A Grid Event Service[1]

Shrideep Pallickara[2]

May 5, 2001

[1]**Version: 1.12** 2001/05/1 21:53:11

[2]Department of Electrical Engineering &Computer Science, Syracuse University

Abstract

The grid event service is a distributed event service designed to run on a very large network of server nodes. Clients interested in using this service can attach themselves to one of the server nodes. Clients specify an interest in the type of events that they are interested in and the service routes events, which satisfy the constraints specified by the clients. Clients can have prolonged disconnects from the server network and can also roam the network (in response to failure suspicions or for better response times) and attach themselves to any other node in the server node network. Events published during the intervening period, of prolonged disconnects and roams, must still be delivered to clients that originally had an interest in these events. The delivery constraints must be satisfied even in the presence of server failures.

The Grid event service provides for a hierarchical dissemination scheme for the delivery of events to relevant clients. The system provides for an efficient calculation of routes to reach relevant destinations. The grid event service also provides for merging streams of related events and delivering these merged streams to relevant clients. The events in these related streams could have spatial and chronological relationships to events within other streams. The GES provides for the resolution of these constraints and the delivery of these resolved event streams to interested clients.

Contents

List of Tables

List of Figures

Chapter 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 is used to describe the event uniquely and completely.
- Control information.
- Destination Lists (explicit or implicit via the topics that a client is interested in).

The attribute information comprises of tags which specify the attributes associated with the event type while the control information specifies the constraints associated with that event viz. ordering, stability. Thus say a person needs to sell stock A - the selling is the event, the general information is his account profile while the control information could be an indication that he wants guaranteed delivery of the event. Events trigger *actions*, through the state transitions induced in a delivering entity, which in turn can trigger events. The event and the associated actions taken by any part of the system share the *cause-effect* relationship. Actions are taken based on the event type and the information contained in the event. The action taken at any node could be influenced not only by different causes but by subsequent effects too. Events define objects, and also define changes in the state of objects. Events are time-stamped messages and also messages with a null timestamp. We think of all communication in the system as being events. The spectrum of relationships between events in traditional systems span from "unrelated" to where events are "related". These events were related through different ordering permutations based on the *local order* imposed by the issuee, *total order* imposed by a deterministic algorithm hosted on multiple nodes and a system determined *causal order*. Events form the basis of our design and are the most fundamental units that entities need to communicate with each other. These events encapsulate expressiveness at various levels of abstractions - content, dependencies and routing. Where, when and how these events reveal their expressive power is what constitutes information flow within our system. The events that we consider exist within event streams and can specify and dictate resolution of complex spatial and chronological dependencies with other events in the system. Related events can be considered to be part of unique abstract merged stream. Clients can express an interest in receiving a merged stream or *bundles* within a stream. It should be noted however that a bundle or the complete merged stream being delivered at a client can have multiple stream sources.

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

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

Distributed messaging systems broadly fall into the three different categories. These include queuing systems, remote procedure call based systems and publish subscribe systems. Message queuing systems with its store-and-forward mechanisms come into play where the sender of the message expects someone to handle the message while imposing asynchronous communication and guaranteed delivery constraints. The two popular products in this area include IBM's MQSeries [IBM00] and Microsofts MSMSQ [Hou98]. MQSeries operates over a host of platforms and covers a much wider gamut of transport protocols (TCP, NETBIOS, SNA among others) while MSMQ is optimized for the Windows platform and operates over TCP and IPX. A widely used standard in messaging is the Message Passing Interface Standard (MPI) [For94]. MPI is designed for high performance on both massively parallel machines and on workstation clusters. Messaging systems based on the classical remote procedure calls include CORBA [OMG00c], Java RMI [Jav99] and DCOM [EE98]. Publish subscribe systems form the third axis of messaging systems and allow for decoupled communications between clients issuing notifications and clients interested in these notifications.

The decoupling relaxes the constraint that publishers and subscribers be present at the same time, and also that the constraint that they be aware of each other. The publisher also is unaware of the number of subscribers that are interested in receiving a message. The publish subscribe model doesn't require synchronization between publishers and subscribers. By decoupling this relationship between publishers and consumers security is enhanced considerably. The routing of messages from the publisher to the subscriber is within the purview of the message oriented middleware (MOM) which is responsible for routing the right content to the right consumers. The publish subscribe paradigm can support both the *pull/push* paradigms. In the case of pull, the subscribers retrieve messages from the MOM by periodic polling. The push model allows for asynchronous operations where there are no periodic pollings. Industrial strength products in the publish subscribe domain include solutions like *TIB/Rendezvous* [TIB99] from TIBCO and *SmartSockets* [Cor00b] from Talarian. Variants of publish subscribe include systems based on content based publish subscribe. Content based systems allows subscribers to specify the kind of content they are interested in. These content based publish subscribe systems include *Gryphon* [BCM+99, ASS+99], *Elvin* [SA97] and *Sienna* [CRW00a]. The system we are looking at, the grid event service, is also in the realm of content based publish/subscribe systems with the additional feature of location transparency for clients.

The shift towards pub/sub systems and its advantages can be gauged by the fact that message queuing products like MQSeries have increased the publish subscribe features within them. This intersection of a mature messaging products with the pub/sub features serves its purpose for a large number of clients. Similarly OMG introduced services that are relevant to the publish subscribe paradigm. These include the Event services [OMG00b] and the Notification service [OMG00a]. The push by Java to include publish subscribe features into its messaging middleware include efforts like JMS [HBS99] and JINI [AOS+99]. One of the goals of JMS is to offer a unified API across publish subscribe implementations. Various JMS implementations include solutions like *SonicMQ* [Cor99] from Progress, *JMQ* [iPl00] from iPlanet, *iBus* [Inc00] from Softwired and *FiranoMQ* [Cor00a] from Firano.

In the systems we are studying, unlike traditional group multicast systems, "groups" cannot be preallocated. Each message is sent to the system as a whole and then delivered to a subset of recipients. The problem of reliable delivery and ordering[1] in traditional group based systems with process crashes has been extensively studied [HT94, BM89, Bir93a]. These approaches normally have employed the "primary partition" model [RSB93], which allows the system to partition under the assumption that there would be a unique partition which could make decisions on behalf of the system as a whole, without risk of contradictions arising in the other partitions and also during partition mergers. However the delivery requirements are met only within the primary partition [GRVB97]. Recipients that are slow or temporarily disconnected may be treated as if they had left the group. This model works well for problems such as propagating updates to replicated sites. This approach doesn't work well in situations where the client connectivity is intermittent, and where the clients can roam around the network. The main differences between the systems being discussed here and traditional group-based systems are:

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

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

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

In this thesis we propose the Grid Event Service (GES) where we have taken a system model that encompasses Internet/Grid messages. The grid event service is designed to include JMS as a special case. However, the topic and profile models provided within GES provides a far richer set of interactions and selectivity between clients than the JMS model. GES is not restricted to Java of course, this is our initial implementation. We envision a system with thousands of server nodes providing a distributed event service in a federated fashion. We also have designed and implemented an optimal routing scheme for the dissemination of events within the system. The architecture that we present in this thesis could also be used to provide an engine for *peer-to-peer* (P2P) [p2p01] interaction between the clients interacting with each other via the grid.

The remainder of this thesis is organized as follows. We begin by presenting a formal specification for the Grid Event Service and provide extensions to this problem by accounting for the presence of event streams in the system. We then provide a discussion on the design of events, a client's connection semantics and also on the server topology that we use to solve the problem. Chapter 4 describes our solution to the event delivery problem and provides detailed explanations of the various protocols that comprise the final solution. In chapter 5 we look at the problem of delivering merged streams and the resolution of spatial and chronological dependencies prior to the delivery of merged streams. We then proceed to describe the approach for guaranteed delivery of events and the detection of network partitions. Chapter 7 outlines a few application domains that the Grid Event Service could be applied to. Finally we include a discussion of results for various scenarios and future directions and conclusions.

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

Chapter 2

Specifications

In this chapter we specify the event service problem. In section 2.2 we present our model of the system in which we intend to solve the problem. In section 2.3 we formally specify our problem. Sections 2.4 and 2.5 deal with the assumptions that we make in our formalism's and the properties that the system and it components must conform to during execution. Section 2.6 provides an introduction to event streams, and how events in a stream can depend on and be related to event in other streams. Section 2.7 formalizes the representation of streams and also the dependencies that exist between events from multiple streams.

2.1 Events

An event is the most fundamental unit that entities need to communicate with each other. An event comprises of a set of properties and have one source and one or more destinations where it would be routed to. The properties could be boolean, or could take specific values within the range specified by the property. The sub vector of this set of properties is what constitutes the type of the event. Events allow separate entities to probe different set of properties, through accessor functions. Any given event is *fixed* except for the added data to reflect its use and routing within the system. This information contained in an event can cause or record mutation of properties of objects within the system. If the information contained in an event needs to be changed a new event would be generated.

Events also possess a set of destinations that comprise the clients which are targeted by the event. This destination list could be explicitly contained within the event itself, or could be computed dynamically as a function of properties list contained within the event. Events induce state transitions in the entities that receive the event. The state transition is followed by a set of actions. The event and the associated actions taken by any part of the system share the *cause-effect* relationship. These induced state transitions and associated actions are based on the values the properties can take.

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

2.2 System Model

The system comprises of a finite (possibly unbounded) set of *server nodes*, which are strongly connected (via some inter-connection network) and special nodes called *client nodes* which can be attached to any of the server nodes in the network. Client nodes can never be attached to each other, thus they never communicate directly with each other. Let **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 $receive(e)$.

Server nodes are responsible for routing/queuing events to the destination lists contained within the event. Each server node instantiates a *service* which is responsible for interacting with service instances on other server nodes to facilitate the calculation of destination lists for the events and the routing/queuing of these events to the relevant clients. 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. As a result of these failures the communication network may *partition.* Similarly *virtual* partitions may stem from an inability to distinguish slow nodes or links from failed ones. Crashed nodes may rejoin the system after recovery and partitions (real and virtual) may heal after repairs.

2.3 The event service problem

Client nodes can issue and receive events. Any arbitrary event e contains implicit or explicit information regarding the client nodes which should receive 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.

Associated with every client node is a *profile* which specifies the type of events the client node is interested in receiving. For events issued by any arbitrary client node, the system is responsible for calculating all the valid destinations associated with the event. This destination list is computed on the basis of the profiles for each and every client node in the system. Considering the volume of events that would be present in the system, it should be ensured that the only events that are routed to a client node are those that it has expressed an interest in. In the event that a client node *roams* and attaches itself to any other server node in the system, the service instances on the server nodes in responsible for relaying/queuing events to the new location of the client node.

For an execution σ of the system, we denote by E_σ the set of all events that were issued by the client nodes. Let $E^i_\sigma \subseteq E_\sigma$ be the set of events e^i_σ that should be relayed by the network and received by client node c_i in the execution σ. During an execution trace σ client node c_i can *join* and *leave* the system.

Node c_i could *recover* from *failures* which were listed in Section 2.2. Besides this, as mentioned earlier client nodes can roam (a combination of leave from an existing location and join at another location) over the network. A combination of join-leave, join-crash, recover-leave and recover-crash constitutes an *incarnation* of c_i within execution trace σ. We refer to these different incarnations, $x \in X = 1, 2, 3...$, of c_i in execution trace σ as $c_i(x, \sigma)$.

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

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

All received events $e^i_\sigma \in E^i_\sigma$ must of course satisfy the causal constraints that exist between them prior to reception at the client node.

2.4 Assumptions

(a) Every event e is unique.

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

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

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

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

Items (d) and (e) constitute the liveness property eliminating trivial implementations in which an event is always lost or all events have zero targeted clients.

2.5 Properties

(a) A client node can receive an event e, only if e was previously issued.

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

(c) If an event e is to be received by client nodes $c, c' \in L_e$, then if c receives e then c' will receive event e.

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

Property (d) pertains 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 $e' \rightarrow e''$.

2.6 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.

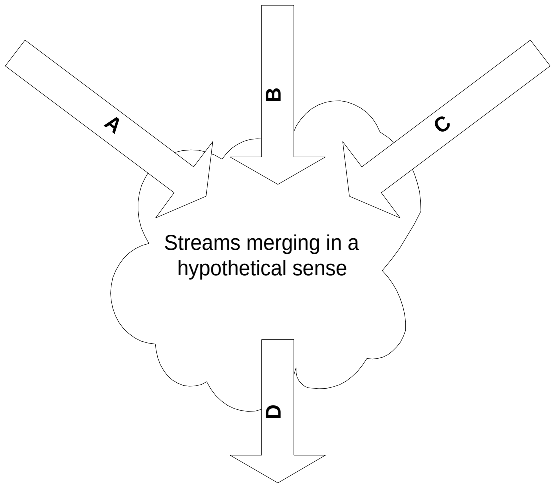


Figure 2.6.1: Existence of multiple event streams.

Events within an event stream could depend[1] 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 2.6.1. This information could point 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 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

[1]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.

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

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

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

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

2.7 Event Stream Specifications

In this section we formally specify the streams, and the dependencies that exist between the events in one stream to the events within other streams. The dependencies are specified by the stream interaction rules within the event streams and controlled by the occurrence vector which dictates the number of events from a specific stream that an event can have a dependency on. We also formulate the resolution of these dependencies and how this subsequently leads to the creation of merged event streams. The event streaming problem is one of routing these merged event streams to clients. To help clarify some of the situations that we are trying to formulate we will refer to the simple example depicted in figure 2.7.1. The scenario is one where an on-line interactive lecture is in progress. The lecture comprises of foil streams of individual foils, mouse streams of mouse events instantiated by the lecturer on different foils and request/response stream where queries are posed by the students and responses posted by the lecturer.

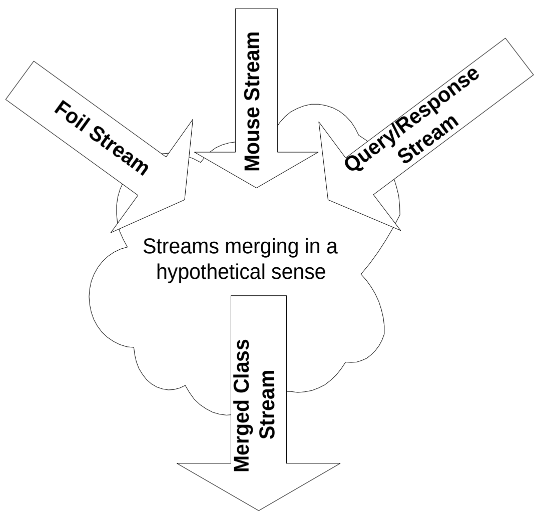


Figure 2.7.1: Merged Streams - Example Scenario

In the foil stream each foil is spatially related to the other foils. Each foil has a specific place in the sequence of foils comprising the foil stream. Mouse streams on the other hand have an additional dependency. Mouse events besides occurring in the sequence that they occurred in, must also maintain the timing delays between any two successive mouse events. Equation (2.7.1) specifies the relationships that exist within the events of an event stream. These relationships exist within the context of space and time. In the spatial domain the events within an event stream could be *precedence related* ($\rightsquigarrow$) 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. In addition to the logical or precedence relationship existing between events within an event stream, events could be constrained by time's arrow. This arrow is a relative notion of time and always points in the same direction.

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

$$E = \overbrace{\{e_0 \overset{\mathrm{t}}{\leadsto} e_1 \overset{\mathrm{t}}{\leadsto} \cdots\}}^{\text{Ordered Set}} \mid \overbrace{\{e_0 \overset{\mathrm{t}}{;} e_1 \overset{\mathrm{t}}{;} e_2 \cdots\}}^{\text{Unordered Set}} \tag{2.7.1}$$

In equation (2.7.2), $\hookrightarrow$ 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 $E^j \in \Pi$. Thus in our example, the stream that the clients specify an interest in is the class stream, and the stream that is routed to the clients is the merged stream comprising of foils, mouse event and queries/responses with the dependencies resolved.

$$E \hookrightarrow \Pi = \{E^1, E^2, E^3, \cdots, E^N\} \tag{2.7.2}$$

The dependency relation $\hookrightarrow$ is the product of the spatial dependency relation $\overset{\mathrm{s}}{\hookrightarrow}$ and the associated chronological dependency $\overset{\mathrm{t}}{\hookrightarrow}$ that exist within the events in streams. Even though there may be no timing constraints imposed on successive events, they are still time constrained, in that they would be released only after $\overset{\mathrm{s}}{\hookrightarrow}$ is resolved. In the example scenario two successive foil stream events f_i,f_{i+1} would still be time constrained since f_i needs to be received before f_{i+1} can be received. The passage of time in the direction of time's arrow is marked by a succession of significant events which have been $\overset{\mathrm{s}}{\hookrightarrow}$ and $\overset{\mathrm{t}}{\hookrightarrow}$ resolved.

$$\hookrightarrow \; = \; \overset{\mathrm{s}}{\hookrightarrow} \times \overset{\mathrm{t}}{\hookrightarrow} \tag{2.7.3}$$

The occurrence vector $\mathcal{O}$ is used to determine the number of events within other individual streams in Π that an event e in E is interested in. In equation (2.7.4) we define the values which elements in the occurrence vector can take. This value specified could be one of ? (once or not at all), + (at least once), $*$(zero or more) and $\star$ (one and only one). In our example for every foil there could be zero or more queries that could be posted.

$$\text{Occurence Vector } \mathcal{O} = \{?, +, *, \star\} \tag{2.7.4}$$

Events within an event stream could have a simple mapping which snapshots their dependencies on events within other streams. This mapping $\leftrightarrow$ could be a simple one to one mapping, or a pre defined mapping which is consistent for all events within an event stream. Equation (2.7.5) is one of the forms that *stream interaction rules* could take. The $\overset{\mathrm{s}}{\hookrightarrow}$ specifies the spatial dependency that exist between events in streams.

$$E \leftrightarrow E^j \Rightarrow E.e_i \overset{\mathrm{s}}{\hookrightarrow} E_j.e_i^j \mid E.e_i \overset{\mathrm{s}}{\hookrightarrow} E^j.e_{i\pm N}^j \text{ where } \leftrightarrow \text{ specifies the mapping rule} \tag{2.7.5}$$

Equation (2.7.6) specifies one of the more complex forms that stream interaction rules can take. The function e^{func} could specify either a *constraint* or a more complex *rule* which needs to be satisfied by the events within other event streams. The equation 2.7.6 snapshots the second half of the stream interaction rules that could exist between different streams and which is used as the basis for the resolution of dependencies that exist within streams.

$$E^j(e^{func}) = \sum e^j \in E^j \ni e^j \text{ satisfies } e_i^{func} \tag{2.7.6}$$

Equation (2.7.7) specifies the resolution of an events dependency in the spatial domain. A specific event within an event stream E has a dependency to events within streams in Π or a subset of the streams contained in Π, denoted Π'. The # operator is the cardinality of a set. The operator $\odot$ is the *refinement* of the stream interaction rules with an element of the occurrence vector $\mathcal{O}$. This refinement pin points the precise event/events in $E^j \in \Pi$ that an event in E is dependent on. As is clear, the result of this dependency resolution is either a Null (if $e_i \hookrightarrow \Pi'$ and $\#\Pi' = 0$) or either an event or an array of events as determined by $\#\Pi'$ and the occurrence vector. The array of events could comprises of zero or single or multiple events from each of the event streams in Π.

$$\begin{aligned} \forall e_i \in E, e_i \overset{\mathrm{S}}{\hookrightarrow} \Pi' \subseteq \Pi \quad &\equiv \quad \overbrace{e_i(data)}^{\text{Implied}} \cup \sum_{j=1}^{\#\Pi'} \overbrace{\{E \leftrightarrow E^j \mid E^j(e_i^{rule}) \mid E^j(e_i^{tags})\}}^{\text{Stream Interaction Rules}} \odot \overbrace{o_i \in \mathcal{O}}^{\text{Occurrance}} && (2.7.7) \\ &\equiv \quad Null \mid e \mid e[\,] && (2.7.8) \end{aligned}$$

In addition to this, the dependency specification also includes timing constraints on the delivery of dependent events. This timing constraint specifies the time after the delivery of an event, that the dependent events should be delivered. This timing constraint between events in E and Π, is in addition to the timing constraints that exist between the events of a stream. Equation (2.7.9) follows from equation (2.7.3) where the product of the spatial resolution and the imposed chronological dependency between events of related streams,specifies the complete dependency resolution.

$$\forall e_i \in E, e_i \hookrightarrow \Pi' \subseteq \Pi \equiv \left(e_i \overset{\mathrm{S}}{\hookrightarrow} \Pi' \subseteq \Pi\right) \times \overbrace{0 \mid t_i \mid t_i[\,]}^{\text{Timing Constraints}} \quad . \qquad (2.7.9)$$

Equation (2.7.10) details the creation of a merged event stream after the resolution of dependencies within Π of every event e_i within an event stream E as specified by the event dependency resolution in equation (2.7.9). The event dependency resolution of every event within E results in the creation of the merged event stream.

$$\sum_{i=0}^{\#E} e_i \hookrightarrow \Pi' \subseteq \Pi = E^{MergedStream} \qquad (2.7.10)$$

2.8 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 receive e_j before e_i. Also clients cannot receive 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 receives e^j before e_i.

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

2.9 Summary

In this chapter we presented a formal specification of the event service problem, and how for a given client in an execution trace spanning multiple incarnations every event that was meant to be received at a client should be received. We also presented a formal representation of a merged stream that would be composed from multiple streams. We formalized a notation for describing the dependencies that events in one stream can have to events in other streams. Finally we outlined the properties that need to be satisfied by the solutions.

Chapter 3

Events, Clients and the Server Topology

In this chapter, we present the anatomy of an event based on our discussions in Chapter 2. We proceed to outline the connection semantics for a client, and also present our rationale for a distributed model in implementing the solution. We then present our scheme for the organization of the server network, and the nomenclature that we would be referring to in the remainder of this thesis.

3.1 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 is used to describe the event uniquely and completely.
- Control information.
- Destination Lists (explicit or implicit via the topics that a client is interested in).

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.

