unit corresponding to the node is system partitioned.

Referring to the connectivity graphs in Figure 8.4 in the case of node 6 in 8.4.(a) the loss of the connection between clusters **a** and **b** results in **b** being unit partitioned within **SC-1** though it connected to SC-3. If however the only link that failed is the one connecting SC-1 and SC-3, no units are partitioned. In the last case, if the link connecting SC-1 and SC-4 hosted at node 6 fails, both node 6 in Figure 8.4.(a) and node 99 in Figure 8.4.(b) conclude that SC-4 has been system partitioned.

For nodes with $\#L_{UL} = 0$ the cost associated with reaching the vertex node $\rightarrow \infty$. All units which have their shortest path to the vertex resulting in a cost that $\rightarrow \infty$ are unit partitioned.

8.6 The event routing protocol - ERP

Event routing is the process of determining the next node that the event must be relayed to. Every event has a routing information associated with it, which could be used by the system to determine the route the event would take next. This routing information is not added by the client issuing this event but by the system to ensure faster dissemination and recovery from failures. When an event is first issued by the client, the server node that the client is attached to adds the routing information to the event. This routing information is the contextual information (see Section 7.1) pertaining to this particular node in system. As the event flows through the system, via gateways the routing information is modified to snapshot its dissemination within the system. This information is then used to avoid routing the event to the same unit twice.

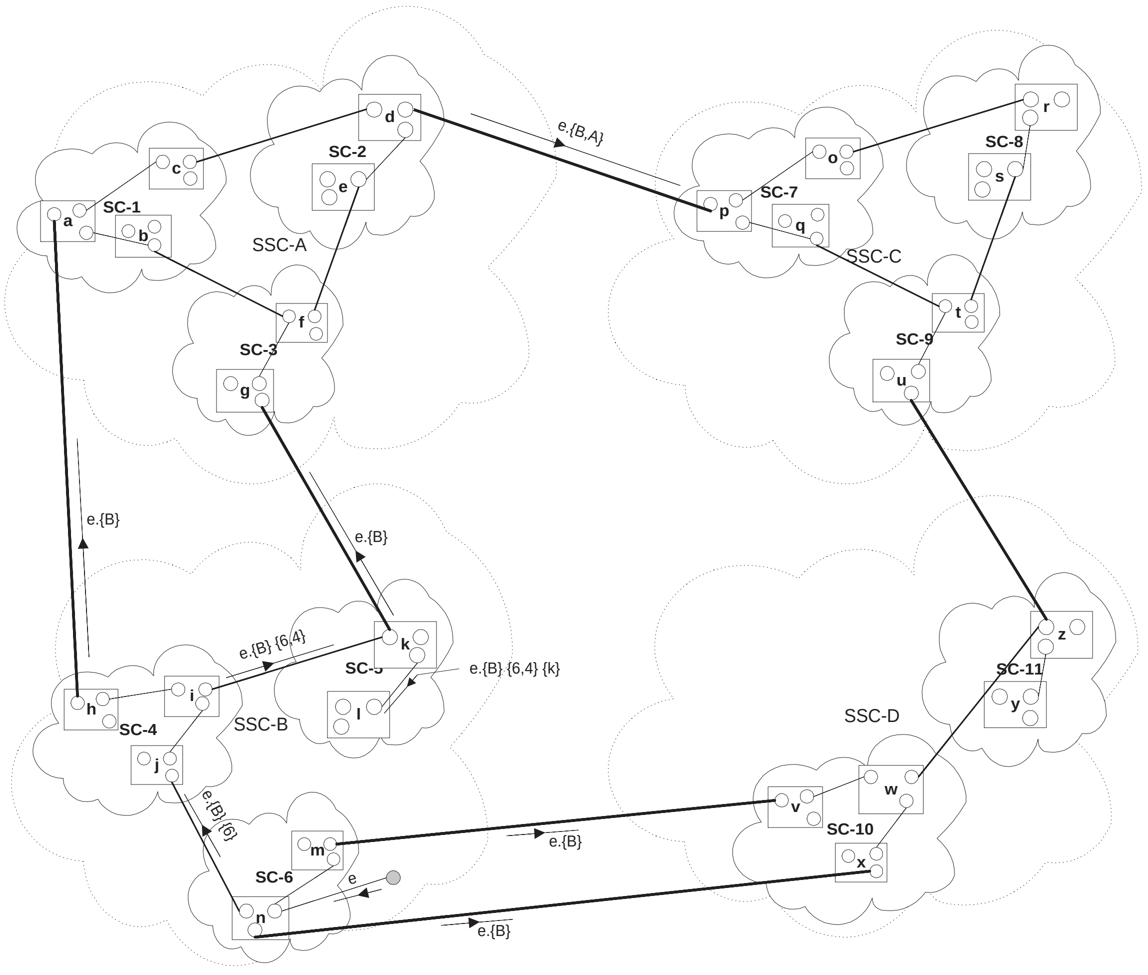

Figure 8.5: Routing events

A gateway $g^{\ell}(C_i^{\ell+1})$ is responsible for the dissemination of events throughout the unit at $level-\ell$ with context $C_i^{\ell+1}$. This is a recursive process and the gateway g^{ℓ} delegates this to the lower level gateways $g^{\ell-1}, \cdots, g^1$ to aid in finer grained dissemination. Thus a super-super-cluster gateway is responsible for disseminating the event to all the super-clusters which comprise the super-super-cluster that it is a part of. A gateway g^{ℓ} is concerned with the routing information from $level-\ell$ to $level-N$. When a event has been routed to a gatekeeper g^{ℓ} the routing information associated with the event is modified to reflect the fact that the event was received at this particular unit. It is the gatekeeper g^{ℓ}'s responsibility to ensure that the event is routed to all the nodes within the $level-\ell$ unit, using the delegation mechanism described earlier. Prior to routing an event across the gateway a $level-\ell$ gatekeeper takes the following sequence of actions -

- Check the $level - \ell$ routing information for the event to determine if the event has already been consumed by the unit at $level - \ell$. If this is the case the event will not be sent over the gateway.

  There could be multiple links connecting a unit to some other unit. This scheme provides us with a greater degree of fault-tolerance. This also leads to the situation[7] where the event could be routed to the same unit over multiple links. The duplicate detection algorithm detects this duplicate event and halts any further routing for this event.

- In case the gateway decides to send the event over the gateway, all routing information pertaining to lower level disseminations are stripped from the event routing information.

  This is because the routing information pertaining to the lower level definitions are within the context of that $level - \ell$ and the unit identifier. Also, in general a higher order gateway would be more overloaded[8] compared to a lower order gateway. Reducing the amount of information being transferred over the gateway helps conserve bandwidth.

Fig 8.5 depicts the routing scheme which we have discussed so far. The routings depicted in the figure are self explanatory and no further explanation is needed in this regard.

In addition to the information regarding where the event has been delivered already, events need to contain information regarding the units which an event should be routed to. Gatekeepers $g^{\ell}(C^{\ell+1})$ decide the $level - \ell$ units which are supposed to the receive the event. This decision is based on the profiles available at the gatekeeper as defined in the profile propagation protocol. The calculation of target units is a recursive process where the lower order disseminations being handled by the lower order gatekeepers. Thus two levels of routing information are contained within an event

(a) Units where an event should be routed within a unit.

(b) Units which have already received the event.

This routing scheme plays a crucial role in determining which events need to be stored to a stable storage during failures and partitions.

Lemma 8.3 *Every event e moves along the shortest path, prior to delivery at a client.*

PROOF: When a gatekeeper g^{ℓ} with context C_i^{ℓ} is presented with an event it computes the $u^{\ell-1}$'s within C_i^{ℓ} that the event must be routed to. At every node the best hop to reach a certain destination is computed. Thus at every node the best decision is taken.

Lemma 8.4 *Nodes and links that have not been failure suspected are the only entities that can be part of the shortest path.*

Theorem 8.5 *The event routing protocol, along with the profile propogation protocol and the gateway information ensure the optimal routing scheme for the dissemination of events in the existing topology.*

8.7 Routing real-time events

Real time events can have destination lists (see section 4.4) which are internal or external to the event. In each case the routing differs, in the case of internal lists the destination's location needs to be precisely located by the system. Routing events with external destination lists involves the system calculating the destinations for delivery.

[7]One of the reasons that this situation arises is a fork in the event's routing which send it to two gateways within the same unit

[8]This is because a lower order gateway is primarily employed for finer grained dissemination of events, and only rarely if at all would be used to get to a higher order gateway. Besides this a higher order gateway $g_i^{\ell}(C_i^{\ell+1})$ is the one responsible for deciding if the event needs to be routed to any of the lower units comprising the $level - \ell$.

8.7.1 Events with External Destination lists

When an event arrives at a gatekeeper g^ℓ, the gatekeeper checks to see if the event satisfies its profile. The profile maintained at g^ℓ snapshots the profile of the $level - \ell$ unit that the gatekeeper belongs to. This check is necessary to confirm if the event needs to be disseminated within the $level - \ell$ unit. Routing events based on the gatekeeper profile is the process which calculates the destination lists. This is a recursive process in which each higher order gatekeeper performs this check before disseminating the event to lower order gatekeepers.

When an event doesn't match the gatekeeper g^ℓ's profile, g^ℓ decides upon the next route that event would take based on the routing information encoded into the event by the event routing protocol.

- The gatekeeper $g^\ell_j(C^{\ell+1}_i)$ checks the routing information provided by ERP to see if it needs to relay the event to other gatekeepers g^ℓ within the context $C^{\ell+1}_i$.
- The gatekeeper also uses the information provided by ERP to check if it could route the event to a higher order gateway which hasn't received the event.

In the event that these steps lead to no actions on part of the gatekeeper g^ℓ the gatekeeper takes no further actions to route this event. If the gatekeeper decides to route this event to other $level - \ell$ and higher order gatekeepers, the system can employ lower order gateways within the context $C^{\ell+1}_i$ to relay this event.

8.7.2 Events with Internal Destination lists

These are events which require the system to be able to route the event to a specific client in the system. Clients which are interested in receiving point-to-point events thus need to include their identifier in their profile. The sequence of steps that are needed to route the event are similar to the steps we take to route events with external destination lists as discussed in section 8.7.1.

8.8 Handling events for a disconnected client

This problem pertains to one of the most important issues that needs to be addressed by our system. A client node has intermittent connection semantics, and is allowed to leave the system for prolonged durations of time and still expect to receive all the events that it *missed* in the interim period, along with real time events. Events are routed based on a clients persistent profile and the persistent profile is what would be stored at its last server node that it was connected to. The server node also has a persistent profile, the cumulation of the profiles of all the client nodes that are attached to it and all the diconnected clients which were last attached to it. The persistent profile of the server node is itself stored at the cluster gatekeeper. Consistency issues pertaining to out of order delivery of real time events and recovery events aside, our solution to this problem delegates this responsibility to the server node that this client was attached to prior to a disconnect/leave.

This node serves as a *proxy* for the client node. Now we could store all the events pertaining to a disconnected client at this node. Storing it within the process executing the node could result in precious main memory utilization in case the client is disconnected for a rather long duration and the number of events it has missed increases steadily over time. Also, it is not possible that every node has access to a stable storage. As mentioned earlier (section 7) one of the requirements of the system is that there should be at least one stable storage within a cluster. This stable storage could be used for storing events. We would deal with the garbage collection scheme for these events in section 8.11

8.9 Routing events to a newly re-connected client

Clients keep track of the last server node that they were connected to. This information is usually stored in their logical address. When a client disconnects the events that need to be routed to that client are still routed to the server. All event routed to a cluster are stored at one of the available stable storages within the cluster. We will address the garbage collection issues pertaining to these stable storages in

Section 8.11. Both the server node and the formerly connected client keep track of the last message that they believe was routed to the client.

Events are routed to the client from the last message that was routed to it. A simple filter routes the appropriate events from the stable storage. These recovery/roam events have a destination list which is internal to the event. This destination list comprises of a single entry - the logical address of the server node that the client is now attached to. Thus the routing scheme for these roam events are significantly different from what we discussed in section 8.7.2.

When the client issues a event recovery process, the logical address of the client is changed to its present address. The client could once again roam while these events are being routed to its present logical address. In that case that server node is now responsible for ensuring that the client doesn't loose any events that it is interested in.

8.10 Duplicate detection of events

Multiple copies of an event can exist in the system. This occurs due to multiple gateways existing between units and also due to events taking multiple routes to the reach destinations in response to failure suspicions. Events need to be duplicate detected because for any event e which is a duplicate event the path taken by the event as dictated by ERP is exactly the same as that taken by the event e which was previously received. In section 4.2.2 we discussed the generation of unique identifiers for events. This scheme of unique ID generation provides us with information pertaining to unrelated events (events issued by different clients) and in the case of related events (events issued by the same client) the order of their occurrence. In our scheme of duplicate event detection we use this unique ID generation as the basis for our duplicate event detection scheme.

