The Grid Event Service (GES) Framework: Research Directions & Issues

Shrideep Pallickara* Geoffrey C. Fox†

Abstract

The Grid Event Service (GES) 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. Server nodes can fail and remain failed forever. Clients need not wait for the failed server nodes to recover. Affected clients can then roam to a new location and thus not experience any denial of service.

GES provides 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. This paper also outlines the GES publish/subscribe model, the natural extension of GES to support dynamic topologies and also the security framework that needs to be in place.

Contents

*Department of Electrical Engineering &Computer Science, Syracuse University
†Department of Computer Science, Indiana University

List of Figures

List of Tables

1 Introduction

Developments in the pervasive computing area, have led to an explosion in the number of devices that users employ to communicate with each other and also to access services that they are interested in. These devices have different computing and content handling capabilities. Further, these users do not maintain an online presence at all times and access services after prolonged disconnects. The channel employed to access services may have different bandwidth constraints depending on the channel type and also on the service being accessed. The system we are considering needs to support communications for 10^9 devices. The users using these devices would be interested in peer-to-peer (P2P) style of communication, business-to-business (B2B) interaction or a system comprising of agents where discoveries are initiated for services from any of these devices. Finally, some of these devices could also be used as part of a computation. The devices are thus part of a complex distributed system. In our model we have the notion of logical clients. These logical clients exist since there could be one or more devices that belong to a single physical user. 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 type of events and servicing that they are interested in.

To support the large number of clients that exist within the system we have a distributed network of servers. These servers are responsible for intelligently routing events within the system. This routing should be such that an event is routed to a server only if it is en route to a valid destination for the event. 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 a 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 is triggered. A highly available distributed solution would have data replication at various server nodes in the network. Solving issues of consistency while executing operations, in the presence of replication, leads to a model where other server nodes can service a client despite certain server node failures. The underlying network that we consider for our problem is one made up of the nodes that are hooked onto the Internet or Intranets.

Communication in the system is through events, which are encapsulated within messages. 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. Clients provide services to other clients using events. These events are routed by the system based on the service advertisements that are contained in the messages published by the client. Events routed to a broker are queued and routing decisions are made based on the service advertisements contained in these events and also based on the state of the network fabric. The network fabric changes with the addition or removal of connections some of which are due to the addition/failure of broker nodes. We assume that the nodes, which participate in the event delivery can experience crash failures or be slow. Similarly the links connecting these nodes may fail or get overloaded. Further failed nodes and connections can remain failed forever. 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. The failure model in which servers fail and remain failed forever allows the system to support dynamic topologies. Servers could be dynamically created to optimize bandwidth utilizations when there are large concentration of clients accessing a remote server. Similarly slow server processes can be purged thus allowing us to eliminate bottlenecks in the system. System services provided within GES include fault-tolerance, replication, guaranteed delivery and security.

1.1 The need for transparent access

Most services allow clients to access the service through a broker. The client is then forced to remain on this broker throughout the entire duration of the time that it is using the service. If the broker fails, the client has to wait till the broker comes back up. In the event that this service is running on a set of brokers the client, since it knows about this set of brokers, could then connect to one of these brokers and continue using the service. Whether the client missed any servicing and whether the service would notify the client of this missed servicing depends on the implementation of the service. In all these implementations the identity of the broker that the client connects to is just as important as the service itself.

Clients are not always online, and when they are, they are not using the same computer. Different clients utilize or communicate with the service using communication channels that have very different bandwidths and associated latencies. Clients access services from different geographic locations, a client may use the service from his home, or from his office or while he is commuting to work or from his hotel room. Access to services should not be tied to specific broker locations or be location sensitive. Client should be able to connect from anywhere to any of the brokers within the system. Concentration of clients from a specific location accessing a remote broker, leads to very poor bandwidth utilization and affects latencies associated with other services too.

A truly distributed service, would allow a client to use the service by connecting to a broker nearest to his geographical location. By having such local broker, a client does not have to reconnect all the way back to the broker that it was last attached to. Also, if the client is not satisfied with the response times that it experiences it could very well choose to connect to some other local broker. This it could also do in the event that the broker, it was attached to, has failed. Also it should not be assumed that a failed broker node, providing this service, would recover within a finite amount of time. Stalling operations for certain sections of the network, and denying service to clients while waiting for failed processes to recover could result in prolonged, probably interminable waits. Also this model potentially forces every broker to be up and running throughout the duration that this service is being provided. Models that require brokers to recover within a finite amount of time generally imply that each broker has some state. Recovery for brokers that maintain state involves state reconstruction. The state reconstruction normally involves reconstructing broker state from neighboring brokers. This model runs into problems when there are multiple neighboring broker failures. Invariably brokers get overloaded, and act as black holes where messages are received but no processing is performed. By ensuring that the individual brokers are stateless (as far as the servicing is concerned), we can allow these brokers to fail and not recover. A failure model that does not require a failed node to recover within a finite amount of time, allows us to purge such slow processes and still provide the service while eliminating a bottleneck.

Systems where clients continuously access a fixed set of brokers, results in a situation where a whole bunch of clients are accessing a certain known broker over and over again. Problems are compounded if the number of clients accessing this broker is large and if these clients are accessing the broker from different geographic locations. Balancing the client load using broker-farms, would still have the same bandwidth constraints caused by concentration of clients at different geographic location. What is indispensable is the service that is being provided and not the brokers which are cooperating to provide the service. Brokers can be continuously added or fail and the broker network can undulate with these additions and failures of brokers. The service should still be available for clients to use. Brokers thus do not have an identity - any one broker should be just as good as the other. Clients however have an identity, and their service needs are very specific and vary from client to client. Any of these brokers should be able to service the needs of every one of these millions and millions of clients. It's the system as a whole, which should be able to reconstruct the service nuggets that a client missed during the time that it was inactive.

Clients just specify the type of events that they are interested in, and the content that the event should at least contain. Clients do not need to maintain an active presence during the time these interesting events are taking place. Once it registers an interest it should be able to recover the missed event from any of the broker nodes in the system. Removing the restriction of clients reconnecting back to the same broker that it was last attached to, also opens up a host of possibilities where brokers could be dynamically instantiated based on the concentration of clients at certain geographic locations. Clients could then be induced to roam to such dynamically created brokers for optimizing bandwidth utilization. Clients which connect to a local broker, instead of the remote broker, after they have moved to a new geographic location (say New York to London) could have some of its interests being serviced by the local service. Resource discovery services (such as search for hotels, printers to print a document etc) should be based on local proximity.

1.2 Possible approaches to building a service infrastructure

The brokers could communicate with each other employing one of three approaches viz. queuing systems, remote procedure call based systems and publish subscribe systems. Message queuing systems with their 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 [46] and Microsoft's MSMQ [45]. 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) [33]. MPI is designed for high performance on both massively parallel machines and workstation clusters. Messaging systems based on the classical remote procedure calls include CORBA [58], Java RMI [49] and DCOM [31]. Publish subscribe systems form the third axis of messaging systems and allow for decoupled communication 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 the constraint that they be aware of each other. The publisher is also unaware of the number of subscribers that are interested in receiving a message. The publish subscribe model does not 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 *pull* and *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* [25] from TIBCO and *SmartSockets* [24] from Talarian. Variants of publish subscribe include systems based on content based publish subscribe. Content based systems allow subscribers to specify the kind of content that they are interested in. These content based publish subscribe systems include *Gryphon* [7, 3], *Elvin* [70] and *Sienna* [18]. 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 mature messaging products with 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 [57] and the Notification service [56]. The push by Java to include publish subscribe features into its messaging middleware include efforts like JMS [43] and JINI [5]. One of the goals of JMS is to offer a unified API across publish subscribe implementations. Various JMS implementations include solutions like *SonicMQ* [23] from Progress, *JMQ* [48] from iPlanet, *iBus* [47] from Softwired and *FioranoMQ* [22] from Fiorano.

1.3 Paper Outline

In this paper we propose the Grid Event Service (GES) where we have taken a system model that encompasses Internet/Grid messages. GES is designed to include JMS as a special case. However, 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 broker nodes providing a distributed event service in a federated fashion. In GES a subscribing client can attach itself to any of the broker nodes comprising the system. This client specifies the type of events it is interested in through its profile. A client such as this could then fail, leave or *roam* (in response to failure suspicions, system induced roams etc.) the system. When such a client reconnects back into the system, this client should receive all the events that it was supposed to receive between the time it left the system and the time it reconnects back into the system.

We have employed a distributed network of broker nodes primarily for reasons of scaling and resiliency. A large number of broker nodes can support a large number of clients while at the same time eliminating the single point of failure in single broker systems. These broker nodes are organized as a set of strongly connected broker nodes comprising a cluster; clusters in turn are connected to other such clusters by long links. This scheme provides for small world networks, which in the spectrum of strongly connected graphs falls in between regular graphs and random graphs. The advantage of such *small world networks* [74] is that the average *pathlength* of any broker node to any other broker node increases logarithmically with the geometric increases in the size of the network.