3.1.1 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’’
```

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.

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.

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.

3.1.2 Attribute Information

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

Tagged Information & the event type

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

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 clients 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

[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.

$sequenceNumber.MAX$ 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.

Systems such as *Gnutella* [gnu00] propagate events through the network without duplication, using the IETF UUID [LS98] which gives a unique 128-bit identifier on demand. The authors guarantee the uniqueness until 3040 A.D. for the ID's generated using their algorithm. Such a scheme of unique ID's could also be very conveniently incorporated into the Grid Event Service for a unique identifier for every event.

3.1.3 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.

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.

Correlation Identifiers

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

Qualities of Service Specifiers

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

3.1.4 Destination Lists

Clients in the system specify an interest in the type of events that are interested in receiving. This interest could be in certain sports events or events sent to a certain discussion group. It is the system which computes the clients that should receive a certain event. A particular event may thus 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.

3.1.5 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 3.1.2. Derived events can be processed as base events and not vice versa.

3.1.6 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 $f(t).attributes \supset g(t).attributes \bigcap f(t).attributes \subset g(t).attributes$

3.1.7 Specifying the anatomy of an event

These sets of equations follow from our discussions in section 3.1 and section 2.7. Equation (3.1.1) follows from our discussions in section 3.1.2 regarding the generation of unique identifiers. This tuple is created by the issuing clients.

$$eventId =< clientId, timeStamp, seqNumber, incarnation > \tag{3.1.1}$$

The tuple in 3.1.2 discriminates between live events and recovery events (which occur due to failures or prolong disconnects).

$$liveness =< live|recovery > \tag{3.1.2}$$

The type of an event is dictated by the event *signature*. These signatures could change, to accommodate these changes we include the concept of versioning in our event signatures. This along with *liveness* (equation 3.1.2) describe the event type completely.

$$eventType =< signature, versionNum, liveness > \tag{3.1.3}$$

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

$$destinationLists =< \overbrace{Implied}^{\text{External}} \mid \overbrace{Explicit}^{\text{Internal}} > \tag{3.1.4}$$

The dependency indicator follows from our discussions in section 2.7 and equations (2.7.4) through (2.7.9).

$$spatialDependency =<? \mid * \mid + \mid \star > \odot < mapping \mid rules \mid constraints > \tag{3.1.5}$$

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

$$\begin{aligned} event \quad = \quad & < eventId, eventType, attributesList, spatialDependency, timingDependency \\ & stream, applicationType, destinationLists > \end{aligned} \tag{3.1.6}$$

3.2 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.

3.2.1 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.

3.2.2 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.

3.2.3 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.

3.3 Client

The system is the sum of clients. Clients can generate and consume events in the system. The three issues which describe a client are

- Connection Semantics
- Client Profile
- Logical Addressing

3.3.1 Connection Semantics

Events in the system 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.

3.3.2 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 3.3.3) in that incarnation.

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

3.3.3 Logical Addressing

Given its connection semantics (Section 3.3.1), 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.

3.4 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 form 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.

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. 3.4.1. Clusters within a super-cluster have one or more links with at least one of the other clusters within that super-cluster. When we refer to the links between two clusters, we are referring to the links connecting the nodes in those individual clusters. Referring to Figure 3.4.1 Cluster-A has links to Clusters B, C and D while Cluster-B has links to Clusters A and C. For two clusters with at least one link between them, any node in either of the clusters can communicate with any other node of the other cluster. In general there would be multiple links connecting a single cluster to several other clusters. This approach provides us with a greater degree of fault-tolerance, by providing us with multiple *routes* to reach nodes within other clusters.

This topology could be extended in a similar fashion to comprise of *super-super-clusters* (level-3) as shown in Fig. 3.4.2, *super-super-super-clusters* (level-4) and so on. A Client thus connects to a server node, which is part of a cluster, which in turn is part of a super-cluster and so on and so forth. We limit the number of super-clusters within a super-super-cluster, the number of clusters within a super cluster and the number of nodes within a cluster viz. the *block-limit* to 64. In a N-level system this scheme allows for $2^6_{N-1} \times 2^6_{N-2} \times \cdots 2^6_0$ i.e 2^{6*N} server nodes to be present in the system.

What we essentially have here is a set of strongly connected server nodes comprising a cluster and a set of links connecting a cluster to other clusters. We are interested in the delays that would be involved

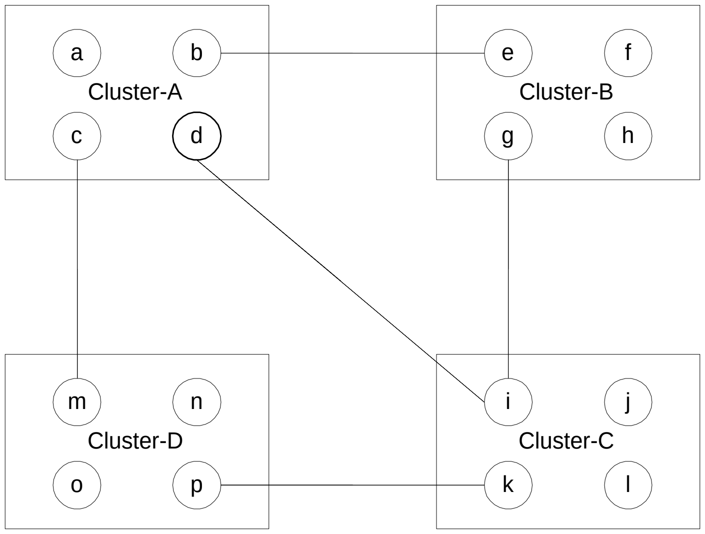


Figure 3.4.1: A Super Cluster - Cluster Connections

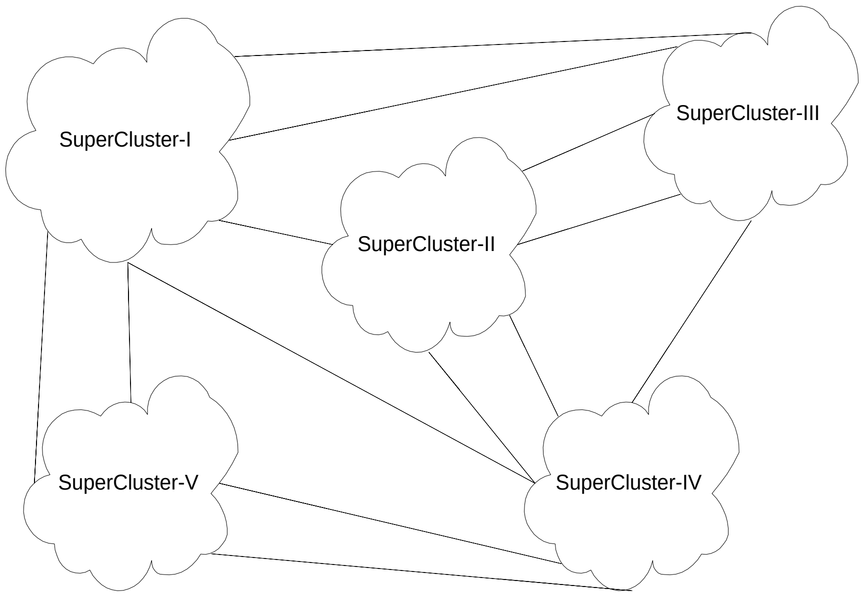


Figure 3.4.2: A Super-Super-Cluster - Super Cluster Connections

in connecting from one node in the network that we have to another server node in the network. This is proportional to the server node hops that need to betaken en route to the final destination.

We now delve into the *small world graphs* introduced in [WS00] and employed for the analysis of real world peer-to-peer systems in [Ora01, pages 207 – 241]. In a graph comprising of several nodes, *pathlength* signifies the average number of hops that need to be taken to reach from one node to the other. *Clustering coefficient* is the ratio of the number of connections that exist between neighbors of node and the number of connections that are actually possible between these nodes. For a regular graph comprising on n nodes each of which is connected to its nearest k neighbors – for cases where n is much larger than k, which in turn is much larger than 1 the pathlength is approximately $n/2k$. As the number of vertices increases to a large value the clustering coefficient in this case approaches a constant value of 0.75.

At the other end of the spectrum of graphs is the *random graph*, which is the opposite of a regular graph. In the random graph case the pathlength is approximately $\log n / \log k$, with a clustering coefficient of k/n. The authors in [WS00] explore graphs where the clustering coefficient is high, and with *long connections* (inter-cluster links in our case). They go on to describe how these graphs have pathlengths approaching that of the random graph graph, though the clustering coefficient looks essentially like a regular graph. The authors refer to such graphs as *small world graphs*. Following this result the conjecture we arrive at is that the for our server node network, the pathlengths will be logarithmic too. Thus in the topology that we have the cluster controllers provide control to local classrooms etc, while the links provide us with *logarithmic* pathlengths and the multiple links, connecting clusters and the nodes within the clusters, provide us with robustness.

3.4.1 GES Contexts

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

$$\begin{aligned} C_i^0 &= (C_j^1, i) && (3.4.1)\\ C_j^1 &= (C_k^2, j) && (3.4.2)\\ &\vdots \\ C_p^{N-2} &= (C^{N-1}, p) && (3.4.3)\\ C_q^{N-1} &= q && (3.4.4) \end{aligned}$$

In an N-level system, a unit at level ℓ can be uniquely identified by $(N-\ell)$ GES context identifiers of each of the higher levels. Of course, the units at any level l within a GES 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*.

3.4.2 Gatekeepers

Within the GES 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 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 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 GES context C_i^3. In a $N - level$ system similar such gateways would exist at every level within a higher GES context. A gateway at level ℓ within a higher GES context $C_j^{\ell+1}$ denoted $g_i^{\ell}(C_j^{\ell+1})$ comprises of –

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

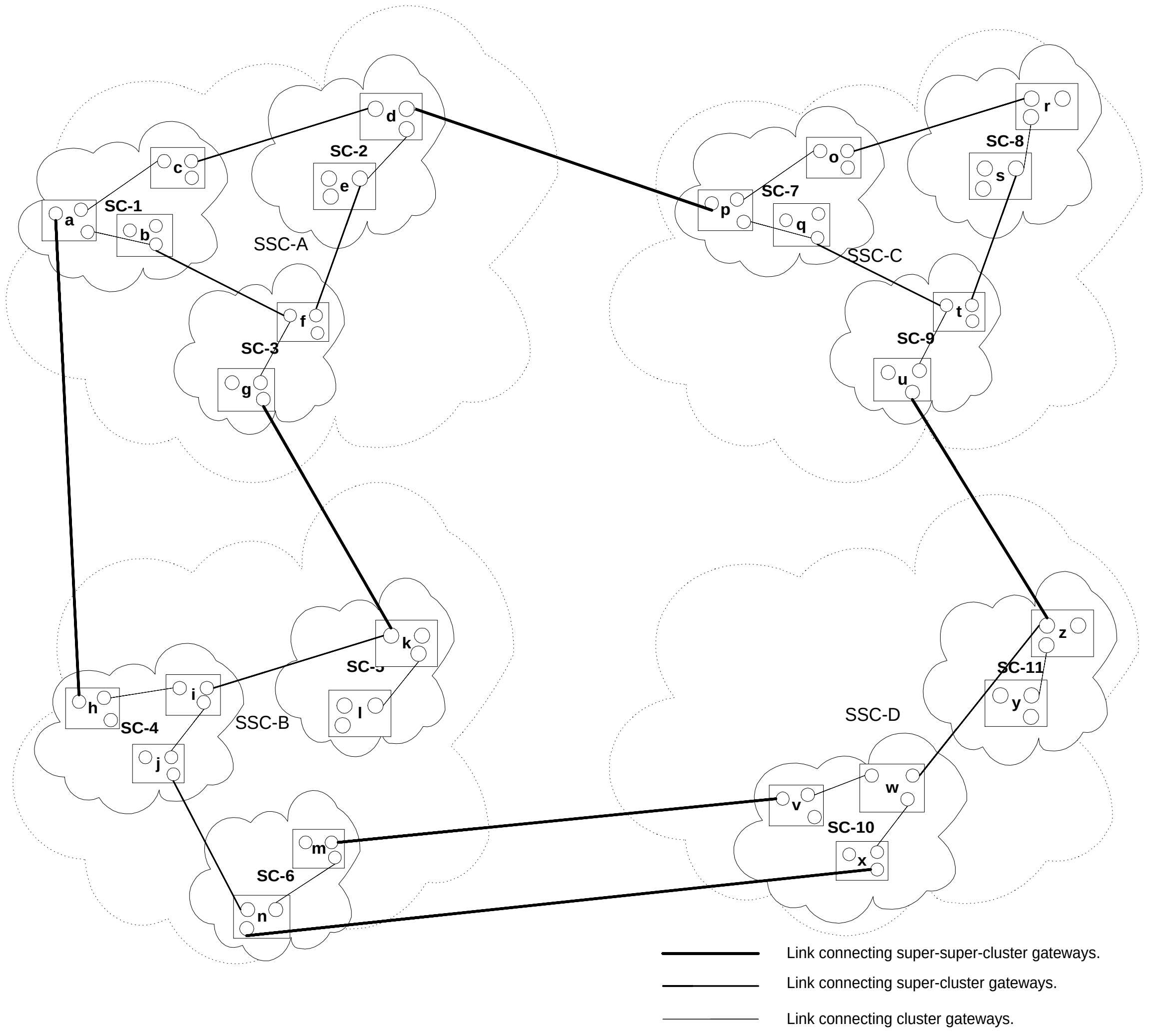

a
b
c
d
e
f
g
SC-1
SC-2
SC-3
SSC-A
o
p
q
r
s
t
u
SC-7
SC-8
SC-9
SSC-C
h
i
j
k
l
m
n
SC-4
SC-5
SC-6
SSC-B
v
w
x
y
z
SC-10
SC-11
SSC-D
Link connecting super-super-cluster gateways.
Link connecting super-cluster gateways.
Link connecting cluster gateways.

Figure 3.4.3: 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 3.4.3 shows a system of 78 nodes organized into a system of 4 super-super-clusters, 11 super-clusters and 26 clusters. When a node establishes a link to another node in some other cluster, it provides a gateway for dissemination of events. If the node it connects to is in a different cluster within the same super-cluster GES context C_i^2 both the nodes are designated as cluster gateways. In general if a node connects to another node, and the nodes are such that they share the same GES context $C_i^{\ell+1}$ but have differing GES 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 3.4.3 we have 12 super-super-cluster gateways, 8 super-cluster gateways (6 each in SSC-A and SSC-C, 4 in SSC-B and 2 in SSC-D) and 4 cluster-gateways in super-cluster SC-1.

3.4.3 The addressing scheme

The addressing scheme provides us with a way to uniquely identify each server node within the system. This scheme plays a crucial role in the delivery and dissemination of events to nodes in the system(discussed in Section 6.2.6). As discussed earlier units at each level are defined within the GES context of a unit at the next higher level. In a N-level system the GES 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.

3.5 Summary

In this chapter we dicussed the design of an event, based on the discussions in chapter 2. We also discussed the rationale for a distributed network of servers, with issues such as scaling, resiliency to failures and load balancing being the most import factors influencing the choice of the distributed model. The chapter also discussed the client connection semantics, which include prolonged disconnects and roam, and the parameters that a client needs to keep track of in its various incarnations within the system. Finally we established a topology for our server nodes, which would be used in building the event service. We also defined the notion of gatekeepers, GES contexts and logical addressing within the system and the nomenclature that would be referred to in the remainder of the thesis.

Chapter 4

The problem of event delivery

The problem of event delivery pertains to the efficient 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 efficiently route the events from the issuing client to the interested clients. A simple approach would be to route all events to all clients, and have the clients discard the content that they are not interested in. This approach would however place a strain on network resources. Under conditions of high load and increasing selectivity by the clients, the number of events a client discards would far exceed the number of events it is actually interested in. This scheme also affects the latency associated with the reception of real time events at the client. The system thus needs to be very selective of the kinds of events that it routes to a client. In this chapter we describe a suite of protocols that are used to aid the process of efficient dissemination of events in the system.

In section 4.1 we describe the Node Addition Protocol (NAP), which provides for adding a server node or a complete unit to an existing system. The Gateway Propagation Protocol (GPP) discussed in Section 4.2 is responsible for the dissemination of connection information within relevant parts of the sub system to facilitate creation of abbreviated system interconnection graphs. Providing precise information for routing of events, and the updating of this information in response to the addition, recovery and failure of gateways is in the purview of the (GPP). To snapshot the event constraints that need to be satisfied by an event prior to dissemination within a unit and subsequent reception at a client we use the Profile Propagation Protocol (PPP) discussed in Section 4.3.5. PPP is responsible for the propagation of profile information to relevant nodes within the system to facilitate hierarchical dissemination of events. Section 4.4 describes the Event Routing Protocol (ERP) which uses the information provided by PPP to compute hierarchical destinations. Information provided by GPP, such as system inter-connections and shortest paths, are then employed to efficiently disseminate events within the units and to clients subsequently. The problem of routing events is a two pronged problem, which needs to address the basic routing scheme and the routing of real-time events (section 4.5). 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 GES context.

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

Different systems address the problem of event delivery to relevant clients in different ways. In [GS95] each subscription is converted into a deterministic finite state automaton. This conversion and the matching solutions nevertheless can lead to an explosion in the number of states. In [SA97] network traffic reduction is accomplished through the use of *quench* expressions. Quenching prevents clients from sending notification for which there are no consumers. Approaches to content based routing in *Elvin* are discussed in [SAB+00]. In [CRW00a, CRW00b] optimization strategies include assembling patterns of notifications as close as possible to the publishers, while multicasting notifications as close as possible to the subscribers. In [BCM+99] each server (broker) maintains a list of all subscriptions within the system in a parallel search tree (PST). The PST is annotated with a trit vector encoding link routing information. These annotations are then used at matching time by a server to determine which of its neighbors should receive that event. The approach adopted by the OMG [OMG00c] is one of establishing channels and registering suppliers and consumers to those event channels. [AAB+00] describes approaches for exploiting group based multicast for event delivery. These approaches exploit universally available multicast techniques. The channel approach in the event service [OMG00b] approach could entail clients (consumers) to be aware of a large number of event channels. The two serious limitations of event channels are the lack of event filtering capability and the inability to configure support for different qualities of service. These are sought to be addressed in the Notification Service [OMG00a] design. However the Notification service attempts to preserve all the semantics specified in the OMG event service, allowing for interoperability between Event service clients and Notification service clients. Thus even in this case a client needs to subscribe to more than one event channel. In *TAO* [HLS97] a real-time event service that extends the CORBA event service is available. This provides for rate-based event processing, and efficient filtering and correlation. However even in this case the drawback is the number of channels that a client needs to keep track of.

4.1 The node organization protocol

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

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

4.1.1 Adding a new node to the system

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

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

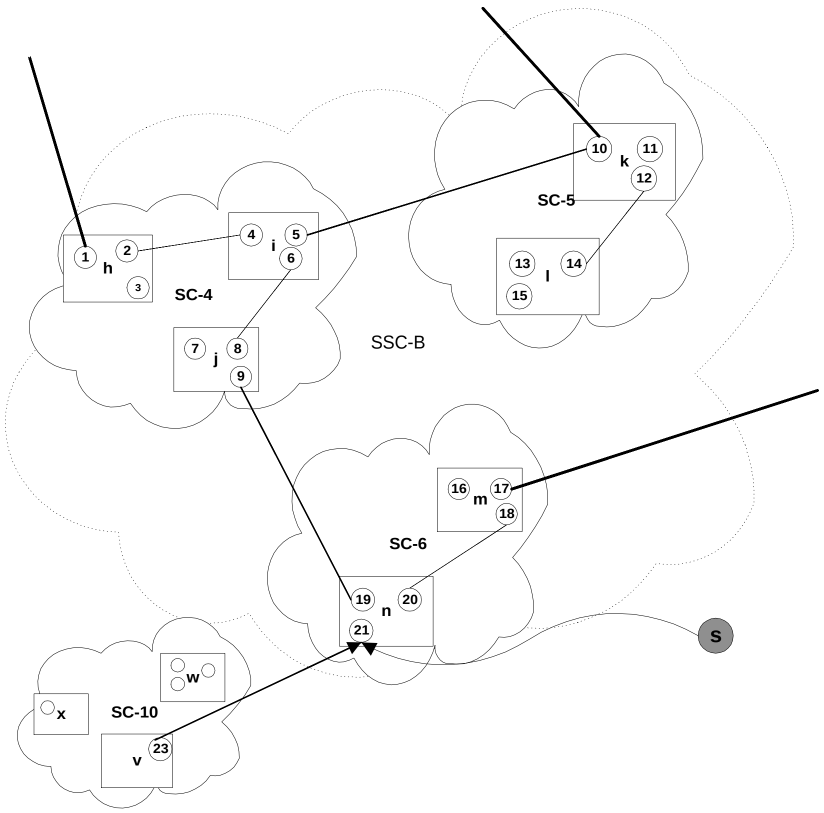


Figure 4.1.1: Adding nodes and units to an existing system

Figure 4.1.1 depicts a node **s** requesting a connection setup request. If **s** requests to be a level-1 node, then it needs to be part of the cluster **n**. Now, if node **n.21** hasn't exceeded the connection threshold limit for level-1 connections and also if the node **s** satisfies the IP-discrimination scheme for accepting nodes within the cluster then node **s** is configured as a level-1 node with a connection to node **n.21**. If however, node **n.21** has reached its connection threshold for level-1 connections but node **s** has satisfied the IP-discrimination requirements for cluster **n** then **n.21** forwards the request to other nodes within the cluster **n**. If there is a node within the cluster **n** which has not reached the connection threshold limit, then node **s** is configured as a level-1 gateway to such a node in cluster **n**. If however, all the nodes have reached their connection threshold limit, the node responds by providing a list of level-2 gatekeepers that are connected to cluster **n**. Node **s** then proceeds with the same process discussed earlier.

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

4.1.2 Adding a new unit to the system

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

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

Figure 4.1.1 depicts the addition of a super cluster SC-10 to the system. Only one node within the unit that needs to be added can issue the connection setup request. The node which issues this request in Figure 4.1.1 is the node **SC-10.v.23**. Since this is a level-2 system that is *unit-added*, node **23** or any other node with SC-10 can not be a level-1 (cluster) gateway to the other nodes within the super-super-cluster SSC-B. Node 21 thus issues a request specifying that it seeks to be a level-3 gateway within super-super-cluster B. Upon a successful connection set up, a new address is assigned for SC-10 (say SSC-8) identifiers for clusters within SC-10 remain the same. However the complete address of these clusters change to **SSC-B.SC-8.w** and so on.

4.2 The gateway propagation protocol - GPP

The gateway propagation protocol (GPP) accounts for the process of adding gateways and is responsible for the dissemination of connection information within relevant parts of the sub system to facilitate creation of abbreviated system interconnection graphs. However, GPP should also account for failure suspicions/confirmations of nodes and links, and provide information for alternative routing schemes.

4.2.1 Organization of gateways

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

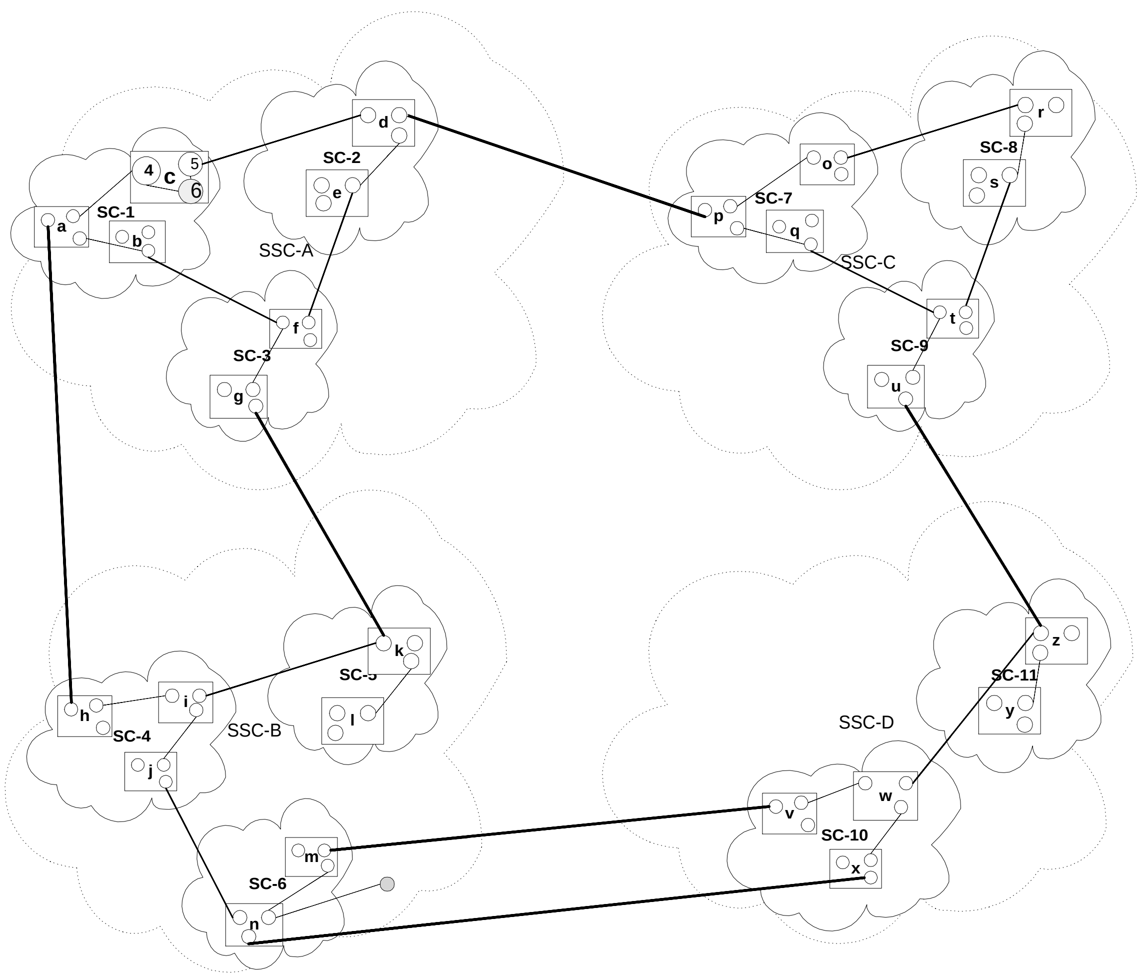


Figure 4.2.1: Connectivities between units

What is required is an abstract notion of the connectivities that exist between various units (sub-units and super-units alike) within the system. This constitutes the *connectivity graph* of the system. At each node the connectivity graph is different while providing a consistent overall view of the system. The view that is provided by the connectivity graph at a node should be of connectivities that are relevant to the node in question. Figure 4.2.1 depicts the connections that exist between various units of the 4 level system which we would use as an example in further discussions.

4.2.2 Constructing the connectivity graph

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

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

4.2.3 The connection

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

A level$-\ell$ connection is the tuple $< n_i^x, n_j^y, \ell > \ where \ x \mid y = \ell \ and \ x, y \leq \ell$. Units u_i^x and u_j^y share the same level-ℓ GES context $C_k^{\ell+1}$. For any given node n_i^ℓ in the connectivity graph we are interested only in the level $\ell, \ell+1, \cdots, N$ gatekeepers that exist within the unit and not the $\ell-1, \ell-2, \cdots, 0$ gatekeepers that exist within that unit. Thus, if a level-ℓ connection is established, the connection information is disseminated only within the higher level GES context $C_i^{\ell+1}$ of the sub-system that the gatekeepers are a part of. This is ensured by never sending a level-ℓ gateway addition information across any gateway $g^{\ell+1}$. Thus, in Figure 4.2.1 for a super-cluster gateway established within **SSC-A**, the connection information is disseminated only within the units, and subsequently the nodes in SSC-A.

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

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

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

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