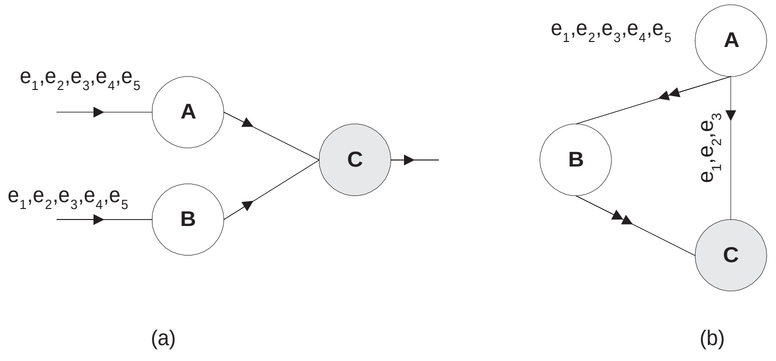

Figure 8.6: Duplicate detection of events

Our unique ID generation scheme allows us to determine which of two related events e and e' was issued earlier. If the last event delivered at a node is e if the node receives a related event e' our duplicate detection scheme works as follows -

- If $e' > e$ then e' was not delivered earlier else it was and is duplicate detected.

Consider the case in Fig 8.6.(a) at nodes A and B events e_1, e_2, e_3, e_4 and e_5 are all events issued by the same client. Node C maintains the last event that was delivered. The links we assume in the system are unreliable and unordered. Since these links allow the events to over take each other, if node C delivers e_3 first node C could errantly conclude that it had received e_1 and e_2. To resolve this we impose the requirement that the events be delivered in order (this is more so in the case of events issued by the same client), that is we do not let events overtake each other in the delivery sequence at any node within the system.

Now even though the events arrive at different times, since they arrive in order, the event e (either from A or B) which arrives first is not duplicate detected while the event e which arrives later is.

Consider the case in Fig 8.6.(b), node A has sent events e_1, e_2 and e_3 over link l_{AC} at time t. At time $t + \delta$ node A suspects a node C failure which could either be due to an overcrowded link l_{AC} or

from-A	e_1			e_2	e_3		e_4	e_5	
from-B		e_1	e_2	e_3				e_4	e_5
at-C	e_1^A		e_2^B	e_3^B			e_4^A	e_5^A	
$t \rightarrow$	1	2	3	4	5	6	7	8	9

Table 8.2: Delivery of events at C

a slow process at C. Now if A were to compute the alternate route to C which goes via B, if it doesn't send e_1, e_2, e_3 prior to sending e_4 and e_5, the events e_1, e_2, e_3 would be duplicate detected if e_4 arrives before e_1. Once we make this minor change of re-sending unacknowledged events across the alternate route in response to suspicions it simply reduces to the case depicted in Fig 8.6.(a). As an optimization feature we could include send *anti-events* down the failed/slow link whenever we resort to computing an alternate route.

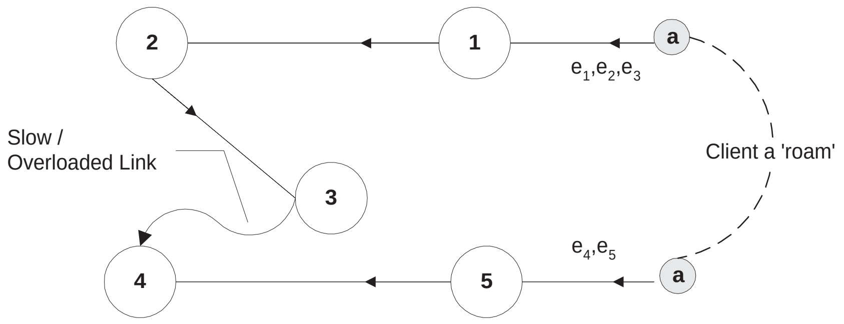

Figure 8.7: Duplicate detection of events during a client roam

Fig 8.7 depicts the scenario where a client roam could lead to duplicate detection of events which are not truly duplicate events. The case in which our duplicate detection scheme breaks down, is detailed in table 8.3. To account for such a scenario we include the incarnation number in our duplicate detection scheme. Incarnation numbers would be incremented for every roam and reconnection of the issuing client. The events would then be treated as events with a different *clientID* thus preventing the duplicate detection of events which shouldn't have been duplicate detected in the first place.

$t \rightarrow$	$t + \Delta$	$t + 2\Delta$	$t + 3\Delta$	$t + 4\Delta$	$t + 5\Delta$
at 2	e_1, e_2, e_3				
at 1		ACK(e_1, e_2, e_3)	$roam + send(e_4, e_5)$		
at 4				e_4, e_5	e_1, e_2, e_3

Table 8.3: Delivery of events at 4 ... client roam

8.11 Garbage collection scheme for the stable storages

Every event is stored along with a *bit vector*, $< 011001 \cdots 010 >$, identifying the server nodes within that cluster which are interested in that event. Each of the nodes have a pre-determined position in this storage vector. A **0** in this bit vector indicates that the event was not routed to the corresponding indexed node, while a **1** indicates that the event was routed to the corresponding node.

A server node issues a storage retrieval message only after it has received an acknowledge from each of the clients (connected to it) which are interested in that event. Upon receipt of this acknowledge the bit vector associated with the event is updated through a logical bitwise & operation with a **0** in place of the index associated with the server node sending the acknowledge and a **1** in every other location. Once the vector associated with the event is zero i.e $< 00 \cdots 0 >$ the event can be garbage collected.

This scheme works fine in the case of a system without network partitions. However in the case of a network partition this scheme would fail during subsequent partition mergers. This failure stems from the fact that the events would get garbage collected due to the 'local' reference counting, and thus would not be available for delivery during mergers. To account for this scenario we augment the scheme to account for gateway failures, by considering the far end of the gateway to be a client attached at the gatekeeper node in the cluster. Thus if an event needs to be sent over a gateway, the event isn't garbage collected till such time that we have received a confirmation regarding the receipt of that event at the intended gatekeeper.

9 The problem of delivering merged streams

For an event stream $E \hookrightarrow \Pi$, the problem of delivering each event within the stream E is one of determining the spatial dependencies $\forall e \in E \overset{s}{\hookrightarrow} \Pi' = e \mid e[\,] \mid null$ and the chronological dependencies $\overset{t}{\hookrightarrow}$ (within the constraints of time's arrow), dependency resolution and subsequent delivery of the events and one or more events within other streams that these events are dependent on. Delivering all events within E ultimately results in the creation of merged streams. Discovery of dependencies in E involves the determining the location of streams $E^j \in \Pi$ where $E \hookrightarrow \Pi$ and the timing constraints that exist within these dependencies. The other factor which plays an important role is the fact that not all stream sources issue events starting at the same time.

The client issuing "dependent event streams" needs to be aware of Π's event stream sources. Stream sources should be able to issue event streams specifying the dependencies and expect the system to resolve these dependencies and provide a coherent representation of the information in both E and Π where $E \hookrightarrow \Pi$ which would ultimately result in the merged event stream. Streams E and $E^j \in \Pi$ need not be aware of the exact and precise location of each other, nor should these stream sources expect a synchronization scheme for issuing events within certain timing constraints. E knows about E^j in an abstract sense, this knowledge needs to be utilized by the system to determine the exact locations of the streams. The issue of discovery of dependent streams doesn't arise once the event streams are merged, recovery for clients interested in E proceeds with the merged event streams.

9.1 Resolution of spatial dependencies

Event streams need to be merged based on the dependencies that exist within different events with a set of *related* event streams. These event streams as we discussed earlier need not to be aware of the precise location or the timing issues pertaining to other event streams. Event streams need to be aware of other event streams in an abstract fashion. We discuss what this *abstraction* should be. The system besides acting as a dependency resolver should aid in the process of dependency resolution before these dependencies are discovered in the first place. To put it simply, it is possible that the related event streams could be issued by sources which exist in different contexts. Dependency resolution involves two distinct steps.

(a) Determination of these dependencies - This involves being able to pin point the dependencies for each event in the stream E.

(b) Being able to resolve these dependencies - This involves ensuring that events being fetched by system and merged into a new stream at the location that these dependencies were discovered. Speeding up the resolution of dependencies enables us to optimize the creation of merged event streams. We thus need the system to be able to route events from streams in a manner which is conducive to the fastest merging.

9.1.1 Profile signatures & $\overset{s}{\hookrightarrow}$ resolution

The values an event's attributes can take comprises the event's *profile signature*. For an event e the profile signature is denoted $\overline{\omega_e}$. All the events within an event stream have identical profile signatures $\overline{\omega}$. Profile signatures dictate the routing characteristics of the event, and not the content. Events with identical profile signatures could have different contents within them. Streams have events with a *profile signature* $\overline{\omega}$ i.e.

$$\forall_{e \in E, E^j} e.profileSignature(\,) = \overline{\omega} \quad where \;\; E \hookrightarrow \Pi \;\; and \;\; E^j \in \Pi \tag{9.1}$$

When a client is interested in a stream, the client is implicitly interested in every event within that stream. This follows from the fact that the if the client's profile ω matches an event $e \in E$ it matches every event in E since all events have the same profile signature $\overline{\omega}$.

Aiding the process of event stream merger is something which should happen prior to and independent of the resolution of dependencies by the system. This issue pertains to the profile signatures which events in dependent event streams possess. Events within event streams are routed in exactly the same manner as individual events are - based on the profiles and the event routing protocol. All streams in Π have events with the same profile signature $\overline{\varpi}$ as the events in E. In addition all stream sources are also clients interested in their own events. This ensures that events are routed to locations where their dependencies would be resolved, and subsequently, lead to a merged stream. This would happen even if there were no true clients which are interested in that event stream during that precise instant of time which wouldn't happen if profiles aren't propagated through the system. Having the sources express an interest in themselves, and not issuing garbage collect notification also ensures that streams survive across system snapshots during which there are no clients interested in those event streams. Thus in most cases during the resolution of dependencies, no more network cycles need to be expended to resolve the dependency, since the related streams $E^j \in \Pi$ have already been routed to the contexts with clients interested in events from E.

9.1.2 When to proceed with resolving spatial dependency of the next event

The occurrence vector $\mathcal{O}$ specifies the number of events within other event streams that are needed to satisfy an events dependency. Elements within the occurrence vector themselves contain two constraints [9]. For delivering an event e we require that only the weakest constraint of the occurrence vector element need be satisfied. Ensuring that constraints are fully satisfied prior to delivery are impractical in a live setting. It is, for example, impossible to know how many events would satisfy the constraint between two related streams.

9.2 Merging of streams and the resolution of chronological dependencies $\overset{t}{\hookrightarrow}$

Merging of event streams requires resolving both the spatial $\overset{s}{\hookrightarrow}$ and the time dependency $\overset{t}{\hookrightarrow}$. Timing dependencies $\overset{t}{\hookrightarrow}$ are imposed either at the source stream E source or are predefined along with the spatial dependencies $\overset{s}{\hookrightarrow}$, that exist between streams. In the former case events in streams $E^j \in Pi$ await their timing constraints from the source stream E prior to delivery at a client.

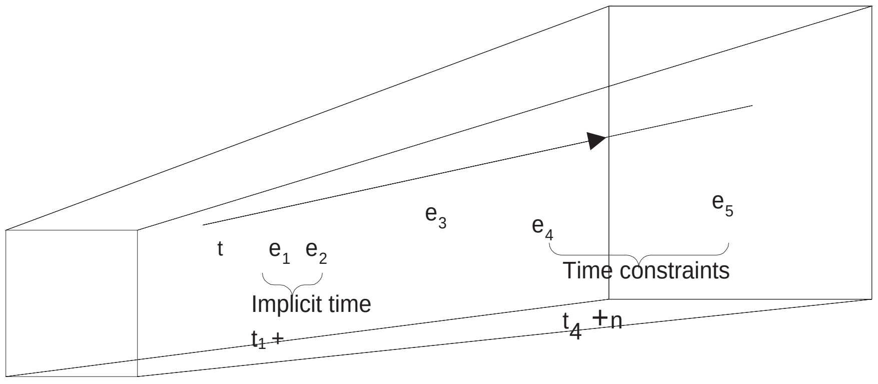

Figure 9.1: Dependencies and chronological ordering.

Newly generated events which add to streams in Π could either be request or response events. Request events do not have a context associated with it. The spatial dependencies may be self contained within the event itself, the timing dependencies are however dictated by the timing considerations at the source of the merged stream i.e. the source for E. It is this timing dependency which dictates the order for these events within the merged stream. Response events are events added to a stream E^j in response to the newly generated event which adds to the stream $E^j \in \Pi$. The timing dependencies for response events are implicitly specified by the system, the constraint imposed by the system is that the response event cannot be delivered at a client till such time that the associated request event has been delivered.

[9]zero or more, one or more, zero or none etc

The rule is simple request events can exist alone, however the response events exist only within the context of the <*request, response*> tuple. The merged stream at the stream E's source is what comprises the playback stream. The spatial and timing dependencies thus dictate the ordering of events within the merged stream. There could be a number of request events that are generated by clients throughout the system, and they would seem to occur in a different order at each interested client. However the total ordering of these requests is determined by the order in which these events were received at the stream E's source.