We employ schemes, which ensure that each broker node maintains abbreviated views of system inter-connectivities. This abbreviated system view is maintained in the *connectivity graph.* The connectivity graph has imposed directional constraints on graph traversal and also dynamic costs associated with the same based on link type and links connecting two system units (brokers, clusters, cluster of clusters etc). This is then used to provide us with the fastest hops to employ to reach any given destination. It is ensured that this graph maintains the *true* state of the system, so that only active nodes and fast links are employed for the routing at every broker node where such decisions are made. To ensure that a client misses no interesting event and also to ensure that uninteresting events are not routed to parts of the system not interested in receiving the events, we employ an intelligent dissemination scheme. This dissemination scheme is hierarchical, as is the calculation of destinations and the propagation of profiles. The profile changes are routed to relevant nodes in the system. A client would thus route profile changes to the broker it is attached to, while the broker propagates its profile changes to its cluster controllers (there could be more than one for a cluster) and so on. The hierarchical destinations computed for an event ensure that only the relevant parts of the sub-system receive the event. This scheme is capable of handling dense and sparse interests in different parts of the system equally well. The logarithmic pathlengths achieved by the organization scheme for the broker nodes, combined with the calculation of fastest routes to reach destinations at every broker node hop and the exact sub-systems to route an event provides a near optimal routing scheme for the events.

To account for failure scenarios, recovery from such failures and the reliable delivery of events to clients in the presence of such failures we need a storage scheme. The replication scheme that we have designed allows different replication strategies to exist in different parts of the system. We employ a scheme that allows us to detect partitions in response to unit (brokers, clusters, cluster of clusters etc.) failures or link failures and take appropriate actions to initiate recovery. The GES failure model allows a unit to fail and remain failed forever. Clients attached to affected units can roam the network, attach themselves to a different broker node and still receive all the events that they were supposed to receive. The GES model allows a stable storage to fail, the only constraint imposed is that these stable storages do not remain failed forever and recover within a finite amount of time. During stable storage failures only certain sections of the subsystem are affected. Similarly a stable storage could be added at different parts of the system and be configured as a finer or coarser grained stable storage at the subsystem that it was added to. Clients need not be notified about the addition of stable storages, the system manages reliable delivery of events to clients transparently. The addition of stable storage is disseminated only within certain parts of the system. The GES publish/subscribe model allows for various flavors in the delivery of events to clients.

The GES system lends itself very well to dynamic topologies. Brokers could be dynamically created to improve bandwidth usage and better servicing of clients. Once these brokers are dynamically created, relevant clients can be induced to roam to the newly created broker, and system routing could be updated to include the newly added broker. This same scheme could be used to reconfigure the broker network by identifying slow broker nodes (which serve as black holes for events) and reconfiguring the network to eliminate the bottlenecks. The GES system could be easily extended to provide services based on the interpretation capabilities of a client. Such a system, e.g. the Grid Event Service Micro Edition (GESME) being developed at Florida State University (FSU), would be a very useful addition. Discovery of services and message transformation switches would provide for very rich content interpretation capabilities for such clients. Other application domains where

GES would be extended into are collaboration and *peer-to-peer* (P2P) systems [2]. GES is intended to be part of the Grid Collaborative Portal (GCP) for distance learning currently being developed at FSU. GES could also be used as an engine providing peer to peer interaction between clients, this interaction being done via the grid.

This paper 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 broker topology that we use to solve the problem. Section 4 describes our solution to the event delivery problem and provides detailed explanations of the various protocols that comprise the final solution. We then proceed to describe the approach for guaranteed delivery of events and the detection of network partitions. The guaranteed delivery of events is in the presence of failures (broker nodes can fail and remain failed forever). Finally we include a discussion of results for various scenarios and future directions and conclusions.

2 Specifications

In this section 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 formalisms and the properties that the system and it components must conform to during execution.

2.1 Events

An event is the most fundamental unit that entities use to communicate with each other. An event consists of a set of properties and has 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. A subset of this set of properties is what constitutes the type of the event. Events allow separate entities to probe different sets 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 the 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 very large) set of *broker nodes*, which are strongly connected (via some inter-connection network) and special nodes called *client nodes* which can be attached to any of the broker 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, brokers and clients, communicate by sending events through the network. This communication is *asynchronous* i.e. there is no bound on communication delays. Also, the message carrying an event can be lost or delayed. A broker 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 broker 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 $receive(e)$.

Broker nodes are responsible for routing/queuing events to the destination lists contained within the event. Each broker node instantiates a *service* which is responsible for interacting with service instances on other broker nodes to facilitate the calculation of destination lists for the events and the routing/queuing of these events to the relevant clients. Client and broker nodes can be on the same physical machine. For increased availability and reduced latency, some of the broker 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 broker nodes) and link failures. The broker 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. Client nodes specify the type of events that they are interested in. This information is contained in the client's *profile*. An event could be addressed to a specific client node or a set of client nodes, we refer to the destinations contained in such events – *explicit* destinations. For events that are not explicitly addressed to a client node or set of client nodes, the system is responsible for computing the destinations associated with the event. These system computed destinations are the *implicit* destinations associated with the event, and are computed based on the profile of each and every client in the system. 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 and can still expect to receive all the events that they missed, in the interim, along with real time events that occur while they have *rejoined*. 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 broker node that 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*. A client could also initiate a roam if it suspects, irrespective of whether the suspicion is correct or not, a failure of the broker node it is attached to. The choice of the broker 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 broker node in the system, the service instances on the broker nodes are 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. Each of the combinations join-leave, join-crash, recover-leave and recover-crash, is 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

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

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 messages.

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

(d) Not all events can be such that there are no clients are interested in them.

(e) Not all messages issued by a client can be lost all the time.

Items (d) and (e) constitute the liveness property, eliminating trivial implementations in which messages carrying the event are 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 any client node in the destination list L_e of an event e receives e, then all client nodes in L_e receive the event e.

(d) For events issued by any client node in the system, the event generation order is preserved by all the client nodes receiving those events.

3 Events, Clients and the Broker Topology

In this section, we present the anatomy of an event based on our discussions in Section 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 broker network, and the nomenclature that we would be referring to in the remainder of this paper.

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 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).

3.1.3 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.

3.1.4 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 $sequenceNumber.MAX$ 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* [1] propagate events through the network without duplication, using the IETF UUID [54] 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.

3.1.5 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 broker hops that are allowed before the event is discarded by the system.

Correlation Identifiers
Correlation identifiers help impose causal delivery constraints on (*request, reply*) events.

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

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

3.1.6 Destination Lists

Clients in the system specify an interest in the type of events that they are interested in receiving. Some examples of interests specified by clients could be 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.

3.1.7 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 3.1.3. Derived events can be processed as base events and not vice versa.

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

3.1.9 Specifying the anatomy of an event

These sets of equations follow from our discussions in section 3.1. Equation (1) follows from our discussions in section 3.1.4 regarding the generation of unique identifiers. This tuple is created by the issuing clients.

$$eventId =< clientId, timeStamp, seqNumber, incarnation > \tag{1}$$

The tuple in equation (2) discriminates between live events and recovery events (which occur due to failures or prolong disconnects).

$$liveness =< live|recovery > \tag{2}$$

The type of an event is dictated by the event's *signature*. These signatures could change. To accommodate these changes we include the concept of versioning in our event signatures. This along with *liveness* describes the event type completely.

$$eventType =< signature, versionNum, liveness > \tag{3}$$

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 case the destination lists would be computed by the system.

$$destinationLists =< \overbrace{Implied}^{\text{External}} \mid \overbrace{Explicit}^{\text{Internal}} > \tag{4}$$

$$spatialDependency =<? \mid * \mid + \mid \star > \odot < mapping \mid rules \mid constraints > \tag{5}$$

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

$$\begin{aligned} event \quad = \quad & < eventId, eventType, attributesList \\ & spatialDependency, timingDependency \\ & stream, applicationType, destinationLists > \end{aligned} \tag{6}$$

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 broker hosting multiple clients. While, this is a 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 broker nodes in the network. Solving issues of consistency while executing operations, in the presence of replication, leads to a model where other broker nodes can service a client despite certain broker node failures.

3.2.1 Scalability

We envision the system to be comprised of hundereds of millions of clients. Having all these clients being serviced by one central broker 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 enables the client to connect to broker 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 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 they were supposed to receive as well as events that they were supposed to receive during their respective 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 broker 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 Broker Node Topology

The smallest unit of the system is a *broker node* and constitutes a unit at level-0 of the system. Broker nodes grouped together form a *cluster*, the level-1 unit of the system. Clusters could be clusters in the traditional sense, groups of broker nodes connected together by high speed links. A single broker node could also decide to be part of such traditional clusters, or along with other such broker nodes form a cluster connected together by geographical proximity but not necessarily high speed links.

Several such clusters grouped together as an entity comprises a level-2 unit of our network and is referred to as a *super-cluster*, shown in figure 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 1 Cluster-A has links to Clusters B, C and D while Cluster-B has links to Clusters A and C. For two clusters with at least one link between them, any node in either of the clusters can communicate with any other node of the other cluster. In general there would be multiple links connecting a single cluster to several other clusters. This approach provides us with a greater degree of fault-tolerance, by providing us with multiple *routes* to reach nodes within other clusters.