When the connection information is being disseminated through the GES context $C_i^{\ell+1}$, it arrives at gatekeepers at various levels. Every gatekeeper $g^p \ni p \leq \ell_c$, at which it is received, checks to see if any of the node information depicts a node n^x where $x < \ell_c$ if this is the case the next check is if $p > x$. If $p > x$ the node information is updated to reflect the node as level-p node by including the level-p contextual information of g^p. If $p \not> x$ the connection information is disseminated *as is*. Thus, in Figure 4.2.1 the connection between **SC-2** and **SC-1** in **SSC-A**, is disseminated as one between node **5** and **SC-2**. When this information is received at **4**, it is sent over as a connection between the cluster **c** and **SC-2**. When the connection between cluster **c** and **SC-2** is sent over the cluster gateway to cluster **b**, the information is not updated. As was previously mentioned, the super cluster connection **(SC-1,SC-2)** information is disseminated only within the super-super-cluster **SSC-A** and is not sent over the super-super-cluster gateway available within the cluster **a** in **SC-1** and cluster **g** in **SC-3**.

4.2.4 Link count

Associated with every connection that is created is a unique identifier for the connection. All connections relevant for a node are maintained in a connection table. This scheme allows to detect if the connection table already contains a certain connection. There could be multiple connections between two specific units, this feature provides for greater fault tolerance. However, what is maintained in the connectivity graph is simply the connection which exists between the two units. The edge thus created also has a link count associated with it, which is incremented by one every time a new connection is established between two units which are already connected. This scheme plays an important role in determining if a connection loss would lead to partitions, this is described in section 6.1.5.

4.2.5 The link cost matrix

The link cost matrix specifies the cost associated with traversing a link. The cost associated with traversing a level-ℓ link from a unit u^x increases with increasing values of both x and ℓ. Thus the cost of communication between nodes within a cluster is the cheapest, and progressively increases as the level of the unit that it is connected to increases. The cost associated with communication between units at different levels increases as the levels of the units increases. One of the reasons we have this cost scheme is that the dissemination scheme employed by the system is selective about the links employed for finer grained dissemination. In general a higher level gateway is more overloaded than a lower level gateway. Table 4.2.1 depicts the cost associated with communication between units at different levels.

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

Table 4.2.1: The Link Cost Matrix

The link cost matrix can be dynamically updated to reflect changes in link behavior. Thus, if a certain link is overloaded, we could increase the cost associated with traversal along that link. This check for updating the link cost matrix could be done every few seconds.

4.2.6 Organizing the nodes

The connectivity graph is different at every node, while providing a consistent view of the connections that exist within the system. This section describes the organization of the information contained in connections (section 4.2.3) and super-imposing costs as specified by the link cost matrix (section 4.2.5) resulting in the creation of a weighted graph. The connectivity graph constructed at the node imposes directional constraints on *certain* edges in the graph.

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

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

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

Figure 4.2.2 depicts the connectivity graph that is constructed at the node SSC-A.SC-1.c.6 in Figure 4.2.1. The set L_{UL} at the node **SC-3** in the figure comprises of node **SC-2** at level-2 and node **b** at level-1. The set L_D at **SC-3** comprises of the node **SSC-B** at level-3. The cost associated with traversal

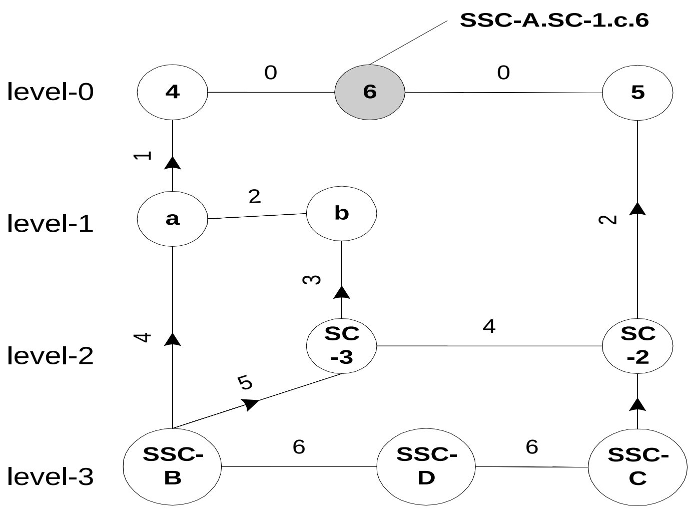


Figure 4.2.2: The connectivity graph at node 6.

over a level-3 gateway between a level-2 unit **b** and a level-3 unit **SC-3** as computed from the linkcost matrix is 3, and is the weight of the connection edge. The directional issues associated with certain edges are imposed by the algorithm for computing the shortest path to reach a node.

4.2.7 Computing the shortest path

To reach the vertex from any given node, a set of links need to be traversed. This set of links constitutes a *path* to the vertex node. In the connectivity graph, the best hop to take to reach a certain unit is computed based on shortest path that exists between the unit and the vertex. This process of calculating the shortest path starts at the node in question. The directional arrows indicate the links which comprise a valid path from the node in question to the vertex node. Edges with no imposed directions are bi-directional. For any given node, the only links that come into the picture for computing the shortest path are those that are in the set L_{UL} associated with every node.

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

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

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

Thus say in the connectivity graph of Figure 4.2.2 we are interested in computing the shortest path to SSC-B from the vertex. This process would start at the node SSC-B. The set of valid links from SSC-B include edges to reach nodes **a**, **SC-3** and **SSC-D**. At each of these three recursions the paths are reflected to indicate the node traversed (SSC-B) and the cost so far i.e 4,5 and 6 to reach **a**, **SC-3** and **SSC-B** respectively. Each recursion at every node returns with the shortest path to the vertex. Thus the recursions from a, SC-3 and SSC-D return with the shortest paths to the vertex. This added with the shortest path to reach those nodes, provides us with the means to decide on the shortest path to reach the vertex, which in this case happens to be 5.

4.2.8 Building and updating the routing cache

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

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

4.2.9 Exchanging information between super-units

When a subsystem u_i^ℓ is added to an existing system $u^{\ell+j+1}$ information regarding $g^{\ell+j}, g^{\ell+j-1}, \cdots, g^\ell$ is exchanged between the system and the sub system. The way the set of connections, comprising the connectivity graph, are sent over the newly established link is consistent with the rules we had set up for sending a connection information over a gateway as discussed in section 4.2.3. Thus if a new super cluster **SC-4** is added to the SSC-A sub-system and a super cluster gateway is established between SC-4 and node SC-1.6 the connectivity graphs at node 6 would be as depicted in Figure 4.2.3.(a) while the connectivity graph at the gatekeeper in SC-4 would comprise of the following connections sent over the newly established gateway.

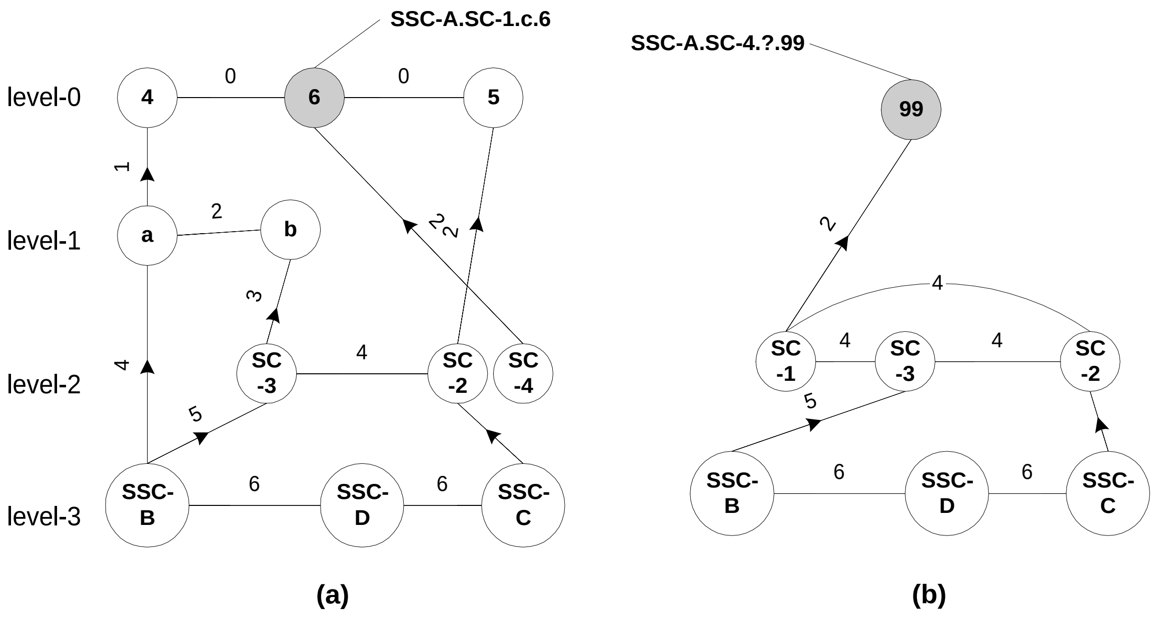


Figure 4.2.3: Connectivity graphs after the addition of a new super cluster SC-4.

Figure 4.2.3.(b) depicts only the connections which describe the connections involving level-2 gateways and upwards at node **99** in SC-4.

4.3 Organization of Profiles and the calculation of destinations

Every event conforms to a signature which comprises of an ordered set of attributes $\{a_1, a_2, \cdots, a_n\}$. The values these attributes can take is dictated and constrained by the *type* of the attribute. Events issued by clients within the system, assign values to the attributes. The values these attributes take comprise the content of the event. All clients are not interested in all the content, and thus are allowed to specify a filter on the content that is being disseminated within the system. Thus a filter allows a client to register its interest in a certain type of content. Of course one can employ multiple filters to signify interest in different types of content.

The organization of these profiles, dictates the efficiency of matching content. A level-ℓ gatekeeper snapshots the profiles of all the level-(ℓ-1) units that share the same GES context C_i^ℓ with it.

4.3.1 The problem of computing destinations

Clients express interest in certain types of content, and events which conform to that content need to be routed to the client. A simple approach would be to route all events to all clients, and have the clients discard the content that they are not interested in. This approach would however place a strain on network resources. Under conditions of high load and increasing selectivity by the clients, the number of events a client discards would far exceed the number of events it is actually interested in. This scheme also affects the latency associated with the reception of real time events at the client. The system thus needs to be very selective of the kinds of events that it routes to a client. In other words the system should be able to efficiently compute destination lists associated with the event. Depending on the event this destination list could be *internal* to the event or *external* to the event. In the case of events with external destination lists, the system relies on information contained within the client's profile as also the content of the event to arrive at a set of destinations that need to receive the event.

These destinations should be computed in such a way that it exploits the network topology in place, as also the routing algorithms which make use of abbreviated views of inter-connections that exist within the system. Profiles need to be organized so that it lends itself to very efficient calculation of destinations upon receiving a relevant event. In our approach a level-ℓ gatekeeper maintains the profiles of all the level-(ℓ-1) units that share the same GES context C_i^ℓ with it. This scheme fits very well with our routing algorithms, since the destinations contained within the event are the destinations which are consistent in the nodes abbreviated view of the system. To allow for a node to maintain profiles contained at different units (server nodes and client nodes) we need to be able to be able to propagate profile additions and changes to nodes responsible for the generation of destination lists.

The problem of computing destinations for a certain event comprises of the following -

(a) Organization of profiles in a profile graph

(b) Propagating profiles to nodes responsible for the calculation of hierarchical destination lists.

(c) Navigation of the profile graph to compute the destinations associated with the content.

A given node can compute destinations only at certain level. Thus the computing of destinations is itself a distributed process in our model.

4.3.2 Constructing a profile graph

As mentioned earlier, events encapsulate content in an ordered set of $< attribute, value >$ tuples. The constraints specified in the profiles should maintain this order contained within the event's signature. Thus to specify a constraint on the second attribute (a_2) a constraint should have been specified on the first attribute (a_1). What we mean by constraints, is the specification of the value that a particular attribute can take. We however also allow for the weakest constraint, denoted $*$, on any of the attributes.

This signifies that the filtered events can take any of the valid values within the range permitted by the attribute's type. By successively specifying constraints on the event's attributes, a client narrows the content type that it is interested in. It is not necessary that a constraint be specified on all the attributes $\{a_1, a_2, \cdots, a_n\}$. What is necessary is that if a constraint is imposed on a_i constraints for $a_1, a_2, \cdots, a_{i-1}$ must be in place, even if some or all of these constraints are the weakest ones. Thus if a constraint is specified till attribute a_i and no constraints are imposed on some of the attributes $a_1, a_2, \cdots, a_{i-1}$, the system assigns these attributes the weakest constraint $*$. It makes more sense imposing the constraint $*$ on higher order attributes $a_{i+1} \cdots a_n$ than on the lower order attributes $a_1, a_2, \cdots a_{i-1}$. Such a scheme has the effect of narrowing content down to the ones which are very closely related to each other.

For every event type we maintain a profile chain. Different profile chains added up together to constitute the profile graph.

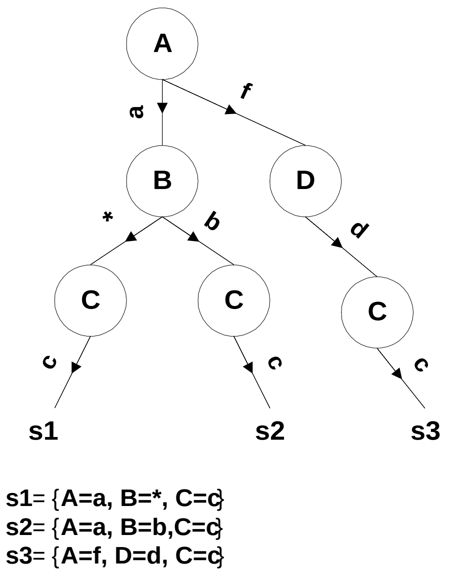


Figure 4.3.1: The profile graph - An example.

We use the organization and matching scheme based on the general matching algorithm presented in [ASS$^+$99] of the Gryphon system to organize profiles and compute the destinations associated with the events. Constraints from multiple profiles are organized in the *profile graph*. Each constraint that is specified on a attribute constitutes a node in the profile graph. When a constraint is specified on a_i, the attributes $a_1, a_2, \cdots, a_{i-1}$ appear in the profile graph. A profile comprises of constraints on succesive attributes of the event's signature. The nodes in the profile graph are linked in the order that the constraints have been specified. Any two successive constraints in a profile result in an edge connecting the nodes in the profile graph. Depending on the kinds of profiles that have been specified by clients, there could be multiple edges, *originating from* a node. Following the scheme in [ASS$^+$99] we do not allow multiple edges *terminating at* a node since it results in a situation where an event constraint matching results in an invalid destination after it has satisfied partial constraints of different profiles from the same unit.

Figure 4.3.1 depicts the profile graph constructed from three different profiles. The example depicts how some of the profiles share partial constraints between them, some of which result in profiles sharing edges in the profile graph. A certain edge is marked as traversed by an event if the two successive constraints, that created the edge, have been satisfied by that event. The presence of an edge signifies the existence of at least one client which is interested in the content satisfying at least two of the constraints contained in that edge. An event's traversal along an edge simply indicates that the event's content has satisfied some partial constraint imposed by one or more of the clients. As we traverse further down the profile chain, the events we are looking for get more fine grained. The final constraint on an attribute leads to the creation of a destination edge. The edges arising out of node **C** in figure 4.3.1 are destination edges.

4.3.3 Information along the edges

To support hierarchical disseminations and also to keep track of the addition and removal of edges, besides the basic organization of constraints, the graph also needs to maintain information along its edges. Associated with every edge we maintain the units that are interested in its traversal. And for each of these units we maintain the number of predicates $\delta\omega$ within that unit that are interested in the traversal of that edge. The first time an edge is created between two constraints as a result of the profile specified by a unit, we add the unit to the route information maintained along the edge. For a new profile ω_{new} added by a unit, if two of its successive constraints already exist in the profile graph, we simply add the unit to the existing routing information associated with the edge. If the unit already exists in the routing information, we increment the predicate count associated with that destination.

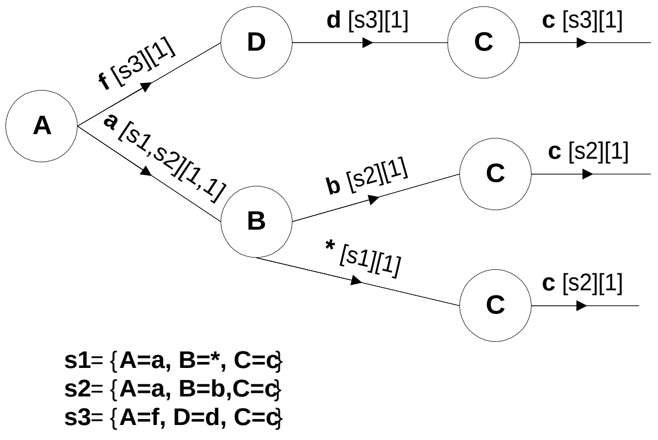


Figure 4.3.2: The complete profile graph with information along edges.

The information regarding the number of predicates $\delta\omega$ per unit that are interested in the two successive constraints allows us to remove certain edges and node when no clients are interested in the constraints any more. Figure 4.3.2 provides a simple example of the information maintained along the edges. We discuss how the profiles are propagated, where they are propagated and how this information along the edges is modified and updated in section 4.3.5.

4.3.4 Computing destinations from the profile graph

Once the profile graph has been constructed, we can compute the destinations that are associated with an event. Traversal along an edge is said to be complete if 2 successive constraints at end points of the edge have been satisfied by the content in question. When an event comes in we first check to see if the profile graph contains the first attribute contained in the event. If that is the case we can proceed with the matching process. When an event's content is being matched, the traversal is allowed to proceed only if -

(a) There exists a wildcard ($*$) edge connecting the two successive attributes in the event.

(b) The event satisfies the constraint on the first attribute in the edge, and the node this edge leads into is based on the next attribute contained in the event.

As an event traverses the profile graph, for each destination edge that is encountered while the event satisfies the destination edge constraint, the destination is added to the destination list associated with the event.

4.3.5 The profile propagation protocol - Propagation of $\pm\delta\omega$ changes

In the hierarchical dissemination scheme that we have, gateways $g^{\ell+1}$ compute destination lists for units u^{ℓ} that it serves as a $g^{\ell+1}$ for. A gateway $g^{\ell+1}$ should thus maintain information regarding the profile graphs at each of the units u^{ℓ}. Profile graph $\mathcal{P}_i^{\ell+1}$ maintains information contained in profiles $\mathcal{P}^{\ell}$ at units u^{ℓ} within $u_i^{\ell+1}$. This should be done so that when an event arrives over a $g^{\ell+1}$ in $u_i^{\ell+1}$ –

(a) The events that are routed to destination u^{ℓ}'s, are those with content such that at least one destination exists for those events within the sub-units that comprise the profile for u^{ℓ}.

(b) There are no events, that were not routed to u_i^{ℓ}, with content such that u_i^{ℓ} would have had a destination within the sub-units whose profile it maintains.

Properties (a) and (b) ensure that the events routed to a unit, are those that have at least one client interested in the content contained in the event. When an event is received over a cluster gateway, there would be at least one client attached to one of the nodes in the cluster which is interested in that event.

When we send the profile graph information over to the higher level gateways g^{ℓ}, the information contained along the edges in the graph needs to be updated to reflect the nodes logical address at that level. Thus when a node propagates the clients profile to the cluster gateway, it propagates the edges created/removed with the server as the destination. Similarly when this is being propagated to a super-cluster gateway the profile change is sent across as a profile change in the cluster. Any profile change in the clients is propagated to gateways at higher levels, that server node in its abbreviated view of the system is aware of. What we are trying to do is to maintain information in the profile graph, in a manner which is consistent with the dissemination constraints imposed by properties (a) and (b). The reason we maintain destination information the way we do is that this model ties in very well with our topology and routing algorithms in place. The connectivity graph provides us with an overall view of the interconnections between units at different levels. Also destinations that are generated at different levels, are invalidated when traversing over a gateway (depending on the level of the gateway). The algorithm for computing shortest paths to reach destinations at different levels relies on this scheme. The organization and calculation of destinations from the abbreviated profiles comprising the profile graph, feeds right into our routing algorithms that compute the shortest path to reach the units (destinations) where the event needs to be routed to. In general for a level-N system and if there's a client with GES context C_j^N and the issuing client has GES context C_i^N the destinations are computed N times. Thus in a system comprising of super-super-clusters, in the worst case the destinations are computed 4 times prior to reception at the client.

For profile changes that result in a profile change of the unit, the changes need to be propagated to relevant nodes, that maintain profiles for different levels. A cluster gateway snapshots the profile of all clients attached to all the nodes that are a part of that cluster. Thus a change in the profile of a client needs to be propagated to its server node. The change in profile of the node should in turn be propagated to the cluster gateway(s) within the cluster that the node is a part of. Similarly a super cluster gateway snapshots the profiles of all the clusters contained in the super cluster. When a profile change occurs at any level, the updates need to be routed to relevant destinations. The connectivity graph provides us with this information. From the connectivity graph, it can be seen that node 4 is the cluster gateway thus changes in profiles at level -0 i.e. $\delta\omega^0$ at any of the node are routed to 4. $\delta\omega^1$ changes need to be routed to level-2 gateways with SSC-1. The way this is calculated is the following -

