On the design of multi-tagged event queues and their effects on Distributed services and Computing*

Shrideep Pallickara   Geoffrey Fox

June 19, 2000

Contents

*Northeast Parallel Architectures Center, Syracuse University, Syracuse NY 13210

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 a indication that he wants guaranteed delivery of the event.

Events trigger *actions*, 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.

This paper, is about events, the organization, retrieval and specification of attributes and constraints associated with that event. This paper is also about event queues comprising of the aforementioned events. We attempt to 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.

2 Specifications

We now try to specify our problem. In section 2.1 we present our model of the system in which we intend to solve the problem. In section 2.2 we formally specify our problem. Sections 2.3 and 2.4 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 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 are can never be attached to each other, thus they never communicate directly with each other. Let **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.

2.2 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 $E_\sigma^i \subseteq E_\sigma$ be the set of events e_σ^i 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 2.1. 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.

2.3 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.

2.4 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 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.

3.1 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’’
```

3.1.1 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.

3.1.2 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.

3.1.3 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.

3.2 Attribute Information

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

3.2.1 Tagged Information & the event type

The tagged information contains values for the 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.

3.2.2 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[1] 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.

3.3 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.

3.3.1 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.

3.3.2 Correlation Identifiers

Correlation identifiers help impose the causal delivery constraints on the request$\rightarrow$reply events.

[1] 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.

3.3.3 Qualities of Service Specifiers

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

3.4 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.

3.5 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 3.2.1. Derived events can be processed as base events and not vice versa.

3.6 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 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.

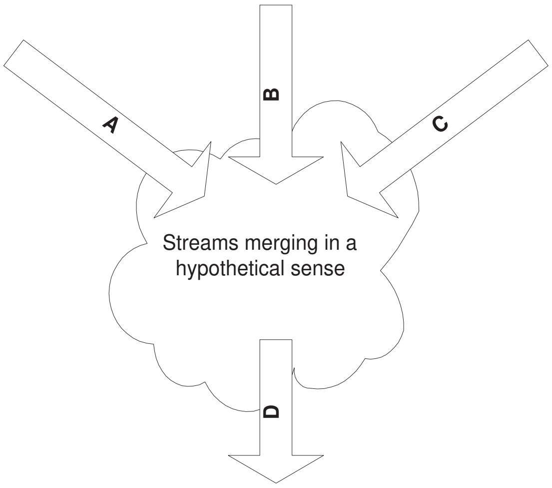

Figure 1: Existence of multiple event streams.