9.3 Resolution of dependencies for newly added events

When we say $e \hookrightarrow e_j$ it implies that e has a spatial dependency $\overset{\text{s}}{\hookrightarrow}$ on e_j for its completeness and also that e and e_j are chronologically related ($\overset{\text{t}}{\hookrightarrow}$) i.e. e_j occurs later than e in the direction of times arrow. For an event e the notion of time's arrow is asymmetric, an event e_i doesn't know when exactly the next event e_{i+1} follows, but it is aware of the occurrence of an earlier event e_{i-1}. For an event $e_i \overset{\text{t}}{\hookrightarrow} e_j$, the $\overset{\text{t}}{\hookrightarrow}$ is either δt or chronological constraint $t_{i,j}$. The δt constraint is determined by the granularity of the clock in the underlying system and is the minimum $\overset{\text{t}}{\hookrightarrow}$ constraint that can exist between two events that are $\overset{\text{s}}{\hookrightarrow}$ related. The notion of time that events have is one of relative times. Constraints are specified based on the intervals between the occurrence of successive events. The $\overset{\text{t}}{\hookrightarrow}$ dependency operates within the context of the spatial dependency $\overset{\text{s}}{\hookrightarrow}$. For $e \overset{\text{s}}{\hookrightarrow} e_j$ and $e \overset{\text{s}}{\hookrightarrow} e_k$ there are no ordering constraints imposed on the delivery of events e_i, e_j with respect to each other. Thus events e_i and e_j have neither a spatial nor a chronological dependency between them though they are events within a merged stream.

In the case of $E^j \in \Pi$ new dependencies could also be generated due to live streams. These are due to the events generated, which add to one or more of the streams in Π. These events have a $\overset{\text{s}}{\hookrightarrow}$ and a $\overset{\text{t}}{\hookrightarrow}$ dependency that is either implicitly conjectured by the system, or explicitly specified within the event. In the case of spatial dependencies the implicit dependency is defined by the context in which the event occurred. and a $\overset{\text{t}}{\hookrightarrow}$ dependency that is either implicitly or explicitly specified. In the case of chronological dependencies the implicit constraint is specified by δt which specifies the minimum time between two spatially related events.

If the existing live event at a client is e the $e \overset{\text{s}}{\hookrightarrow}$ is being resolved, when the event was added to one of the streams in Π then $e \overset{\text{s}}{\hookrightarrow} e^N$ (where $\overset{\text{s}}{\hookrightarrow}$ is a transitive relationship).

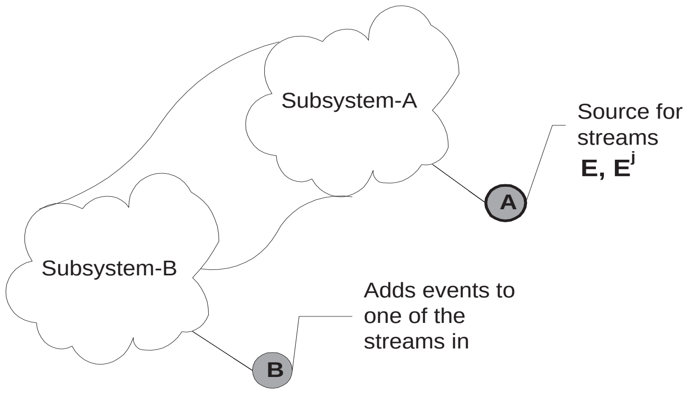
Subsystem-A
Source for streams E, E^j
A
Subsystem-B
Adds events to one of the streams in
B

Figure 9.2: Resolving dependencies.

Fig 9.2 depicts one of the possible scenarios for resolving dependencies. Client A in the figure is source for streams E and $E^j \in \Pi$. Of course all the streams in Π could have been hosted at A, which is where ultimately the merged stream charactheteristics are specified. The session is a live session where all the streams have events that would be issued as time progresses in the system. There could zero[10] or

[10]We are excluding stream sources which express an interest in their own events in order to avoid garbage collection of

more clients interested in a merged stream, we consider one such client B. Clients interested in a merged stream can add one or more events to zero or more streams which constitute the merged stream.

Let us first consider the case for client A. If the spatial context at A is $E.e$ and an event $e_j$is added to E^j then $e \overset{\text{t}}{\hookrightarrow} e_j$. This spatial dependency must be consistent with the dependencies that exist between events in E^j and E. The timing constraint is specified by the difference between the time when event e was delivered and the event e_j was added to the event stream E^j.

Now consider an event e_k being added to one of the streams in Π by the client B. For a sequence of dependency resolved events e^1, e^2, e^3 this event was added within the context e^3. However at A the spatial context is now e^5 where $e^5 \hookrightarrow e^3$, i.e the event e^3 has already be deliverd. The spatial context thus needs to be resolved (a process that would take place during playbacks). The $\overset{\text{t}}{\hookrightarrow}$ is assigned based on the receipt of the events at E. Clients, other than B and A, need to await the chronological context (from A) associated with events added by B to streams $\in \Pi$, prior to the events delivery at the client.

For playbacks the context is assigned by A in the following manner. For a dependency chain $e^2 \hookrightarrow e^3 \hookrightarrow e^4 \hookrightarrow e^5$ at A. Event e_k was received at A when e^5 was the active context, but has a spatial context in e^3. Now $e^3 \hookrightarrow e_x \hookrightarrow \cdots e_z \hookrightarrow e^4$ is the complete dependency chain between e^3 and e^4 in the merged stream existing at A during the receipt of e_k. In this case e_k is attached to the last spatial sequence of e^3 with a chronological increment of δt relative to the last chronological event which is consistent with e^3's dependency chain i.e e_z.

9.4 Playback of event streams

All the interested clients may not have registered their interest during the *live stream.* Playbacks ensure the delivery of these missed streams during a subsequent time. Playbacks are initiated by a profile change $\delta\omega$ or a subsequent join into the system after a prolonged disconnect during which the clients had missed several events. In the case of the profile change $\delta\omega$ all the events in the event streams are routed to it while in the case of a client re-entering after a disconnect only the relevant events are played back. During playbacks what a client gets is the merged event stream, with all the dependencies resolved, in response to the interest in event stream E.

In addition during playbacks a client interested in $E \hookrightarrow \Pi$ could add one or more new events to one or more streams in Π. Subsequent playbacks for other clients should include the updated streams with the requests and <*request, response*> tuples added during a prior playback. Merged streams should be able to reside on different stable storages of the system, reconstruction would need to determine the locations of storages for $E \hookrightarrow \Pi$.

9.5 Streams & interpretation capabilities

Different clients have different interpretation characteristics. This interpretation capability is a function of the underlying system at the client. The streams that actually need to be routed[11] to a client are a function of the interpretation capabilities that are available at the client i.e $\Pi_{client} \subseteq \Pi$, this of course needs to be taken care by PPP[12]. The interpretation capabilities are dependent also on the event transformation switches that are available within the system. The switches are responsible for transforming the streams into something that can be deciphered by the client. Also if $E^j \notin \Pi_{Client}$ replace $E^j.e < data >$ with some value signifying the inability to represent content on the specific client device.

events in their streams. This is an artifact of the PPP and ERP which would automatically garbage collect those events which were issued when no clients had registered an interest in the stream sources.

[11]We are of course referring to the fact that though it is a merged stream that is routed, we only route those events within the streams that can be interpreted by the client

[12]Profiles would also need to contain information about the devices that are present within the unit that it is *snap-shot'ing.*

10 Issues in Reliability & Fault Tolerance

The system we are considering could have the failures listed in section 2. Each of these failures could lead to network partitions. In a distributed asynchronous system, it is impossible to distinguish a crashed process from a failed one, and a failed link from an overloaded one. In addition to the failures we are considering, incorrect suspicions may result due to overloaded links and slow processes. These failure suspicions, both correct and incorrect, can also lead to network partitions. We need to ensure that partitions make safe progress, during the network partitions in concurrent views of the network and also that there are no contradictions during the partition merges after the partition has been repaired.

Failures could also manifest themselves in the form a node failure, consecutive node failures, cluster failures and so on. The objective that we are trying to meet is to ensure safe progress of operations and meeting system guarantees in the presence of failures. In the remainder of these sections we address each issue separately and then come up with solutions which solve this problem.

10.1 Message losses and error correction

With respect to mechanisms for error correction, protocols can be broadly separated into two categories: *sender-initiated* and *receiver-initiated*. A sender-initiated protocol is one in which the sender gets positive acknowledgments (ACKs) from all the receivers periodically and releases messages from its buffer only after an indication that the message has been received at all the intended destinations. A receiver-initiated protocol is one in which the receivers send negative acknowledgments (NAKs) when they detect message losses. In receiver initiated protocols the assumption at the sender is that the message has been received at the receiver unless indicated otherwise by the NAKs. The NAKs indicate the holes in message sequences, also the receivers never send any ACKs to the sender.

We employ a combination of ACk's and NAK's to address this problem. In short, error correction on the link is handled using NAKs while garbage collection is performed using the ACKs.

10.1.1 Message losses due to consecutive node failures

In Fig 10.1.(a) we have a situation where the two nodes ensure reliable delivery using a series of positive acknowledgements (ACKs). Node A will not garbage collect a message m until it has received an $ACK(m)$ from B. However it is possible that node B experiences a crash-failure immediately after issuing an $ACK(m)$ to A. Message m would thus never be delivered by C. We could try and rectify this situation as in Fig 10.1.(b) by requiring that a receiving node issue an ACK only after it has forwarded the message. This would solve our earlier problem, but this simply pushes the problem further in space, since the scheme would breakdown in case of successive broker failures after an ACK(m) has been issued by the soon to fail node B (the other one being C). Brokers B,C fail after B has issued an $ACK(m)$ and C has been unable to forward m to D. Thus, m is lost since A has already garbage collected it and D doesn't know if it should have received m (for that matter it wouldn't even know about the existence of m to even detect its loss) in the first place.

This problem can be circumvented by augmenting the client nodes with re-issue behavior till such time that the event has been stored onto a stable storage. Once an event is stored onto stable storage, the guarantee is that it can be recovered in the event of failures which could take place. For every event e issued by a client, and held in the client's local queue, there is a timer associated with the event. Unless the client receives a storage notification before the timer's expiry the message would be re-issued and the timer reset. The timers associated with events in the local queue are updated every Δt. The timer associated with the event is reduced by Δt after every failure to receive a storage notification within the Δt, prior to the timer expiry. If a storage notification is received prior to this timers expiry the corresponding event is garbage collected from the clients local queue.

10.2 Node failures

The state of a server node is the list of profiles of clients connected to the server node. The state of a server could be reflected to server nodes to allow reconstruction of state from other nodes. The state is updated during the profile changes as defined by the PPP (section 8.1).Thus a server node could behave

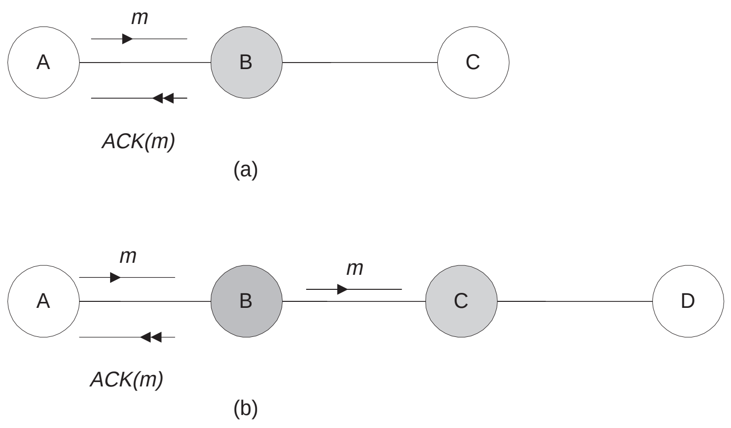

Figure 10.1: Message losses due to successive node failures

as some other server node. If node A fails node B could take over the role of node A while also being node A at the same time.

10.3 Gateway Failures

There could be multiple gateways connecting different units. Gateways could also suffer transient failures which could be a result of overloaded links etc. It could also suffer a permanent failure due to a failure of the link or the gatekeeper at the other end which comprises the gateway.

10.3.1 Transient gateway failures

In this case the events are stored at the gatekeeper experiencing problems. The gatekeeper node regularly tries to re-send these events over the gateway. In addition some of the events could be garbage collected based on the gateway's awareness of the units interconnection scheme and information provided by gatekeepers which provide gateways to the same unit.

We use multiple gateways to provide us with a greater degree of fault tolerance. We need to use this information to also determine whether certain events need to be stored at a gatekeeper, the gateway which it provides having transient or permanent failures.

10.3.2 Permanent gateway failures

This would call for an update of the information by the gateway propagation protocol. This information would be used by the nodes in tandem with the routing information contained in the event to decide the next route that the event would take.

10.4 Unit Failures

When we refer to unit failures, we are referring to the failure of all the nodes and gateways within that unit. The cases that we need to consider include failures that last forever, recovery of the complete unit and partial recoveries of the unit. The issues to be considered in each case and the associated recovery mechanism are described in the sections below.

10.4.1 Unit remains failed forever

In the event that a unit never recovers, all the nodes with this unit would eventually be deemed failed by the attached client nodes. This failure confirmation would result in a roam of all the attached client nodes. The system would already have treated all the client nodes within that unit as disconnected clients, and would have proceeded to store events for eventual routing. The re-routing of events to the client which has *'roamed'* to a new location is identical to the scheme which we discussed in section 8.8 with one notable difference. The difference stems from the fact that the original node to which the client was attached to is no longer available. Thus the unit which has stored the events which should have been routed to the client needs to intercept the request for a re-route and then proceed with applying the filter operation to recovery of events.