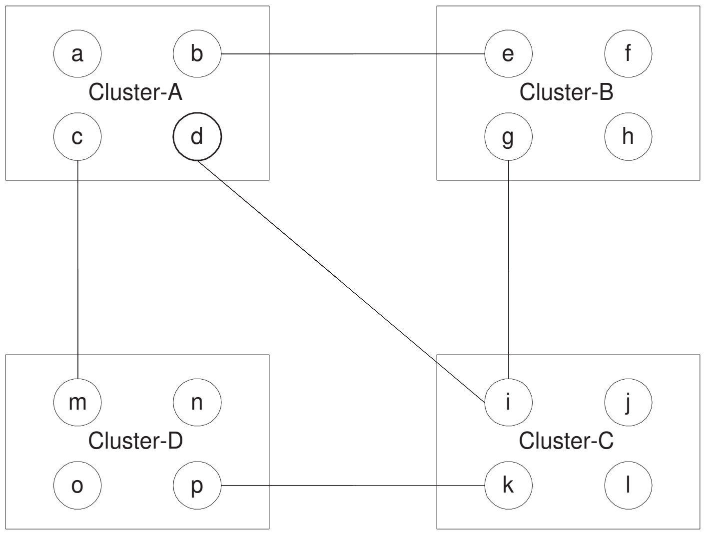

Figure 1: A Super Cluster - Cluster Connections

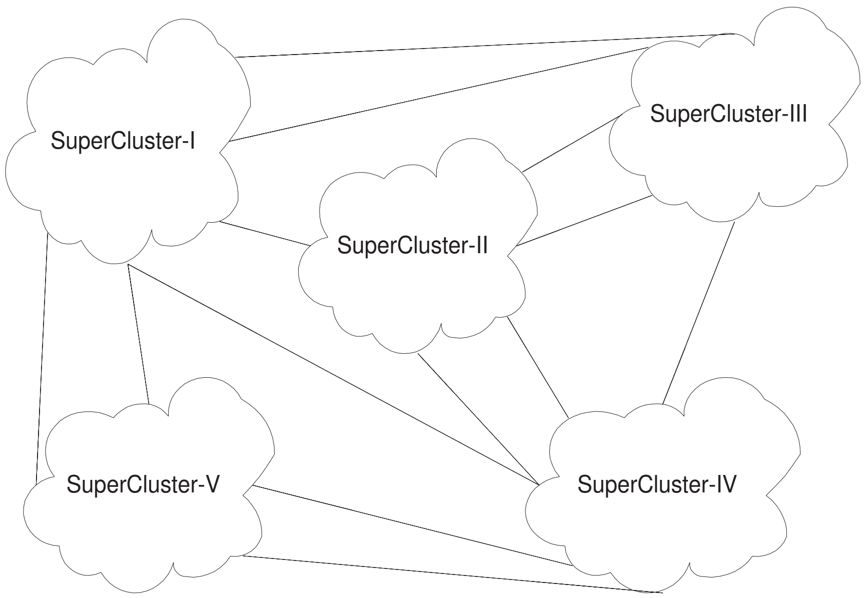

Figure 2: A Super-Super-Cluster - Super Cluster Connections

This topology could be extended in a similar fashion to comprise of *super-super-clusters* (level-3 units) as shown in figure 2, *super-super-super-clusters* (level-4 units) and so on. A Client thus connects to a broker 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 an N-level system this scheme allows for $2^6_N \times 2^6_{N-1} \times \cdots 2^6_0$ i.e $2^{6*(N+1)}$ broker nodes to be present in the system.

What we essentially have here is a set of strongly connected broker 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 in connecting from one node in the network to another node in the network. This is proportional to the broker node hops that need to be taken en route to the final destination.

We now delve into the *small world graphs* introduced in [74] and employed for the analysis of real world peer-to-peer systems in [59, pages 207 – 241]. In a graph comprising 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 consisting of n nodes, each of which is connected to its nearest k neighbors – for cases where $n \gg k \gg 1$, 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 [74] 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, though the clustering coefficient looks essentially like a regular graph. The authors refer to such graphs as *small world graphs*. This result is consistent with our conjecture that for our broker 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 broker exists within the GES context C^1_i of a cluster, which in turn exists within the GES context C^2_j of a super-cluster and so on. In general a GES context C^ℓ_i at level ℓ exists within the GES context $C^{\ell+1}_j$ of a level $(\ell+1)$. In an N-level system the following hold —

$$\begin{aligned} C^0_i &= (C^1_j, i) && (7)\\ C^1_j &= (C^2_k, j) && (8)\\ &\vdots \\ C^{N-2}_p &= (C^{N-1}, p) && (9)\\ C^{N-1}_q &= q && (10) \end{aligned}$$

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 $C^{\ell+1}_i$ should be able to reach any other unit within that same level. If this condition is not satisfied we have a *network partition*.

3.4.2 Gatekeepers

Within the GES context C^2_i of a super-cluster, clusters have broker 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. This architecture provides a greater degree of fault tolerance by providing multiple routes to reach the same cluster. Some of the nodes in the cluster thus maintain connections to the nodes in other clusters. Similarly, some nodes in a cluster could be connected to nodes in some other super-cluster. We refer to such nodes as *gatekeepers*. Nodes, which maintain connections to other nodes in the system, have different GES contexts. Depending on the highest level at which there is a difference in the GES contexts of these node, the nodes that maintain this active connection are referred to as the gatekeeper at that level. Nodes, which are part of a given cluster, have GES contexts that differ at level-0. Every node in a cluster is connected to at least one other node within that cluster. Thus, every node in a cluster is a gatekeeper at level-0.

Let us consider a connection, which exists between nodes in a different cluster, but within the same super-cluster. In this case the nodes that maintain this connection have different GES cluster contexts i.e. their contexts at level-1 are different. These nodes are thus referred to as gatekeepers at level-1. Similarly, we would have connections existing between different super-clusters within a super-super-cluster GES context C_i^3. In an N-level system gatekeepers would exist at every level within a higher GES context. The link connecting two gatekeepers is referred to as the *gateway*, which the gatekeepers provide, to the unit that the other gatekeeper is a part of. A gatekeeper 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 gatekeeper identifier i
- The list of gatekeepers at level ℓ that it is connected to, within the GES context $C_j^{\ell+1}$.

It should be noted that a gatekeeper at level ℓ can be a gatekeeper at any other level. In fact, every node within the system is a gatekeeper at level-0. Figure 3 shows a system comprising 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 the 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 gatekeepers. 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 as gatekeepers at $level - \ell$ i.e. $g^\ell(C^{\ell+1})$. Thus, in figure 3 we have 12 super-super-cluster gatekeepers, 18 super-cluster gatekeepers (6 each in **SSC-A** and **SSC-C**, 4 in **SSC-B** and 2 in **SSC-D**) and 4 cluster-gatekeepers in super-cluster **SC-1**.

3.4.3 The addressing scheme

The addressing scheme provides us with a way to uniquely identify each broker 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 5.4.9). As discussed earlier, units at each level are defined within the GES context of a unit at the next higher level. In an 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 broker 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 broker 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**. This addressing scheme is very similar to the IP addressing scheme.

3.5 Summary

In this section we dicussed the design of an event, based on the discussions in section 2. We also discussed the rationale for a distributed network of brokers, with issues such as scaling, resiliency to failures and load balancing being the most important factors influencing the choice of the distributed model. The section 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

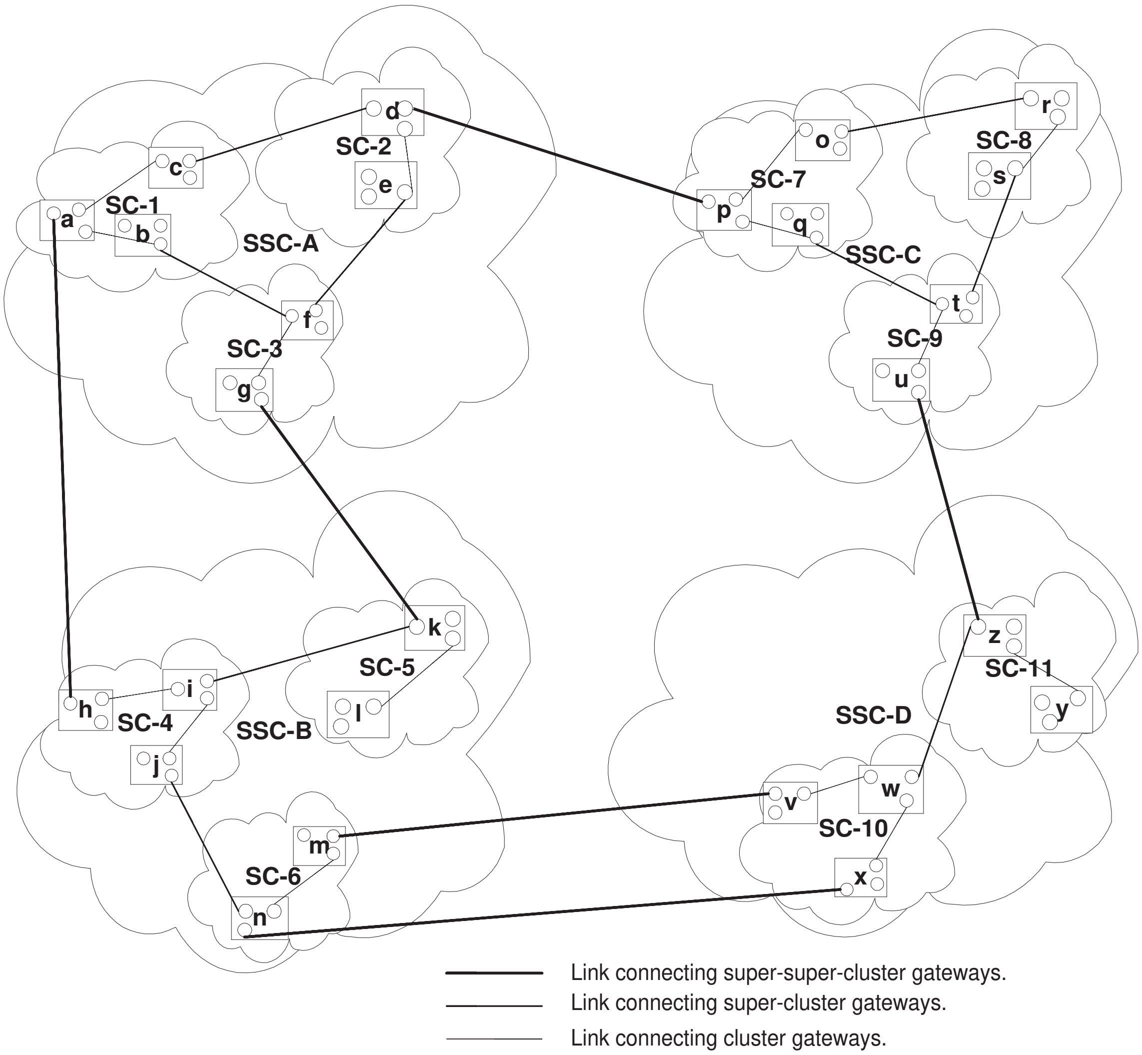

Figure 3: Gatekeepers and the organization of the system

within the system. Finally we established a topology for our broker 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 paper.