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 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 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

[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.

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.

4.1 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 occurence 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 (4.1) specifies the relationships that exist within the events of an event stream. The events within an event stream could be *precedence related* ($\rightsquigarrow$) 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.

$$E = \overbrace{\{e_0 \rightsquigarrow e_1 \rightsquigarrow \cdots\}}^{\text{Ordered Set}} \mid \overbrace{\{e_0, e_1, e_2, \cdots\}}^{\text{Unordered Set}} \tag{4.1}$$

In equation (4.2), $\hookrightarrow$ 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 $E^j \in \Pi$.

$$E \hookrightarrow \Pi = \{E^1, E^2, E^3, \cdots, E^N\} \tag{4.2}$$

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 (4.3) we define the values which elements in the occurence 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).

$$\text{Occurence Vector } \mathcal{O} = \{?, +, *, \star\} \tag{4.3}$$

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 (4.4) is one of the forms that *stream interaction rules* could take.

$$E \leftrightarrow E^j \Rightarrow E.e_i \hookrightarrow E_j.e_i^j \mid E.e_i \hookrightarrow E^j.e_{i\pm N}^j \text{ where } \leftrightarrow \text{ specifies the mapping rule} \tag{4.4}$$

Equation (4.5) 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 4.5 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.

$$E^j(e^{func}) = \sum e^j \in E^j \ni e^j \text{ satisfies } e_i^{func} \tag{4.5}$$

Equation (4.6) specifies the resolution of an events dependency. 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.

$$\begin{aligned} \forall e_i \in E, e_i \hookrightarrow \Pi' \subseteq \Pi \quad &\equiv \quad \overbrace{e_i(data)}^{\text{Implied}} \cup \sum_{j=1}^{\#\Pi'} \overbrace{\{E \leftrightarrow E^j \mid E^j(e_i^{rule}) \mid E^j(e_i^{tags})\}}^{\text{Stream Interaction Rules}} \odot \overbrace{o_i \in \mathcal{O}}^{\text{Occurrance}} && (4.6) \\ &\equiv \quad Null \mid e \mid e[\,] && (4.7) \end{aligned}$$

Equation (4.8) 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 (4.6). The event dependency resolution of every event within E results in the creation of the merged event stream.

$$\sum_{i=0}^{\#E} e_i \hookrightarrow \Pi' \subseteq \Pi = E^{MergedStream} \tag{4.8}$$

4.2 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 evetually delivers the merged event stream $\sum_{i=0}^{\#E}(e_i \hookrightarrow \Pi' \subseteq \Pi)$.

4.3 Specifying the anatomy of an event

These sets of equations follow from our discussions in section 3 and section 4.1. Equation (4.9) follows from our discussions in section 3.2.2 regarding the generation of unique identifiers. This tuple is created by the issuing clients.

$$eventId =< clientId, timeStamp, seqNumber, incarnation > \tag{4.9}$$

The tuple in 4.10 discriminates between live events and recovery events (which occur due to failures or prolong disconnects).