(a) Locate the level-(ℓ) node in the connectivity graph.

(b) The uplink from this node of the connectivity graph to any other node, indicates the presence of a level-ℓ gateway at that node.

This scheme provides us with information regarding the level-ℓ gateway, within the part of the system that we are interested in. We are not interested in the lateral links since they provide us within information regarding all the level-ℓ gateways within the next higher level GES context $C^{\ell+1}$.

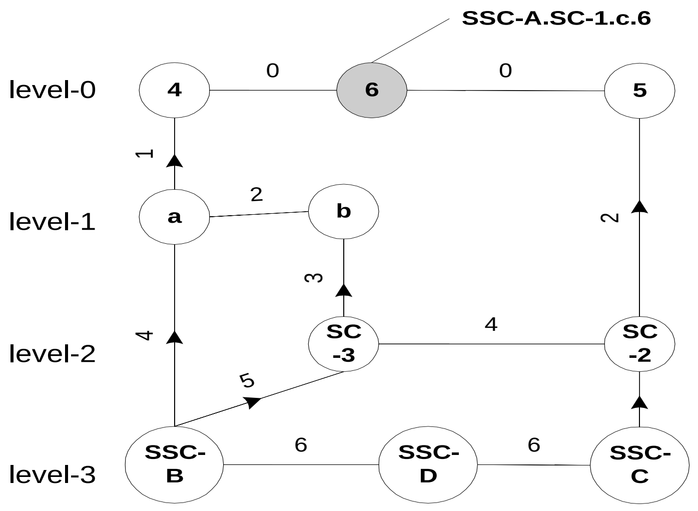


Figure 4.3.3: The connectivity graph at node 6.

In the figure 4.3.3, any $\delta\omega^0$ changes need to be routed to node **4**. Any $\delta\omega^1$ changes at node **4** need to be routed to **5**, and to a node in cluster **5**. Similarly $\delta\omega^2$ changes at node **5** needs to be routed to the level-3 gatekeeper in cluster **a** and superclusters **SC-3**, **SC-2**. When such propagations reach any unit/super-unit the process is repeated till such time that the gateway that the node seeks to reach is reached. Every profile change has a unique-id associated it, which aids in ensuring that the reference count scheme that we have does not fail due to delivery of the same profile change multiple times within the unit.

Summarizing the discussion so far, the profile graph snapshots content of units at a certain level, and as such can compute destinations for only these set of units. The profile snapshot that is created ensures that there is at least one sub-unit attached to one of the units within the super unit under consideration which should receive the event. Thus the profile matching scheme ensures that there is at least one client which will receive the event when it is received within a unit. If we do not have a scheme which snapshots profiles in the following manner, we could end up in a scenario where none of the events received in a unit have any clients which are interested in that event.

4.3.6 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 GES 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 received by the client. When we say that a profile is *unit-active*[1] what we mean is

[1]The unit we are referring to in this case are the clusters, super-clusters, super-super-clusters etc. Of course these units

that for every event that is being routed within that unit the destination lists calculated would include information to facilitate routing to the client. Since a client profile is unit active, all events, issued within the unit, will be routed to the client if it satisfies the client profile.

4.4 The event routing protocol - ERP

Event routing is the process of disseminating events to relevant clients. This includes matching the content, computing the destinations and routing the content along to its relevant destinations by 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 3.4.1) pertaining to this particular node in system. As the event flows through the system, via gateways the routing information is modified to snapshot its dissemination within the system. This information is then used to avoid routing the event to the same unit twice. What a node also needs to decide is when it would be futile to try and find a higher order gateway, and also when all the higher level units that could possibly be covered have been covered. Of course it should also know if there's a higher order gateway that needs to be reached. This decision is based on the event routing information and the information pertaining to gateways that's available at a node. If there are no such units that need to be reached, the event routing would proceed with lower order disseminations. However if there is a unit that needs to be reached, gateways would have to be employed to reach this unit as fast as possible. The event routing information contained with an event simply indicates the units which were present en route to reception at the node.

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

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

  There could be multiple links connecting a unit to some other unit. This scheme provides us with a greater degree of fault-tolerance. This also leads to the situation[2] 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 GES context of that $level-\ell$ and the unit identifier. Also, in general a higher order gateway would be more overloaded[3] compared to a lower order gateway. Reducing the amount of information being

are assumed to be within some higher level GES context of the server node to which the interested client is attached to or was last attached to

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

[3]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

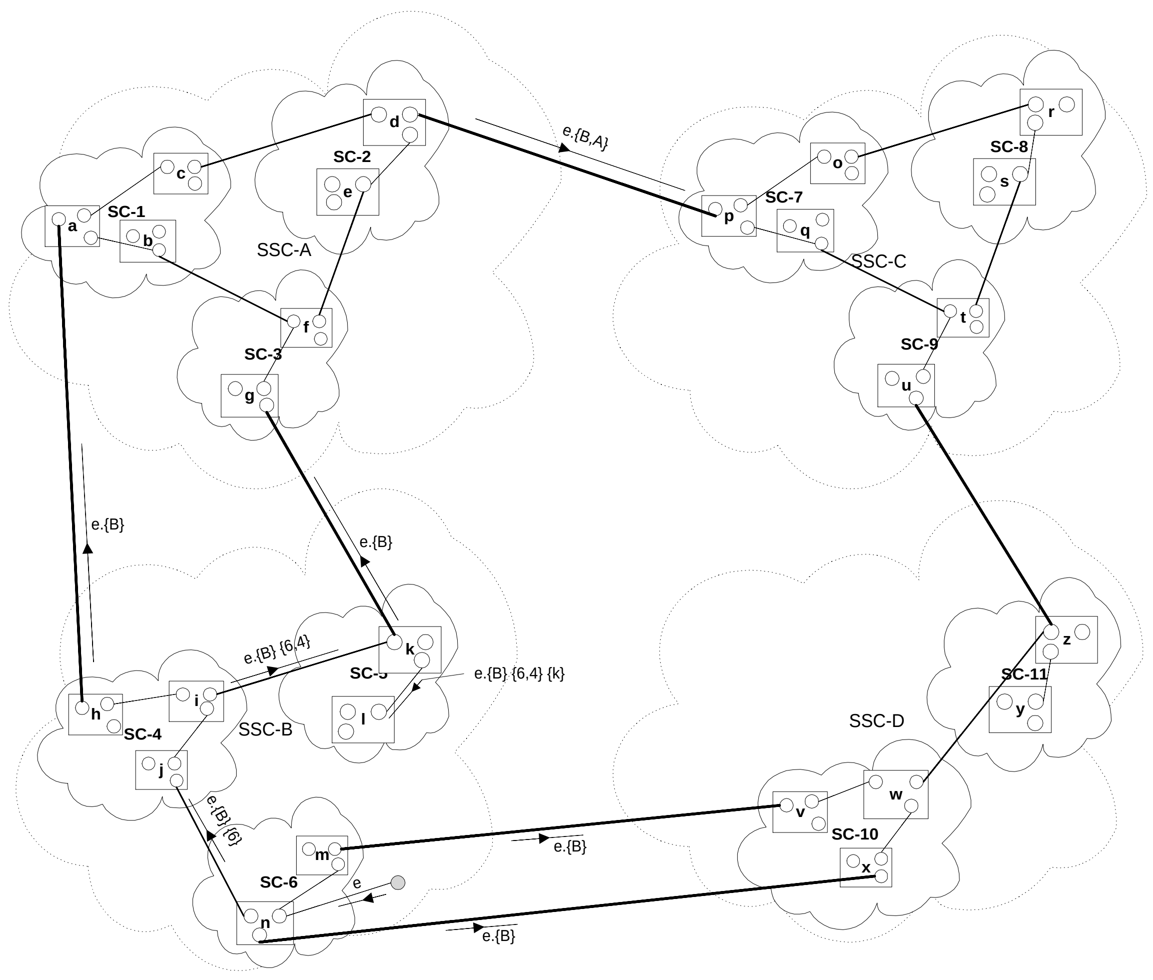


Figure 4.4.1: Routing events

transferred over the gateway helps conserve bandwidth.

Fig 4.4.1 depicts the routing scheme which we have discussed so far. The routings depicted in the figure are self explanatory and no further explanation is needed in this regard.

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

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

(b) Units which have already received the event.

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

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

for deciding if the event needs to be routed to any of the lower units comprising the $level-\ell$.

computed. Thus at every node the best decision is taken. Nodes and links that have not been failure suspected are the only entities that can be part of the shortest path. The event routing protocol, along with the profile propagation protocol and the gateway information ensure the optimal routing scheme for the dissemination of events in the existing topology.

4.5 Routing real-time events

Real time events can have destination lists (see section 3.1.4) which are internal or external to the event. In each case the routing differs, in the case of internal lists the destination's location needs to be precisely located by the system. Routing events with external destination lists involves the system calculating the destinations for delivery.

4.5.1 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 GES 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 GES context $C_i^{\ell+1}$ to relay this event.

4.5.2 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 4.5.1.

4.6 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 3.1.2 we discussed the generation of unique identifiers for events. This scheme of unique ID generation provides us with information pertaining to unrelated events (events issued by different clients) and in the case of related events (events issued by the same client) the order of their occurrence. In our scheme of duplicate event detection we use this unique ID generation as the basis for our duplicate event detection scheme.

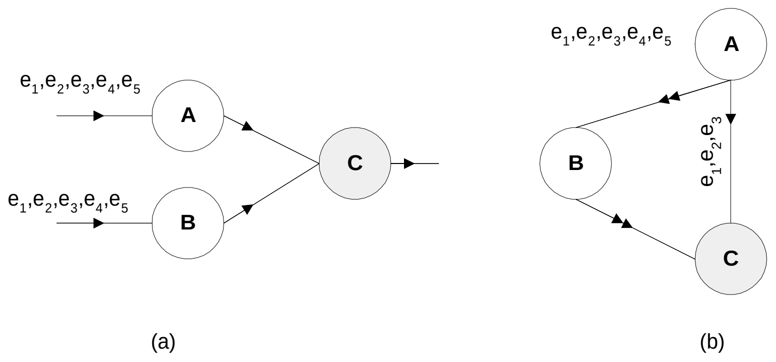


Figure 4.6.1: Duplicate detection of events

Our unique ID generation scheme allows us to determine which of two related events e and e' was issued earlier. If the last event received 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 received earlier else it was and is duplicate detected.

Consider the case in Fig 4.6.1.(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 received. 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 receives 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 received 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 reception sequence at any node within the system.

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

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	
$t \longrightarrow$	1	2	3	4	5	6	7	8	9

Table 4.6.1: Reception of events at C

Consider the case in Fig 4.6.1.(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, the events e_1, e_2, e_3 would be duplicate detected if e_4 arrives before e_1. Once we make this minor change of re-sending unacknowledged events across the alternate route in response to suspicions it simply reduces to the case depicted in Fig 4.6.1.(a). As an optimization feature we could include send *anti-events* down the failed/slow link whenever we resort to computing an alternate route.

Fig 4.6.2 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 4.6.2. 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.

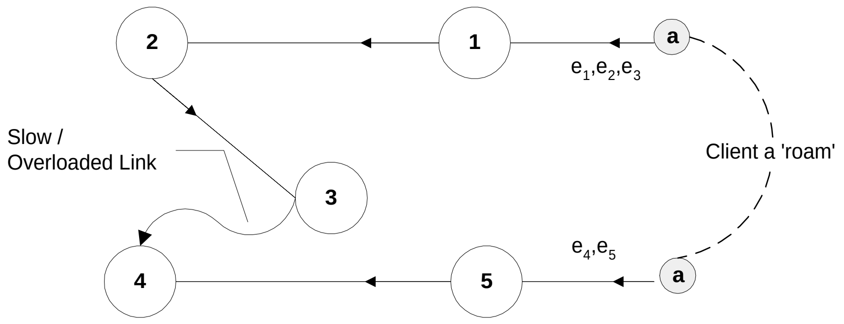


Figure 4.6.2: Duplicate detection of events during a client roam

$t \rightarrow$	$t+\Delta$	$t+2\Delta$	$t+3\Delta$	$t+4\Delta$	$t+5\Delta$
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 4.6.2: Reception of events at 4: Client roam

4.7 Summary

In this chapter we described a suite of protocols used in the design of the event service. This included the node addition protocol which is used to organize server nodes within the topology scheme that we introduced in Chapter 3. With the ability to add nodes/units to an existing sub-system, we proceeded to discuss the creation and organization of abbreviated system views at each server node in the system. We update this system inter-connection graph at each node with a link cost and link count for every edge within the graph reflecting the cost for link traversal and the number of links connecting two units respectively. We use this graph to compute shortest paths with graph traversal rules which restrict the paths that can be taken to reach a certain node. We proceeded to outline the organization of profile predicates and calculation of destinations using the matching algorithm discussed in [ASS$^+$99], and the modifications we made to this algorithm to include information along the edges to account for the number of predicates interested in a given edge and also the destinations associated with each edge to account for hierarchical propagation of profiles. The profile propagation protocol discussed the propagation of profile predicates to relevant nodes within the system, to support hierarchical dissemination of events within the system. This calculation of the nodes, to route profile updates to, is done based on the information encapsulated within the connectivity graph. To effectively disseminate messages within the system we introduced the event routing protocol. We also presented our approach to routing events with internal/external destination lists. Finally, we presented our scheme for the duplicate detection of messages.

Chapter 5

The problem of delivering merged streams

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

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

5.1 Resolution of spatial dependencies

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

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

(b) Being able to resolve these dependencies - This involves ensuring that events being fetched by system and merged into a new stream at the location that these dependencies were discovered.

Speeding up the resolution of dependencies enables us to optimize the creation of merged event streams. We thus need the system to be able to route events from streams in a manner which is conducive to the fastest merging.

5.1.1 Profile signatures and aiding the process of stream mergers

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

$$\forall_{e \in E, E^j}\, e.profileSignature(\,) = \overline{\varpi} \quad where \quad E \hookrightarrow \Pi \; and \; E^j \in \Pi \tag{5.1.1}$$

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

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

5.1.2 The spatial dependency $\overset{s}{\hookrightarrow}$ resolution

The distributed messaging mechanism, on the other hand, is responsible for resolving the spatial constraints that exist between events.

The stream source for $E \hookrightarrow \Pi$ is aware of the valid inter-dependencies that could exist between events in multiple related streams. This stream source constructs the stream context chain, much similar to the profile chain within a profile graph, that snapshots the spatial dependencies that exist between these related streams. Figure 5.1.1 shows a sample stream context graph between 4 related streams. The dotted edges originating from a node in the graph and terminating in another attribute node comprises the spatial dependency that exists between 2 related streams. You can have knowledge only about past events, at the same time how much of a knowledge can you store at each node. So what you store is something that allows you to conjecture what has been received so far. This is provided by *numbered stream contexts*. Also, if such a conjecture is not possible what you store is what you would like future events to satisfy. This is provided by *logical stream contexts*.

5.1.3 Propagating the dependency graph

The stream source for $E \hookrightarrow \Pi$ propagates the dependency graph to relevant nodes in the system. To start with in a N-level system, this information is propagated to all the g^ℓ where $\ell = 0, 1, \cdots, N$ that exist within the server node, that the stream source is attached to, system view. In other words the

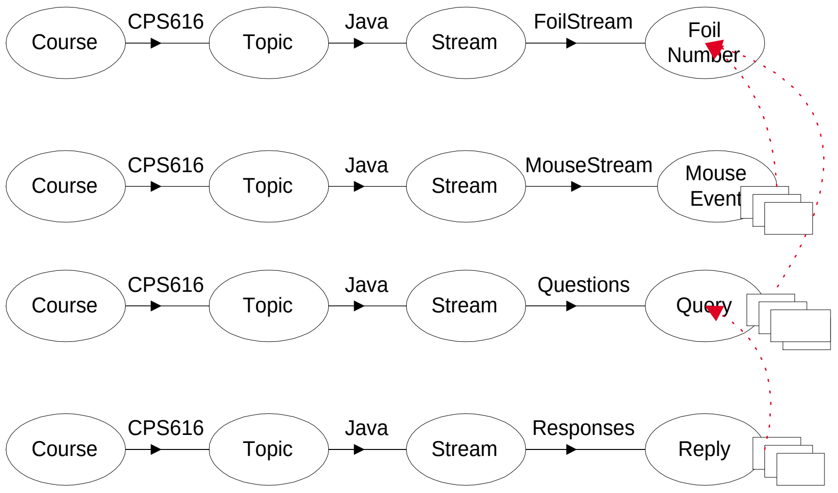


Figure 5.1.1: The stream context graph for 4 related streams.

server node propagates this information to its cluster gateways, super-cluster gateways and so on. This process of calculation of the nodes to route the graph to is identical to the profile propagation protocol.

Pushing the dependency graph to the client nodes which have an interest in this. In the case whenever a new stream context graph is propagated check to see if any valid destinations exists for the newly added elements to the context graph, and in case it exists push the dependencies to the relevant locations. Similarly when you have a disinterest remove it from the stream context graph if the destination list is reduced to zero.

5.1.4 Resolution of dependencies

As events are processed and dependencies resolved, the stream context information associated with the nodes is updated. When there is a sinking edge at the attribute node, we maintain contextual information pertaining to the last value of this attribute. This contextual information is maintained for every destination that is interested in receiving these events. These destinations that we refer to are hierarchical

Associated with the stream context graph, is the stream context at each of the nodes. This is of course hierarchical in much the same way as the profile graph's destinations are. When an event arrives a check is performed to see the constraints that the event satisfies. Depending on the results that this operation returns, events are either released for delivery to certain units or are garbage collected. The garbage collection scheme allows us to prevent unnecessary routing of events in the system.

Consider the example stream context graph in figure 5.1.1. In this example a bundle of mouse events $m_1^i, m_2^i, \cdots, m_n^i$ in the mouse stream can occur only within the stream context of the foil f_i within the foil stream. These mouse events would be invalid within any other foil. Lets denote the stream contexts for mouse event m as $m.c$ and those for the foils as $f.c$. In this case the foil stream context is a numbered context, and is advanced with the passage of time and the reception of successive foil events. For any given unit interested in receiving the merged stream, the following scenarios are possible

- $f.c = m.c$ In this case route the events in mouse stream, with context $m.c$ to the available destination list.
- $f.c > m.c$ discard the events in the mouse stream.
- $f.c < m.c$ queue these events in the mouse stream.

In the case of logical constraints, we are waiting for future constraints to be satisfied. In this case events are queued pending notifications regarding the receipt of appropriate events leading to the creation of queues of dependent events. As successive events arrive, some of the constraints would be satisfied and some of the queued events would be released for dissemination within the system. Each of these queues could have system imposed garbage collection constraints associated with individual queue elements to ensure that system resources are not overloaded.

5.1.5 Routing stream events

Clients in the system would specify an interest in $E \hookrightarrow \Pi$ and the system delivers the merged stream Π. The propagation of this interest $\delta\omega$ is identical to the profile propagation scheme we discussed earlier. Thus in the example depicted in figure 5.1.1 there could be a client specifying an interest `Course=CPS,Topic=Java`. When an event arrives the destinations are computed hierarchically from the profile graphs of g^{ℓ}'s for $\ell = N, N-1, \cdots, 0$, as discussed in the earlier chapter. These destinations form the preliminary destination list for the event. Further a check is made to see if the event is spatially constrained by an event from a related stream. If this event is constrained by events in other streams, a check is made to see if that constraint has been satisfied. If the constraint has been satisfied the event is routed to those destinations for which the constraint has been satisfied. If it is not the event is either discarded or queued for subsequent delivery. The arrival of an event which signifies that a future stream context or a certain numbered stream context is not likely to occur, will cause the garbage collection of events which were queued pending receipt of such a releasing event.

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

The occurrence vector $\mathcal{O}$ specifies the number of events within other event streams that are needed to satisfy an events dependency. Elements within the occurrence vector themselves contain two constraints [1]. For delivering an event e we require that only the weakest constraint of the occurrence vector element need be satisfied. Ensuring that constraints are fully satisfied prior to delivery is not practical in a live setting. It is, for example, impossible to know how many events would satisfy the constraint between two related streams.

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

Merging of event streams requires resolving both the spatial $\overset{s}{\hookrightarrow}$ and the time dependency $\overset{t}{\hookrightarrow}$. Timing dependencies $\overset{t}{\hookrightarrow}$ are imposed either at the source stream E source or are predefined along with the spatial dependencies $\overset{s}{\hookrightarrow}$, that exist between streams. In the former case events in streams $E^j \in Pi$ await their timing constraints from the source stream E prior to reception at a client. Newly generated events which add to streams in Π could either be request or response events. Request events do not have a context associated with it. The spatial dependencies may be self contained within the event itself, the timing dependencies are however dictated by the timing considerations at the source of the merged stream i.e. the source for E. It is this timing dependency which dictates the order for these events within the merged stream. Response events are events added to a stream E^j in response to the newly generated event which adds to the stream $E^j \in \Pi$. The timing dependencies for response events are implicitly specified by the system, the constraint imposed by the system is that the response event cannot be received at a client till such time that the associated request event has been received.

The rule is simple — request events can exist alone, however the response events exist only within the context of the <*request, response*> tuple. The merged stream at the stream E's source is what comprises

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

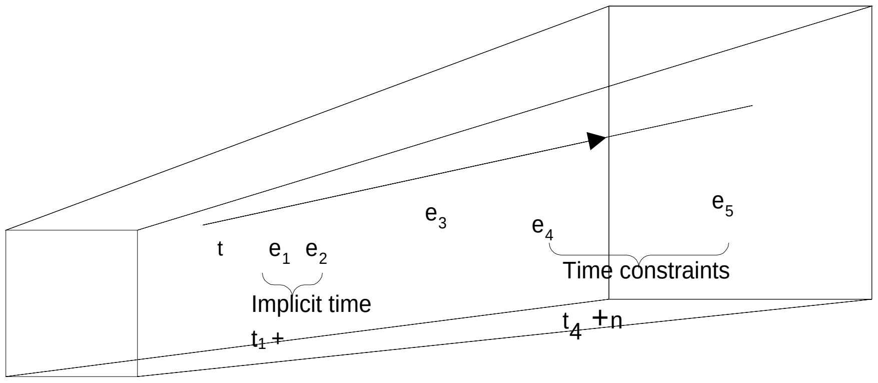


Figure 5.2.1: Dependencies and chronological ordering.

the playback stream. The spatial and timing dependencies thus dictate the ordering of events within the merged stream. There could be a number of request events that are generated by clients throughout the system, and they would seem to occur in a different order at each interested client. However the total ordering of these requests is determined by the order in which these events were received at the stream E's source.

5.3 Resolution of dependencies for newly added events

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

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

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

Fig 5.3.1 depicts one of the possible scenarios for resolving dependencies. Client A in the figure is source for streams E and $E^j \in \Pi$. Of course all the streams in Π could have been hosted at A, which is where ultimately the merged stream characteristics are specified. The session is a live session where

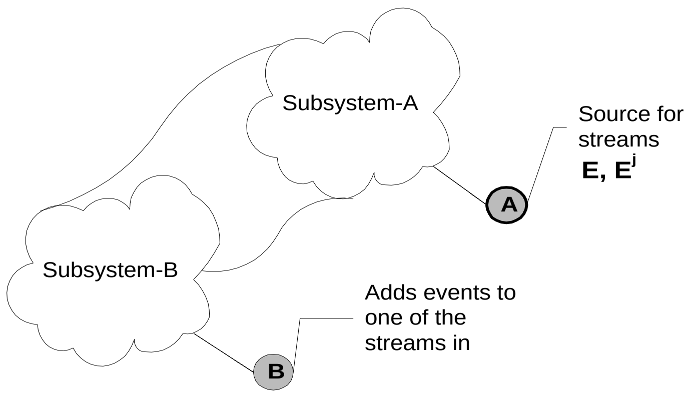


Figure 5.3.1: Resolving dependencies.

all the streams have events that would be issued as time progresses in the system. There could zero[2] or more clients interested in a merged stream, we consider one such client B. Clients interested in a merged stream can add one or more events to zero or more streams which constitute the merged stream.

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

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

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