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 events 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 that 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 increase in latency is due to the cumulation of queuing delays associated with the *uninteresting/flooded* events. The system thus needs to be very selective of the kinds of events that it routes to a client. In this section 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 broker 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 the 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 within the system 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. 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.

Different systems address the problem of event delivery to relevant clients in different ways. In [39] 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 [70] network traffic reduction is accomplished through the use of *quench* expressions. Quenching prevents clients from sending notifications for which there are no consumers. Approaches to content based routing in *Elvin* are discussed in [71]. In [18, 19] 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 [7] each 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 broker to determine which of its neighbors should receive that event. [6] describes approaches for exploiting group based multicast for event delivery. These approaches exploit universally available multicast techniques.

The approach adopted by the OMG [58] is one of establishing channels and registering suppliers and consumers to those event channels. The channel approach in the event service [57] 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 [56] 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* [44], 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.

In some commercial JMS implementations, events that conform to a certain topic are routed to the interested clients. Refinement in subtopics is made at the receiving client. For a topic with several subtopics, a client interested in a specific subtopic could continuously discard uninteresting events addressed to a different subtopic. This approach could thus expend network cycles for routing events to clients where it would ultimately be discarded. Under conditions where the number of subtopics is far greater than the number of topics, the situation of client *discards* could approach the flooding case.

In the case of servers that route static content to clients such as Web pages, software downloads etc. some of these servers have their content mirrored on servers at different geographic locations. Clients then access one of these mirrored sites and retrieve information. This can lead to problems pertaining to bandwidth utilization and servicing of requests, if large concentrations of clients access the wrong mirrored-site. In an approach sometimes referred to as *active mirroring*, websites powered by *EdgeSuite* [21] from Akamai, redirect their users to specialized Akamized URLs. *EdgeSuite* then accurately identifies the geographic location from which the clients have accessed the website. This identification is done based on the IP addresses associated with the clients. Each client is then directed to the server farm that is closest to the client's network point of origin. As the network load and server loads change clients could be redirected to other servers.

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 pertains to the number of gateways at each level that the node can maintain concurrent connections to 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 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 whether the node is either part of the existing system or is part of a new unit being merged into the system. In the former case no new logical address are assigned, while in the latter 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-0 gatekeeper with the node in question, the two nodes would end up sharing a similar GES context C_i^1. The level-0 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-0 gatekeeper, the GES contextual information varies at the lowest level C_i^0. In the event that the requester seeks a *to* relationship with the addressee, the GES 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.

A node requests the connection setup in a bit vector specifying the kind of gatekeeper it seeks to be. The position of 0's and 1's dictates the kind of gatekeeper that 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-2 gateway to supercluster **SC-6** and as a level-1 (cluster) gateway within the super-cluster (SC-4/SC-5) that it would be a part of.

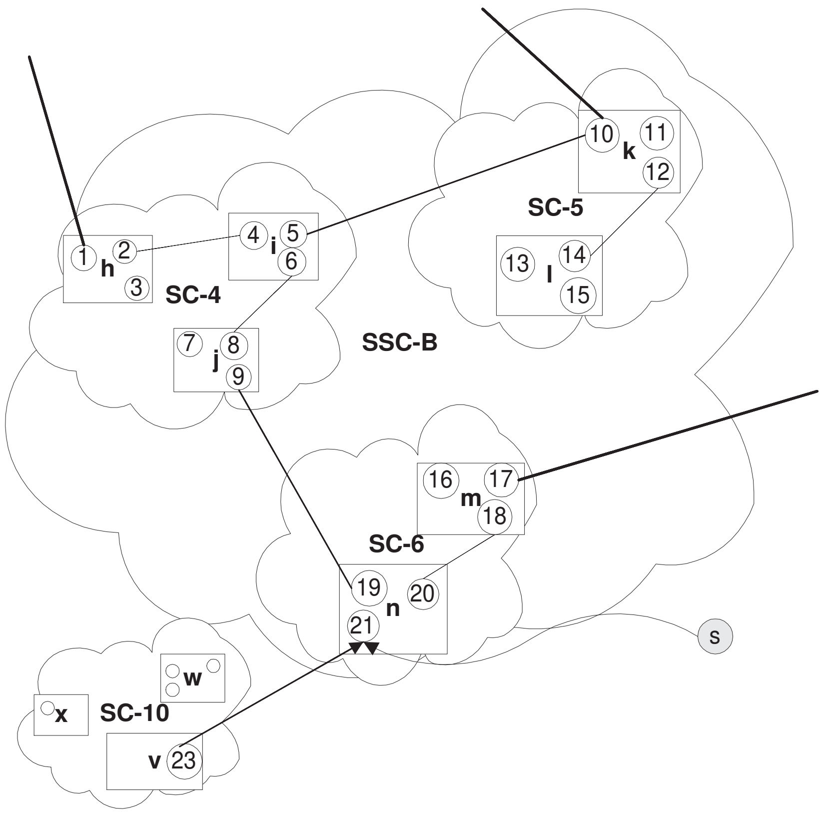

Figure 4: Adding nodes and units to an existing system

Figure 4 depicts a node **s** requesting a connection setup request. If **s** requests to be a level-0 node, then it needs to be part of the cluster **n**. Now, if node **n.21** has not exceeded the connection threshold limit for level-0 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-0 node with a connection to node **n.21**. If however, node **n.21** has reached its connection threshold for level-0 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-0 gateway to that node in cluster **n**. If however, all the nodes have reached their connection threshold limit, the node responds by providing a list of level-1 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-0 gatekeeper within cluster **n** but seeks to be a level-ℓ ($\ell > 0$), gateway *to* cluster **n** the procedures 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 node seeks to be a level-1 gatekeeper *to* cluster **n**, then node **n.21** confirms the connection threshold vector. If all the nodes have reached their connection threshold for level-1 gateways the cluster returns a failed response. If however there is such a node in cluster **n** which has not reached its threshold for level-1 connections node **n.21** provides the address for such a node, and also the addresses of level-1 gatekeepers within supercluster **SC-6** to which it is connected. Node **s** then tries to be a level-0 gateway within cluster **m** which is also a level-1 gateway to the nodes in cluster **n**. If there are no clusters within super-cluster **SC-6** other than cluster **n** which can accept **s** as a level-0 gatekeeper, then the request fails.

4.1.2 Adding a new unit to the system

A 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-0 gatekeeper of only one cluster. Thus a node in an existing cluster cannot seek to be part of another cluster in the system. In general for a unit at level-ℓ which is being added to the system, any node in the unit being added cannot seek to be a level-$(\ell - i)$ (where $i = 1, 2, \cdots, \ell$) 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. Nodes within a cluster 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 C^0 of the nodes with respect to newly assigned GES cluster context C_j^1 remains the same.

Figure 4 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 is the node **SC-10.v.23**. Since this is a level-2 system that is *unit-added*, node **23** or any other node within **SC-10** can not be a level-1 (cluster) gateway to the other nodes within the super-super-cluster SSC-B. Node **23** thus issues a request specifying that it seeks to be a level-3 gateway within super-super-cluster **SSC-B**. Upon a successful connection set up, a new address is assigned for **SC-10** (say **SC-8**), the 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 with 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 the event. At each $g^{\ell+1}(C_i^{\ell+1})$ finer grained disseminations targeted for units u^ℓ within $C_i^{\ell+1}$ are computed. When presented with such a list of destinations, based on the gateway information the best hops to take to reach the destinations 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 $\ell = 0, 1, \cdots, N - N$ being the system level).