$$liveness =< live|recovery > \tag{4.10}$$

The type of an event is dictated by the event *signature*. These signatures could change, to accomodate these changes we include the concept of versioning in our event signatures. This along with *liveness* (equation 4.10) describe the event type completely.

$$eventType = < signature, versionNum, liveness > \tag{4.11}$$

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.

$$destinationLists = < \overbrace{Implied}^{\text{External}} \mid \overbrace{Explicit}^{\text{Internal}} > \tag{4.12}$$

The dependency indicator follows from our discussions in section 4.1 and equations (4.3) through (4.6).

$$dependencyIndicator = <? \mid * \mid + \mid \star > \odot < mapping \mid rules \mid constraints > \tag{4.13}$$

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

$$\begin{aligned} event \quad = \quad & < eventId, eventType, attributesList, dependencyIndicator, \\ & stream, applicationType, destinationLists > \end{aligned} \tag{4.14}$$

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.

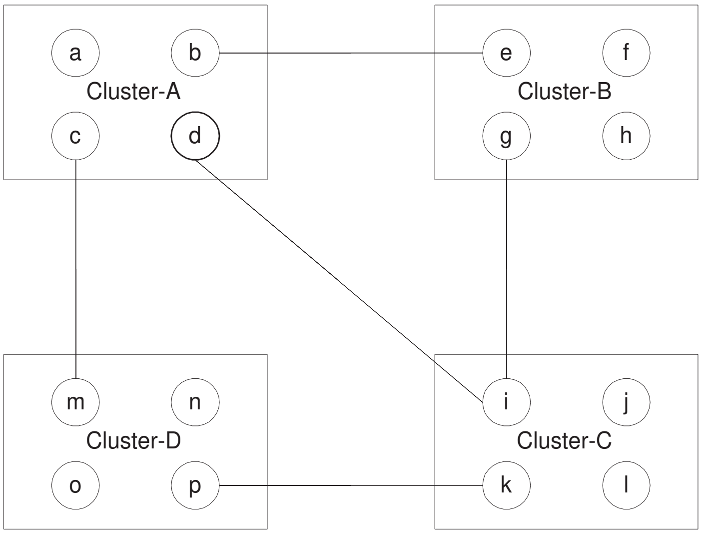

Figure 2: A Super Cluster - Cluster Connections

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. 2. Clusters within a super-cluster have one or more 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 2 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 other clusters.

This topology could be extended in a similar fashion to comprise of *super-super-clusters* (level-3) as shown in Fig. 3, *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

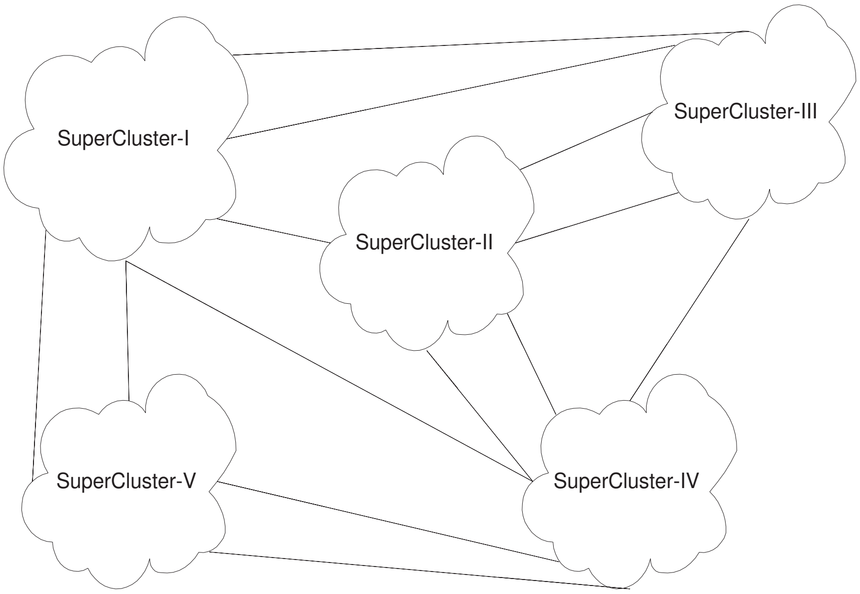

Figure 3: A Super-Super-Cluster - Super Cluster Connections

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_i^1 of a cluster, which in turn exists within the context C_j^2 of a super-cluster and so on. In general a context C_i^ℓ at level l exists within the context $C_j^{\ell+1}$ of a level $(\ell+1)$. In a N-level system the following hold -

$$C_i^0 = (C_j^1, i) \tag{7.1}$$

$$C_j^1 = (C_k^2, j) \tag{7.2}$$

$$\vdots$$

$$C_p^{N-2} = (C^{N-1}, p) \tag{7.3}$$

$$C_q^{N-1} = q \tag{7.4}$$

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 $C_i^{\ell+1}$ 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_i^2 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_i^2. 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_i^3. 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 $C_j^{\ell+1}$ denoted $g_i^{\ell}(C_j^{\ell+1})$ comprises of -

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

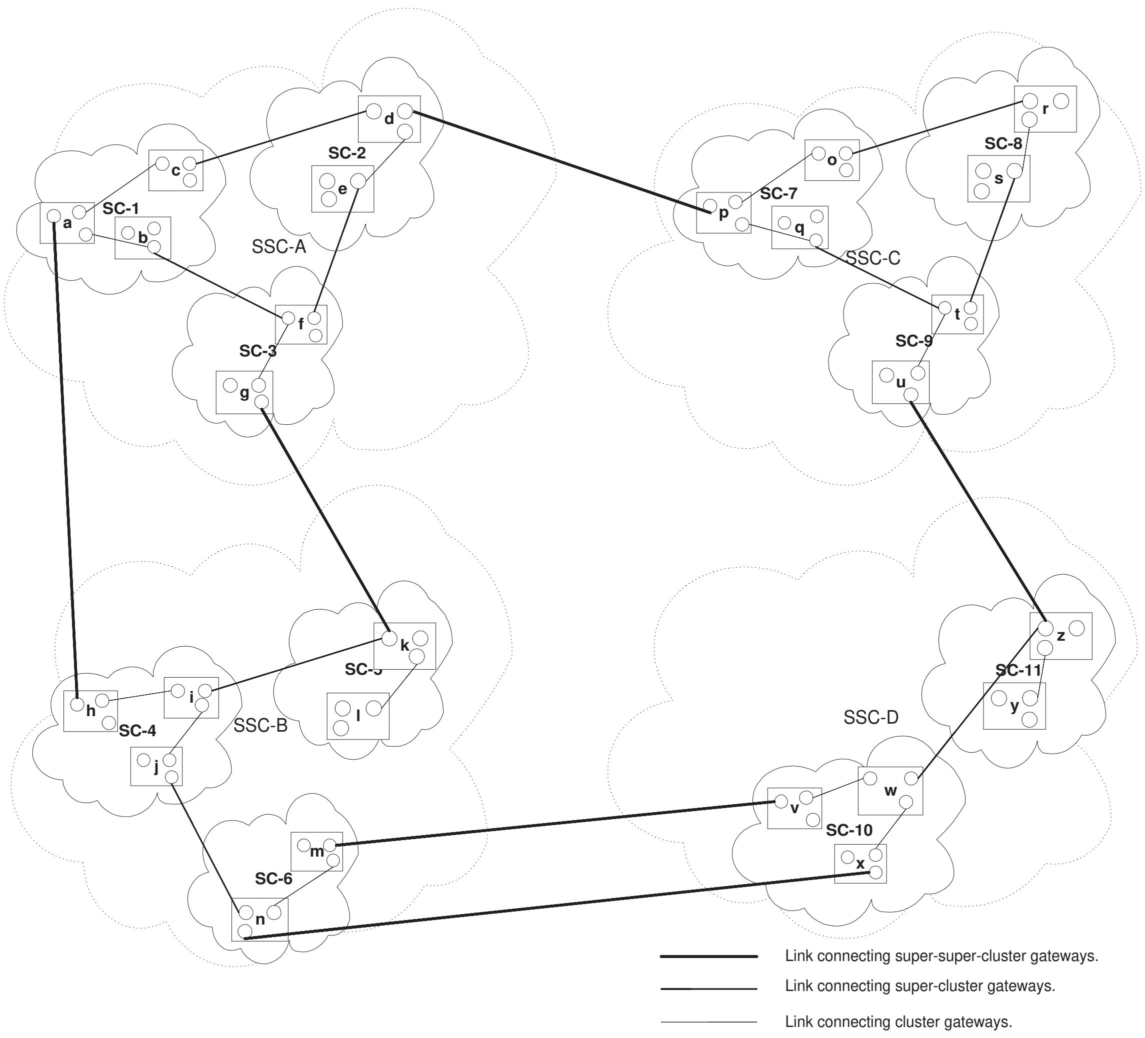

Figure 4: Gatekeepers and the organization of the system

It should be noted that a gatekeeper at level l need not be a gatekeeper at level $(\ell + 1)$ and vice-versa. Fig 4 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_i^2 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 $C_i^{\ell+1}$ but have differing contexts C_j^ℓ, C_k^ℓ, the nodes are designated gateways at $level - \ell$ i.e. $g^\ell(C^{\ell+1})$. Thus in Fig 4 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.