5.4 Playback of event streams

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

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

[2]We are excluding stream sources which express an interest in their own events in order to avoid garbage collection of events in their streams. This is an artifact of the PPP and ERP which would automatically garbage collect those events which were issued when no clients had registered an interest in the stream sources.

to reside on different stable storages of the system, reconstruction would need to determine the locations of storages for $E \hookrightarrow \Pi$.

5.5 Streams & interpretation capabilities

Different clients have different interpretation characteristics. This interpretation capability is a function of the underlying system at the client. The streams that actually need to be routed[3] to a client are a function of the interpretation capabilities that are available at the client i.e $\Pi_{client} \subseteq \Pi$, this of course needs to be taken care by PPP[4]. The interpretation capabilities are dependent also on the event transformation switches that are available within the system. The switches are responsible for transforming the streams into something that can be deciphered by the client. Also if $E^j \notin \Pi_{Client}$ replace $E^j.e < data >$ with some value signifying the inability to represent content on the specific client device.

5.6 Summary

In this chapter we presented a solution to the creation of merged streams. We discussed the resolution of spatial and chronological dependencies that exist between multiple streams, and how the merged streams can be created even in the absence of clients interested in the streams.

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

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

Chapter 6

The Reliable Delivery Of Events

The problem of reliable delivery [HT94, Bir93a] and ordering[1] [BM89, Bir93b] in traditional group based systems with process crashes has been extensively studied. The approaches normally have employed the *primary partition* model [RSB93], which allows the system to partition under the assumption that there would be a unique partition which could make decisions on behalf of the system as a whole, without risk of contradictions arising in the other partitions and also during partition mergers. However the delivery requirements are met only within the primary partition [GRVB97]. Recipients that are slow or temporarily disconnected may be treated as if they had left the group. This model, adopted in Isis [Bir85], works well for problems such as propagating updates to replicated sites. This approach doesn't work well in situations where the client connectivity is intermittent, and where the clients can roam around the network. Systems such as Horus [RBM96] and Transis [DM96] manage *minority partitions*, and can handle concurrent views in different partitions. The overheads to guarantee consistency are however too strong for our case. DACE [BEGS00] introduces a failure model, for the strongly decoupled nature of pub/sub systems. This model tolerates crash failures and partitioning, and does not rely on consistent views being shared by the members. DACE achieves its goal through a self-stabilizing exchange of views through the Topic Membership protocol. In [BBT96] the effect of link failures on the solvability of problems (which are solved with reliable links) in asynchronous systems has been rigorously studied, while [Sch90] describes approaches to building fault-tolerant services using the state machine approach.

SmartSockets [Cor00b] provides high availability/reliability through the use of software redundancies. Mirror processes receiving the same data, and performing the same sequence of actions as the primary process and allows for the mirror process to take over in the case of process failures. *SmartSockets* also allows for routing tables to be updated in real time in response to link failures and process failures. *TIB/Rendezvous* [TIB99] integrates fault tolerance through delegation to another software *TIB/Hawk* which provides it with immediate recovery from unexpected failures or application outages. This is achieved through the distributed *TIB/Hawk* micro-agents which support autonomous network behavior which continue to perform local tasks even in the event of network failures.

Message queuing products are statically pre-configured to forward messages from one queue to another. This leads to the situation where they generally don't handle changes to the network (node/link failures) very well. They also require these queues to recover within a finite amount of time to resume operations. To achieve guaranteed delivery, JMS provides two modes: persistent for sender and durable for subscriber. When messages are marked persistent, it is the responsibility of the JMS provider [Cor99, iPl00, Inc00, Cor00a] to utilize a store-and-forward mechanism to fulfill its contract with the sender (producer). A durable subscription is one that outlasts a clients connection with a message server.

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

6.1 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.

6.1.1 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.

Message losses due to consecutive node failures

In Fig 6.1.1.(a) we have a situation where the two nodes ensure reliable delivery using a series of positive acknowledgements (ACKs). Node A will not garbage collect a message m until it has received an $ACK(m)$ from B. However it is possible that node B experiences a crash-failure immediately after issuing an $ACK(m)$ to A. Message m would thus never be received by C. We could try and rectify this situation as in Fig 6.1.1.(b) by requiring that a receiving node issue an ACK only after it has forwarded the message. This would solve our earlier problem, but this simply pushes the problem further in space, since the scheme would breakdown in case of successive broker failures after an ACK(m) has been issued by the soon to fail node B (the other one being C). Brokers B,C fail after B has issued an $ACK(m)$ and C has been unable to forward m to D. Thus, m is lost since A has already garbage collected it and D doesn't know if it should have received m (for that matter it wouldn't even know about the existence of m to even detect its loss) in the first place.

Augmenting the client nodes with re-issue behavior till such time that the event has been stored onto a stable storage circumvents this problem. 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.

Depending on the client's processing power, this re-issue behavior could be delegated to the server node that the client is attached to. The server node then is responsible for ensuring that the event

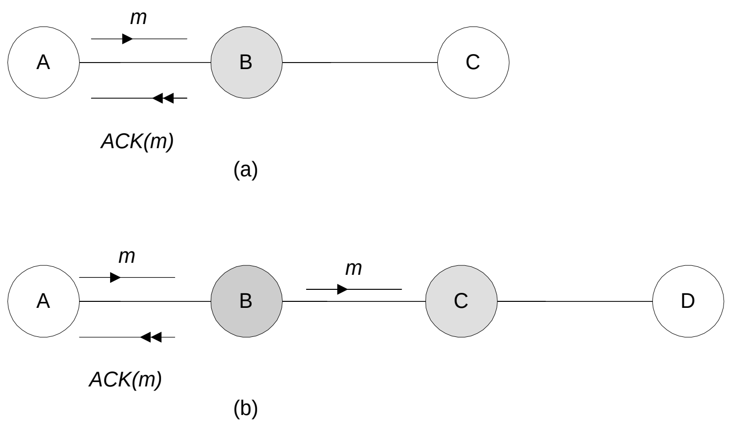


Figure 6.1.1: Message losses due to successive node failures

is written to stable storage. The Δt associated with the event on the client side could be increased, and checks only need to be made to ensure that the server node the client is attached to is functioning correctly.

6.1.2 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.

Transient gateway failures

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

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

Permanent gateway failures

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

6.1.3 Unit Failures

When we refer to unit failures, we are referring to the failure of all the nodes and associated gateways within that unit. In the case of unit failures, all the nodes within 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 based on the replication granularity that is available in the system that the node is a part of. 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.

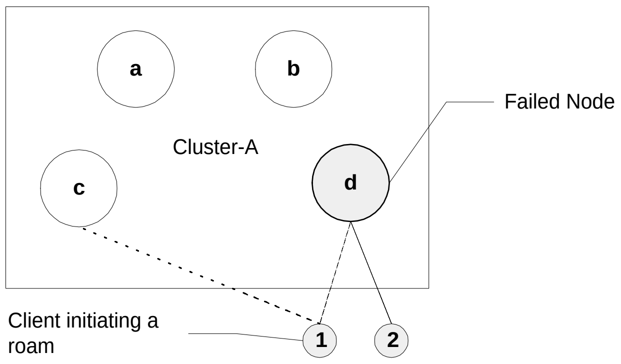


Figure 6.1.2: Client 'roam' in response to a node failure.

In sections 6.2.5 we discuss the process of handling events for a disconnected client, while in section 6.2.6 we discuss the process of handling events for a newly reconnected client.

6.1.4 Network Partitions

Network partitions can be caused both by link failures and node[2] 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.

6.1.5 Detection of partitions

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

Figure 6.1.3 depicts the connections that exist between various units of the 3 level system which we would use as an example in our discussions. The nodes within the connectivity graph are organized as nodes at various levels. Associated with every level-ℓ node in the graph are two sets of links, the set L_{UL} which comprises connections to nodes $n_i^a \ni a \leq \ell$ and L_D with connections to nodes $n_i^b \ni b > \ell$.

Figure 6.1.4 depicts the connectivity graph that is constructed at the node SSC-A.SC-1.c.6 in Figure 6.1.3. The set L_{UL} at the node **SC-3** in the figure comprises of node **SC-2** at level-2 and node **b** at level-1. The set L_D at **SC-3** comprises of the node **SSC-B** at level-3. The information contained in the loss

[2]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

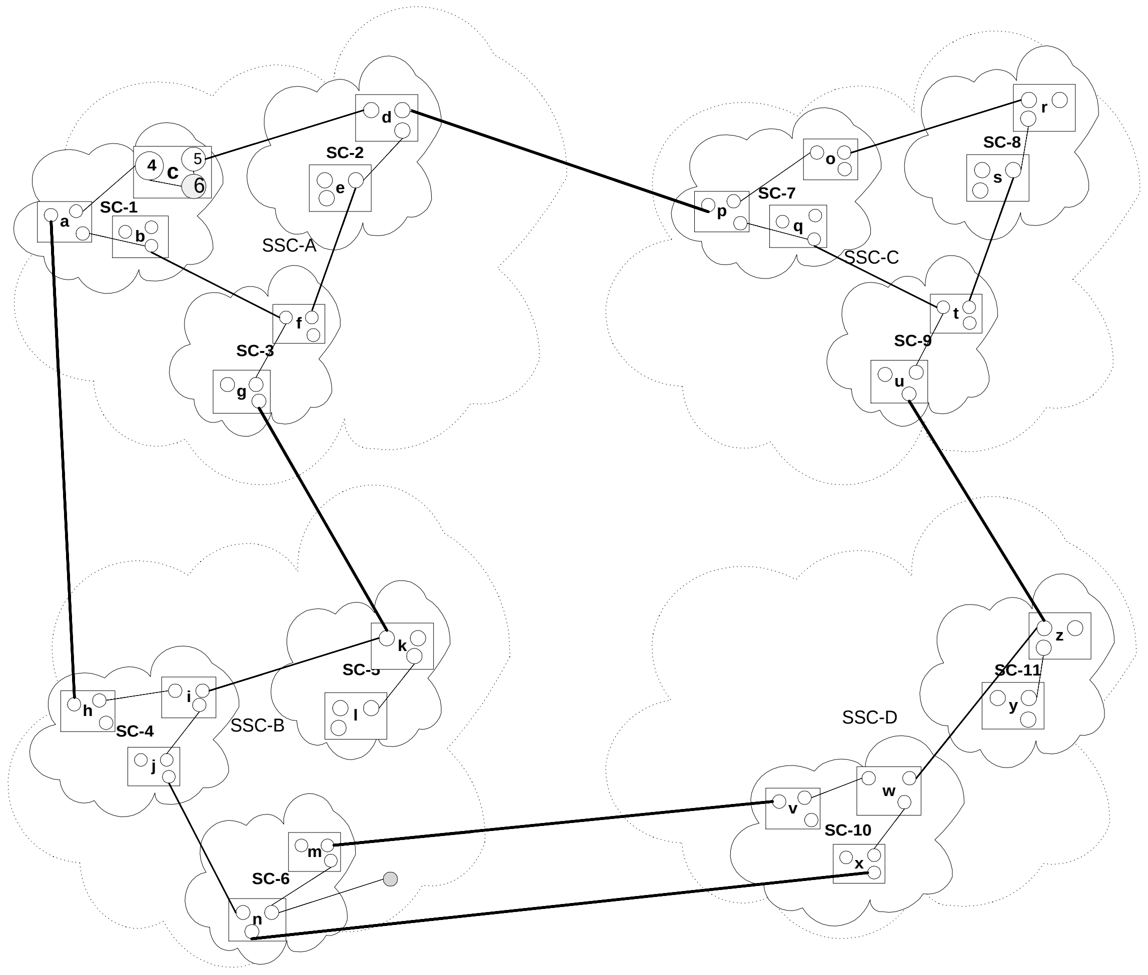


Figure 6.1.3: Connectivities between units and the detection of partitions

of connections is identical to that contained in the addition of a new connection. Also the dissemination of this loss of connection is dealt with in exactly the same way as additions are as described in section 4.2.3.

When a connection is lost we remove the connection from the connection table maintained at every node. This is based on the unique identifier associated with every connection. Next we check to see if there are other connections that exist between the corresponding nodes in the connectivity graph. If the link count associated with the connection edge is greater than one, we conclude that the connection loss is compensated by other existing connections. A situation where the link count is reduced to zero results in the removal of link information from the sets L_{UL} and L_D associated with the node. If the connection that was lost is the connection $< n_i^x, n_j^y, \ell >$ $where\ x \mid y = \ell\ and\ x, y \leq \ell$, if $y \leq x$ node n_j^y is removed from the set L_{UL} associated with node n_i^x and n_i^x is removed from the set L_D associated with the node n_j^y. The process is reversed if $x \leq y$. The detection of partitions is very simple. At the node whose L_{UL} is updated to reflect the connection loss. If $\#L_{UL} = 0$ the unit corresponding to the node is unit partitioned. If $\#L_{UL} = 0 \bigcap \#L_D = 0$ the unit corresponding to the node is system partitioned.

Referring to the connectivity graphs in Figure 6.1.4 in the case of node 6 in 6.1.4 the loss of the connection between clusters **a** and **b** results in **b** being unit partitioned within **SC-1** though it connected to SC-3. If however the only link that failed is the one connecting **SC-1** and **SC-3**, no units are partitioned. In the last case, if the link connecting clusters **a** and **b** fails and the link connecting cluster **b** and super-cluster **SC-3** also fails, node **6** in Figure 6.1.4 concludes that cluster **b** has been system partitioned.

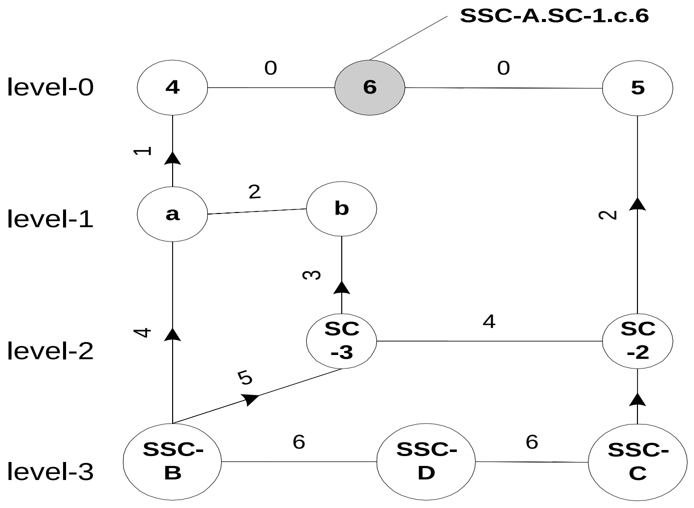


Figure 6.1.4: The connectivity graph at node 6 and the detection of partitions.

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

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.

Partition Mergers

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

6.2 Stable Storage Issues

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. We also need to address the issues pertaining to the control of the replication scheme. In section 6.2.1 we discuss the replication scheme for our system, and the process of adding stable storages within a sub-system. Section 6.2.3 describes the need for epochs, the assigning of epochs and the storage scheme for events. Section 6.2.4 describes the guaranteed delivery of events to all units within the subsystem. Finally in section 6.2.6 we describe the recovery scheme for roaming clients or clients connecting back after a prolonged disconnect.

6.2.1 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 and also on the *replication granularity* which exists within a specific unit. For a level-ℓ system if there's a stable storage set up for servicing all the server nodes within that unit, we denote the replication granularity for nodes within 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 received by all the clients within the system would be replicated $N \times N \times N$ 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 at different levels within the same unit. We do not impose a homogeneous replication granularity through out the system. Instead, we impose a constraint on the minimum replication scheme for the system. In a N-level system we require that every node within any level-N unit have a replication granularity of at least r_N. Thus in a system comprising of super-super-clusters we require that every server node within every super-super-cluster have a replication granularity of at least r_3. 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 other constraint we impose is that within a unit u_i^ℓ there can be only one stable storage at level ℓ.

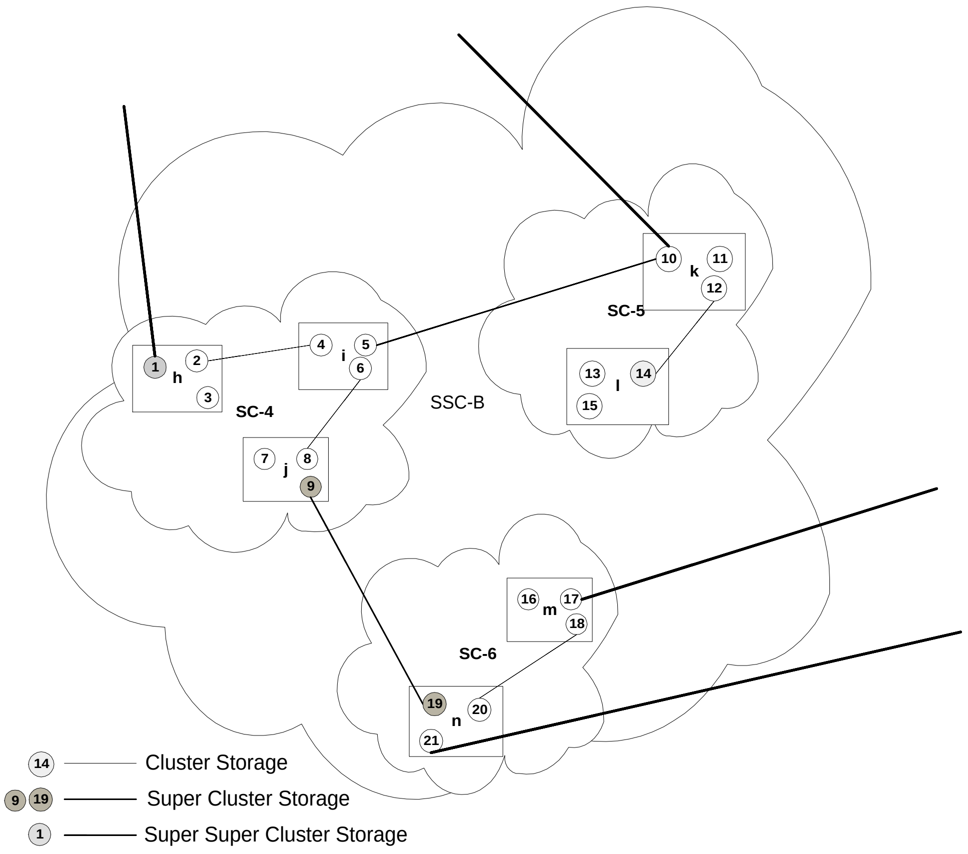


Figure 6.2.1: The replication scheme

The interaction between the stable storages of a unit and the stable storages within the sub units needs to address both the redundancy and garbage collection issues. Stable storages store events that the unit it is servicing, is interested in. This is ensured by the ERP which would ensure the routing of only the interesting events. The node which best serves this purpose is the gatekeeper node. As discussed earlier (section 4.3.5) PPP ensures that a gatekeeper $g_i^\ell(C_j^{\ell+1})$ snapshots the profile of level-ℓ unit i within the GES 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 GES 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 6.2.1 depicts the different replication strategies that can exist within different parts of a sub system. As can be seen super-super-cluster SSC-B has a replication granularity r_3, while super-cluster SC-4 within SSC-B has a replication granularity r_2. Cluster l has a replication granularity of r_1. Also in the depicted replication scheme there could be no other node in SSC-B which serves as a stable storage to provide nodes in SSC-B with a replication granularity of r_3. Similarly there could be no other stable storages which try to service units SC-4 and SC-6 with a replication granularity of r_2. Table 6.2.1 lists the replication granularities available at different nodes within the sub system depicted in fig 6.2.1.

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	**19**
13,14,15	r_1	**14**

Table 6.2.1: Replication granularity at different nodes within a sub system

Requirements (6.2.1), (6.2.2) and (6.2.3) snapshot the various constraints that we impose on our replication strategy.

Requirement 6.2.1 *In a N-level system, the replication granularity at each and every node in the system must be at least* r_N.

Requirement 6.2.2 *A level-ℓ stable storage can only be set up at node which serves as a level-ℓ gatekeeper.*

Requirement 6.2.3 *For a level-ℓ unit u_i^ℓ only one of the gatekeepers g^ℓ can be configured as a level-ℓ stable store.*

Adding stable storages and updates of replication granularity

When a stable storage is configured as a level-ℓ storage, we try to update the replication granularities associated with the nodes within the level-ℓ unit u_i^ℓ that the store is a part of. For a node x if the node's replication granularity is r_m^x, there are two possible outcomes. If $m > \ell$ the node's replication granularity is updated to ℓ i.e. r_ℓ^x. On the other hand if $m < \ell$ the replication granularity for the node is left unchanged. Thus for example if the unit had a granularity of r_3^x and r_2 has been added, the granularity is changed to r_2^x. A condition where $m = \ell$ is an error condition since it depicts the presence of multiple stable storages at the same level, a situation which should not arise because of the constraint that we have imposed. Every node also keeps track of $r_\ell(min)$ and $r_\ell(next)$ which refers to the minimum replication granularity and the next highest one. This comes into the picture during the guaranteed delivery of events, and is used to retrieve data from other stable storages when a finer grained store is added (discussed in section 6.2.3). To sum up our discussions so far, if in figure 6.2.1 we were to set up a stable storage at node **SSC-B.SC-6.m.18**. Further if we configure this stable store as a cluster storage with replication granularity of 1, the replication granularity at nodes **16,17** and **18** is updated to r_1.

6.2.2 Stability

For a N-level system with a minimum replication granularity of r_N, the responsibility for ensuring stability of messages is delegated to finer grained stable stores for those sub systems where the replication granularity is less than N. In the case of multiple stable stores at different levels within a single super-unit, the stability requirements for individual nodes are delegated to the finest grained store servicing the node. Thus in fig 6.2.1 stability for nodes **13,14** and **15** are handled by the cluster store at **14** while those for **10,11,12** are handled by the level-3 store at node **1**. Every event in the system should be stable since we should be able to retrieve it in case of failures or roams initiated by clients. Stable storages need to wait for notifications prior to the garbage collection of events. To aid in this process of garbage collection of events from stable storages we make a small change to way an event's destinations are computed at a storage node. When a node is hosting a stable storage with r_ℓ, the node is responsible for computing destinations at level-(ℓ-1) within the level-ℓ unit that the node belongs to. Associated with these destinations the node also computes the number of predicates per destination that are interested in receiving the event. The predicate count per destination allows us to garbage collect events upon receiving acknowledgements from destinations. The acknowledgements include the predicates that were serviced at the destinations. The destinations associated with the acknowledgement is updated depending on the gateway that it is transmitted over. For acknowledgements issued by a server node which has a replication granularity of r_ℓ the acknowledgement is never sent over a level-ℓ gateway g^ℓ. Thus acknowledgements to decrement predicate count for a cluster storage should never be allowed to leave the cluster.

If finer grained stable storages are present within the subsystem with r_ℓ the receipt notification is slightly different. As soon as the event is stored to the finer grained stable storage, a notification is sent to the coarser grained storage indicating the receipt of the event and predicate count that can be decremented for the sub-unit. Thus in figure 6.2.1 when an event stored at node **1** is received at node **19** we can assume that all nodes in unit **SC-6** can be serviced and decrement the reference counts at the level-3 stable store at node **1** accordingly.

6.2.3 The need for Epochs

We digress here to discuss the need for *epochs*. When a node is hosting a stable storage with r_ℓ, the node is responsible for computing destinations at level-($\ell - 1$) within the level-ℓ unit that the node belongs to. Associated with these destinations the node also computes the number of predicates per destination that are interested in receiving the event. The predicate count per destination allows us to garbage collect events upon receing acknowledgements from destinations. Consider the following scenario where the predicate count equals the client count for a destination associated with the event. Unit s_A has a total of 156 clients attached to it and unit s_A fails. Clients which detect this failure would initiate a roam. Local queues could be constructed for each client that has initiated a roam in response to this failure. For each queue constructed and sent across the system to it new hosting unit, the reference count associated with every event contained within the queue is decremented by one. However, it is conceivable that a client could have been attached to s_A, which had joined the system for the first time prior to the unit failure. This client is thus *not* the intended recipient of any of the local queues that would be constructed in response to the servicing of roaming clients. If this client is one of the first clients to initiate a roam, local queues would be constructed for it and the reference counts of the events contained within this local queue would be decremented by one. This operation would lead to the *starvation* of at least one client, if any of the 156 clients contained a profile which partially matched that of the new client.

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

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

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

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

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

Epoch generation