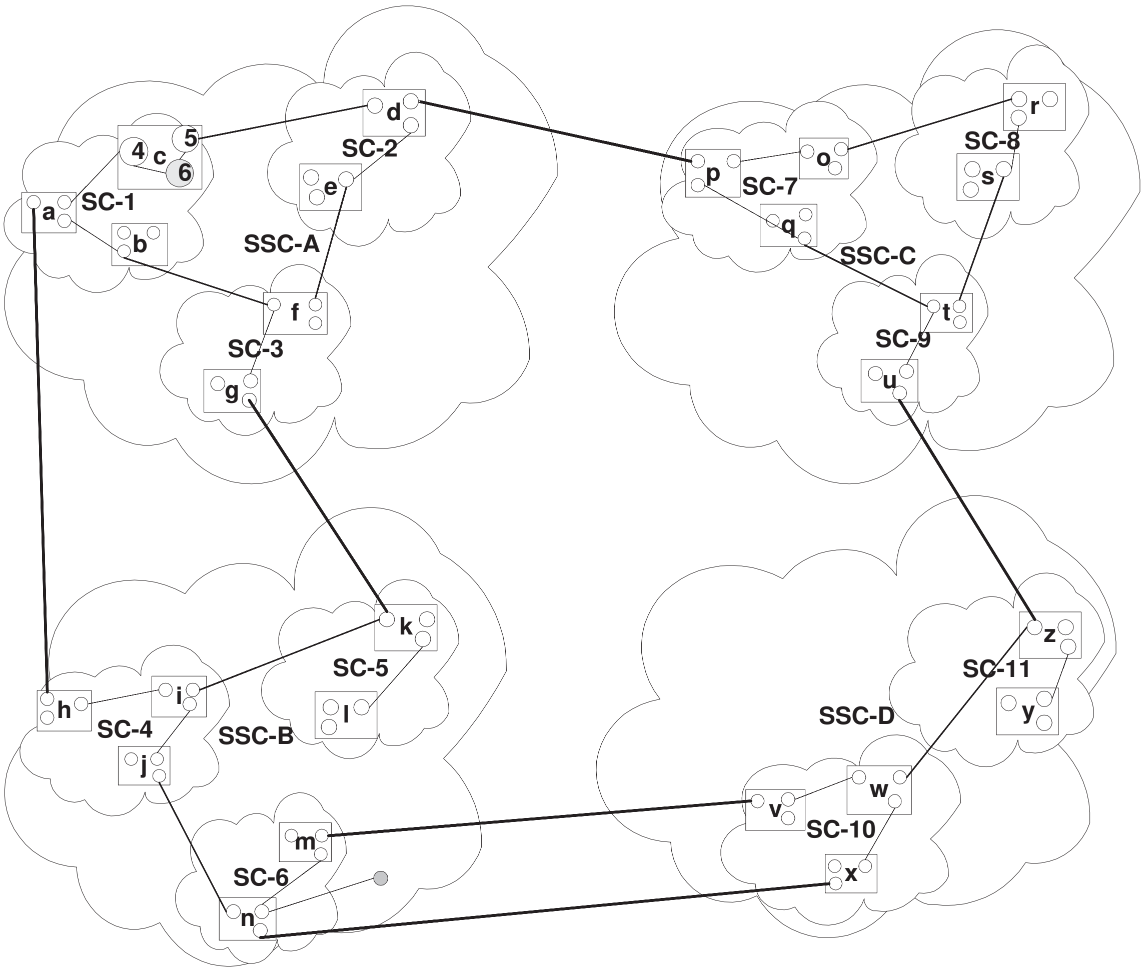

Figure 5: Connectivities between units

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 the connectivities that are relevant to the node in question. Figure 5 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 one 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 $C_i^{\ell+1}$ 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 other 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 snapshot 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-$(\ell + 1)$ GES context $C_k^{\ell+1}$. For any given node n_i^ℓ in the connectivity graph we are interested only in the level $\ell, \ell+1, \cdots, N$ connections that exist within the unit and not the $\ell-1, \ell-2, \cdots, 0$ connections 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 5 for a super-cluster gateway established within **SSC-A**, the connection information is disseminated only within the super-clusters **SC-1**, **SC-2** and **SC-3**, and subsequently the nodes in super-super-cluster **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 using its level-0 GES context.

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

When the connection information is being disseminated throughout the GES context $C_i^{\ell+1}$, it arrives at gatekeepers at various levels. Depending on the kind of link this information is being sent over, the information contained in the *connection* is modified. Every gatekeeper $g^p \ni p \leq \ell_c$, at which the connection information 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 to see if $p > x$. If $p > x$ the node information is updated to reflect the node as level-p node by including the level-p GES contextual information of g^p. If $p \not> x$ the connection information is disseminated *as is*. Thus, in figure 5 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

For every connection that is created there is a unique identifier associated with that connection. All connections relevant for a node are maintained in a connection table. This scheme allows us 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 that were already connected. This scheme also plays an important role in determining if a connection loss would lead to partitions, this is described in section 5.2.1.

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 why 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 1 depicts the cost associated with communication between units at different levels.

level	**0**	**1**	**2**	**3**	$\boldsymbol{\ell_i}$	$\boldsymbol{\ell_j}$
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	$\ell_i + 3$	$\ell_j + 3$
$\boldsymbol{\ell_i}$	ℓ_i	$\ell_i + 1$	$\ell_i + 2$	$\ell_i + 3$	$2 \times \ell_i$	$\ell_i + \ell_j$
$\boldsymbol{\ell_j}$	ℓ_j	$\ell_j + 1$	$\ell_j + 2$	$\ell_j + 3$	$\ell_j + \ell_i$	$2 \times \ell_j$

Table 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 node*, which is the level-0 broker 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 of 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 graph nodes (representing the corresponding units at different levels) is present in the connectivity graph. If any of the units within the connection is not present in the connectivity graph, the corresponding graph node is added to the connectivity 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 $y \leq x$ then –

- Graph node n_j^y is added to the set L_{UL} associated with node n_i^x
- Graph node n_i^x is added to the set L_D associated with 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 6 depicts the connectivity graph that is constructed at the node SSC-A.SC-1.c.6 in figure 5. 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

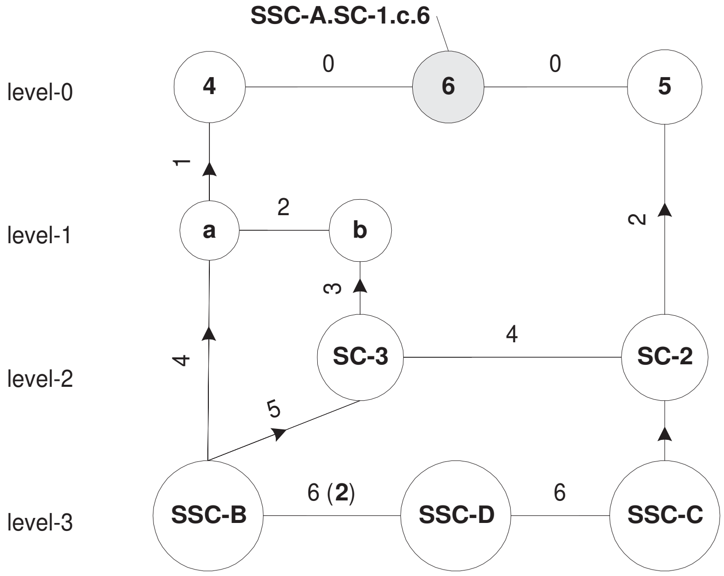

Figure 6: The connectivity graph at node 6.

traversal over a level-3 gateway between a level-2 unit **b** and a level-3 unit **SC-3** as computed from the linkcost matrix is 3, and is the weight of the connection edge. There are two connections between the super-super-cluster units **SSC-B** and **SSC-D**, this is reflected in the link count associated with the edge connecting the corresponding graph nodes. 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 the shortest path that exists between the unit and the vertex. This process of calculating the shortest path, from the node to the vertex, 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 directional constraints 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 any of the nodes in a valid path.

The algorithm proceeds by recursively computing the shortest paths to reach the vertex node, along every valid link (L_{UL}) originating at every node that 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 we do not keep track of the nodes that were visited, we could end up in an infinite recursion where we revisit the same node over and over again.

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

For example in the connectivity graph of figure 6 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 along with the shortest path to reach those nodes, provides us with the means to decide on the shortest path to reach the vertex.

4.2.8 Building and updating the routing cache

The best hop to take to reach a certain unit is the last node that was reached prior to reaching the vertex, when traversing the shortest path from the corresponding unit graph node to 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 to ensure efficient dissemination. 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 information contained in the routing cache to reach units at level $\ell, \ell+1, \cdots N$ needs to be updated for all the units within $u_i^{\ell+1}$. The cases of gateway failures, node failures, detection of partitions and the updating of the routing cache in response to these failures are 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 $g^{\ell+j}, g^{\ell+j-1}, \cdots, g^\ell$ connections are exchanged between the system and the newly added sub system. Thus when a super cluster is added to an existing system comprising of super-super-clusters, the existing system routes information regarding super-cluster and super-super-cluster connections to the newly added super-cluster. The way the set of connections, comprising the connectivity graph, is sent over the newly established link is consistent with the rules, which 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.c.6**, then, the connectivity graphs at node **6** would be as depicted in figure 7.(a) while the connectivity graph at the gatekeeper in **SC-4** would comprise of the connections that were sent over the newly established gateway by node **6**.