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 of events (discussed in Section 8.7). 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_j^ℓ is $C_i^\ell = \overbrace{C_j^N(C_k^{N-1}(\cdots(C_m^{\ell+1}(C_i^\ell))\cdots))}^{N-l}$. Thus in a 4-level system, to identify a server node, the addressing scheme specifies the super-super-cluster C_i^3, super-cluster C_j^2 and cluster C_k^1 that the node is a part of along with the node-identifier within C_k^1. 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 updation 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 8.4 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 8.5) and events to a newly reconnected client (section 8.7). 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 3.6 on page 6) 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 flooded by events that don't 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 exits 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 ever 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.

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

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

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

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

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_i^ℓ within the context $C_j^{\ell+1}$, all the level-$(\ell+1)$ gatekeepers, $g^{\ell+1}$, at context $C_j^{\ell+1}$ within $C_k^{\ell+2}$ are notified about this change. In an $N - level$ system this process of 'higher level' gatekeeper profile updation 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 updation 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_i^2 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*[3] what we mean is that for every event that is being routed within that unit the destination

[3]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

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 8.4) pertaining to contexts different from the node it is attached to, the highest such unit where this difference occurs forms part of the profiles active unit. Reception of events with differing contextual information also implies that the profile propagation has 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 2.4 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 it 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[4] (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.

8.2 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 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.

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

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[5] 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 $g_j^\ell(C_i^{\ell+1})$ the problem lies in the fact that 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 $0 < l < N$, gateway that exists within the cluster it belongs to.