Epochs, denoted ξ, are truly determined by the replication granularities that exist in different parts of the system. In the case of a client its the GES context of the server node that it is attached to which determines the epoch. A client could be operating in disconnected mode. Such a client is nevertheless still serviced based on its profile, the destination for delivery being the node or unit (in case the node fails) at which the client was last present. This profile along with its last logical address serves as a proxy for the client in its absence. Some of the details pertaining to epoch generation are listed below –

(a) Epochs should monotonically increase.

(b) Epochs for client nodes exist within the context of the finest grained stable storage that the server node (that it is attached to) is a part of. Thus if the server node has a replication granularity of r_2 valid epochs for events received by the client would those that have been assigned by the corresponding level-2 storage.

(c) For every client with a profile ω there is a epoch ξ^{ω} associated with it.

The fact that there is only one epoch associated with every ω, follows from property (b) and also from the constraint that there can be only 1 stable storage configured for servicing a unit u_i^{ℓ} with a granularity of r_{ℓ}.

Requirement 6.2.4 *A persistent client will not receive an event e unless there is an epoch, ξ_e, associated with the event.*

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. However, every event received at a client needs to have an epoch associated with it to aid in the recovery and servicing of events that have not been received by the client. The arrival of such an event results in an update of the corresponding epoch associated with the client's profile. Profile changes initiated by a client also have an epoch associated with it. This is discussed in a later section. The reason why we don't need an epoch for every $\delta\omega$ is that the epochs are assigned to a client by the stable store, and irrespective of the addition of stable stores at different levels these epochs monotonically advance, and the reception of an epoch easily allows us to conjecture about the events that should be (or have been) received.

The replication granularity within the system could be different in different sub systems. Within a sub system having a replication granularity r_{ℓ}, it is possible that there is a "sub system" with replication granularity $r_{\ell-1}, r_{\ell-2}, \cdots, r_0$, in such cases the epochs assigning process is delegated to the corresponding replicators. If a node within u_i^{ℓ} has a granularity of r_{ℓ}, it needs to await the receipt of an epoch assigned by the level-ℓ storage at u_i^{ℓ}. Thus the epoch associated with the same event could be different at different clients in the system. In figure 6.2.1 it is possible that by the time an event arrive at node **15** there will be two different epochs associated with it, only one of which is valid for clients attached to node **15**. Also epochs associated with the same event could be the same at different parts in the system. Thus clients attached to any of the nodes in **SC-6** in figure 6.2.1 have the same epochs (assigned by the store at node **19**) for events that they would all receive.

The storage format

When an event is written to a stable storage, there are epoch numbers associated with it. Since all events are not routed to all destinations we maintain the destinations associated with the event. Besides the destinations associated with the event, the matching operation at the stable storage nodes also return the predicate count associated with the event. This information is used to service roaming clients or clients re-joining after a duration of time, and still be able to garbage collect events from the stable storages. We also maintain information pertaining to the type of the event, and the length of the serialized representation of the event. Finally we also maintain the serialized representation of the event.Thus the storage format is the following tuple $< \xi^e, (d^e_0, d^e_1, \cdots, d^e_n), (p^e_0, p^e_1, \cdots, p^e_n), e.type, e.length, e.serialize >$.

Epochs and profile changes

Whenever a profile change is made, there needs to be an epoch associated with the profile change. The epoch, assigned by a replicator, depending on the subsystems granularity is an indicator of the time from which point on, that change would be serviced by the system. If an epoch is not associated with profile changes, it is conceivable that starvation of some client would occur. Consider the following scenario, a client receives a sequence of events $e_1, e_2, \cdots, e_n$. For an extended period of time this client doesn't receive any events. The last epoch that it received was ξ_n. This client then proceeds to make a profile change, and leaves the system. When it rejoins the system at a later time, this client would expect to receive all it missed events, which would include events which satisfy the profile change it last made, starting with its last known epoch ξ_n.

We thus have an epoch associated with every profile change, and require that the client to wait till it receives the epoch notification, before it can disconnect from the system.

Epochs and the addition of finer grained stable storages

When a new stable storage is being added within a unit, if the replication granularity of that unit is r_m and if the new storage for that unit is at level-n. For nodes where $m < n$ the new stable storage should access the storage with r_n and retrieve stored data with epoch numbers in ascending sequence for events which were meant to be disseminated within the unit. The predicate count associated with the destinations for each individual event needs to be updated accordingly depending on the predicate count associated with the destination. This is especially crucial since there could be clients which have epoch numbers associated with the ones assigned by storage hosting r_m. The addition of a stable storage at level ℓ is disseminated only within the u^ℓ_i that the hosting node belongs to. The first event that the newly added store receives provides it with the epoch number from which point on the epoch numbers assigned by the old store and the new store can deviate.

6.2.4 Ensuring the guaranteed delivery of events

For a level-N system the stable storages servicing level-N units are designated also as *system storages*. Figure 6.2.2 depicts a system comprising of 4 super-super-clusters and the replication schemes that exist in different parts of the system. For events issued by clients attached to nodes within these u^N units, these system storage nodes have the additional responsibility that they maintain events in stable storage till such time that they are sure that all the other u^N's within the system have received that event. When an event is issued within a super unit u^N_i, the destinations are computed as described in the event routing protocol. However, before the event is allowed to leave u^N_i, it must stored onto the stable storage which provides nodes with the minimum replication granularity of r_N. Thus in figure 6.2.2 for an event issued by a client attached to a node in **SSC-B**, the node must be stored to the system store in **SSC-B.SC-4.h** before it can be routed to **SSC-A,SSC-C** and **SSC-D**.

The node maintains the list of all known u^N destinations within the system. This destination list is associated with every event that is stored by the system. Associated with these events is a sequence

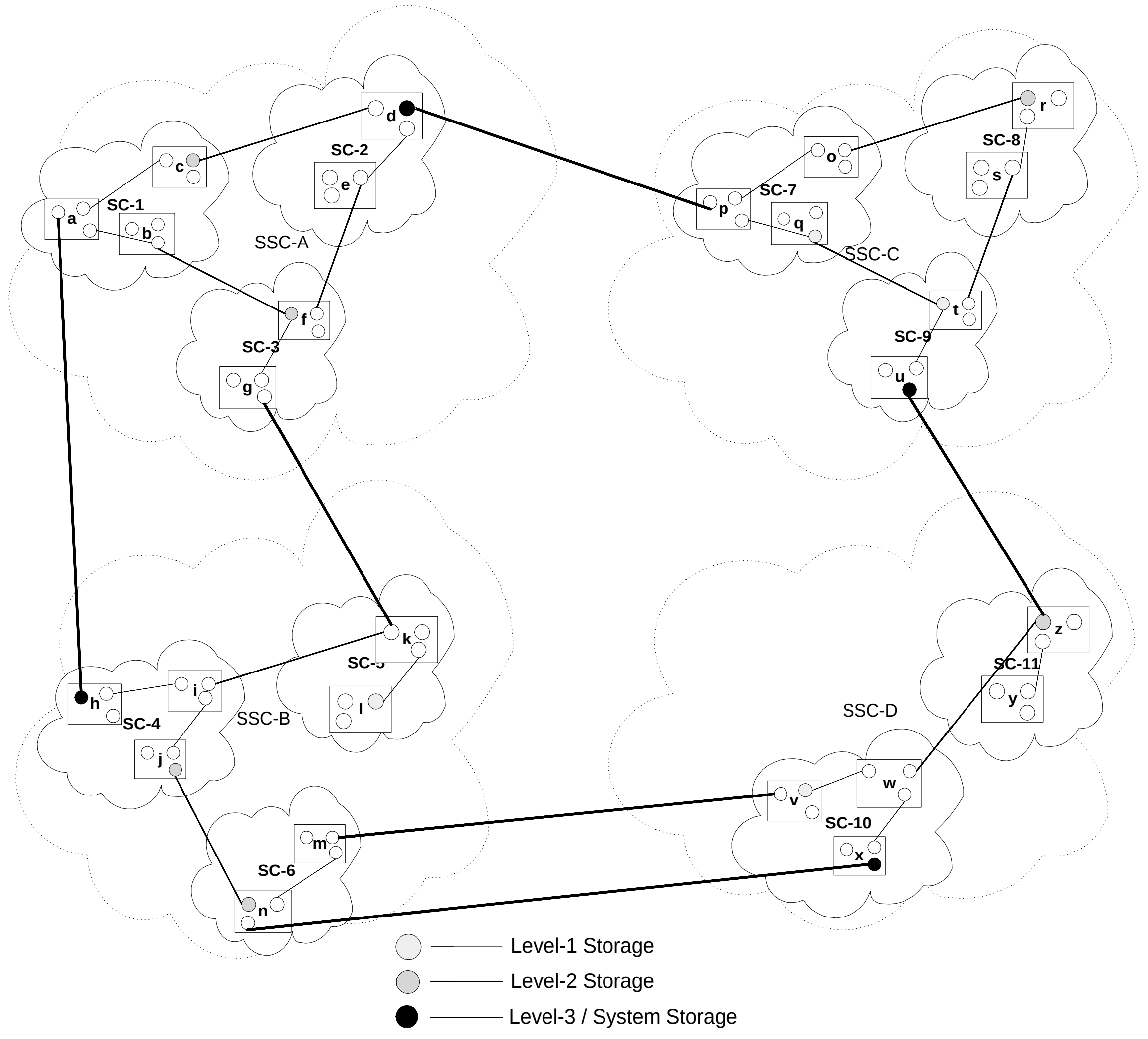


Figure 6.2.2: Systems storages and the guaranteed delivery of events

number, which is different from the epoch number associated with the events that clients receive. Once it is stored to such a storage the event is ready to be sent across to other u^N destinations within the system. Also, for an event that is issued by a client within u_i^N, the event is stored to stable storage (to ensure routing to other u^N's within the system) within u_i^N and not at any other system storages at the other u^N's within the system. When the events are being sent across gateway g^N for dissemination to other u^N's they have a sequence number associated with them and also the unit u_i^N in which the event was issued. This is useful since the r_N replicators (which serve as system storages) in other units can know which unit to send the acknowledgements (either positive or negative) to. Thus for an event e issued by a client in **SSC-B** what we store is - $< seqNumber, e, (SSC-A, SSC-C, SSC-D) >$.

Every system storage node also keeps track of the last sequence number that was received from a certain unit u_i^N. Thus the system store in **SSC-B** would keep track of the last received sequence numbers for events published by clients in **SSC-A,SSC-C** and **SSC-D**. Every system storage node can now keep track of any events that it should receive from a certain unit u_i^N. Consider the case where the system store node at **SSC-A** has received events with sequence numbers $s_1^B, s_2^B, \cdots, s_{100}^B$ from unit **SSC-B**. When this system store receives an event with sequence number s_{103}^B from **SSC-B** based on the last sequence number that it received, s_{100}^B, it knows that it has missed events with sequence numbers s_{101}^B, s_{102}^B. This storage node then issues a NAK to retrieve those events. The storage node at **SSC-B** when it receives this NAK(s^B_{101}, s^B_{102}), it re-issues those events to requesting system store. The system store node at **SSC-A** does not assign an epoch or route the event with sequence number s^B_{103} till such time that it receives events with s^B_{101}, s^B_{102} from **SSC-B**. If an event with sequence number s^B_{98} (assigned by **SSC-B**) is received, the store at **SSC-A** discards this event since it knows that it has already processed this event just as it has processed events with sequence numbers $s^B_{97}, s^B_{96}, \cdots s^B_0$.

Upon receipt of an event with an associated sequence number, other system stores issue an *ACK(seqNum)* to facilitate garbage collection. The fact that sequence numbers assigned by any system store increases monotonically allows us to sustain the loss of acknowledgement messages. This is because the receipt of an acknowledgement for an event e (stored with sequence number n) *ACK(n)* implies the receipt of events with sequence numbers $n, n-1, n-2, \cdots$ issued by the system store. The receipt of an ACK from a certain unit results in that unit being removed from the destination lists associated with events with sequence numbers $n, n-1, n-2, \cdots$. When the destinations associated with an event is reduced to zero the event is garbage collected. Thus in our example for events $e_1, e_2, \cdots, e_n$ issued by clients in **SSC-B** and stored to the system store at **SSC-B.SC-4.h** with sequence numbers $s_1, s_2, \cdots, s_n$ with destination lists **SSC-A,SSC-C** and **SSC-D**. The receipt of an ACK(n) from **SSC-A** results in the removal of **SSC-A** from the destination lists associated with the stored event. When the destination list associated with any of the stored events is reduced to zero that event is garbage collected.

The upward propagation of events

When an event is issued by a client attached to a server node, other clients interested in that event do not receive the event till such time that there is an epoch associated with the event. This epoch is dependent on the replication granularity that exists at the corresponding server nodes. The epoch that is associated with an event should be the epoch that is assigned by the servicing storage for the server node in question. Events with epochs assigned by replicators r_ℓ are valid only within the u^ℓ_i that the node belongs to. When an event is issued by a client, the reissue behavior ensures that the event is stored onto a stable storage. If this stable storage is not the system storage responsible for r_min, the node is responsible for storing this event and not scheduling it for garbage collection till it receives a notification from the system storage regarding the receipt of that event. Besides this for a N-level system, the event is not allowed to leave the unit u^N till such time that there is a sequence number (assigned by the system storage) associated with it. We use figure 6.2.3 to explain the routing of events to persistent clients.

For an event e issued by a client attached to node **SSC-B.SC-5.l.15** the client re-issue behavior ensures that the event is stored to the cluster storage at node **14**. Even if the reference count associated with this event is reduced to zero, the cluster storage cannot garbage collect this event till such time that the event is stored to the system storage at node **1**. It is now the responsibility of node **14** to ensure the storage of the event e to the system store and corresponding re-issues to ensure the same. This event is not allowed to leave **SSC-B** though a gateway does exist at the super-cluster **SSC-B.SC-5**. Once the event is stored to the system storage at node **1** it is allowed to leave the super-super-cluster. The system store at node **1** should also send a notification to node **14** indicating that the event was stored to the system storage. The event e is then replicated at node **9** and **19**, if there are any clients attached to super-clusters **SC-4** and **SC-6** respectively. This event which is being disseminated in say **SC-6**, would have 2 epochs associated with it — one assigned by the level-3 store at node **1** and the other assigned by the level-2 store at node **19**.

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

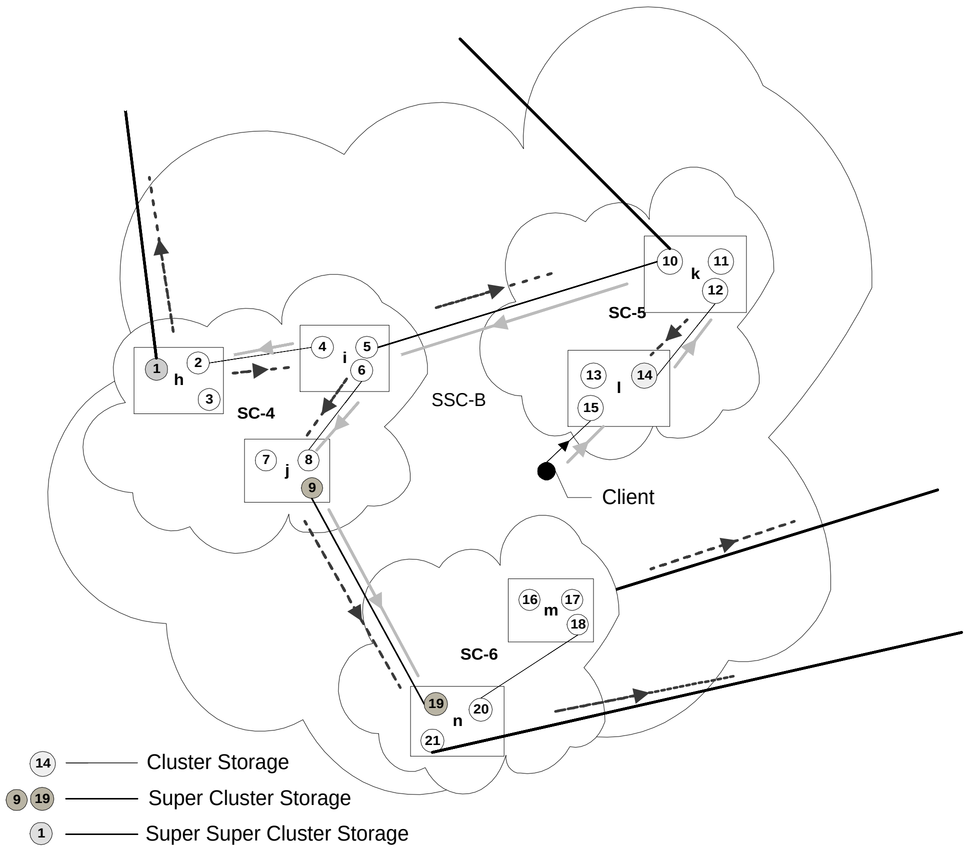


Figure 6.2.3: The propagation of events in the system

6.2.5 Handling events for a disconnected client

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

When a client is not present in the system, the event is not acknowledged and thus can not be garbage collected by the replicator for the system that this client was a part of. The events are thus available for the construction of recovery queues when the client connects back into the system.

6.2.6 Routing events to a re-connected client

The client in question could be both a roaming client or a client which has reconnected after a prolonged disconnect. Associated with the client is the epoch number which was associated with the last event that it received or the last profile change initiated by the client. The routing for the client is based on the node to which the client was last attached to. It is this node that serves as a proxy for the client. If this node fails, it is the cluster gateway of the cluster that the node belonged to which serves as a proxy for the client. As mentioned earlier, in our system a node/unit can fail and remain failed forever.

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 GES context of the clients logical address, the system would use the higher level stable storage to retrieve the client's interim events. Otherwise the system would delegate this retrieval process to the stable storage which services the client's lower level GES context.

Conjecture 6.2.6 *A client's logical address provides the system with the stable storage that should be used for the construction of queues containing events that were missed by the client.*

It is possible that this stable storage is unavailable during a subsequent client reconnect and construction of event queues. From Requirement 6.2.5 it is clear that these storages would recover within some finite amount of time. During such a recovery the system should be able to reconstruct the event queues which it failed to and route the event queues to the client. This requires that –

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

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

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

(a) Its logical address from its previous incarnation.

(b) The last epoch ξ received from the replicator within the replication granularity r_ℓ of the sub system that it was formerly attached to.

    The replication granularity of the sub-system that the client was formerly attached to would have changed. The client however does not need to deal with this. The process of adding finer/coarse grained stable storages ensures that the epoch associated with the client is sufficient to complete a full recovery.

(c) A list of the profile ID's associated with client's profile ω.

Item (a) provide us with the stable storage that has stored events for the client, while item (b) provides us with the precise instant of time from which point on event queues of events needs to be constructed and routed to the client's new location.

Locating the stable store

When the client reconnects back into the system, based on the logical address of the server node that this client was connected to, the stable storage responsible for assigning epochs to clients attached to that particular server node is located. This works as follows, the request is first forwarded to the server node that the client was last attached to. Based on the replication granularity r_ℓ currently available at the node, the recovery request is forwarded to level-ℓ servicing storage. The replication granularity of the sub-system that the client was formerly attached could have been greater than r_ℓ. However the process of adding stable storages accounts for the fact that the epochs would be consistent for recovery. In the event that the server node is down, the information could be retrieved from the cluster gateway for the cluster that the node is a part of. Since a unlike a server node without a stable store, a storage node is not allowed to remain failed forever - the servicing storage will always be retrieved.

The epochs used in the recovery process

Let ξ_n be the last epoch contained in the event routed to a client in its last incarnation. As discussed in section 6.2.3 an event e is stored in the following format - $< \xi^e, (d_0^e, d_1^e, \cdots, d_n^e)$, $(p_0^e, p_1^e, \cdots, p_n^e)$, *e.type*, *e.length*, *e.serialize* $>$. Among the events that have been stored at this stable storage what we are interested in are those events which have an epoch greater than ξ_n. We then compute the second epoch ξ_m associated with the recovery request. This is the epoch at the stable store when the recovery request was received. This epoch indicates the point from which queues need not be constructed, since the real time events would be routed to the client by the sub-system that it is presently attached to. Thus, the set of events which form the preliminary set of events to be considered for recovery are events with epochs greater than ξ_n through ξ_m. The number of events in this preliminary set would be less than or equal to $(m - n)$ since some of intermediate events could have been garbage collected. Within this set of events, the events that could potentially be routed to the client are those for which the unit that the server node is a part of, is one of the destinations. This operation of computing potential recovery events based on the epochs and the destination list can be performed by a simple filter.

Profile ID's and the recovery events

The individual profile predicates $\delta\omega$ corresponding to the profile ID's are marked for removal from the profile graph. Using the profile-ID's we can compute the events that need to be received by the client within the set of recovery events computed in the earlier section. This is very important since with the set computed in the previous section, the number of events that should not be received at the recovering client far exceeds the number of events that should be received by the client. The stable store then proceeds to propagate this removal of the profile ID's both to higher level gatekeepers just as in the profile propagation protocol and also to the lower level gatekeepers down to the server node which last hosted this client.

When the client issues a event recovery process, the logical address of the client is changed to its present address. The recovery events each 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. These recovery events are now managed by the stable store servicing the server node that the client is now attached to. This stable store is responsible for issuing acknowledgements in response to the receipt of the recovery events. Upon receipt of acknowledgements from the new store the corresponding predicate count associated with the event at the old store is decremented by one. If this count is reduced to zero the destination is removed from the destination list, and if the destination list is reduced to zero the event is garbage collected by the stable store. Upon receiving every such recovery event, the epoch associated with the client's profile is advanced to the epoch contained in the latest recovery event that the client received. The epoch in this case would be assigned by the stable store that is presently servicing the client. The profile predicates associated with the client's profile is propagated using the profile propagation protocol.

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. In case of client roam or storage failures during reconnection there's another epoch that is associated with the client. This pertains to the time from which point on events *need not* be routed. Of course every recovery of a failed stable storage is a new epoch, and for clients which couldn't be serviced during the time the storage had failed, this is the epoch from which no events should be used in the construction of local queues.