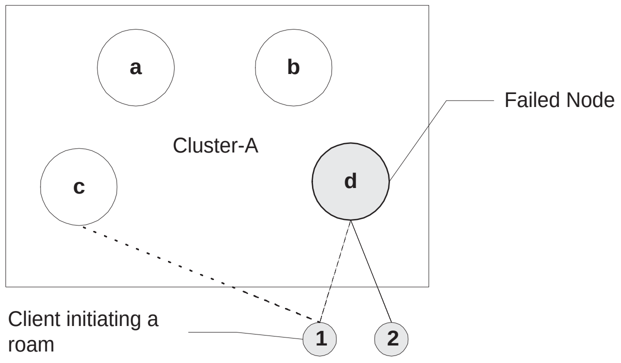

Figure 10.2: Client 'roam' in response to a node failure.

10.4.2 Unit doesn't remain failed forever

If a unit doesn't remain failed forever and the associated nodes recover and the gateways are reinstated, events that were missed by this unit (excluding the ones which were garbage collected due to clients recovering events during the roam and subsequent reconnect), are routed to the unit. It is possible that there were some clients which were disconnected during the unit failure, and have just re-connected after the failure. These clients would need to be serviced by this unit.

Profiles for such clients may be lost, we also need a mechanism for recovering the profiles of the clients which were connected. Besides we also need information on which clients shouldn't be reconstructed. How we know which client was attached to which node is another issue which we need to address.

10.4.3 Partial recovery of a unit

Not all units may recover at the same time or at all. Some units though they have recovered based on their earlier connection schemes would still be network partitioned. We discuss recovery mechanisms for such partitioned units in the section 10.5.

10.5 Network Partitions

Traditional approaches [Bir85, Bir93] in group based systems, include applications that require a certain message be delivered to all correct processes in a group. However these approaches maintain delivery guarantees only within the major partition during node failures. Also delivery constraints are not satisfied during partition mergers. In [BBT96] the effect of link failures on the solvability of problems (which are solved with reliable links) in asynchronous systems has been rigorously studied. Network partitions can be caused both by link failures and node[13] failures. The issues to deal with in the case of network partitions differ considerably from the unit failure cases. Unlike the unit failure cases where the clients can initiate a roam, it is possible that a client is attached to a node within a partition which is fully functional. Thus we need mechanisms to -

- Detect partitions.

[13]In this case the node could be a gatekeeper, or is on the route to a gatekeeper. If this is the only node which leads to a specific gatekeeper, a failure in this node leads to a network partition

- Ensure safe progress in concurrent partitions.
- Merge partitions while maintaining consistency.

10.5.1 Detection of Network partitions

When a gateway is deemed failed, a decision needs to be reached about the existence of a partitions. When $g^{\ell}(C_j^{\ell+1})$ fails all other $g_i^{\ell}(C_j^{\ell+1})$'s are notified or attempts are made to submit such notifications. Gatekeepers could then decide on whether a partition exists or not. Detection of these partitions are far more complex in this case since not only g^{ℓ}'s are involved in this decision making process but also higher order gatekeepers $g^{\ell+1}$'s and lower order gatekeepers $g^{\ell-1}$'s. A unit which has experienced a g^{ℓ} failure could have other g's which could compensate or offset such failures from causing a network partition. Thus nodes need to be aware of every other $level - \ell$ gateway with the unit-*ell* that it is a part of. We would need such information to arrive at partitioning decisions and garbage collection of replicate data and avoiding too many replications of events.

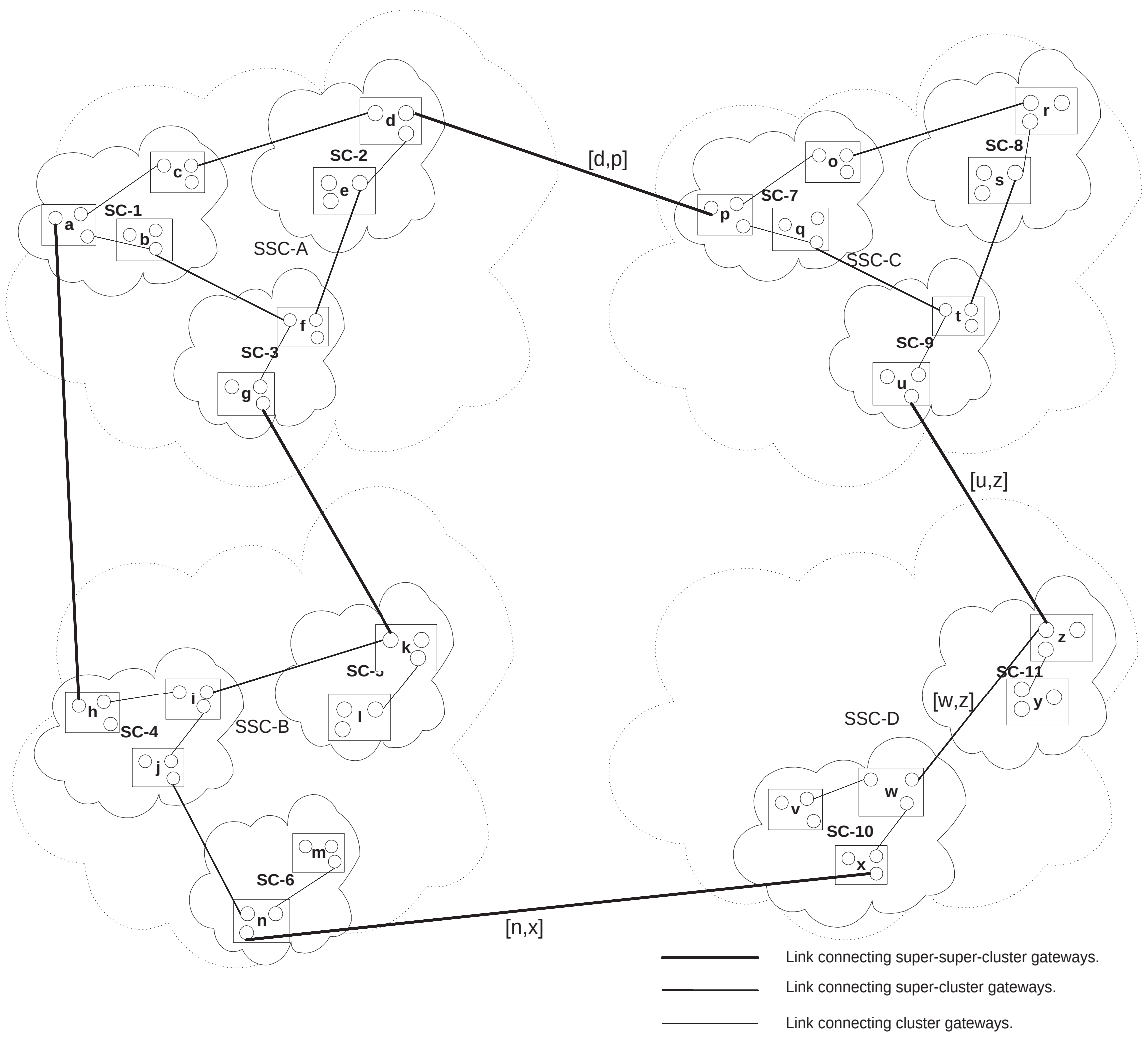
d
SC-2
e
c
a
SC-1
b
SSC-A
f
SC-3
g
[d,p]
r
SC-8
s
o
p
SC-7
q
SSC-C
t
SC-9
u
[u,z]
z
SC-11
y
[w,z]
SSC-D
k
SC-5
l
h
i
SC-4
SSC-B
j
m
SC-6
n
[n,x]
v
w
SC-10
x
Link connecting super-super-cluster gateways.
Link connecting super-cluster gateways.
Link connecting cluster gateways.

Figure 10.3: Partitioning issues related to node failures

10.6 Node failures

If a node fails, it may lead to partition if it is en route to other nodes within the cluster. If this node is a gatekeeper, three issues need to be taken into consideration -

(a) This gatekeeper is the only gatekeeper within the cluster. This would result in a minor partition within the cluster.

(b) There are additional gatekeepers g^{ℓ}, where $\ell = 1, 2, \cdots, N-1$, within this cluster.
In such cases we need to decide if the presence of these gatekeepers within the cluster helps compensate or offset the failure of a gatekeeper node. We would specifically be interested in the presence of higher order gatekeepers than the one which has just failed.

(c) The node could be providing us with multiple gateways (not necessarily of the same order). In such cases we need to address the effects of losing each gateway that this node was providing us.

In Fig. 10.3 SSC-C and SSC-D should know that it has suffered a partitioning if [d,p] and [n,x] have suffered failures. Similarly SC-10 would know that it has suffered a partitioning if [n,x] and [w,z] have failed. This decision can be reached based on the routing information available at the gatekeepers. Failures in gatekeepers are propagated to the appropriate units. Thus if $g^{\ell}(C_j^{\ell+1})$ experiences a failure all the units within $C_j^{\ell+1}$ are notified about this failure. Higher order gatekeeper failures would thus be much more expensive than lower order gatekeeper failures.

However its possible that a gatekeepers failure doesn't lead to a partition. This could be due to alternate routing route provided by the higher-order gatekeepers, or similar-order gatekeepers within the unit in question. Thus [w,z] could fail and we wouldn't have a partition due to the existence of higher order gateways [u,z] and [n,x]. However lower order gatekeepers cannot compensate/offset the loss of higher order gatekeepers within the unit. Thus the rule is -

Conjecture 10.1 *If a gatekeeper g^{ℓ} fails (or rather a gateway fails) check to see if there are gatekeepers $g^{\ell+x}$ where $x = 0, 1, \cdots, (N-\ell-1)$ within that unit which can compensate for this failure.*

10.6.1 Issues in event routing and profile propagation

This section specifically tries to address the issues, pertaining to event routing and profile propagation, which arise out of the partitioning between units of a super-unit, but not necessarily a network partition within the system. ERP provides information regarding the units which an event must be routed to and also the units where an event has been routed to. One of the objectives of having multiple gateways between units is to have a higher degree of resilience to failures which may take place within the system.

ERP, however delegates the responsibility of event dissemination recursively[14] to the units which exist within a super-unit. Thus in Fig 10.3 if [w,z] has failed , leading to a unit partition between SC-10 and SC-11 within SSC-D, it is conceivable that an event e may arrive at SSC-D within an indication that the event was routed to SSC-D. If [w,z] has failed the premise that since the event routing information contains a reference to SSC-D the event was routed within SSC-D is not true. This case needs to be accounted for, where SSC-D notifies SSC-B and SSC-C about the unit partition which exists within SSC-D. Thus we need to relax routing rules for units which sub-unit partitioning, since the system could otherwise conclude that the sub-units have received the relevant events.

PPP also experiences problems in this regard, where it needs to decide on the profile of a super-unit whose units have been partitioned. Since its the gatekeeper $g^{\ell+1}(C_j^{\ell+2})$ which maintains the profile of the units at $level - \ell$ the profile of all the sub-units wouldn't be lost. However it is also possible that the partitioned sub-units receive events which satisfy the other sub-units profile.

10.6.2 Ensuring progress in concurrent partitions

Concurrent partitions may contain clients which issue events and also other clients which are interested in those events. The interested clients should thus be able to receive events which are currently being

[14] A super cluster gatekeeper delegates the responsibility to the clusters, which in turn delegate it to the server nodes

issued within that partition. All these events would of course need to be stored onto a stable storage, for re-routing during partition mergers.

10.6.3 Partition Mergers

Each partition keeps track of the last events that were received by the gatekeepers in individual partitions. Based on this information appropriate events are routed. Of course prior to this we need to also account for the profile reconstruction since there could be clients which have initiated a roam. Similarly events issued by clients, either during disconnected mode operations or server node failures, and subsequently held in the client's local queue would be fed back in to the system.

10.7 Stable Storage Issues

Systems can slow down considerably because of this process of storing to stable storages. Storages exist en route to destinations but decisions need to be made regarding when and where to store and also on how many replications we intend to have. Events can be forwarded only after the event has been written to stable storage. Thus the greater the stable storage hops the greater the latency in delivering events to their destinations. Of course we would be optimizing this scheme. We also need to address the issues pertaining to the control of the replication scheme.

10.7.1 Replication Granularity

In our storage scheme data can be replicated a few times, the exact number being proportional to the number of units within a super unit also on the *replication granularity* which exists within a specific unit. For a level-ℓ system if there's at least one stable storage servicing the unit, we denote the replication granularity of that part of the sub system as r_ℓ. Thus if the replication strategy is one of replicating within every cluster in case of a 3-level system with N units at each level a certain event which would be delivered by all the clients within the system would be replicated $2^N \times 2^N \times 2^N$ times. Of course what we are considering here is the extreme case, but nevertheless its an exemplar of how the replication strategy is a crucial element of the system. Besides, the garbage collection also ensures that the storage space doesn't increase exponentially.

Stable storages exist within the context of a certain unit, with the possibility of multiple stable storages within the same unit. We do not impose a homogeneous replication granularity through out the system. Instead we impose a minimum replication scheme for the system. This is of course the coarsest grained replication scheme, there could be units present in the system which have a replication strategy which is more finely grained.