Every gateway at level-l is aware of all the other $g^l(C_j^{l+1})$ 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 is futile to try and find a higher order gateway, and also when all the higher level units that could possibly be covered are covered. Of course it also must 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 8.4) and the information pertaining to gateways that's available at a node. If there's no such unit that needs to be reached the event routing would proceed with lower order disseminations. However if there's a unit that needs to be reached gateways would need 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 to across units within or across levels. 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.

[5]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

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.

8.4 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 the system which is then used to avoid routing the event to the same unit twice.

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 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[6] 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

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

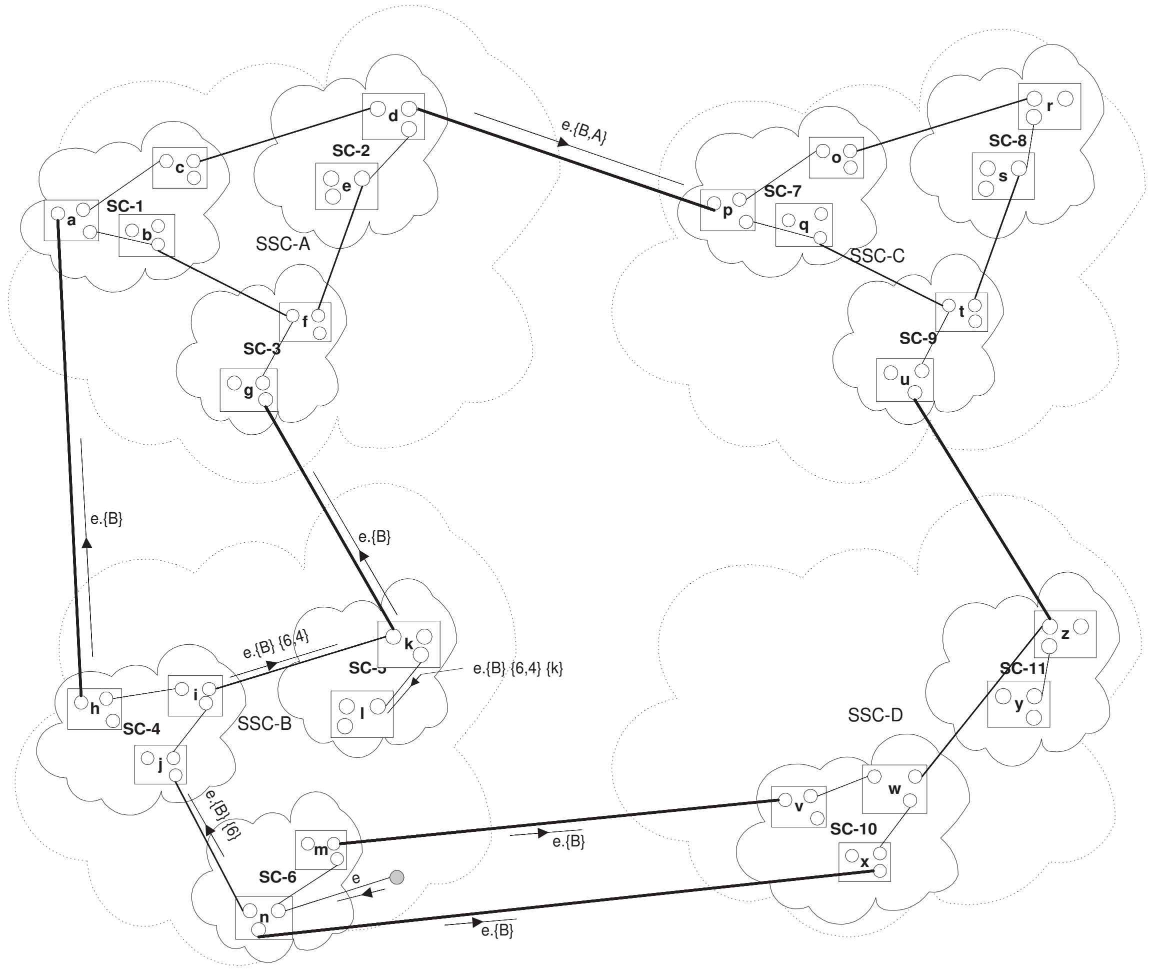

Figure 5: Routing events

more overloaded[7] compared to a lower order gateway. Reducing the amount of information being transferred over the gateway helps conserve bandwidth.

Fig 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 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.

[7]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 $level - \ell$.

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

8.5 Routing real-time events

Real time events can have destination lists (see section 3.4) 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 destination's for delivery.

8.5.1 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 $g_j^\ell(C_i^{\ell+1})$ checks the routing information provided by ERP to see if it needs to relay the event to other gatekeepers g^ℓ within the context $C_i^{\ell+1}$.
- 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 $C_i^{\ell+1}$ to relay this event.

8.5.2 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 8.5.1.

8.6 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 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. Consistency issues pertaining to out of order delivery 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 on 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 8.9

8.7 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 8.9. 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 8.5.2.

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.

8.8 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 3.2.2 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.

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 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.

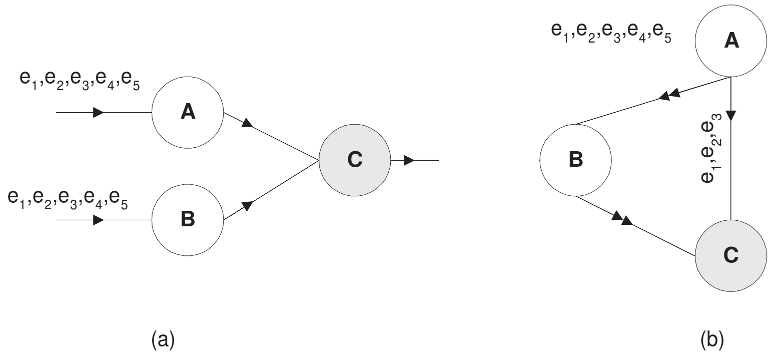

Figure 6: Duplicate detection of events

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.

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 1: Delivery of events at C