6.2.7 Advantages of this scheme

This scheme ensures that any given event is received by all the persistent clients that had expressed an interest in it. The scheme withstands the failure of nodes/units, with all nodes within these units remaining failed forever. The only constraint that is imposed is that the stable storage not remain failed for ever, and that it recover within a finite amount of time. In the case of stable storage failures, the higher level stable store would have stored events destined for the unit, and release it to the store when it comes back up. The scheme allows us to respond to node failures (associated link failures), gateway failures and network partitions. When partitions heal the units exchange data destined for each half of the partition. This scheme supports the roaming of clients and also accounts for the garbage collection of events stored onto stable storage in response to clients initiating roam and constructing local queues to receive missed events.

6.3 Summary

In this chapter we presented our replication scheme, and outlined how different nodes within a given super-unit could be served by different stable storages. We laid down the restrictions imposed on the number/location of stable storages within the system. We outlined some of the common failure scenarios involved, and presented a scheme for the detection of partitions that these failure scenarios lead to. We then introduced the concept of epochs, and discussed issues which demonstrate its usefulness. Combining the replication scheme and the concept of epochs, and adding the notion of a system storage we arrived at our scheme for guaranteed delivery. We then describe our scheme for handling messages for disconnected clients and also for clients reconnecting back into the system.

Chapter 7

Application Domains For the Grid Event Service

In this chapter we discuss the possible application domains for the Grid Event Service (GES). In section 7.1 we discuss employing GES for developing collaboratory applications. Section 7.2 describes how GES could be used to provide an illusion of devices at the edge of the internet communicating with each other. Finally in section 7.3 we describe a possible extension to the GES the Grid Event Service Micro Edition (GESME) which handles messages and events for hand held devices.

7.1 Collaboration

Collaboration systems based on the client server model tend to suffer from performance drawbacks due to factors such as scaling and response times. Also the inherent simplicity in these systems, is offset by the fact that they constitute a single point of failure. Distributed collaboration systems such as JSDT [Bur99], ISAAC [LE99] and TANGO [FTD+98] have always relied on the notion of a Session object which is responsible for the taking decisions for a given collaboration session. The *Pragmatic Object Web* [FFPO99, FFOP98] concept offers a combination of object/component reusability with the global access, while setting up a multiple-standards based framework in which the best assets of various approaches complement each other. JDCE [Pal98] employed this concept for collaborative systems with an RMI and CORBA based collaboratory system. However, this too required the concept of a Session object and also in case of the CORBA implementation relied on the establishment of a CORBA Event Channel like Session Object. The TANGO collaboratory system was limited by its support for synchronous communications. Systems such as JSDT and ISAAC provides support for both synchronous and asynchronous style of communications. Approaches to building scalable systems using JSDA (the precursor to JSDT) and JDCE have been described in [FFN+99, DFF+98]. However these systems though distributed all maintain the notion of a Session object, which could serve as a single point of failure for clients who are part of that Session. In this section we proceed to make a case for collaboration systems designed using the Grid Event Service.

In the Grid Event Service, the notion of a Session would be an interest in receiving a certain type of an event, and does not reside on a single node. In traditional collaboration systems the Session is aware of the precise location and number of clients that are registered to a session. In GES the calculation of destination lists is itself a distributed process. GES could be used to provide both the synchronous and asynchronous style of communication. The *roam* features and the delivery guarantees in the presence of server node failures provides for a greater resiliency in collaborative applications. In case of a set of *roam-join*, the client operates in asynchronous mode for missed events and synchronous mode for real time events.

Also typical collaboration applications include clients being part of multiple collaboration sessions.

When the number of sessions increase exponentially the dissemination of content should be a judicious process being able to handle sparse interest in sessions at certain locations and dense interest in others. The dissemination scheme,, comprising of the routing based on the profiles should ensure that the dense and sparse interest cases are handled equally effectively.

7.2 P2P Systems

Another important trend is peer-to-peer computing (P2P) [p2p01] with recent work typified by JXTA [Kay01] from Bill Joy at Sun Microsystems. P2P systems provide a linkage of computers *at the edge* of the Internet. Collaborative systems create P2P networks although in our approach (and most other systems), this is an *illusion* as the P2P environment is created by the routing of messages through a network of servers.

7.3 Grid Event Service Micro Edition

One extension of importance GESME (GES Micro Edition) handles messages and events on hand held and other small devices. This assumes an auxiliary (personal) server or adaptor handling the interface between GES and GESME and offloading computationally intense chores from the handheld device.

Chapter 8

Results

8.1 Experimental Setup

The system comprises of 22 server node processes organized into the topology shown in the Figure 8.1.1. Each server node process is hosted on 1 physical Sun SPARC Ultra-5 machine, with no SPARC Ultra-5 machine hosting two or more server node processes. The run-time environment for the server node processes is JDK-1.2. For the purpose of gathering performance numbers we have 1 publisher in the system and 200 client node processes with 5 client nodes attached to every server node within the system. The 100 client node processes reside on a SPARC Ultra-60 machine. The publisher is responsible for issuing events, while the subscribers are responsible for registering their interest in receiving events. The publisher and the *measuring* subscriber reside on another SPARC Ultra-5 machine.

8.2 Factors to be measured

Once the publisher starts issuing events the factor that we are most interested in is the *latency* in the reception of events. This latency corresponds to the response times experienced at each of the clients. We measure the latencies at the client under varying conditions of *publish rates*, *message sizes* and *matching rates*. Publish rates and message sizes correspond to the rate at which messages are being issued by the publisher and the size of these individual messages respectively. Matching rate is the percentage of events that are actually supposed to be receieved at a client. In most publish subscribe systems, at any given time for a certain number of events being present in the system, any given client is generally interested in a very small subset of these events. Varying the matching rates allows us to simulate such a scenario, and perform measurements under conditions of varying selectivity. For a sample of messages received at a client we calculate the *mean latency* for sample of received messages, the *variance* in the sample of these messages and the *system throughput* measured in terms of the number of messages received per second at the measuring subscriber. We also measure the highest and lowest message latencies within the sample of messages that have been received.

8.2.1 Measuring the factors

For events published by the publisher the number of tag-value pairs contained in every event is 6, with the matching being determined by varying the value contained in the fourth tag. The profile for all the clients in the system, thus have their first 3 <*tag=value*> pairs identical to the first 3 pairs contained in every published event. This scheme also ensures that for every event for which destinations are being computed there is some amount of processing being done. Clients attached to different server nodes specify an interest in the type of events that they are interested in. This matching rate is controlled by the publisher, which publishes events with different footprints. Since we are aware of the footprints for

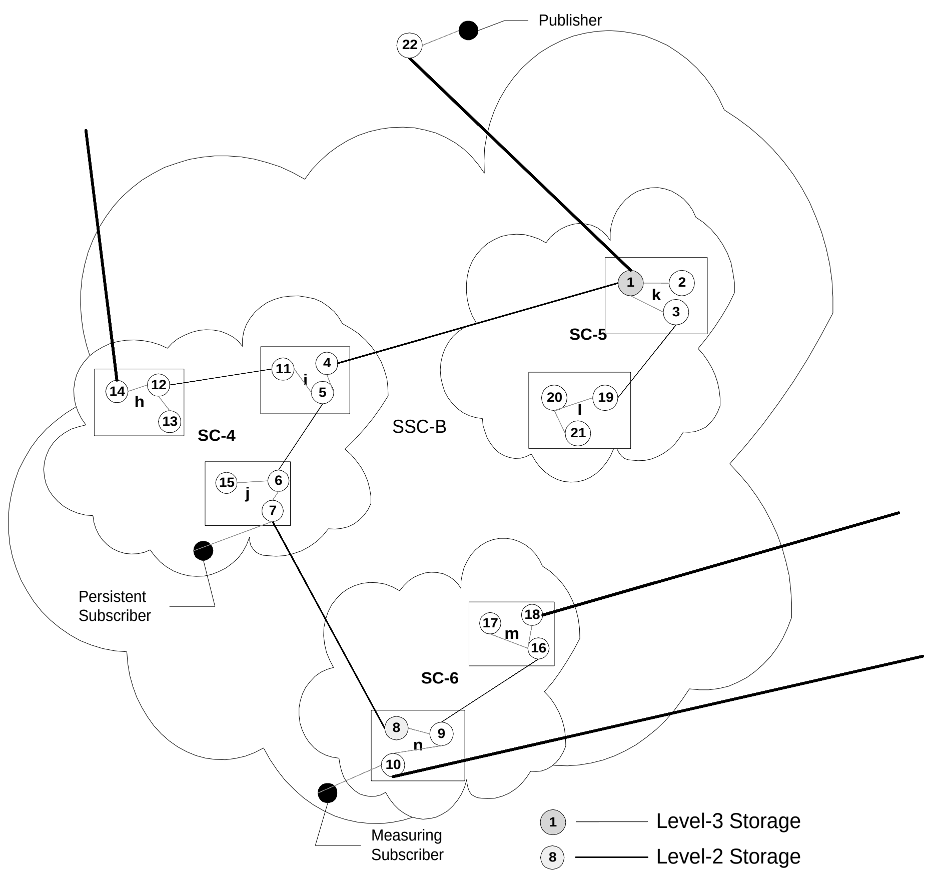


Figure 8.1.1: Testing Topology

the messages published by the publisher, we can accordingly specify profiles, which will allow us to control the dissemination within the system. When we vary the matching rate we are varying the percentage of events published by the publisher that are actually being received by clients within the system. Thus when we say that the matching rate is set at 50%, any given client will receive only 50% of the events published by the publisher. To vary the publish rates, we control the *sleep time* associated with the publisher thread, and also the number of messages that it publishes at a time once the publisher thread *wakes up*. This requires some preliminary tuning, once the values for the *sleep time* and the number of messages that are published at a time have been fixed for the publisher and the corresponding server node in question, we proceed to compute the real publish rates for the sample of messages that we send. This is the publish rate that we report in our results.

For each matching rate we vary the size of the messages from 30 to 500 bytes, and vary the publish rates at the publisher from 1 to 1000 Messages per second. For each of these cases we measure the latencies in the reception of events. To compute latencies we have the publishing client and one the *measuring*subscriber residing on the same machine. Event's issued by the publisher are *time-stamped* and when they are received at the subscribing client the difference between the *present time* and the *time-stamp* contained in the received message constitutes the latency in the dissemination of the event at the subscriber via the server network. In case the publisher and the subscriber were on two different machines, with acess to different underlying system clocks we would need to synchronize the clocks and also account for the drift in clock rates prior to computing the latencies in message reception. Having the publisher and one of the subscribers on the same physical machine with access to the same underlying clock, obviates this need for clock synchronization and also accounts for clock drifts. It should be noted that though the publisher and the *measuring* subscriber are on the same machine, they are connected to two different server nodes within the server network, as depicted in figure 8.1.1. In fact it takes 10 hops for a message issued by the publisher to be received at the measuring subscriber.

8.3 Discussion of Results

In this section we discuss the latencies gathered for varying values of publish rates, message sizes and matching rates. We then proceed to include a small discussion on system throughputs at the clients, and another discussion that outlines the trends in variance in latencies of messages received at a client. The results also discuss the latencies involved in the delivery of events to persistent clients in units with different replication schemes.

8.3.1 Latencies for the routing of events to clients

At high publish rates and increasing message sizes, the effects of queuing delays come into the picture. This queuing delay is a result of the messages being added to the queue faster than they can be processed. In general, the mean latency associated with the delivery of messages to a client is directly proportional to the size of the messages and the rate at which these messages were published. The latencies are the lowest for smaller messages issued at low publish rates. The mean latency is further influenced by the matching rates for events issued by the publisher. The results clearly demonstrate the effects of flooding/queuing that take place at high publish rates and high message sizes and high matching rates at a client. It is clear that as the matching rate reduces the latencies involved also reduce, this effect is more pronounced for cases involving messages of a larger size at higher publish rates.

Figures 8.3.1 through 8.3.7 depict the pattern of decreasing latencies with decreasing matching rates. The latencies vary from 391.85 *mSecs* to 48.2 *mSecs* with the <*publish rate, message size*> varying from <952 *messages/Sec* , 450 *Bytes*> for a matching rate of 100% to <961 *messages/Sec*, 425 *Bytes*> for a matching rate of 5%. This reduction in the latencies for decreasing matching rates, is a result of the routing algorithms that we have in place. These routing algorithms ensure that events are routed only to those parts of the system where there are clients which are interested in the receipt of those events. Thus events are queued only at those server nodes which

- Have attached clients interested in those events
- Are en route to server nodes which are interested in these events. These server nodes generally fall in the shortest path to reach the destination node.

In the flooding approach, all events would still have been routed to all clients irrespective of the matching rates.

Figure 8.3.1 depicts the case for matching rates of 100%. In this case the mean latency for the sample of messages varies from 15.54 *mSec* for <1 *message/Sec*, 50 *Bytes*> at a throughput of 1 message/Sec to 391.85 *mSec* for <952 *messages/Sec*, 450 *Bytes*> with a throughput of 78 *messages/Sec* at the client. The variance in the sample of messages varies from 2.3684 $mSec^2$ to 69,713.93 $mSec^2$ for the 2 cases respectively. The maximum throughput achieved was 480.76 *messages/Sec* at publish rates of 492 *messages/Sec* with messages of size 75 bytes.

Figure 8.3.2 depicts the case for matching rates of 50%. In this case the mean latency for the sample of messages varies from 13.02 *mSec* for <20 *messages/Sec*, 50 *Bytes*> to 178.66 *mSec* for <952 *messages/Sec*, 350 *Bytes*>. The variance in the sample of messages varies from 56.8196 $mSec^2$ to 14,634 $mSec^2$ for the 2 cases respectively.

Figure 8.3.3 depicts the case for matching rates of 33%. In this case the mean latency for the sample of messages varies from 15.18 *mSec* for <20 *messages/Sec*, 50 *Bytes*> to 95.969 *mSec* for <952

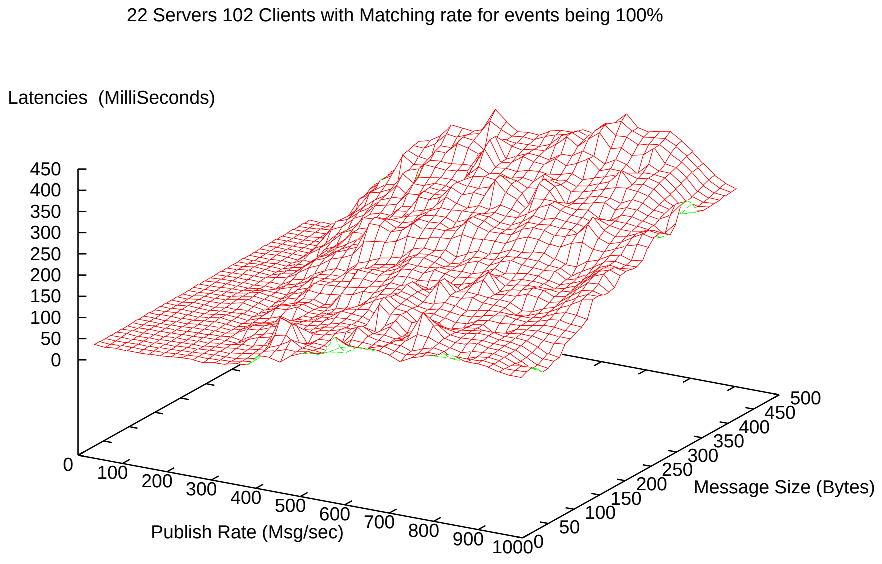


Figure 8.3.1: Match Rates of 100

messages/Sec, 425 *Bytes*>. The variance in the sample of messages varies from 26.57 *mSec*^{2} to 1,263 *mSec*^{2} for the 2 cases respectively.

Figure 8.3.4 depicts the case for matching rates of 25%. In this case the mean latency for the sample of messages varies from 14.40 *mSec* for <20 *messages/Sec*, 50 *Bytes*> to 66.6 *mSec* for <961 *messages/Sec*, 400 *Bytes*>. The variance in the sample of messages varies from 0.24 *mSec*^{2} to 587.04 *mSec*^{2} for the 2 cases respectively.

Figure 8.3.5 depicts the case for matching rates of 20%. In this case the mean latency for the sample of messages varies from 15.35 *mSec* for <20 *messages/Sec*, 50 *Bytes*> to 62.35 *mSec* for <952 *messages/Sec*, 400 *Bytes*>. The variance in the sample of messages varies from 12.027 *mSec*^{2} to 312 *mSec*^{2} for the 2 cases respectively.

Figure 8.3.6 depicts the case for matching rates of 10%. In this case the mean latency for the sample of messages varies from 14.40 *mSec* for <20 *messages/Sec*, 50 *Bytes*> to 52.0 *mSec* for <952 *messages/Sec*, 400 *Bytes*>. The variance in the sample of messages varies from 0.44 *mSec*^{2} to 103 *mSec*^{2} for the 2 cases respectively.

Figure 8.3.7 depicts the case for matching rates of 5%. In this case the mean latency for the sample of messages varies from 14.0 *mSec* for <20 *messages/Sec*, 50 *Bytes*> to 47.6 *mSec* for <961 *messages/Sec*, 425 *Bytes*>. The variance in the sample of messages varies from 0.44 *mSec*^{2} to 87.44 *mSec*^{2} for the 2 cases respectively.

8.3.2 System Throughput

We also depict the system throughputs at the client under conditions of varying message sizes and publish rates. We choose to depict the system throughputs at a Matching rate of 100% since at other matching rates only the relevant events are being routed to the clients, and thus does not reveal the true throughputs that can be achieved at a client. Figure 8.3.8 depicts the system throughputs achieved at a client under conditions of different publish rates and message sizes. The maximum throughput achieved was 480.76 *messages/Sec* at publish rates of 492 *messages/Sec* with messages of size 75 bytes.

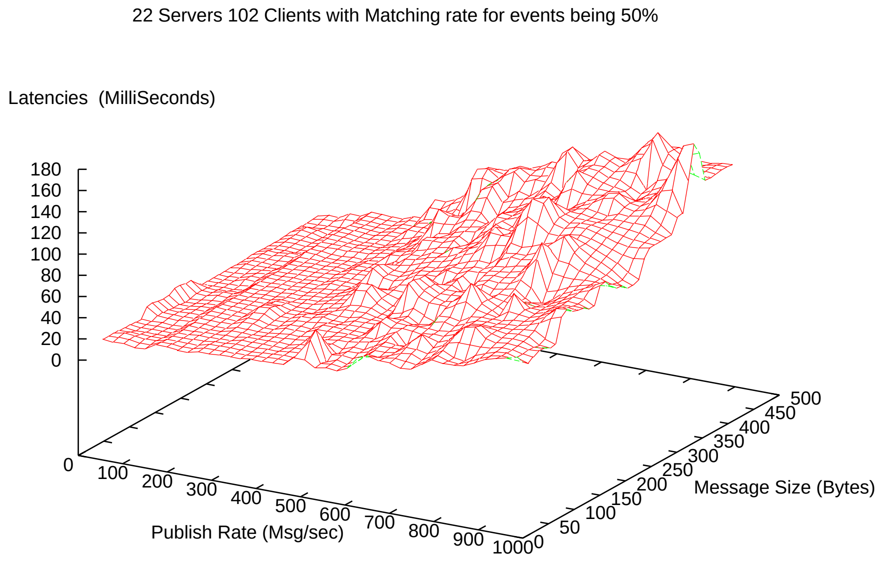


Figure 8.3.2: Match Rates of 50

8.3.3 Variance

Variance for the sample of received messages at a client, demonstrate how queueing delays can add up to increase the mean latency, and also how this mean latency has high deviations from the highest and lowest latencies contained in the sample of latencies for messages received at a client. The variance in the sample of messages varies from 69713 $mSec^2$ to 133.76 $mSec^2$ for <952 *messages/Sec* , 450 *Bytes*> at matching rates of 100% to <877 *messages/Sec*, 450 *Bytes*> at matching rates of 5%. Thus variance in the sample of messages for higher message sizes at higher publish rates also reduces with decreasing matching rates for the published events.

8.3.4 Persistent Clients

In figure 8.1.1 we have also outlined the replication scheme that exists in the system. When an event arrives at node **1** the event is first stored to the level-3 stable store so that it has an epoch associated with it. The event is then forwarded for dissemination within the unit. Clients attached to node in super-cluster **SC-6** have a replication granularity of r_2, thus when the events issued by the publisher in the test topology is being disseminated when clients attached to nodes in **SC-6** receive that event, the event would have been replicated twice. For testing purposes we set up another *measuring* subscriber at node **7** in addition to the subscriber that we would set up at node **10**. When an event is received by the subscriber attached to node **7** the event would have been replicated only once, at node **1**. These *measuring* subscribers allow us to measure the response times involved for singular and double replications experienced at clients attached to nodes **7** and **10** respectively. Every node in the system has 5 persistent clients attached to it, for a total of 102 persistent clients. The publisher and the 2 *measuring* subscribers are all hosted on the same machine for reasons discussed earlier. Figures 8.3.9 and 8.3.10 depict the latencies in delivery of events at persistent clients, with singular and double replications.

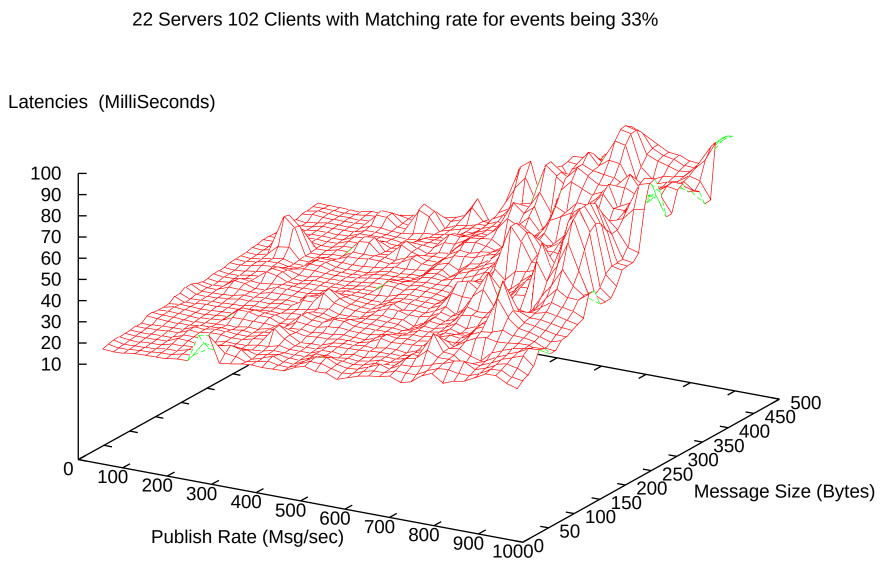


Figure 8.3.3: Match Rates of 33

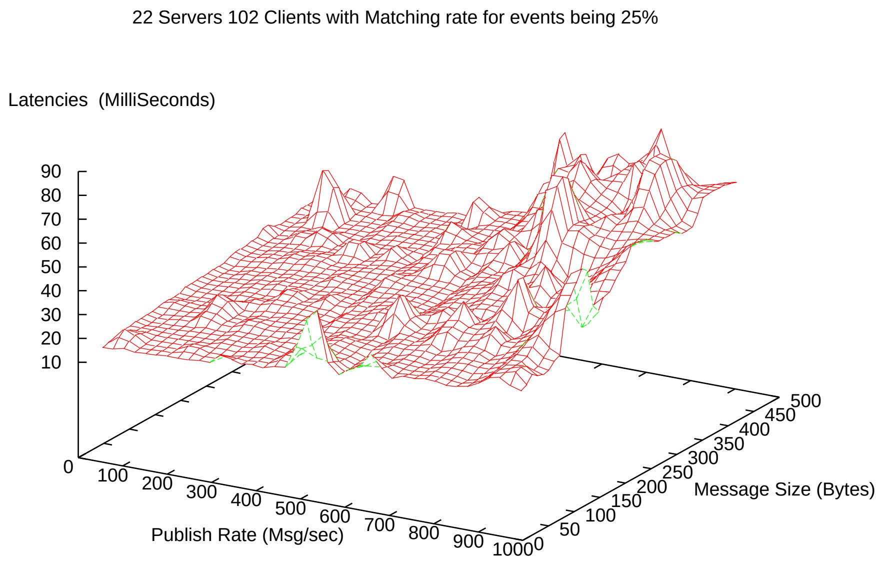


Figure 8.3.4: Match Rates of 25

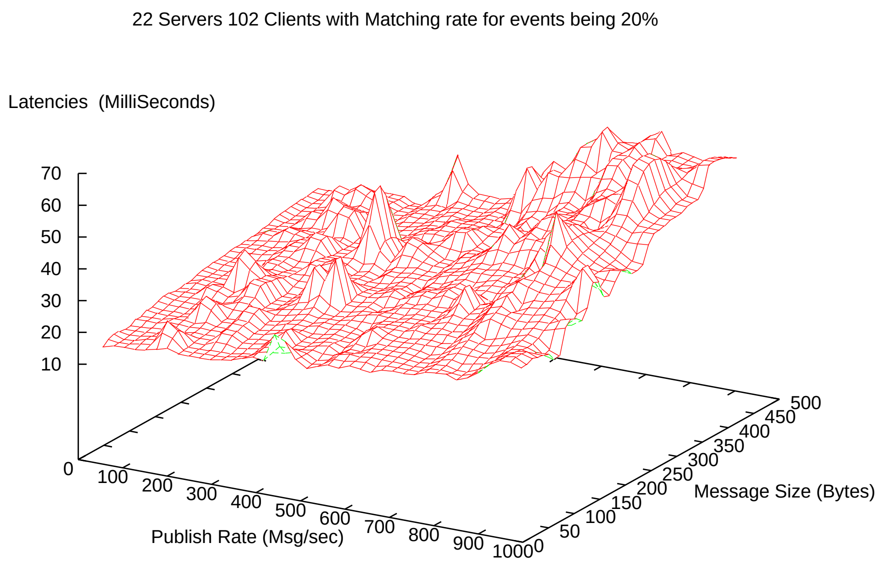


Figure 8.3.5: Match Rates of 20

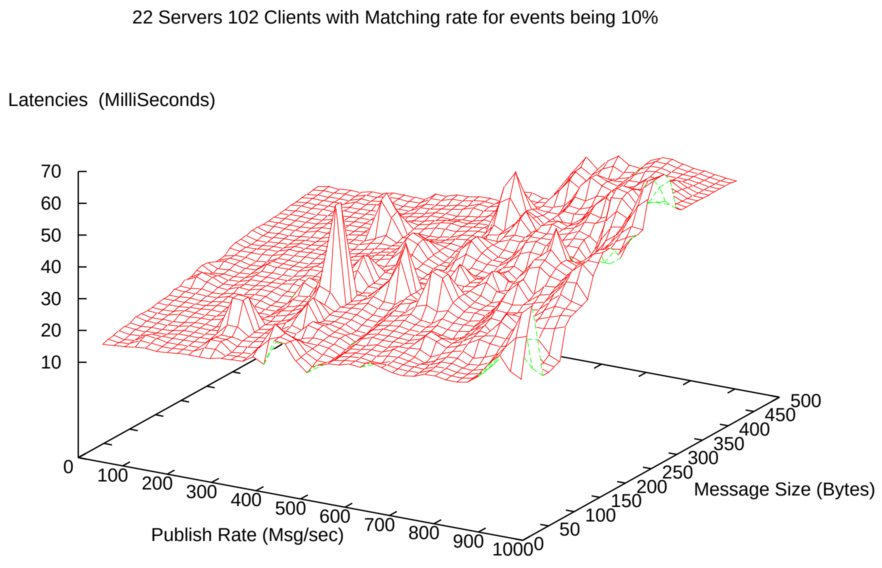


Figure 8.3.6: Match Rates of 10

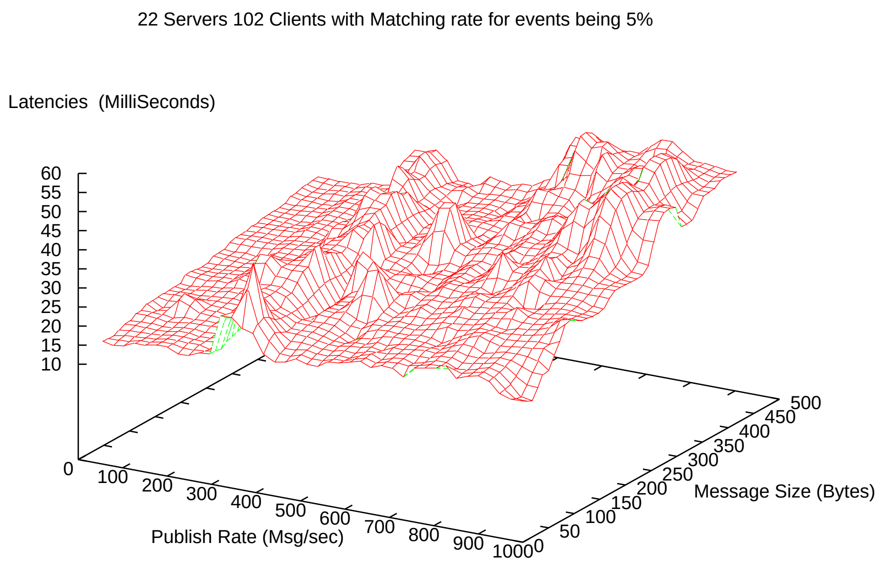


Figure 8.3.7: Match Rates of 5

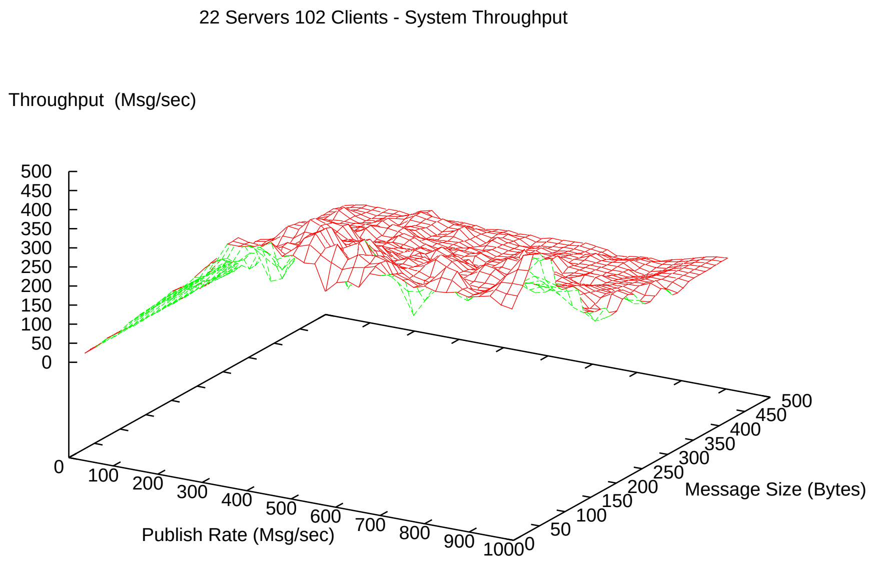


Figure 8.3.8: System Throughput

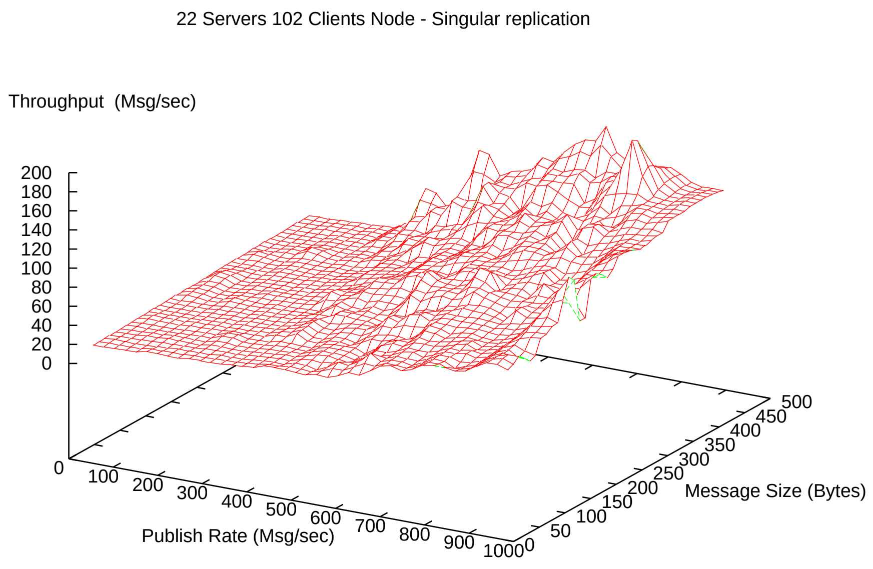


Figure 8.3.9: Match Rates of 50% - Persistent Client (singular replication)

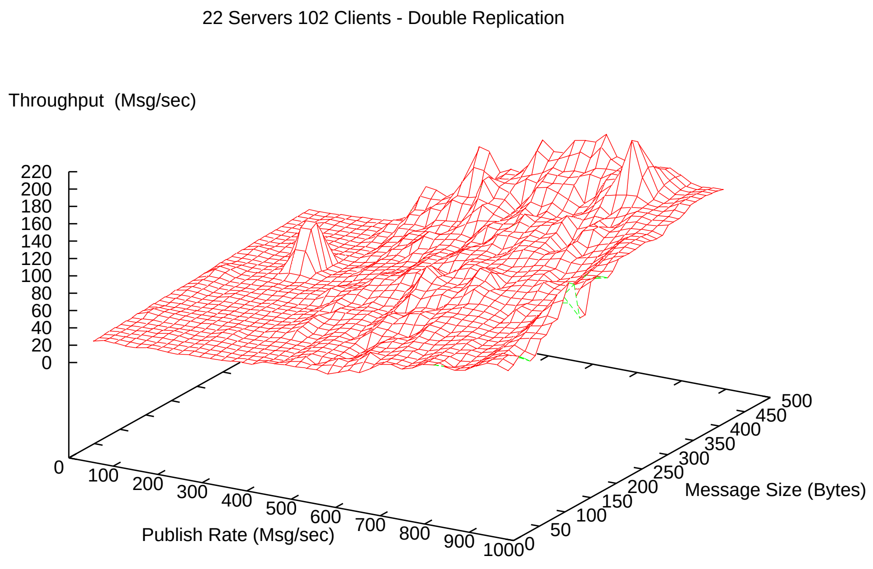


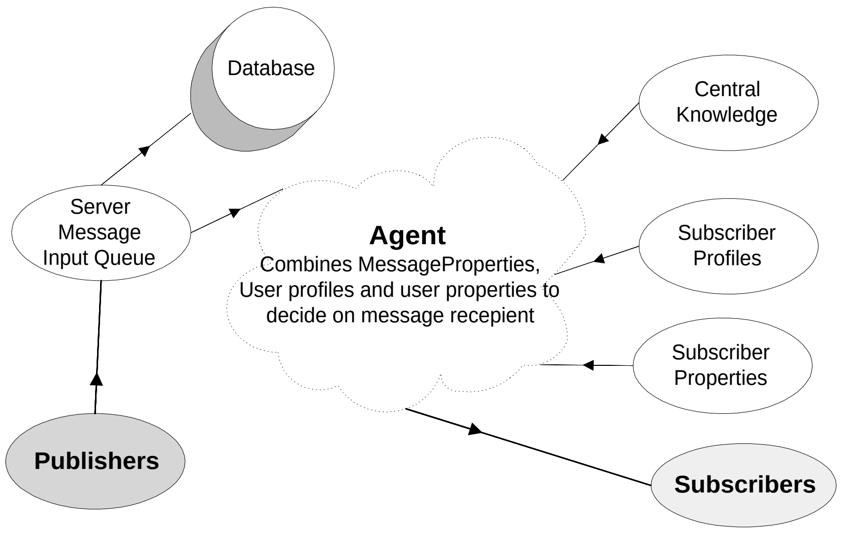
Figure 8.3.10: Match Rates of 50% - Persistent Client (double replication)

Chapter 9

Future Directions

9.1 GMS software architecture

A client's profile comprises of a set of predicates which the client mandates that a certain event satisfy prior to the client being targetted as one of the destinations for the event. In addition to this associated with the client are a set of properties which could be used to further refine the destinations associated with an event. The refinement process is carried out by a server side agent responsible for further refining the events targetted for a client. This scheme is depcited in figure 9.1.1.



Database
Central Knowledge
Server Message Input Queue
Agent
Combines MessageProperties, User profiles and user properties to decide on message recepient
Subscriber Profiles
Subscriber Properties
Publishers
Subscribers

Figure 9.1.1: An agent based approach

9.1.1 Object-based Matching