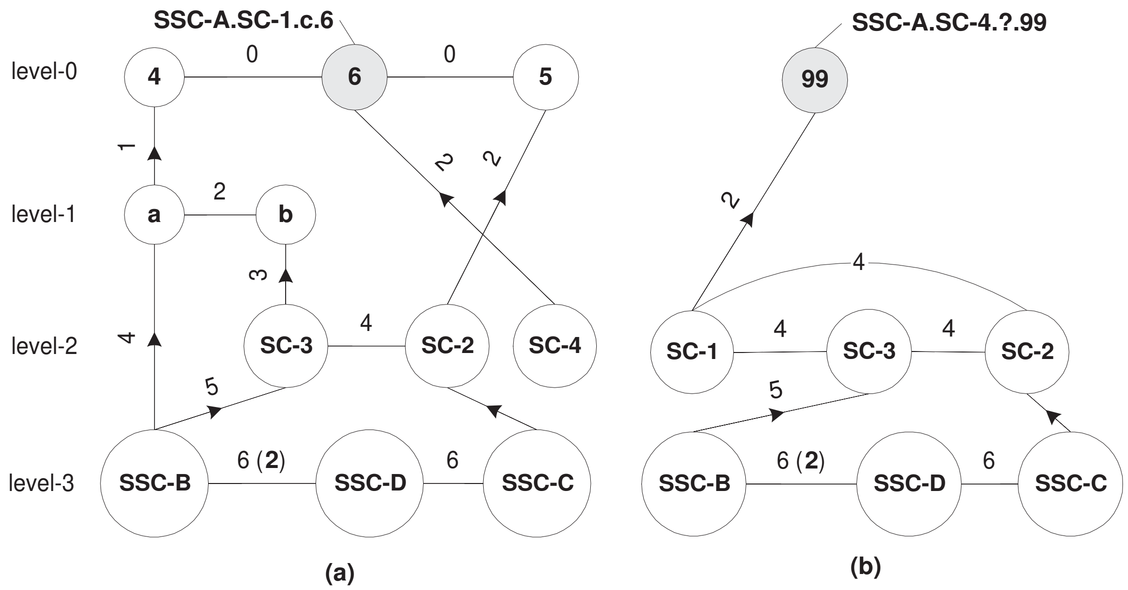

Figure 7: Connectivity graphs after the addition of a new super cluster SC-4.

Figure 7.(b) depicts only the connections which describe the connections involving level-2 gateways and upwards at node **99** in SC-4. There would be clusters comprising of strongly connected broker nodes in **SC-4**, we however do not need to depict these, in figure 7.(b), for the present discussion regarding connection information exchange.

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 are dictated and constrained by the *type* of the attribute. Clients within the system that issues these events, assign values to these attributes. The values these attributes take comprise the content of the event. All clients are not interested in all the content, and 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. These filters specified by the client constitutes its *profile*. 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 and also the content of the event to arrive at the 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 that make use of abbreviated views of inter-connections existing within the system. Profiles need to be organized so that they lend themselves 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 those that are consistent with the node's abbreviated view of the system. To allow for a node to maintain profiles contained at different units (clusters, brokers, clients etc.) 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) Propagation of profiles to the nodes that are 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 computation 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. The $*$ 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 an attribute a_i constraints for attributes $a_1, a_2, \cdots, a_{i-1}$ must be in place, even if some or all of these constraints is the weakest constraint $*$. 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 when added up constitute the profile graph.

We use the general matching algorithm, presented in [3], 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.* Every attribute on which a constraint is specified constitutes a node in the profile graph. When a constraint is specified on an attribute a_i, the attributes $a_1, a_2, \cdots, a_{i-1}$ appear in the profile graph. A profile comprises of constraints on successive attributes in an 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 [3] we do not allow multiple edges *terminating at* a node since it results in a situation where the event matching results in an invalid destination, due to that event having satisfied partial constraints of different profiles from within the same unit.

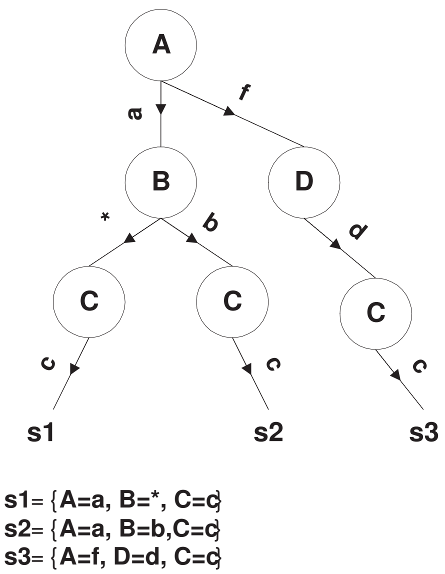

Figure 8: The profile graph - An example.

Figure 8 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 8 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 needs to maintain additional information along its edges. This additional information also plays a very important role in the reliable delivery of events to clients (we discuss this in a later section). Along every edge we maintain information regarding the units that are interested in its traversal. For each of these units we also 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.

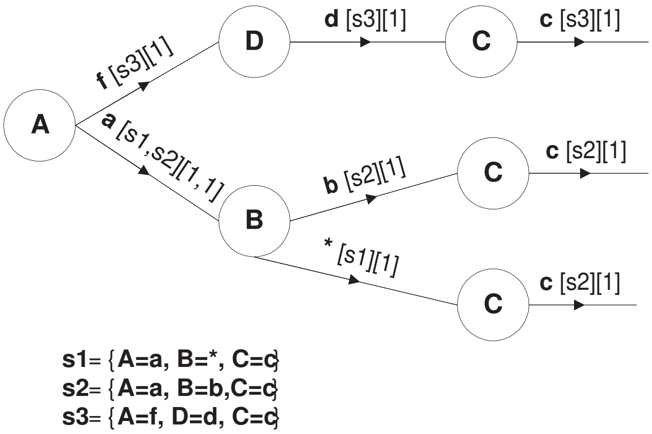

Figure 9: The complete profile graph with information along edges.

The information regarding the number of predicates $\delta\omega$ per unit that are interested in two successive constraints allows us to remove certain edges and nodes from the profile graph, when no clients are interested in the constraints any more. Figure 9 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 two 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 that 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 if the event satisfies the destination edge constraint, that 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, gatekeepers $g^{\ell+1}$ compute destination lists for the u^{ℓ} units that it serves as a $g^{\ell+1}$ for. A gatekeeper $g^{\ell+1}$ should thus maintain information regarding the profile graphs at each of the u^{ℓ} units. Profile graph $\mathcal{P}_i^{\ell+1}$ maintains information contained in profiles $\mathcal{P}^{\ell}$ at all the u^{ℓ} units 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 gatekeeper 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 gatekeeper, it propagates the edges created/removed with the broker as the destination associated with the profile predicate. Similarly, when this is being propagated to a super-cluster gatekeeper the profile change is sent across as a profile change in the cluster. Any change in the client's profile is propagated to gatekeepers at higher levels, that the broker 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 the routing algorithms that are in place. The connectivity graph provides us with an overall view of the interconnections between units at different levels. The organization and calculation of destinations from the profiles comprising the profile graph, feeds right into our routing algorithms that compute the shortest path to reach the units (destinations) where an event needs to be routed. In general for a level-N system, if there is a subscribing client with GES context C_j^N and the issuing client has GES context C_i^N the destinations are computed (N+1) times. Thus, in a system comprising of super-super-clusters, the destinations are computed four 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 any of the broker nodes that are a part of that cluster. Thus a change in the profile of a client needs to be propagated to its broker node. The change in profile of the broker 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 nodes in cluster **SSC-A.SC-1.c** are routed to node **4**. $\delta\omega^1$ changes need to be routed to level-2 gateways within **SSC-A**. In general the gatekeepers to which the profile changes need to be propagated are computed as follows —

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

(b) The uplink from this node of the connectivity graph to any other node in the graph, indicates the presence of a level-ℓ gateway at the unit corresponding to the graph 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 with information regarding all the level-ℓ gateways within the next higher level GES context $C^{\ell+1}$.

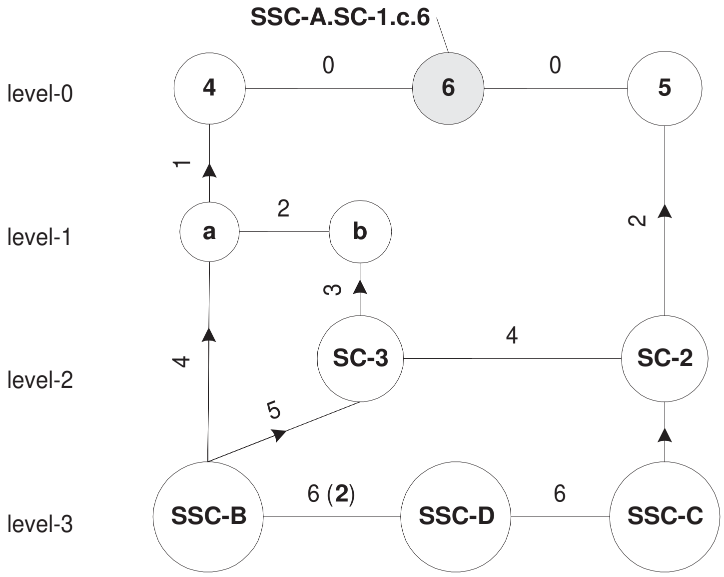

Figure 10: The connectivity graph at node 6.

In the figure 10, any $\delta\omega^0$ changes at any of the nodes within cluster **c**, are need to be routed to node **4**. Any $\delta\omega^1$ changes at node **4** need to be routed to node **5**, and also to a node in cluster **b**. 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* does not fail due to delivery of the same profile change multiple times within the same unit.

Summarizing the discussion so far, the profile graph snapshots the profiles of units at a certain level, and as such can compute destinations only for this 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 Unit additions and the propagation of profiles

When a unit (with publishing and subscribing clients) is being added to a larger existing broker network, besides the sequence of actions pertaining to the generation/update of logical addresses and the exchange of system inter connectivities, profiles would need to be propagated in exactly the same way that we described. Thus when a cluster is added to the system, the broker nodes within the cluster route their profiles to the newly created cluster gatekeeper. This gatekeeper is in turn responsible for the propagation of profiles to the super-cluster gatekeepers in the newly merged system.