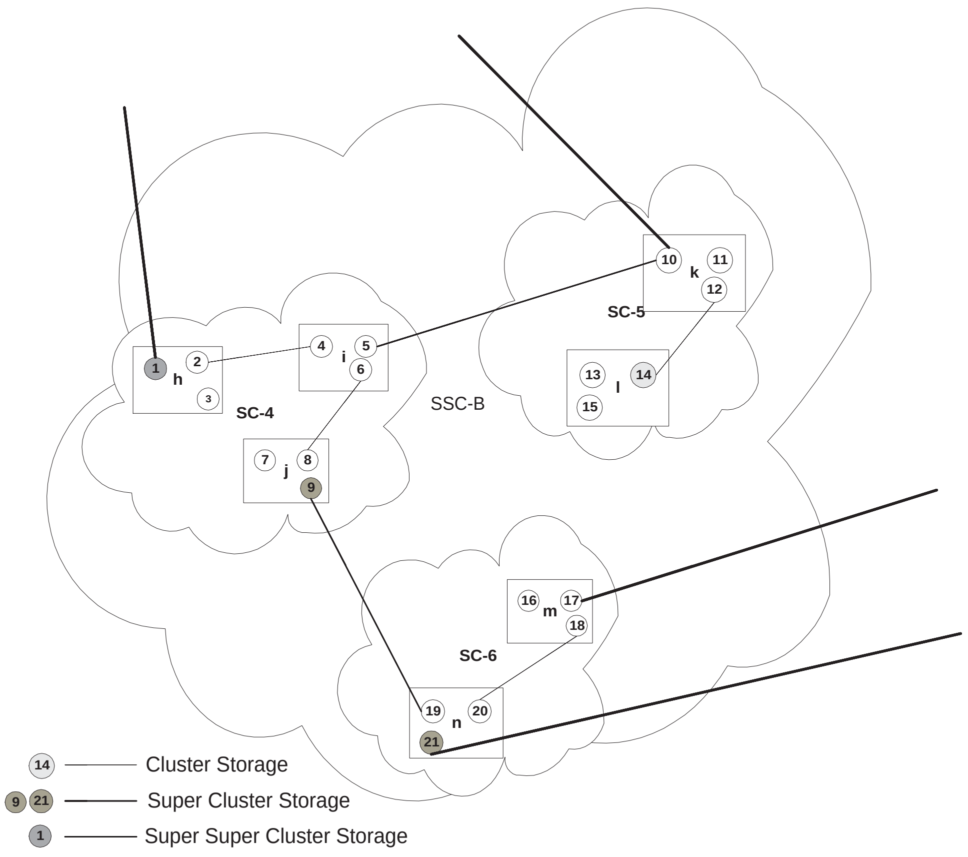

Figure 10.4: The replication scheme

The interaction between the stable storages of a unit and the stable storages within the sub units needs to address both the redundancy and garbage collection issues. Stable storages store events that the unit it is servicing, is interested in. This is ensured by the ERP which would ensure the routing of only the interesting events. The node which best serves this purpose is the gatekeeper node. As discussed earlier (section 8.1) PPP ensures that a gatekeeper $g_i^\ell(C_j^{\ell+1})$ snapshots the profile of $level - \ell$ unit i within the context C_j. Thus if we fix the replication granularity at ℓ at least one gatekeeper $g^\ell(C_j^{\ell+1})$ among others within the context $C_j^{\ell+1}$ is responsible for the event storage. One of the advantages of this scheme is that we store only those events that we are interested in.

Fig 10.4 depicts the different replication strategies that can exist within a sub system. As can be seen super-super-cluster SSC-B has a replication granularity r_3, while super-cluster SC-4 within SSC-B has a replication granularity r_2. Cluster l has a replication granularity of r_1. The figure demonstrates the different replication granularities that can exist within different parts of the sub system. Table 10.1 depicts the replication granularities available at different nodes within the sub system depicted in fig 10.4.

Nodes	Granularity r_ℓ	Servicing Storage
10,11,12	r_3	**1**
1,2,3,4,5,6,7,8,9	r_2	**9**
16,17,18,19,20,21	r_2	**21**
13,14,15	r_1	**14**

Table 10.1: Replication granularity at different nodes within a sub system

Requirement 10.2 *There should be at least one stable storage present in the system.*

10.7.2 Stability

The presence of a stable storage within a unit at $level - (\ell - 1)$ for a system with replication granularity ℓ delegates the stability responsibilities for events within the $level - (\ell - 1)$ to the stable storage at the lower level. Every event in the system should be stable because we should be able to retrieve it in case of failures. Stable storages need to wait for notifications prior to the garbage collection of events. This notification varies from unit to unit. For a unit which possesses no stable storage this notification is issued only if

1. All the clients attached to the nodes within this unit have received this event.
2. There are no other units where this event needs to be forwarded to. This is determined by the event routing protocol. If there is at least one such unit where the events are forwarded to then no notifications can be used till a notification is received from all the units to which this event was forwarded to.

In case of unit with a stable storage, this notification is issues once the stable storage has stored this event to stable storage.

10.7.3 Delivering events without storage notifications

In section 10.1.1 we have described the scheme which ensures the storage of events to the first stable storage. This re-issue behavior of the issuing client continues till the event is stored to at least one stable storage within the system. Thus we do not really need to be very conservative when we are delivering events, we could in effect deliver events without the receipt of storage notifications. There are a few reasons which contribute to this optimistic approach despite the possible failures within any unit -

Lemma 10.3 *The client reissue behavior ensures the storage of the event to at least one stable storage.*

Proof: Section 10.1.1 and Requirement 10.2 on page 46.

Lemma 10.4 *If there is at least one client which hasn't delivered the event, then there is at least one stable storage which hasn't garbage collected that event.*

Proof: Section 10.7.2.

Lemma 10.5 *If there is such a stable storage, for the scenario described in Lemma 10.4, then the system will locate this stable storage when the client missing this event issues a request for it.*

Proof: Conjecture 10.8 on page 47 and requirement 10.7 on page 47.

Theorem 10.6 *For an event e issued by a client, all clients within the implicit or explicit destination list contained within the event will eventually deliver e.*

10.7.4 Storage scheme

Events need to also indicate or provide information if they have been stored to stable storage somewhere in the system and also if they have been stored to stable storage at one of the locations within the unit.

The primary issues which we need to address are -

(a) Finding the route to the nearest stable storage.

(b) A single unit could be served by multiple stable storages. During partition mergers how do we deal with the routing scenario and how does this scenario come into the picture. Is there any way around such a situation

(c) When a client has initiated a roam in response to a failure, how does the system decide where to go and fetch the missed events. This follows directly from (b). This scenario holds true for a disconnected client too, which isn't yet aware of the unit failure.

10.7.5 Stable storage failures

When a stable storage node fails, the events that it stored wouldn't be available to the system. A new client trying to retrieve its events is prevented from doing so. The stable storage also misses any garbage collect notifications that were intended for it.

Requirement 10.7 *A stable storage cannot remain failed forever, and must recover within a finite amount of time.*

Stable storages could be removed, however the delegation of storage responsibilities can be delegated only to a higher level storage.

10.7.6 Finding stable storages which have stored a certain event

One of the disadvantages of having a client keep track of the servicing stable storages is that when the client is operating in the disconnected mode, there could be other stable storages which are servicing the unit to which the client was last connected. However, the client is not aware of this new stable storage and could possibly loose events which it was supposed to receive.

Stable storages at a higher level (minimum replication granularity) are aware of the finer grained replication schemes that exist within its unit. If a higher level unit is managing the lower level context of the clients logical address, the system would use the higher level stable storage to retrieve the client's interim events else the system would delegate this retrieval process to the stable storage which services the client's lower level context.

Conjecture 10.8 *A client's logical address provides the system with the list of stable storages that should be used for the construction of queues containing events that were missed by the client.*

It is possible that one or more of these stable storages (in case of multiple stable storages within a context) are unavailable during a subsequent client reconnect and construction of event queues. From Requirement 10.7 it is clear that these storages would recover within some finite amount of time. During such a recovery the system should be able to reconstruct the event queues which it failed to and route the event queues to the client. This requires that

(a) The unit keeps track of all the requests for event queue construction that it failed to service.

(b) Unserviced clients notify the unit about its location, every time it issues a roam.

10.7.7 How to decide when to store and where to store

10.7.8 Storing to stable storage and location proximity

In the event of failure suspicions, data needs to be logged with location proximity. To elucidate the point further in Fig 7.3 on page 16 if the links connecting SSC-B.SC-6 and SSC-D.SC-10 are suspected failed and if x receives and event e{B,A,C}{11,10}{x}, you've to acknowledge the receipt of that event since otherwise the event would be logged in SC-6.n.

10.8 The need for Epochs

We digress here to discuss the need for *epochs*. The reference count scheme which we discussed in section 8.11 pertains to the number of units/clients that are interested in a certain event. Consider the following scenario. Unit s_A has a total of 156 clients attached to it and unit s_A fails. Clients which detect this failure would initiate a roam. Local queues could be constructed for each client that has initiated a roam in response to this failure. For each queue constructed and sent across the system to it new hosting unit, the reference count associated with every event contained within the queue is decremented by one. However, it is conceivable that a client could have been attached to s_A, which had joined the system for the first time prior to the unit failure. This client is thus *not* the intended recipient of any of the local queues that would be constructed in response to the servicing of roaming clients. If this client is one of the first clients to initiate a roam, local queues would be constructed for it and the reference counts of the events contained within this local queue would be decremented by one. This operation would lead to the *starvation* of at least one client, if any of the 156 clients contained a profile which partially matched that of the new client.

The second scenario is for a client c_A which has delivered events $e_0 \cdots e_{25}$ in its incarnations (past and present) prior to a disconnect in its present incarnation. During the time that c_A was in disconnected the only event targetted to it was e_{26}. When c_A reconnects back the only event that should be routed to it should be e_{26} and not the events that it already delivered in its previous incarnations.

The two scenarios dictate that we need epochs. The two primary issues that we seek to address are -

(a) We should not construct recovery queues for clients that would comprise of events that a client was not originally interested in. This as we discussed earlier could lead to starvation of some of the clients.

(b) We need a precise indication of the time from which point on a client should receive events. This besides leading to client starvations would also cause the system to expend precious network cycles in routing these events.

Epochs are used to aid the re-connected clients and also the recovery from failures. The reason why we can not delegate the event queue generation scheme to the individual units is that a unit can fail and remain failed forever. It is best that the event queue generation is handled by the system as there could be stable storages that could be added within the system and the storage could be delegated to multiple storages within the same context.

10.8.1 Epoch generation

Epochs, denoted ξ, are truly determined by the replication granularity of the system within the context of which the client[15] is a part of. For a replication granularity ℓ, epoch generation schemes would differ depending on whether the event was issued either outside or within the context of the replicator. In the former case, generation of epochs is within purview of one or more[16] gatekeepers $g^{\ell}(C_j^{\ell+1})$ within the context $C_j^{\ell+1}$. In the latter case epoch generation is determined by the finest grained stable storage within the context of the issuing client. Some of the details pertaining to epoch generation are listed below.

(a) Epochs should monotonically increase.

(b) For every $\delta\omega$ associated with a profile ω there is a list of epochs $\xi^{\delta\omega}$ associated with it.

The list of epochs exists because of the presence of multiple gatekeepers and stable storages within the replication granularity.

(c) Epoch advances need to be coordinated between the multiple stable storages that could exist within a context.

[15] A client could be operating in disconnected mode. Such a client is nevertheless still serviced based on its last logical address. The logical address serves as a proxy for the client in its absence.

[16] This arises primarily out of the fact that there could multiple links connecting two units, thus the same event could be assigned different epochs by each gatekeeper receiving the same event.

(d) Epochs exist within the context of the finest grained stable storage and the gatekeepers g^ℓ with the context having a replication granularity ℓ.

In section 10.8.2 we discuss the generation of epochs for events issued by clients attached to node outside the context of the sub system in question, while in section 10.8.3 we discuss the epoch generation scheme for events issued by clients attached to nodes within the sub system. Fig. 10.5 details the two scenarios for generation of epochs associated with events.