Traditionally matching of events and calculation of destinations have been based on text properties with SQL like selections or on a static set of *tag-value* pairs contained in the client's profile. JMS employs the earlier approach while most content based pub/sub systems employ the latter approach. We seek to augment this matching process by allowing for topics to be matched to clients based on not just the profiles, but also the properties associated with the client. In addition to the matching based on string properties or tag-value matching, the advantage of this scheme is that it allows for matching to be based on more dynamic features like the state of the system (bandwidth constraints etc.), a client's content handling capabilities and other similar constraints . This operation is performed by a server side agent which is responsible for this more powerful matching. We define published topics and subscriber profiles and device properties in an XML syntax GXOS which is interpreted by the matching agent. As summarized in next section, GXOS also can be used to describe GMS messages and application meta-data. The decision to route an event is based not only the properties contained in the event, but also on the constraints specified/detected within the user property set. As an example an event would be routed based on not just the headers describing the event but also on the clients content handling capabilities. Thus we would use the publish/subscribe matching engine for routing, but we will narrow the destination lists associated with the event based on the client's content handling capabilities.

9.1.2 The execution Model - GXOS, MyXoS & RDF

We are developing GMS as the message service to support a collaborative portal to be used in education and computing. All objects in the system are self defining and the objects and meta-data describing them are separately managed. This allows us to useful powerful technologies for managing the meta-objects while using classic high performance computing approaches for the raw objects. This allows to combine high performance with high functionality. The object model GXOS specifies three types of meta-objects

(a) The events in GMS.

(b) The resources (users, computers, programs, educational curricula etc.)

(c) The user view or portalML including object rendering, portal layout and subscription profiles.

The environment MyXoS that manages these objects and meta-objects is driven by XML commands - initially in an RDF (Resource Description Framework) syntax. GMS is used both for asynchronous and synchronous collaboration – it handles asynchronous information channels, the updates for shared display or shared event collaborations and the Jabber [SA01] instant messenger built into our collaborative framework.

We are currently researching different ways of reading into memory the XML meta-objects as needed by programs running under MyXoS. SAX and DOM XML parsers are not efficient for tens of millions of XML instances at a time. Converting XML schema into Java data structures is possible [exo01] but efficiency requires this be combined with *lazy* parsing so that we expand GXOS trees only as needed to refine our access. We see this as a particularly challenging and having important programming style implications as we look at models where data structures are defined in XML and not directly as C++ or Java classes. Further papers will describe these parts of our research.

9.2 The DTD for the event

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

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

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

To work around items (a) and (b) we employ the following approach. The first time that an event type is encountered at a server node, the DTD definition is fetched[1] and cached at the server node. Thus

[1] 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.

we circumvent the network operation.

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

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

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

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

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

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

Earlier we discussed our approach to circumventing network operations while parsing the XML events. DTD's could however change, and the cache rendered useless, to account for this scenario we need to include the concept of version Number within the DTD fields. When the event is parsed a look at the `Mspaces:versionNumber` field could tell us if the cache needs to be updated. If the DTD definition for the event is changed the clients interested in the events conforming to the old DTD definition need to be notified about this change. These clients could then decide if their profiles need to be updated to reflect this change. This notification of the change in the DTD of the event that a client is interested is included in the field `Mspaces:LatestVersionNumber`. Also nodes need to maintain the DTD definitions for different versions of the same DTD. It is conceivable that there are events being published within the system or there are recovery events which would conform to the old versions of the DTD. Information regarding these version numbers along with the security constraints and liveness indicator constitute `Mspaces:EventType`.

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

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

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

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

A brief note about the `Mspaces:EventLinkage` is in order. If an event is allowed to be part of multiple bundles within the same stream with multiple BundleNumber’s the `?` should be `*` in the `Mspaces:BundleId, Mspaces:BundleNumber` grouping. The listing of the DTD in section 9.2.1 and element analysis in table 9.2.1 assumes the `?` occurrence operator.

9.2.1 The complete DTD

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

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

<!ELEMENT Mspaces:StreamId         (#PCDATA)>

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

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

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


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

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

Table 9.2.1 depicts the various elements, the nested elements and occurrence bounds for the nested elements within a specific element. The table also snapshots our discussions so far with brief descriptions of the purpose of each element with the event element hierarchy.

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

Table 9.2.1: Mspaces:Event Hierarchy

Chapter 10

Conclusion

In this thesis we have presented the Grid Event Service (GES), a distributed event service designed to run on a very large network of server nodes. GES comprises of a suite of protocols, which are responsible for the organization of nodes, creation of abbreviated system views, management of profiles and the hierarchical dissemination of content based on these profiles. These protocols exchange information collected and processed by the other protocols. Thus when a new connection is added the information is used to update the connectivity graph, which is used to identify the relevant nodes for the propagation of profiles to. This information contained in the profile graphs is used for the hierarchical dissemination of content. All these protocols can run concurrently, adding a lot of flexibility to the overall system.

The system views at each of the server nodes respond to changes in system inter-connections, aiding in the detection of partitions and the calculation of new routes to reach various units. The protocols ensure that the only events routed to a node are those, which satisfy the profile of at least one of the clients attached to the server node, or the node is en route to destinations that do so. Thus the protocols in GES ensure that the routing is intelligent and can handle sparse/dense interest in certain sections of the system. GES's ability to handle the complete spectrum of interests equally well, lends itself as a very scalable solution and conditions of varying publish rates, matching rates and message sizes.

We also provided a scheme for the creation of merged streams from a set of related streams. This delivery of merged streams is done by the resolution of spatial and chronological constraints that exist between events in multiple related streams. The thesis outlined a scheme for the delivery of events in the presence of failures. The replication strategy, epochs associated with received events and profile ID's associated with client profiles allowed us to account for a very precise recovery of events for clients with prolonged disconnects or those which have roamed the network.

GES is intended to be a part of the Grid Collaborative Portal (GCP). The features of location transparency, intelligent routing, replication and persistent delivery of events lends itself very easily to aid in the development of the GCP. The application domains into which GES can be easily extended include collaborative systems, peer-to-peer (P2P) systems and messaging for hand-held devices such as the Grid Event Server Micro Edition (GESME). GESME, which would account for the conversion of GCP messages to be handled by the hand-held devices could add considerable value to GES.

The location transparency feature contained within GES where a client can roam the network in response to failure suspicions (correct or incorrect) or re-join the system after a prolonged disconnect, and attach itself to any node within the system and still receive all the events it was supposed to receive is a significant contribution. The failure model of the system, which allows a server node or a unit/super-unit of server nodes to fail and remain failed forever and still satisfy delivery guarantees is another significant contribution which also allows the system to be easily extensible. The replication strategy presented in this thesis, could be augmented to include mirror storages, which maintain information identical to that of the stable storages, and take over in the event of stable storage failures. This feature would add an additional robustness and reduce the time required to recover from stable storage failures.

The results in Chapter 8 demonstrated the efficiency of the routing algorithms and confirmed the advantages of the dissemination scheme which intelligently routes messages. Industrial strength JMS solutions, which support the publish subscribe paradigm generally are optimized for a small network of servers. The seamless integration of multiple server nodes and the failure model that we impose on server nodes without stable storages provides for easier maintenance of the server network.

The explosive developments in the area of pervasive computing have resulted in the need for building distributed network centric services. However these network services should also be easily extensible, scalable and have a reasonable failure model, which causes a denial of service only under the most extreme of failure scenarios. But the most important feature is the ability to access another part of this distributed service for better response times or convenience. This thesis makes significant contributions towards providing a solution to this issue.

Bibliography

[AAB$^+$00] Mark Astley, Joshua Auerbach, Guruduth Banavar, Lukasz Opyrchal, Rob Strom, and Daniel Sturman. Group multicast algorithms for content-based publish subscribe systems. In *Middleware 2000*, New York, USA, April 2000.

[Aka00] Akamai. Delivering a better Internet - Technology. http://www.akamai.com, 2000.

[AOS$^+$99] Ken Arnold, Bryan O'Sullivan, Robert W. Scheifler, Jim Waldo, and Ann Wollrath. *The Jini Specification*. Addison-Wesley, June 1999.

[ASS$^+$99] Marcos Aguilera, Rob Strom, Daniel Sturman, Mark Astley, and Tushar Chandra. Matching events in a content-based subscription system. In *Proceedings of the 18th ACM Symposium on Principles of Distributed Computing*, May 1999.

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

[BCM$^+$99] Gurudutt Banavar, Tushar Chandra, Bodhi Mukherjee, Jay Nagarajarao, Rob Strom, and Daniel Sturman. An Efficient Multicast Protocol for Content-Based Publish-Subscribe Systems. In *Proceedings of the IEEE International Conference on Distributed Computing Systems*, Austin, Texas, May 1999.

[BEGS00] Romain Boichat, Patrick Th. Eugster, Rachid Guerraoui, and Joe Sventek. Effective Multicast programming in Large Scale Distributed Systems: The DACE Approach. *Concurrency: Practice and Experience*, 2000.

[BF96] Ken Birman and Roy Friedman. Trading consistency for availability in distributed systems. Technical Report TR 96-1579, Dept. Of Computer Science, Cornell University, Ithaca, NY-14853, April 1996.

[BG00] Dan Brickley and R.V. Guha. Resource Description Framework (RDF) Schema Specification 1.0. Technical report, W3C, March 2000.

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

[Bir93a] Kenneth Birman. The process group approach to reliable distributed computing. *Communications of the ACM*, 36(12):36–53, 1993.

[Bir93b] Kenneth Birman. A response to cheriton and skeen's criticism of causal and totally ordered communication. Technical Report TR 93-1390, Dept. Of Computer Science, Cornell University, Ithaca, NY 14853, October 1993.

[BM89] Kenneth Birman and Keith Marzullo. The role of order in distributed programs. Technical Report TR 89-1001, Dept. Of Computer Science, Cornell University, Ithaca, NY 14853, May 1989.

[BPSMM00] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, and Eve Maler. Extensible Markup Language (XML) 1.0 (Second Edition). Technical report, W3C, October 2000.

[Bur99] Rich Burridge. *Java Shared Data Toolkit User Guide*. Sun Microsystems, 2.0 edition, October 1999.

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction To Algorithms*. McGraw-Hill Book Company, 1990.

[Cor99] Progress Software Corp. SonicMQ :The role of Java messaging and XML In Enterprise application Integration. Technical report, http://www.progress.com/sonicmq, October 1999.

[Cor00a] Firano Corporation. A Guide to Understanding the Pluggable, Scalable Connection Management (SCM) Architecture - White Paper. Technical report, http://www.fiorano.com/products/fmq5_scm_wp.htm, 2000.

[Cor00b] Talarian Corporation. Everything you need to know about middleware: Mission critical interprocess communication. Technical report, http://www.talarian.com/ products/smartsockets, 2000.

[CRW00a] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Achieving scalability and expressiveness in an internet-scale event notification service. In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 219–227, Portland OR, USA, July 2000.

[CRW00b] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Content-based addressing and routing: A general model and its application. Technical Report CU-CS-902-00, Department of Computer Science, University of Colorado, Jan 2000.

[DFF$^+$98] Daniel Dias, Geoffrey Fox, Wojtek Furmanski, Vishal Mehra, Balaji Natarajan, H.Timucin Ozdemir, Z. Odcikin Ozdemir, and Shrideep Pallickara. Exploring JSDA, CORBA and HLA based MuTechs for Scalable Televirtual (TVR) Environments . In *Virtual Reality Modeling Language Symposium - VRML98*, Monterey, California, February 1998.

[DM96] D Dolev and D Malki. The transis approach to high-availability cluster communication. In *Communications of the ACM*, volume 39(4). April 1996.

[DVM$^+$97] Bhatia D, Burzevski V, Camuseva M, Fox G, Premchandran G, and Furmanski W. WebFlow-A Visual Programming Paradigm for Web/Java-based Coarse Grain Distributed Computing. *Concurrency: Practice and Experience*, 9(6):555–577, June 1997.

[EE98] Guy Eddon and Henry Eddon. Understanding the DCOM Wire Protocol by Analyzing Network Data Packets. *Microsoft Systems Journal*, March 1998.

[exo01] exolab.org. Castor: The Source Code Generator. http://castor.exolab.org/sourcegen.html, 2001.

[FFN$^+$99] G.C. Fox, W. Furmanski, B. Natarajan, H. T. Ozdemir, Z. Odcikin Ozdemir, S. Pallickara, and T. Pulikal. Integrating Web, Desktop, Enterprise and Military Simulation Technologies to Enable World-Wide Scalable Televirtual Environments. *Information and Security: An International Journal*, Volume 3, 1999.

[FFO98] G Fox, W Furmanski, and H Ozdemir. JWORB-Java Web Object Request Broker for Commodity Software based Visual Dataflow Metacomputing Programming Environment. In *Seventh IEEE Symposium on High Performance Distributed Computing HPDC7*, Chicago, IL, July 1998.

[FFOP98] G.C. Fox, W. Furmanski, H. T. Ozdemir, and S. Pallickara. New Systems Technologies and Software Products for HPCC: Volume III - High Performance Commodity Computing on the Pragmatic Object Web . Technical report, Research Consortium Inc, June 1998.

[FFPO99] G. C. Fox, W. Furmanski, S. Pallickara, and H. Ozdemir. *Online book: Building Distributed Systems on the Pragmatic Object Web- The Best of Web, Java, CORBA and COM.* http://www.npac.syr.edu/projects/webtech/index.htm. 1st edition, July 1999.

[For94] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. Technical report, Message Passing Interface Forum, May 1994.

[FTD$^+$98] G. Fox, Scavo T., Bernholdt D., Markowski R., McCracken N., Podgorny M., Mitra D., and Malluhi Q. Synchronous Learning at a Distance: Experiences with TANGO Interative. In *Supercomputing 98*, November 1998.

[gnu00] Gnutella. http://gnutella.wego.com, 2000.

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

[GS95] John Gough and Glenn Smith. Efficient recognition of events in a distributed system. In *Proceedings 18th Australian Computer Science Conference (ACSC18)*, Adelaide, Australia, 1995.

[GWvB$^+$00] Steven D. Gribble, Matt Welsh, Rob von Behren, Eric A. Brewer, David Culler, N. Borisov, S. Czerwinski, R. Gummadi, J. Hill, A. Joseph, R.H. Katz, Z.M. Mao, S. Ross, and B. Zhao. The Ninja Architecture for Robust Internet-Scale Systems and Services . In *Special Issue of Computer Networks on Pervasive Computing.* 2000.

[HBS99] Mark Happner, Rich Burridge, and Rahul Shrama. Java message service. Technical report, Sun Microsystems, November 1999.

[HLS97] T.H. Harrison, D.L. Levine, and D.C. Schmidt. The design and performance of a real-time CORBA object event service. In *Proceedings of the OOPSLA'97*, Atlanta, Georgia, October 1997.

[Hou98] Peter Houston. Building distributed applications with message queuing middleware - white paper. Technical report, Microsoft Corporation, 1998.

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

[IBM00] IBM. IBM Message Queuing Series. http://www.ibm.com/software/mqseries, 2000.

[Inc00] Softwired Inc. iBus Technology. http://www.softwired-inc.com, 2000.

[iPl00] iPlanet. Java message queue documentation. Technical report, http://docs.iplanet.com/docs/manuals/javamq.html, 2000.

[Jav99] Javasoft. Java Remote Method Invocation - Distributed Computing for Java (White Paper). http://java.sun.com/marketing/collateral/javarmi.html, 1999.

[Kay01] Kammie Kayl. JOY POSES JXTA INITIATIVE: Pushing the Boundaries of Distributed Computing. Technical report, Sun Microsystems, February 2001.

[LE99] Jackson L. and Grossman E. Integration of synchronous and asynchronous collaboration activities. In *ACM Computing Surveys*, volume 31 (2es). ACM, June 1999.

[LS98] Paul J. Leach and Rich Salz. UUIDs and GUIDs. Technical report, Network Working Group, February 1998.

[LS99] Ora Lassila and Ralph R. Swick. Resource Description Framework (RDF) Model and Syntax Specification. Technical report, W3C, February 1999.

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

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

[OMG00c] The Object Management Group OMG. Omg's corba services. http://www.omg.org/ technology/documents/, June 2000. Version 3.0.

[Ora01] Andy Oram, editor. *Peer-To-Peer - Harnessing the Benefits of a Disruptive Technology.* O'Reilly & Associates, Inc., 1.0 edition, March 2001.

[p2p01] The O'Reilly Peer-to-Peer Conference. http://conferences.oreilly.com/p2p, February 2001.

[Pal98] Shrideep B. Pallickara. Java distributed collaborative environment as test–bed for distributed object technologies. Masters thesis, Syracuse University, August 1998.

[PF01] Shrideep Pallickara and Geoffrey Fox. Towards A Grid Message Service. Submitted to the Tenth IEEE International Symposium on High Performance Distributed Computing (HPDC), San Francisco, California, August 2001.

[PSS99] Shrideep Pallickara, Rob Strom, and Daniel Sturman. Algorithms for reliable delivery in content based publish subscribe systems. Work done over Spring/Summer 99 at the IBM Watson Research Center, November 1999.

[RBM96] R Renesse, K Birman, and S Maffeis. Horus: A flexible group communication system. In *Communications of the ACM*, volume 39(4). April 1996.

[RML95] Rajkumar R, Gagliardi M, and Sha L. The Real-Time Publisher/Subscriber Communication for Inter-Process Communication in Distributed Real-Time Systems. In *The First IEEE Real-time Technology and Applications Symposium*, May 1995.

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

[SA97] Bill Segall and David Arnold. Elvin has left the building: A publish/subscribe notification service with quenching. In *Proceedings AUUG97*, pages 243–255, Canberra, Australia, September 1997.

[SA01] Peter Saint-Andre. Jabber technology overview. Technical report, Jabber.org, March 2001.

[SAB$^+$00] Bill Segall, David Arnold, Julian Boot, Michael Henderson, and Ted Phelps. Content based routing with elvin4. In *Proceedings AUUG2K*, Canberra, Australia, June 2000.

[Sch90] Fred Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. In *ACM Computing Surveys*, volume 22(4), pages 299–319. ACM, December 1990.

[TIB99] TIBCO. TIB/Rendezvous White Paper. http://www.rv.tibco.com/whitepaper.html, 1999.

[WS00] D.J. Watts and S.H. Strogatz. Collective Dynamics of 'Small-World' Networks. *Nature*, 393:440, 2000.