4.3.7 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 has not 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 broker 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*[2] what we mean is that for every event that is being routed within that unit the destination lists calculated would include information to facilitate routing to the client. Since a client profile is unit active, all events, issued within the unit, will be routed to the client if it satisfies the client profile.

Events contain routing information in them, which indicate the units where these events were disseminated. The routing information contained in an event thus includes the unit in which the event was issued. Since the dissemination is hierarchical, an event will not be routed to a client till such time that the client's profile change has been propagated to higher level gatekeepers. If a profile change issued by a client c_A is routed to a super-cluster gatekeeper, all events issued by clients attached to any of the nodes within this super-cluster, will be routed to the client c_A if these events match the corresponding profile change. The routing information, for events issued by clients in this super-cluster, indicate the dissemination within the units in that super-cluster. If this event matches the profile change initiated by one of the attached clients, and if this event is routed to such a client then the profile change associated with that client is said to be super-cluster active. In an N-level system if the routing information depicts the dissemination of the event within another level-N unit within the system, the profile change issued by the client is said to be *system active*. When a profile change initiated by a client is system active, events issued by any other clients within the system will be routed to this client, if those events match the system active profile change that was initiated by this client.

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 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 broker node that the client is attached to adds the routing information to the event. This routing information is the GES contextual information (see section 3.4.1) pertaining to this particular node in the 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

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

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 is 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.

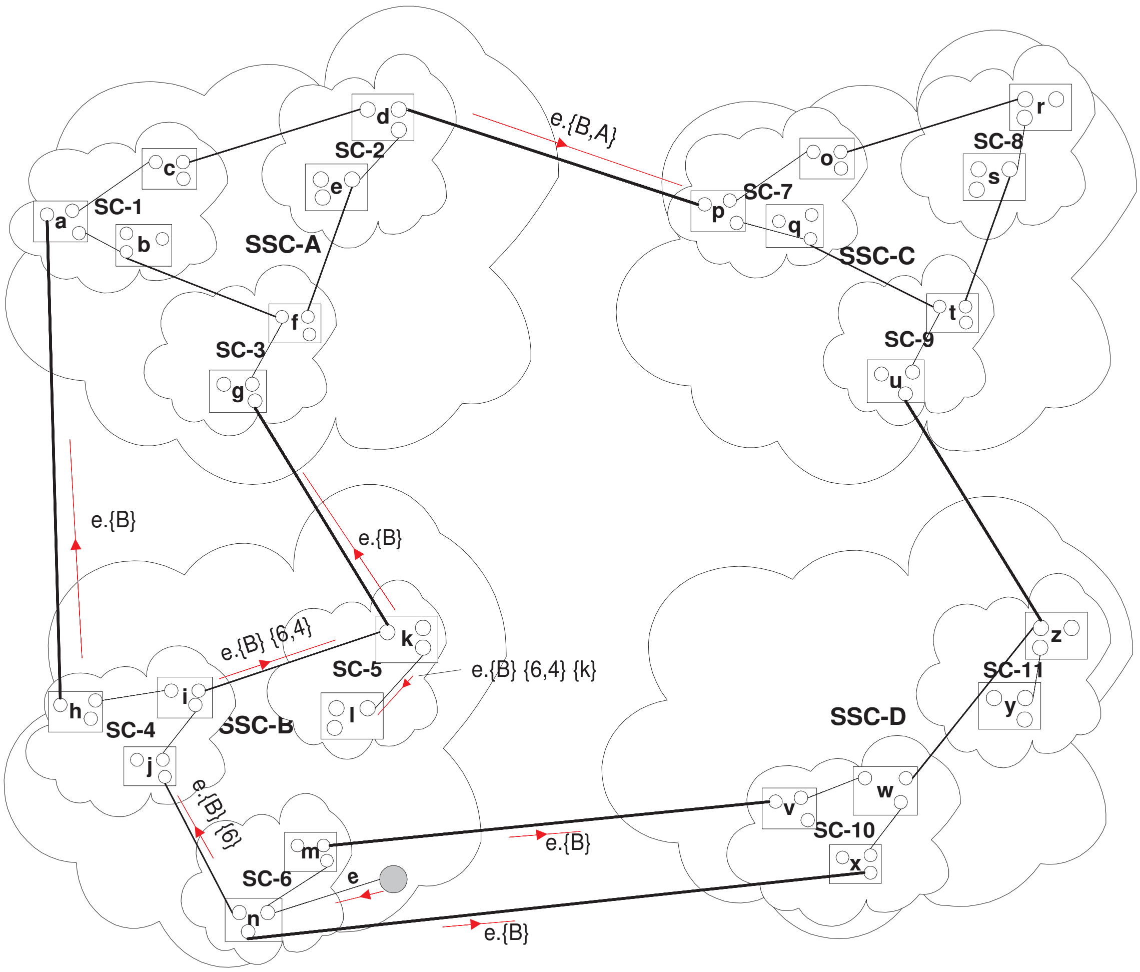

Figure 11: Routing events

A gateway $g^{\ell+1}$ in $u_i^{\ell+1}$ is responsible for the dissemination of events throughout the relevant u^ℓ units within $u_i^{\ell+1}$. This is a recursive process and the gateway $g^{\ell+1}$ delegates this dissemination process to the lower level gateways $g^\ell, g^{\ell-1}, \cdots, g^1$ to aid in finer grained disseminations. 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-ℓ to level-N. When an 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 that particular unit. It is the gatekeeper g^ℓ's responsibility to ensure that the event is routed to all the relevant nodes within the level-ℓ unit, using the delegation mechanism described earlier. Prior to routing an event across the gateway a level-ℓ gatekeeper takes the following sequence of actions –

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

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[3] where the event could be routed to the same unit over multiple links. In this case 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 disseminations is within the GES context of a specific level-ℓ unit and will not be valid within other level-ℓ units. Also, in general a higher order gateway would be more overloaded[4] compared to a lower order gateway. Reducing the amount of information being transferred over the gateway helps conserve bandwidth.

Figure 11 depicts the routing scheme which we have discussed so far. The routings depicted in the figure outline how routing information is updated to reflect the traversal at units in different levels.

In addition to the information regarding where the event has been delivered already, events also need to contain information regarding the units which an event should be routed to. Gatekeepers $g^\ell(C^{\ell+1})$ decide the level-$(\ell-1)$ units which are supposed to the receive the event. This decision is based on the profiles available at the gatekeeper as outlined in the profile propagation protocol. This calculation of the targeted units is a recursive process with the lower order disseminations being handled by the corresponding 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 hops to reach the destinations are 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 3.1.6) 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-ℓ unit that the gatekeeper belongs to. This check is necessary to confirm if the event needs to be disseminated within the level-ℓ unit. Routing events based on the gatekeeper profile is the process which calculates the destination

[3]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

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

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 has not 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-ℓ 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 that 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 3.1.4 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.

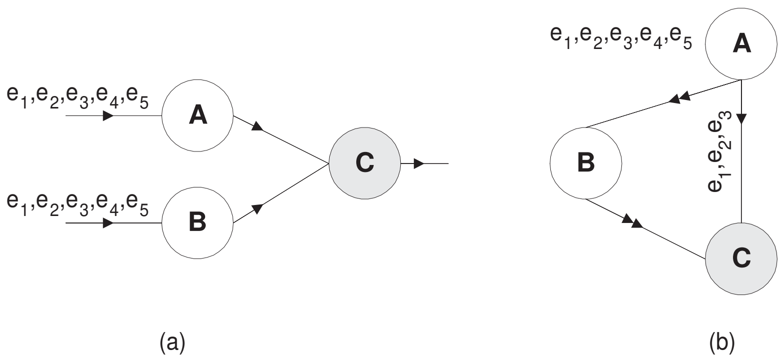

Figure 12: Duplicate detection of events

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

- If $e' > e$ then e' was not received earlier, else it was and it is duplicate detected. The $>$ relation between two related events is based on the timestamp and the sequence number that is associated with the two events.

Consider the case in figure 12.(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 overtake 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), i.e. 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) that arrives first is not duplicate detected while the event e that arrives later is duplicate detected.

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

Table 2: Reception of events at C

Consider the case in figure 12.(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 that 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 resending unacknowledged events across the alternate route in response to suspicions it simply reduces to the case depicted in figure 12.(a). As an optimization feature we could also send *anti-events* down the failed/slow link whenever we resort to computing an alternate route.

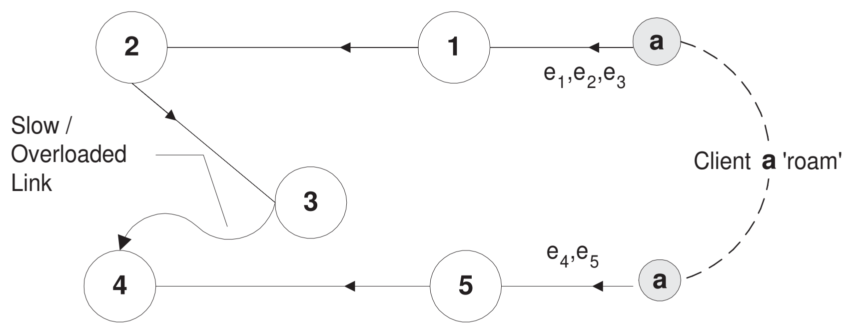

Figure 13: Duplicate detection of events during a client roam

Figure 13 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 3. To account for such a scenario we include the incarnation number in our duplicate detection scheme. Incarnation numbers would be incremented for every roam and reconnection of the issuing client. The events would then be treated as events with a different *clientID* thus preventing the duplicate detection of events which should not have been duplicate detected in the first place.