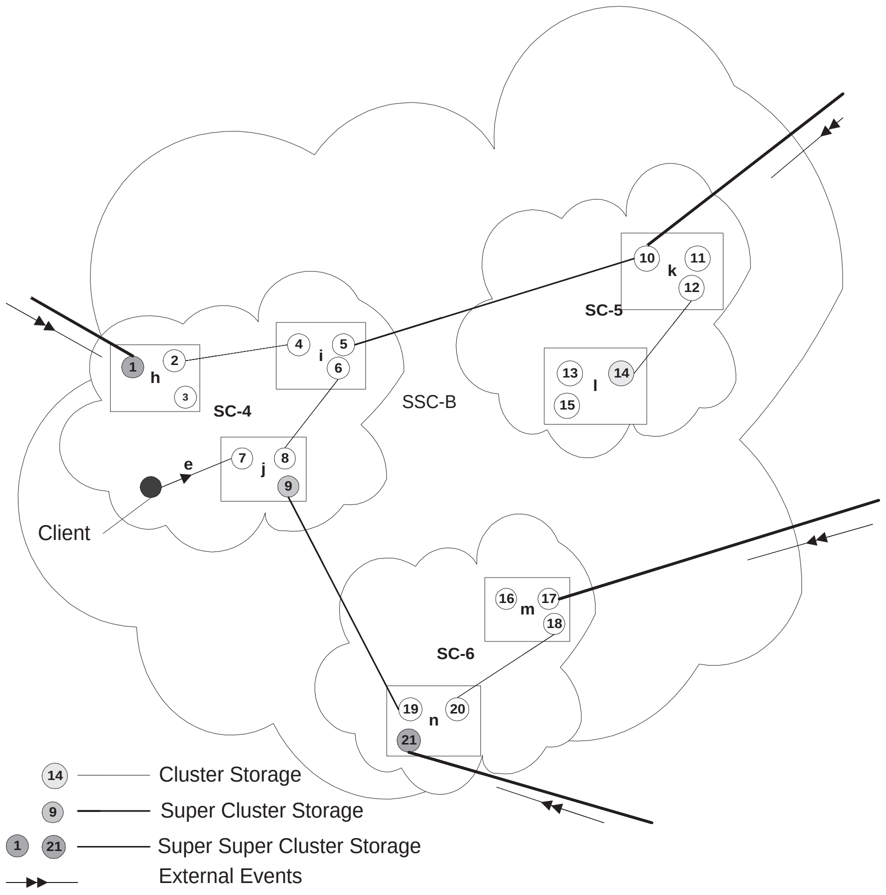

Figure 10.5: Generation of epochs associated with events

Axiom 10.9 *A client will not deliver an event e unless there is an epoch, ξ_e, associated with the event.*

10.8.2 Epoch generation for external events

In this section we are focusing on the sub system which is a level-ℓ unit(context C_i^ℓ) within the context $C_j^{\ell+1}$ and has a replication granularity r_ℓ. We restrict our scope of events to only those events which are issued by clients attached to nodes outside the context C_i^ℓ.

Conjecture 10.10 *For an event e issued by a client attached to a node outside a context $C_i^\ell(C_j^{\ell+1})$, if the event needs to be routed within C_i^ℓ, the event needs to arrive at one of the gatekeepers $g^\ell(C_j^{\ell+1})$ first.*

Based on conjecture 10.10 in our scheme of epoch generation for external events, for a level-ℓ unit within the context $C_j^{\ell+1}$ and a replication granularity r_ℓ, epochs ξ_e for an event e are assigned by gatekeepers $g^\ell(C_j^{\ell+1})$. Depending on the connectivities between different units u^ℓ the same event could arrive at different gatekeepers g^ℓ of the unit u_j^ℓ. For such events, the epochs would of course be different.

However, our duplicate detection scheme (section 8.10) ensures that the duplicate events would be duplicate detected. These events e with the epochs ξ_e assigned to them then are stored at the stable storage. For a replication granularity r_ℓ the stable storage exists at one or more of the gatekeepers g^ℓ. One of the reasons why we didn't delegate the epoch generation scheme to the replicators is that in that case the clients would need to receive epochs generated by the replicator prior to delivering events which arrived at gatekeeper g^ℓ which is not a replicator node.

For a profile ω associated with a client, we denote the smallest individual profile unit as $\delta\omega$. Events are routed to a client based on the $\delta\omega$ that exist within a profile ω. Events conforming to the same $\delta\omega$ can be assigned epochs by any of gatekeepers within the replication granularity. For every $\delta\omega$ associated with a profile ω there is a list of epochs $\xi^{\delta\omega}[\ ]$ associated with it, the $[\ ]$ depending on the number of gatekeepers which have assigned epochs to events conforming to $\delta\omega$. For each $\delta\omega$ the arrival of a new event results in the update/addition of the last epoch received from the epoch-assigning gatekeeper.

The replication granularity within the system could be different in different sub systems. Within a sub system having a replication granularity r^ℓ, it is possible that there is a "sub system" with replication granularity $r_{\ell-1}, r_{\ell-2}, \cdots, r_0$, in such cases the epochs would be assigned by the gatekeepers $g^{\ell-1}, g^{\ell-2}, \cdots, g^0$ respectively.

10.8.3 Epoch generation in units containing issuing clients

In section 10.1.1 we discussed augmenting the client with reissue behavior to account for losses due to consecutive node failures. As opposed to the scheme we discussed earlier (section 10.8.2, when a node receives an event issued by clients attached to some node within the same context C_i^ℓ, there are no epoch numbers associated with it. However, every event needs to have an epoch associated with it to aid in recovery and servicing of undelivered events. This epoch is provided by the stable storage which issues a notification to stop the reissue associated with a certain event. With respect to clients attached to nodes within the same context as the clients issuing logical address, the clients shouldn't deliver events till they have received an epoch from the stable storage servicing the context of which the client's logical address is a part of. The clients/node may receive a certain event but may not deliver it till such time that epoch is received for that event.

10.8.4 Servicing newly reconnected clients

For a profile ω associated with a client, when a disconnected client joins the system it presents the node it connects to in its present incarnation the following -

(a) Its logical address from its previous incarnation.

(b) For every $\delta\omega$ associated with its profile, it presents an array $\xi[\ ]$ of epochs containing the last epoch received from each gatekeeper node g_i^ℓ within the replication granularity r_ℓ of the sub system that it was formerly attached to.

Conjecture 10.8 and item (a) provide us with the list of stable storages that has stored events for the client, while item (b) provides us with the precise instant of time from which point on event queues of events needs to be constructed and routed to the client's new location. In case of client roam or failed storage during reconnection there's another epoch that is associated with the client. This pertains to the time from which point on events *need not* be routed. Of course every recovery of a failed stable storage is a new epoch, and for clients which couldn't be serviced during the time the storage had failed, this is the epoch from which no events should be used in the construction of local queues.

10.8.5 Epochs and profile changes

11 Related work

11.1 OMG Event and Notification Services

The Event Service [OMG00b] provides basic capabilities that can be configured together in a very flexible and powerful manner. Asynchronous events (decoupled event suppliers and consumers), event ”fan-in,” notification ”fan-out,” and (through appropriate event channel implementations) reliable event delivery are supported. There is no requirement for a centralized server or dependency on any global service. In addition, the event service does not impose higher level policies (e.g., specific event types) allowing great flexibility on how it is used in a given application environment. Both push and pull event delivery models are supported: that is, consumers can either request events or be notified of events, whichever is needed to satisfy application requirements.

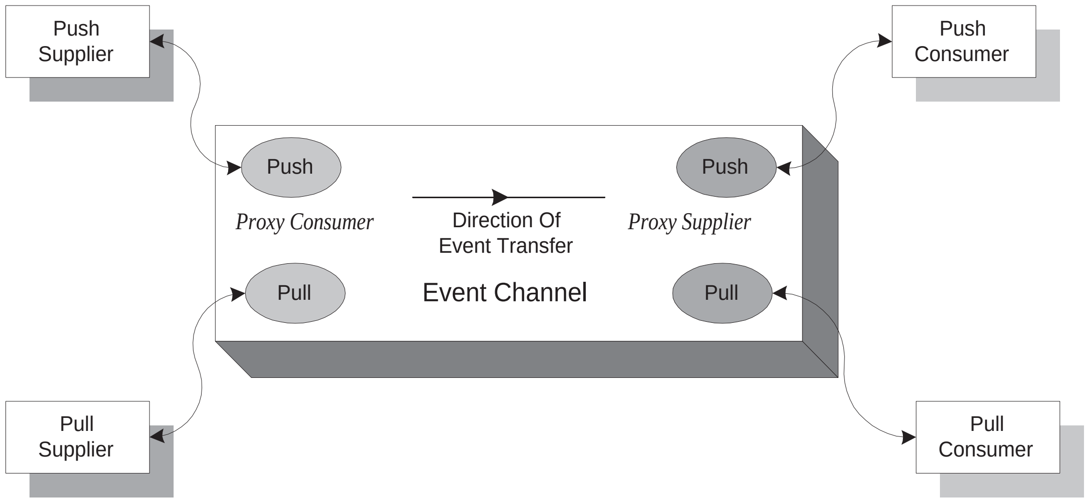

Figure 11.1: OMG Event Service.

There can be multiple consumers and multiple suppliers of events. Suppliers can generate events without knowing the identities of the consumers. Conversely, consumers can receive events without knowing the identities of the suppliers. The event channel interface can be subtyped to support extended capabilities. The event consumer-supplier interfaces are symmetric, allowing the chaining of event channels (for example, to support various event filtering models). Event channels can be chained by third-parties. The Clients could be push/pull-Client, and the Server could be a pull/push Server. The clients play the dual role of suppliers and consumers, though these notions are independent and unaware of the other. An additional advantage of using the EventChannel is that, the events can be buffered to accommodate consumers of differing speeds. The suppliers and the consumers both register with the EventChannel since otherwise its not possible to determine the source of the event in case of the supplier and also since its not possible to invoke the appropriate notification method on the consumer. Typed event channels extend basic event channels to support typed interaction. Because event suppliers, consumers and channels are objects, advantage can be taken of performance optimizations provided by ORB implementations for local and remote objects. No extension is required to CORBA.

Thus to use the event service we would essentially have a large number of event channels. Clients interested in events register in some kind of change notifications. If the client is not present, the notification would of course be lost. Lack of transparency since channels could fail and issuing clients would receive exceptions. Events themselves are not objects since the CORBA distributed model CORBA distributed object model does not support passing objects by value. One of the most serious drawback is that all consumers registered to a particular event channel receive all events supplied to that event channel using the *at most once* best effort delivery. Thus all events would receive all events issued by all suppliers.

In case of explicit confirmations that suppliers may need in response to concumption by consumers, what is employed is a *reverse* event channel through which consumers can send confirmations back as normal events. Also, the Event Service interfaces allow implementations that provide different qualities of service to satisfy different application requirements. The two serious limitations of event channels are the lack of event filtering capability and the inability to configure support for different qualities of service. These are sought to be addressed in the Notification Service [OMG00a] design. However the Notification service attempts to preserve all the semantics specified in the OMG event service, allowing for interoperability between Event service clients and Notification service clients. Thus even in this case the a client needs to subscribe to more than one event channel.

11.2 Ninja

In [GWvB$^+$00] we have the following scenario. [Aka, Sof, ORA00].

12 Snapshots of innovations

(a) No assumptions made regarding the client's computing power or the transport layer over which it communicates.

(b) Reliable delivery in the presence of node and link failures, intermittent connection semantics at the client and prolonged disconnects at a client.

(c) Ability to specify, resolve and detect cyclic dependencies in the delivery of spatially and chronologically constrained events.

(d) Routing events to a client in a manner consistent with the profile changes, while reducing latencies in the delivery of an event which matches the profile changes at the client.

(e) Scalable solution. Easy configuration and extensibility (in terms of adding new units and links) of the system.

(f) The PPP, ERP, NOP, GPP and failure detections responses are organized in a way such that they operate independently of each other. Resulting in maximum concurrency of operations.

(g) Lower event, bundle, merged stream delivery latencies.

(h) Self-sustaining and persistent merged streams. You could continuously add events, bundles and streams to an existing merged stream.

(i) The replication scheme goes a long way in obviating issues related to "digital distances" by reducing response times for disconnected clients who reconnect back to its server node.

(j) Ability to withstand stable storage failures, while servicing clients with in a finer or coarser grained unit.

13 Implementation Strategy - Transport level Details

Each node contains the following -

- The maximum number of nodes that are allowed in the cluster.
- The IP-discrimination scheme employed by the nodes within the cluster. This scheme is used to determine if a node can be part of the cluster. One of the reasons why we employ such a scheme is that, nodes within a cluster would communicate using UDP. Losses are very low, and communication very fast while using UDP between nodes on the same token ring in the network. In the case of a node not on the same token ring as the rest of the cluster, losses incurred due to UDP communication would be inordinately high. These high losses would serve to divert the rest of the nodes, towards error correction and handling of messages over the erring link, from their basic task of efficient dissemination of messages received at the cluster gateways.
- There is also a connection set up vector, which specifies the maximum number of concurrently active connections that a node can maintain at any given time. Associated with every connection between nodes (server or client) there is a receiver thread per transport protocol, and sender threads proportional to the number of active connections and the number of transport protocols supported by the node and nodes connected to it. There is thus a price to be paid (CPU-dollars) for maintaining active concurrent connections to multiple nodes. In addition using UDP, Multicast require regular polling since there are no outtage notifications (akin to ICMP provided by the TCP core protocol set) in response to socket failures. A node hosted on a slower machine could thus tune its processing to the capabilities existing in the underlying system and also on the number of processes active on the system.