Consider the case in Fig 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 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 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 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.

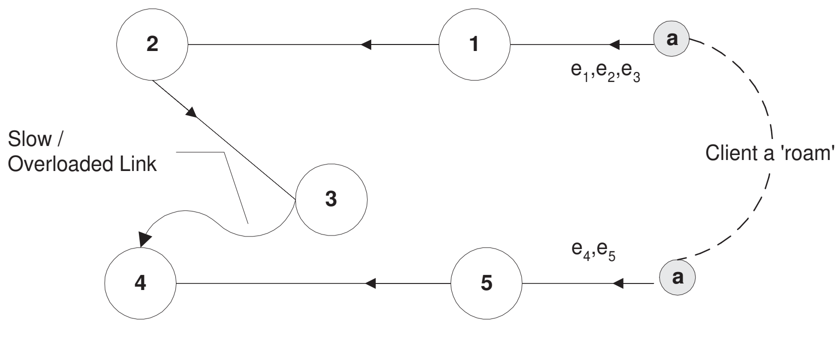

Figure 7: Duplicate detection of events during a client roam

Fig 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 2. 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 2: Delivery of events at 4 ... client roam

8.9 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 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 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.

9.1 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.

9.1.1 Message losses due to consecutive node failures

In Fig 8.(a) we have a situation where the two nodes ensure reliable delivery using a series of positive acknowledgements (ACKs). Node A won't 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 8.(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.

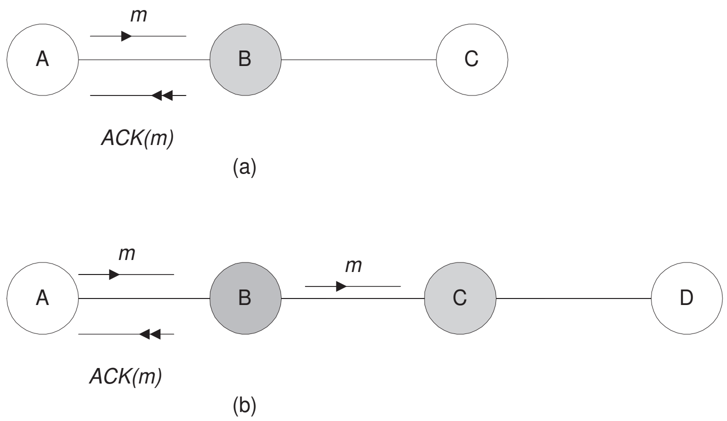

Figure 8: Message losses due to successive node failures

9.2 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 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.

9.3 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.

9.3.1 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 gateways 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.

9.3.2 Permanent gateway failures

This would call for updation 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.

9.4 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.

9.4.1 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 8.6 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.

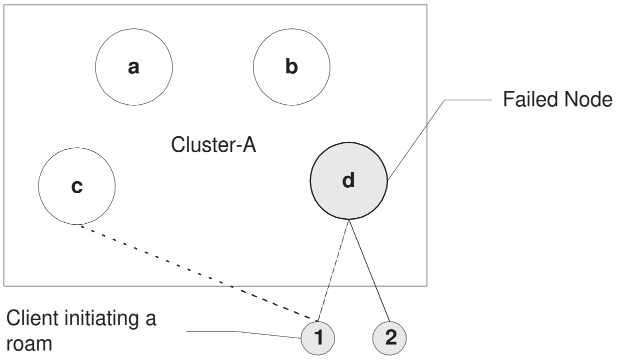

Figure 9: Client 'roam' in response to a node failure.

9.4.2 Unit doesn't remain failed forever

If a unit doesn't remain failed forever and the associated nodes recover and the gateways are re-instated, 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 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.

9.4.3 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 9.5.

9.5 Network Partitions

Network partitions can be caused both by link failures and node[8] 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.
- Ensure safe progress in concurrent partitions.
- Merge partitions while maintaining consistency.

9.5.1 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.

9.6 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 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 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

[8] 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

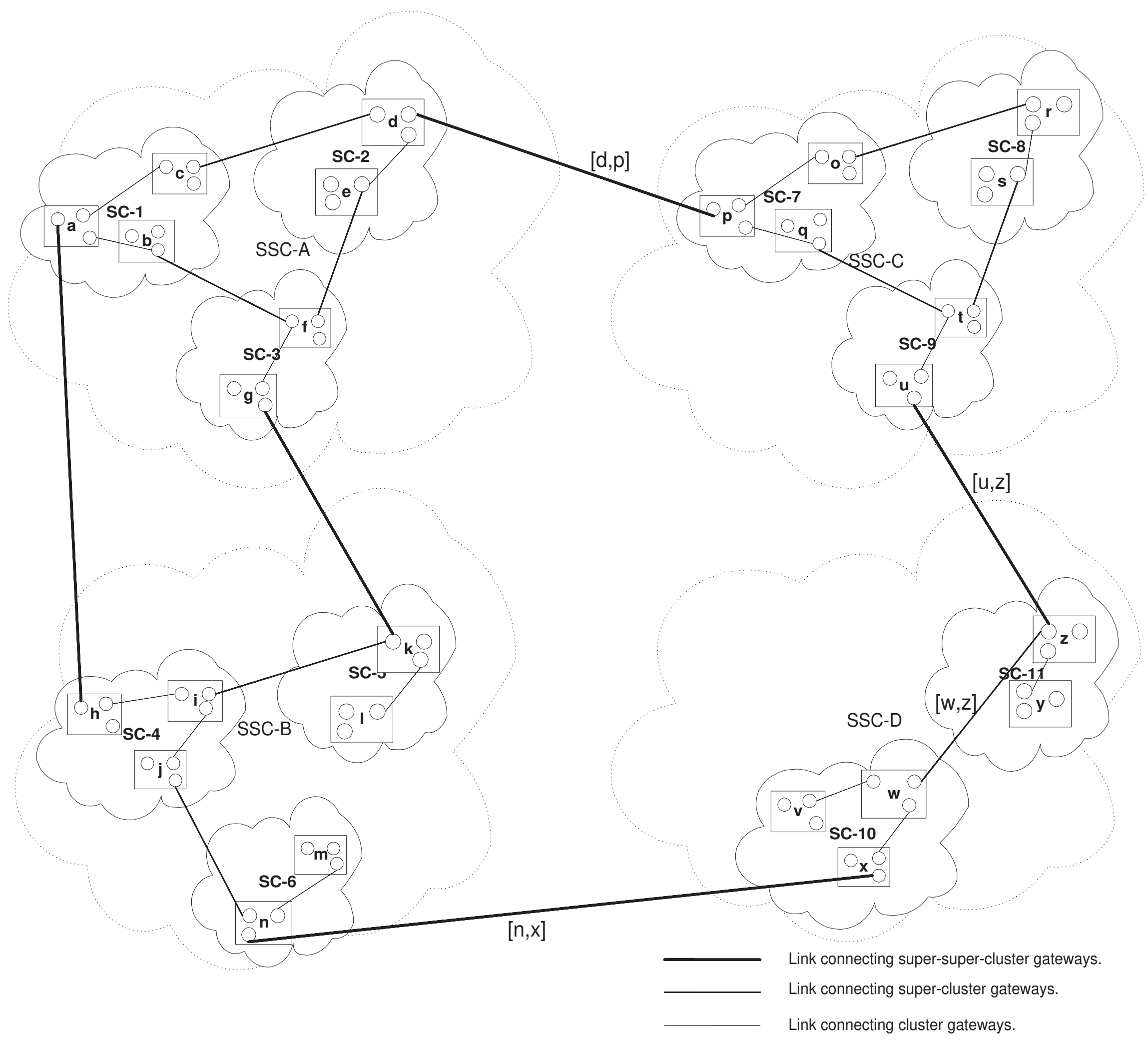

Figure 10: Partitioning issues related to node failures

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 9.1 *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.*

9.6.1 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[9] to the units which exist

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

within a super-unit. Thus in Fig 10 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.