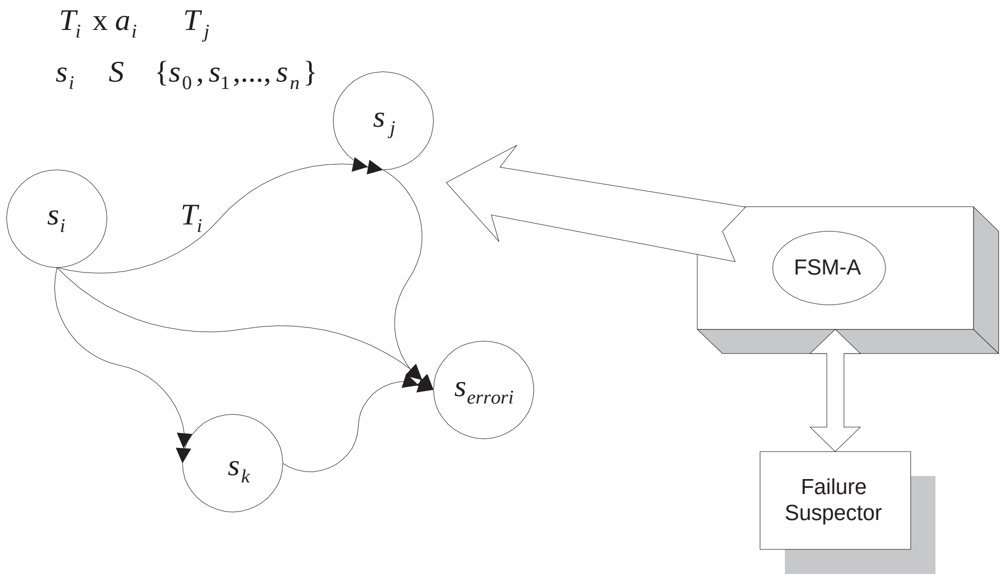

Figure 13.1: The State at each node.

In addition to this every node, can communicate using TCP, UDP, Multicast. Every node publishes the TCP port, UDP port and the Multicast group over which it monitors connection requests and message interchange. In the case of UDP and Multicast, though the communication is fast the communication is unreliable. UDP and Multicast communication provide no guarantees on the delivery and ordering of message packets. As a result messages received could contain holes in the intended sequence and the messages could also be out of order. Thus we employ an error detection and correction mechanism atop the UDP and Multicast transports for our protocol needs. The other disadvantage of UDP communication is that there are no outtage notifications.

14 The DTD for the event

Events conform to XML DTDs. Not all fields within the DTDs need to be present, some fields are however mandatory. At every server node hop, the DTD definition for the event needs to be referenced. There are two ways for this information to be included within the XML event

(a) Include the DTD definition within the event itself. This is ruled out as the information contained within the XML event would increase.

(b) Include a pointer to the DTD definition. This would entail a lot of network traffic with every arrived event resulting in a network operation to fetch the document definition.

To work around items (a) and (b) we employ the following approach. The first time that an event type is encountered at a server node, the DTD definition is fetched[17] and cached at the server node. Thus we circumvent the network operation.

An event exists within the context of a stream, thus the specification of an event includes the stream that this event is a part of, this is specified by the StreamId. Every event needs to have an Id, `Mspaces:EventId`, that is unique in space and time. Events also should be able to specify the linkages that exist between them and events within other streams, this constitutes the `Mspaces:EventLinkage`. Resolution of the event linkage is a precursor to the creation of merged streams. We also need an indication of the type of event that this event is, i.e. live or recovery and the security constraints contained within the event. This is included in `Mspaces:EventType`. Events could also possibly specify zero or applications that it is a part of. The event summary, which could occur once or not at all, provides a synopsis of the event itself. Thus an Event definition could be the following.

```
<!ELEMENT Mspaces:Event (Mspaces:StreamId, Mspaces:EventId,  Mspaces:EventType,
                         Mspaces:EventLinkage, Mspaces:ApplicationType*,
                         Mspaces:Summary?)>
```

This specification dictates that the various elements should appear in the order `Mspaces:StreamId` first, then `Mspaces:EventId` and so on. The `StreamId` representation is a simple (`#PCDATA`) representation.

```
<!ELEMENT Mspaces:StreamId         (#PCDATA)>
```

The ID associated with every event, `Mspaces:EventId`, needs to be unique in space and time. Having a unique Client Id, `Mspaces:ClientId` reduces the uniqueness problem to a point in space. `Mspaces:TimeStamp` provides the uniqueness in the time domain, while the sequence number (contained in `Mspaces:SequenceNumber`) scheme ensures issue rates which are higher than that dictated by the constraint imposed on uniquely identifiable events by the granularity of the underlying clock. `Mspaces:IncarnationNumber`'s are essential to avoid conflicts when an issuing client initiates a roam. The duplicate detection loop hole exists since no process has access to a global clock and also since the clocks on individual machines are never synchronized. Even if the clocks were synchronized, the rates at which these individual clocks tick are different. The following definition for the eventId specifies a an ID unique in space and time.

```
<!ELEMENT Mspaces:EventId (Mspaces:ClientId, Mspaces:TimeStamp,
                           Mspaces:SequenceNumber, Mspaces:Incarnation)>
<!ELEMENT Mspaces:ClientId         (#PCDATA)>
<!ELEMENT Mspaces:TimeStamp        (#PCDATA)>
<!ELEMENT Mspaces:SequenceNumber   (#PCDATA)>
<!ELEMENT Mspaces:Incarnation      (#PCDATA)>
```

Earlier we discussed our approach to circumventing network operations while parsing the XML events. DTD's could however change, and the cache rendered useless, to account for this scenario we need to include the concept of version Number within the DTD fields. When the event is parsed a look at the

[17]This DTD definition could be fetched either from the pointer contained within the event or from the node which routed the event to this node in the first place.

Mspaces:versionNumber field could tell us if the cache needs to be updated. If the DTD definition for the event is changed the clients interested in the events conforming to the old DTD definition need to be notified about this change. These clients could then decide if their profiles need to be updated to reflect this change. This notification of the change in the DTD of the event that a client is interested is included in the field Mspaces:LatestVersionNumber. Also nodes need to maintain the DTD definitions for different versions of the same DTD. It is conceivable that there are events being published within the system or there are recovery events which would conform to the old versions of the DTD. Information regarding these version numbers along with the security constraints and liveness indicator constitute Mspaces:EventType.

```
<!ELEMENT Mspaces:EventType (Mspaces:VersionNum, Mspaces:LatestVersionNum?)
<!ATTLIST Mspaces:EventType
         Securitylevel    (low | med | high) ‘‘med’’
         Liveness         (live|recovery)  ‘‘live’’>
<!ELEMENT Mspaces:VersionNum       (#PCDATA)>
<!ELEMENT Mspaces:LatestVersionNum (#PCDATA)>
```

If an Mspaces:EventType created does not specify values for the SecurityLevel and Liveness attributes, the EventType is assumed to be a "live" event of "med" security. Recovery events are the events which clients have missed either during a *roam* operation or during a prolonged disconnect.

The Mspaces:EventLinkage specifies the dependencies that exist between events in multiple streams. The linkage should provide for resolution of the spatial and timing dependencies in an implicitly or explicitly specified order. Besides these we also need the ability to create *bundles* of events within a given stream. The bundles that we create need an identifier, this is provided by Mspaces:BundleId. However, there could be situations where the bundle we consider is the stream itself. Bundles need to also indicate the methodology that needs to be in place to decide upon the merging schemes. This is provided by the enumeration of Mspaces:TimeConstraint and Mspaces:MergeScheme. Some bundles however, may not impose any scheme on the merging of bundles. We account for such a scenario by including "None" in the enumeration for the linkage schemes which we mentioned earlier. Events within a bundle also have monotonically increasing sequence numbers assigned to events within the bundle. This is in addition to the sequence numbers that events possess to determine a uniqueID. The Mspaces:BundleNumber however, comes into play only in the presence of a Mspaces:BundleId within the event stream. The Mspaces:BundleOrder specifies the ordering scheme that should be in place for events which are "concurrent" based on the merging methodology that is specified by Mspaces:BundleLinkage.

```
<!ELEMENT Mspaces:EventLinkage ((Mspaces:BundleId, Mspaces:BundleNumber)? ,
                                Mspaces:BundleLinkage, Mspaces:BundleOrder)
<!ELEMENT Mspaces:BundleId         (#PCDATA)>
<!ELEMENT Mspaces:BundleNumber     (#PCDATA)>
<!ELEMENT Mspaces:BundleLinkage     NONE | (Mspaces:TimeConstraint?,
                                           Mspaces:MergeScheme?)>
<!ELEMENT Mspaces:TimeConstraint   (#PCDATA)>
<!ELEMENT Mspaces:MergeScheme      (#PCDATA)>
<!ELEMENT Mspaces:BundleOrder      (Mspaces:StreamId+ | Mspaces:BundleId+)>
```

A brief note about the Mspaces:EventLinkage is in order. If an event is allowed to be part of multiple bundles within the same stream with multiple BundleNumber's the ? should be * in the Mspaces:BundleId, Mspaces:BundleNumber grouping. The listing of the DTD in section 14.1 and element analysis in table 14.1 assumes the ? occurrence operator.

14.1 The complete DTD

The event routing information as specified by the event routing protocol (ERP) and the information contained within the event during recoveries are not included within the definition for the DTDs. The event itself is encapsulated within an XML document, however the routing is not. Below we include the complete definition of the event, which follows from our discussions so far.

```
<!ELEMENT Mspaces:Event (Mspaces:StreamId, Mspaces:EventId,  Mspaces:EventType,
                         Mspaces:EventLinkage, Mspaces:ApplicationType*,
                         Mspaces:Summary?)>

<!ELEMENT Mspaces:StreamId         (#PCDATA)>

<!ELEMENT Mspaces:EventId (Mspaces:ClientId, Mspaces:TimeStamp,
                           Mspaces:SequenceNumber, Mspaces:Incarnation)>
<!ELEMENT Mspaces:ClientId         (#PCDATA)>
<!ELEMENT Mspaces:TimeStamp        (#PCDATA)>
<!ELEMENT Mspaces:SequenceNumber   (#PCDATA)>
<!ELEMENT Mspaces:Incarnation      (#PCDATA)>

<!ELEMENT Mspaces:EventType (Mspaces:VersionNum, Mspaces:LatestVersionNum?)
<!ATTLIST Mspaces:EventType
         Securitylevel    (low | med | high) ‘‘low’’
         Liveness         (live|recovery)  ‘‘live’’>
<!ELEMENT Mspaces:VersionNum       (#PCDATA)>
<!ELEMENT Mspaces:LatestVersionNum (#PCDATA)>


<!ELEMENT Mspaces:EventLinkage ((Mspaces:BundleId, Mspaces:BundleNumber)? ,
                                 Mspaces:BundleLinkage, Mspaces:BundleOrder)
<!ELEMENT Mspaces:BundleId         (#PCDATA)>
<!ELEMENT Mspaces:BundleNumber     (#PCDATA)>
<!ELEMENT Mspaces:BundleLinkage    NONE | (Mspaces:TimeConstraint?,
                                           Mspaces:MergeScheme?)>
<!ELEMENT Mspaces:TimeConstraint   (#PCDATA)>
<!ELEMENT Mspaces:MergeScheme      (#PCDATA)>
<!ELEMENT Mspaces:BundleOrder      (Mspaces:StreamId+ | Mspaces:BundleId+)>

<!ELEMENT Mspaces:ApplicationType  (#PCDATA)>
<!ELEMENT Mspaces:Summary          (#PCDATA)>
```

Table 14.1 depicts the various elements, the nested elements and occurrence bounds for the nested elements within a specific element. The table also snapshots our discussions so far with brief descriptions of the purpose of each element with the event element hierarchy.

Element	**Allowed Nested Elements**	**Num**	**Purpose of the Element**
`Mspaces:Event`	Mspaces:StreamId Mspaces:EventId Mspaces:EventType Mspaces:EventLinkage Mspaces:ApplicationType Mspaces:Summary	1 1 1 1 $0 \cdots N$ 0/1	Overall root element of the Event
`Mspaces:StreamId`	None		Stream the event belongs to
`Mspaces:EventId`	Mspaces:ClientID Mspaces:TimeStamp Mspaces:SequenceNumber Mspaces:Incarnation	1 1 1 1	The unique event ID.
`Mspaces:ClientID`	None		The issuing Client ID.
`Mspaces:TimeStamp`	None		Time Stamp usually in milliseconds
`Mspaces:SequenceNumber`	None		Issue events at rate greater than the granularity of the timeStamp.
`Mspaces:Incarnation`	None		Allows for duplicate detection during a issuing client *roam* .
`Mspaces:EventType` (*attributes* : live, secure)	Mspaces:VersionNum Mspaces:LatestVersionNum	1 0/1	Information about the versioning and liveness of an event.
`Mspaces:VersionNum`	None		The version number of the DTD that XML event conforms to
`Mspaces:LatestVersionNum`	None		Inform clients about the version change to a DTD.
`Mspaces:EventLinkage`	Mspaces:BundleId Mspaces:BundleNumber Mspaces:BundleLinkage Mspaces:BundleOrder	0/1 0/1 1 1	Specification for the linkage of events in multiple streams.
`Mspaces:BundleId`	None		Identifies a specific bundle within the stream.
`Mspaces:BundleNumber`	None		Specifies the numbering with in the bundle of a stream. Depends on the presence of the Bundle.
`Mspaces:BundleLinkage` *(Enumeration)*	Mspaces:TimeConstraint Mspaces:MergeScheme	0/1 0/1	Specifies the method for merging streams/bundle.
`Mspaces:TimeConstraint`	None		Specifies merging based on time.
`Mspaces:MergeScheme`	None		Specifies a merge scheme.
`Mspaces:BundleOrder` *(Enumeration)*	Mspaces:StreamId Mspaces:Bundle	$1 \cdots N$ $1 \cdots N$	Specifies ordering for concurrent events

Table 14.1: Mspaces:Event Hierarchy

References

[Aka] Akamai. Delivering a better internet - technology. http://www.akamai.com.

[BBT96] Anindya Basu, Bernadette Charron Bost, and Sam Toueg. Solving problems in the presence of process crashes and lossy links. Technical Report TR 96-1609, Dept. Of Computer Science, Cornell University, Ithaca, NY-14853, September 1996.

[Bir85] Kenneth Birman. Replication and fault tolerance in the isis system. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles*, pages 79–86, Orcas Island, WA USA, 1985.

[Bir93] Kenneth Birman. The process group approach to reliable distributed computing. *Communications of the ACM*, 36(12):36–53, 1993.

[GRVB97] Katherine Guo, Robbert Renesse, Werner Vogels, and Ken Birman. Hierarchical message stability tracking protocols. Technical Report TR97-1647, Dept. Of Computer Science, Cornell University, Ithaca, NY 14853, 1997.

[GWvB$^+$00] Steven D. Gribble, Matt Welsh, Rob von Behren, Eric A. Brewer, David Culler, N. Borisov, S. Czerwinski, R. Gummadi, J. Hill, A. Joseph, R.H. Katz, Z.M. Mao, S. Ross, and B. Zhao. The ninja architecture for robust internet-scale systems and services. In *Special Issue of Computer Networks on Pervasive Computing*. 2000.

[HT94] Vassos Hadzilacos and Sam Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical Report TR94-1425, Dept. Of Computer Science, Cornell University, Ithaca, NY-14853, May 1994.

[OMG00a] The Object Management Group OMG. Corba notification service. http://www.omg.org/technology/documents/formal/notificationservice.htm, June 2000. Version 1.0.

[OMG00b] The Object Management Group OMG. Omg's corba event service. http://www.omg.org/technology/documents/formal/eventservice.htm, June 2000. Version 1.0.

[ORA00] ORACLE. *Oracle Portal - Production Release*, 3.0 edition, August 2000.

[RSB93] Aleta Ricciardi, Andre Schiper, and Kenneth Birman. Understanding partitions and the "no partition" assumption. In *Proceedings of the Fourth Workshop on Future Trends of Distributed Systems*, Lisbon, Portugal, September 1993.

[Sof] Softwired. ibus technology. http://www.softwired-inc.com.

A Appendix - Data Structures, Message Formats etc

A.1 Event routing protocol

`Protocol.DATA` $\rightarrow$ byte[0].
`DataLength` $\rightarrow$ byte[1] $\cdots$ byte[4]

```
DataLength = Event.length = StreamId.length + EventId.length +
             EventLinkage.length +  EventType.length + event.length
```

Now we proceed to the information pertaining to routing. The event routing protocol, dictates that different nodes include/exclude certain levels of routing information based on the kind of gateway it serves to the context it is routing the information to. In the routing information we indicate the routing information included for the event, this is provided by `LevelInfo`.

`LevelInfo` $\hookrightarrow$ byte[2 + `DataLength`]

`RoutingInformation` $\rightarrow$ byte[3 +`DataLength`] $\cdots$ byte[3+`DataLength` + 4 $\times$ `LevelInfo`]

`Protocol.DATA` size = 1 + 4 + `DataLength` + 1 + 4$\times$`LevelInfo`

When a the routing information is received at a node, the routing information is reflected to account for the fact that the event was received at this node. This is done by performing a bit-wise `Or` operation on the routing information received with the context information of the node. Before routing data over a gateway check to see if the intended receiving node has already received the event, if it has do not route the event to that node.

A.2 Gateway Propogation Protocol

The first byte in the setup request indicates the type of the request.
`Protocol.GPP_SETUP_REQUEST` $\rightarrow$ byte[0].
We then need to indicate the setup request. This setup request vector indicates the kind of gateway it seeks to be. A byte or 8-bits more than suffice to handle all the levels that a system could have. The request should be consistent with the rules rules for adding a gateway. A node within a context C_i^ℓ can not seek to be a level $\ell \cdots 0$ gateway node to another node in context C_j^ℓ. A 1 in the bit-vector along with the position in the bit-vector, starting with the least significant bit, indicates the kind of gatekeeper the node seeks to be. Thus a request of the form 00000010 seeks to be a level-2 node. The request could not be satisfied if the connection vector at the node in question has exceeded its saturation point for handling connections at that level.
`SetupRequestVector` $\rightarrow$ byte[1].

The response to this request could either be

(a) A success - In which case the gateway information at all the nodes within the context's, C_i^ℓ and C_j^ℓ within the context $C_k^{\ell+1}$, connected by the new gateway, are updated to reflect this new addition. In addition all other level-ℓ gateways within $C_k^{\ell+1}$ are updated to reflect this information.

(b) An error - This results from a malformed request or from a request which is logically incorrect. Also this error could take place during namespace conflicts which could occur when a node seeks to establish the gateway. In other words the node should be able to detect the fact that the protocol that should be used is NOP and not GPP.

(c) The request could also be failure if it fails to meet the terms dictated by the connection setup vector. The request could also fail if the gateway already exists between the nodes in question.

(d) Partially Satisfied - The node to which the request was sent to can not satisfy the request, but it forwards a set of addresses that can do the same.

The first byte indicates that it is a setup response message.
`Protocol.GPP_SETUP_REQUEST` size = 1+1
`Protocol.GPP_SETUP_RESPONSE` $\rightarrow$ byte[0]
`ResponseCode` $\rightarrow$ byte[1] which could be `{Success, Error, Fail, Partial}`.

If `ResponseCode.Partial` the second byte indicates the number of other nodes that can be contacted. The total number of such nodes is included in `NumOfNodes` $\rightarrow$ byte[2]. For each of the nodes, the contact information is split into `NodeInformation.Length` and `NodeInformation`. Thus the total size of the response packet in case of `ResponseCode.PartiallySatisfied` is 1 + 1 +

`Protocol.GPP_INFO` $\rightarrow$ byte[0]. This is used to update the routing information for level-ℓ gateways within its immediate context $C_i^{\ell+1}$. This is of course intended to allow for the addition and removal of gateway information. This indication is included in `Addition` and `GatewayInfoVector`, which depicts the level of the gateway that this gateway information would affect. We also need to include the logical and physical address of the node in question. Using these information we can update the gateway information at nodes within the context $C_i^{\ell+1}$.

A.3 Node Organization Protocol

Indication of whether a node/unit is being and the level-ℓ of the unit is implicit in the request that arrives at a node.
`Protocol.NOP` $\rightarrow$ byte[0].
`Protocol.ConnectionSetup` $\rightarrow$ byte[1] provides information regarding the kind of gateway that the node seeks to be. This request could either -

- Succeed -
- Fail -
- Error -
- Partially Satisfied -

In the case of a successful setup of the system, the logical address of the added node, unit (in this case what changes is the highest level context) and this should be included in the `NOP_RESPONSE`.
`NOP_RESPONSE` $\rightarrow$ byte[0]
`RequestNumber` $\rightarrow$ byte[1]
`{Success , Fail, Error, Partial}` $\rightarrow$ byte[2]
If `Response.Partial` we would have additional information pertaining to the Nodes that would be included. This includes the `NumOfNodes` as also the `NodeInformationLength` and the `NodeInformation` for each of the `NumOfNodes` that have been included in this information.

In the case of a success, the logical address associated with node/nodes in the subsystem that is being added changes at its most significant context. Once this address change is reflected across the sub-system, we proceed with updating the gateway information at each of the nodes. Once tha address change is complete, the gateway setup status is updated to `Setup.Complete` and then GPP proceeds with providing the relevant information regarding the gateways at level-ℓ and upwards.

A.4 Protocol Constants

These are usually the first byte in a message. Primarily used to determine the type of message that we would be dealing with. Within each protocol, there would be additional constants which are specific to the protocol in question. This would be handled within each individual protocol.

```
Protocol.DATA
Protocol.GPP_SETUP_REQUEST
Protocol.GPP_SETUP_RESPONSE
Protocol.GPP_INFO
Protocol.NOP_SETUP_REQUEST
Protocol.NOP_SETUP_RESPONSE
Protocol.TCP
Protocol.UDP
Protocol.Multicast
Protocol.
```

A.5 Exception Hierarchy

Exceptions need to be in place to provide information regarding the various exceptions that could take place during either the addition of a node or the setup of a gateway and so on.

A.5.1 Overall Design Strategy

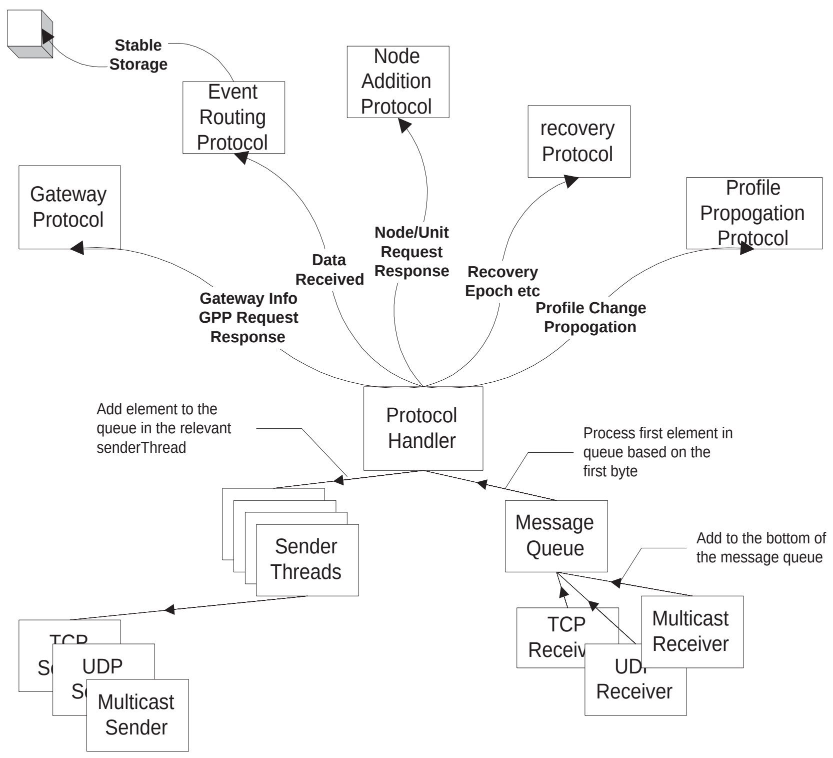
Stable Storage
Event Routing Protocol
Node Addition Protocol
recovery Protocol
Profile Propogation Protocol
Gateway Protocol
Data Received
Node/Unit Request Response
Recovery Epoch etc
Profile Change Propogation
Gateway Info GPP Request Response
Protocol Handler
Add element to the queue in the relevant senderThread
Process first element in queue based on the first byte
Sender Threads
Message Queue
Add to the bottom of the message queue
TCP Receiver
UDP Receiver
Multicast Receiver
TCP Sender
UDP Sender
Multicast Sender

Figure A.1: Module Interaction - Higher Level.

ERP also needs to know the precise `SenderThread` that should be used for routing individual events. All these protocol modules deal with the logical addresses of the nodes. Th actual mapping of physical and logical addresses is maintained in the `ProtocolHandler`.

When we receive an event, the routing information contained within the event needs to be updated to reflct the fact that event was delivered at the node. This is done by performing a logical `Or` operation on the received routing information and the logical address of the node. To determine the unit that an event needs to be routed to, `XOR` the updated[18] routing information with the gateway information available at the node. We of course would do this operation first for the highest level gateways.

The gateway info and the connection vector information can be updated only by the GPP and NOP modules. The Gateway information is also used by the ERP and PPP modules to decide the next hop for routing events and propogation profile changes respectively.

Organization of gateways: The objective of storing gateway information efficiently is to

(a) Compute efficient paths to reach a destination.

(b) Compute alternate paths to reach the same destination in the event of node/gateway failures.

The premise of this information is a deterministic approach to deciding the next hop that an event should take. For this we need precise information of the connections within a unit and also a snapshot of the

[18]When we say updated, we are referring to the fact that the node updates the received events routing information to reflect the fact that the event was received by the node within a unit.

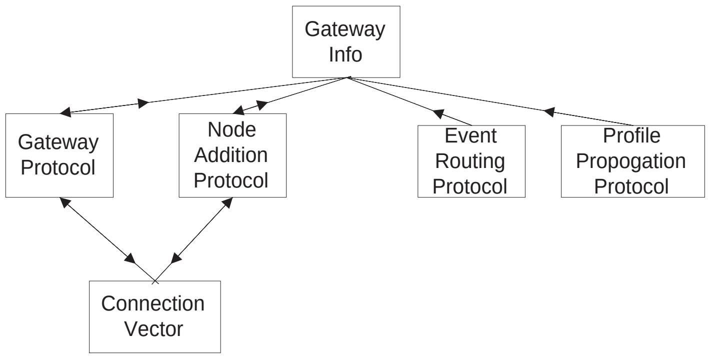

Figure A.2: Data Structures being shared.

overall connection setup outside of the system. Even if a unit u_i^ℓ may not be interested in a certain event, the unit has to ensure disseminations to other units u_j^ℓ within the same level-ℓ context, or other super units.