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

9.6.3 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 system.

9.7 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.

9.7.1 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 don 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.

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 11 depicts the different replication strategies that can exist within a sub system. As can be seen super-super-cluster SCC-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 3 depicts the replication granularities available at different nodes within the sub system depicted in fig 11.

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 3: Replication granularity at different nodes within a sub system

Requirement 9.1 *There should be at least one stable storage present in the system.*

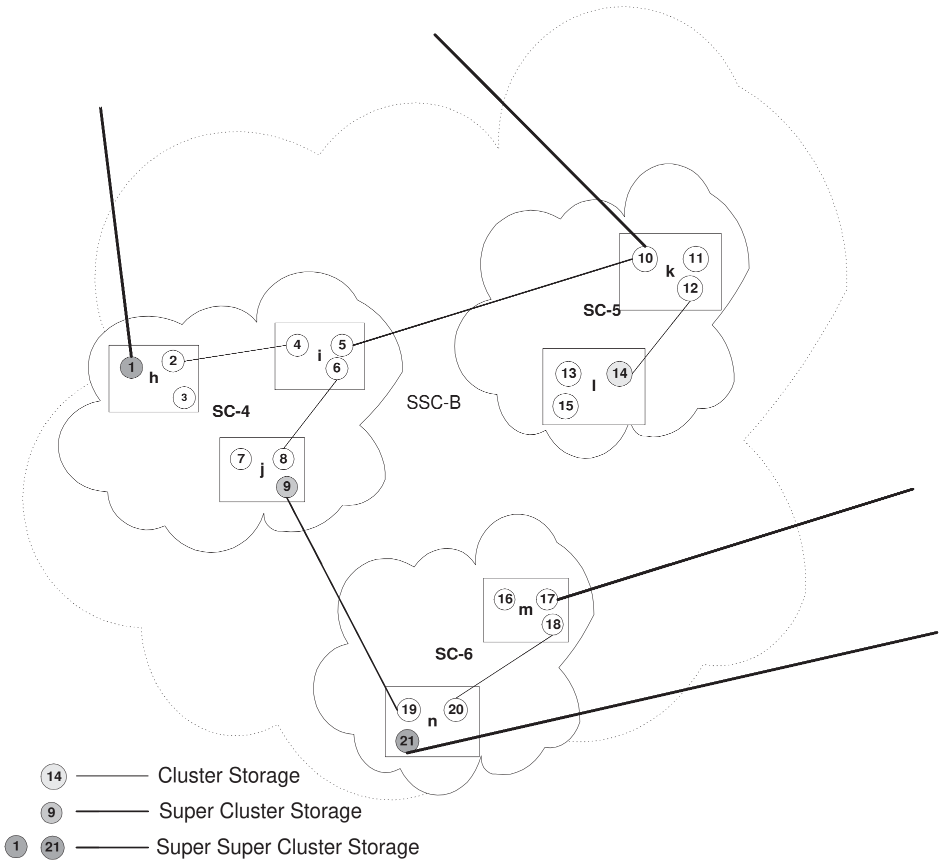

Figure 11: The replication scheme

9.7.2 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 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.

9.7.3 Delivering events without storage notifications

In section 9.1.1 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 don't 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 9.1 *The client reissue behavior ensures the storage of the event to at least one stable storage.*

Proof: Section 9.1.1 and Requirement 9.1 on page 32.

Lemma 9.2 *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 9.7.2.

Lemma 9.3 *If there is such a stable storage, for the scenario described in Lemma 9.2, then the system will locate this stable storage when the client missing this event issues a request for it.*

Proof: Conjecture 9.2 on page 35 and requirement 9.2 on page 34.

Theorem 9.1 *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.*

9.7.4 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.

9.7.5 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 9.2 *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.

9.7.6 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 9.2 *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 9.2 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 keep 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.

9.7.7 How to decide when to store and where to store

9.7.8 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 4 on page 14 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.

9.8 The need for Epochs

We digress here to discuss the need for *epochs*. The reference count scheme which we discussed in section 8.9 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.

We thus need some concept of epochs, without this queues would be constructed for clients which weren't originally interested in those events contained within the queue. Epochs provide us with a precise indication of the time from which point on a client should receive events. Epochs are used to aid the re-connected clients and also the recovery from failures. The reason why we can't 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.

9.8.1 Epoch generation

Epochs, denoted ξ, are truly determined by the replication granularity of the system within the context of which the client[10] 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[11] 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 coordinated between the multiple stable storages that could exist within a context.

(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.

[10] 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.

[11] This arise 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.

In section 9.8.2 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 9.8.3 we discuss the epoch generation scheme for events issued by clients attached to nodes within the sub system. Fig. 12 details the two scenarios for generation of epochs associated with events.

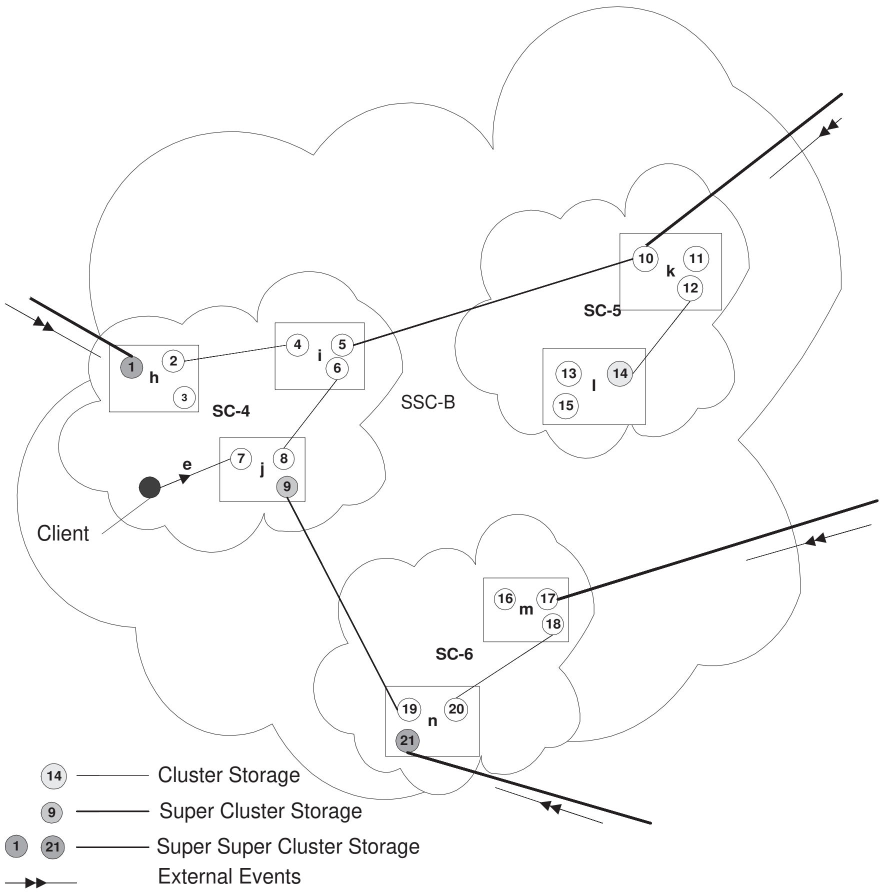

Figure 12: Generation of epochs associated with events

Axiom 9.1 *A client will not deliver an event e unless there is an epoch, ξ_e, associated with the event.*

9.8.2 Epoch generation for external events

In this section we are focussing 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 9.3 *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 9.3 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 8.8) 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 updation/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 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.

9.8.3 Epoch generation in units containing issuing clients

In section 9.1.1 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 9.8.2, 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.

9.8.4 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.

From conjecture 9.2 (a) provides us with the list of stable storages that has stored events for the client, while (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 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.

9.8.5 Epochs and profile changes

10 Implementation Details and such

Events conform to XML DTDs. Not all fields with 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[12] and cached at the server node. Thus we circumvent the network operation. 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 version Number 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, so that profiles could be updated to reflect this change.

```
<!ELEMENT EVENT (EVENT-ID, APPLICATION_TYPE*, SUMMARY*, NOTE?)>
<!ATTLIST EVENT
        versionNum      CDATA #REQUIRED
        securitylevel   (low | med | high) ‘‘low’’
        eventType       (normal|recovery)  ‘‘normal’’>

<!ELEMENT EVENT-ID (CLIENT-ID, TIME-STAMP, SEQUENCE-NUMBER, INCARNATION)>
<!ELEMENT CLIENT-ID         (#PCDATA)>
<!ELEMENT TIME-STAMP        (#PCDATA)>
<!ELEMENT SEQUENCE-NUMBER   (#PCDATA)>
<!ELEMENT INCARNATION       (#PCDATA)>

<!ELEMENT APPLICATION-TYPE  (#PCDATA)>
<!ELEMENT SUMMARY           (#PCDATA)>
<!ENTITY  Description    ‘‘This is the base event type’’>
```

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.

[12]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.