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

Table 3: Reception of events at 4: Client roam

4.7 Interaction between the protocols and performance gains

In our system the node organization protocol could be used in the creation of *small world* [74, 59] networks. This organization, which comprises of strongly connected broker nodes in clusters connected by *long links* ensures that the pathlength increases logarithmically for geometric increases in the size of the broker node network. The feature of having multiple links between two units/super-units ensures a greater degree of fault tolerance. Links could fail, and the routing to those units could still be performed using the alternate links. The organization of connection information ensures that connection losses (or additions) are incorporated into the connectivity graph hosted at relevant nodes. Certain sections of the routing cache are invalidated in response to this addition (or loss) of connections. This invalidation and subsequent calculation of best *hops* to reach units (at different levels) ensure that the paths computed are consistent with the state of the network, and include only valid/active links. The ability to compute routes to reach destinations at different levels lends the scheme very useful for hierarchical disseminations.

In our scheme for the organization of profiles we employ an approach where profiles of sub-units are maintained at the unit gatekeeper. Events almost always arrive at the unit gatekeepers first, since they provide a gateway to the unit. The only exception is in the cluster where a client issues an event. Having this unit gatekeeper intelligently decide on the sub-units, which should receive an event helps eliminate redundant routing of events. By maintaining sub-unit profiles at the unit gatekeeper we ensure that the only events that are routed to a unit are those for which there is at least one client, attached to one of the broker nodes in that unit, which is interested in the specific event. We obtain information regarding the nodes/units to route profile changes based on the information contained in the connectivity graph. We then employ hops (at every broker node en route) obtained from the routing cache to ensure that this profile dissemination is the fastest. The information maintained in the profile graph is consistent with the dissemination scheme and can be used to compute destinations at different levels. In an N-level system, an event is matched (N+1) times prior to routing the event to a client.

The event routing protocol uses the profile information available at the unit gatekeepers to compute the sub-units that the event should be routed to. To reach these destinations every node, at which this event is received, employs the *best hops* to reach the destinations. This *best hop* is computed based on the cost of traversal as also the number of links connecting the different units. Thus in our system, based on the organization of profiles and subsequent matching of events, the only units to which an event is routed are those that have clients interested in that event. Further, based on the connectivity graph and the associated routing cache we compute the fastest/reliable hops to take to reach the relevant destinations. The routing information encoded into the event along with the duplicate detection scheme ensures that we eliminate *continuous event echoing*, where the event is routed to the same unit over and over again.

These approaches result in only the relevant links and functioning nodes being employed for disseminations. The *small world* behavior that would exist in broker network, when appropriately organized, ensures that the pathlengths for these disseminations would only increase logarithmically with the number of broker nodes.

4.8 Summary

In this section 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 broker nodes within the topology scheme that we introduced in section 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 broker 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 the calculation of destinations using the matching algorithm discussed in [3]. Further we modified 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 the 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 nodes, to route a profile update to, is done based on the information encapsulated within the connectivity graph. To effectively disseminate messages within the system, we formulated 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.

5 The Reliable Delivery Of Events

The problem of reliable delivery [42, 11] and ordering[5] [13, 12] in traditional group based systems with process crashes has been extensively studied. The approaches normally have employed the *primary partition* model [65], 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 [41]. Recipients that are slow or temporarily disconnected may be treated as if they had left the group. This model, adopted in Isis [10], works well for problems such as propagating updates to replicated sites. This approach does not work well in situations where the client connectivity is intermittent, and where the clients can roam around the network. Systems such as Horus [64] and Transis [30] manage *minority partitions*, and can handle concurrent views in different partitions. The overheads to guarantee consistency are however too strong for our case. DACE [14] introduces a failure model, for the strongly decoupled nature of pub/sub systems. This model tolerates crash failures and partitioning, while not relying 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 [8], the effect of link failures on the solvability of problems (which are solved with reliable links) in asynchronous systems, has been rigorously studied. [69] describes approaches to building fault-tolerant services using the state machine approach.

Systems such as *Sienna* [19, 18] and *Elvin* [71, 39, 70] focus on efficiently disseminating events, and do not sufficiently address the reliable delivery problem in the presence of failures. In *Gryphon* the approach to dealing with broker failures is one of reconstructing the broker state from its neighboring brokers. This approach requires a failed broker to recover within a finite amount of time, and recover its state from the brokers that it was attached to prior to its failure. *SmartSockets* [24] 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, allows for the mirror process to take over in the case of process failures. The *mirror process* approach runs into scaling problems as the number of processes increase, since each process needs to have a mirror process. Since there is an entire server network that would be mirrored in this approach the network cycles expended for dissemination also increases as the number of server nodes increases. *SmartSockets* also allows for routing tables to be updated in real time in response to link failures and process failures. What is not clear though, is how the system is affected if both the process and its mirror counterpart fail. *TIB/Rendezvous* [25] 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, while continuing 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 do not 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 [23, 48, 47, 22] to utilize a store-and-forward mechanism to fulfill its contract with the sender (producer). A durable subscription is one that outlasts a client's connection with a message broker.

5.1 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 that 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.

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

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 mergers after the partition has been repaired.

Failures could also manifest themselves in the form of node failures, 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 that solve this problem.

5.1.1 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 the problem of message losses and garbage collection. In short, error correction on the link is handled using NAKs while garbage collection is performed using the ACKs.

5.1.2 Message losses due to consecutive node failures

In figure 14.(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 figure 14.(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 approach 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**). Nodes **B** and **C** fail after **B** issued an $ACK(m)$ and before **C** could forward m to **D**. The message m is lost since **A** has already garbage collected it and **D** does not know if it should have received m (for that matter it would not even know about the existence of m to even detect its loss) in the first place.

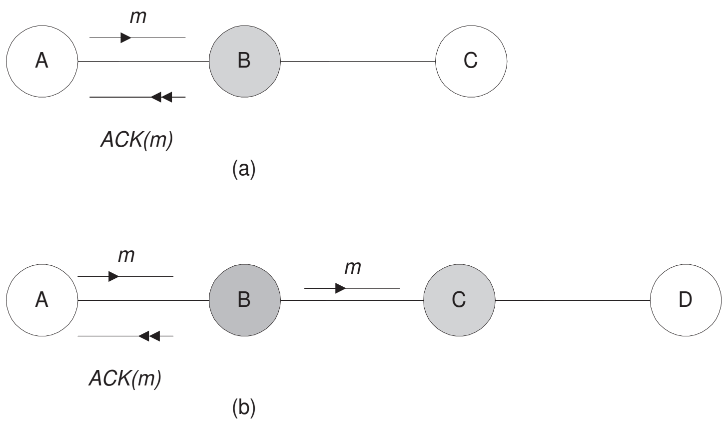

Figure 14: Message losses due to successive node failures

Augmenting the client nodes with reissue behavior till such time that the event has been stored onto a stable storage circumvents this problem. Once an event is stored onto a stable storage, the guarantee is that the event can be recovered even in the presence of failures that could take place. There is a timer associated with every event e that is issued by a client and held in the client's local queue. Unless the client receives a storage notification before the timer's expiry the event would be reissued 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's expiry. If a storage notification is received prior to this timer's expiry the corresponding event is garbage collected from the clients local queue. If such a notification is not received, the event is reissued.

Depending on the client's processing power, this reissue behavior could be delegated to the broker node that the client is attached to. The broker node then is responsible for ensuring that the event 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 broker node, which the client is attached to is functioning correctly.

5.1.3 Gateway Failures

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

Transient gateway failures

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

We use multiple gateways to provide us with a greater degree of fault tolerance. We also 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 connection 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.

5.1.4 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 rerouting of events to the client, which *roamed* to a new location, is based on the replication scheme that presently exists in the part of the system, which the node it was last attached to is a part of. The unit which has stored the events that should have been routed to the client needs to intercept the request for a *reroute* and then proceed with applying the filter operation for the recovery of events.

In sections 5.4.8 we discuss the process of handling events for a disconnected client, while in section 5.4.9 we discuss the process of handling events for a newly reconnected client.

5.2 Network Partitions

Network partitions can be caused both by link failures and node[6] failures. The issues to deal with in the case of network partitions differ considerably from the unit failure cases. Unlike the unit failure

[6] 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

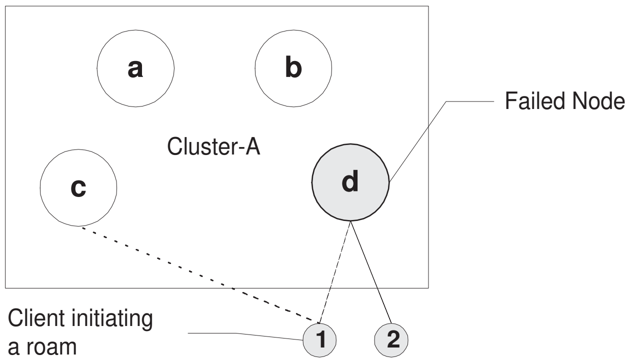

Figure 15: Client roam in response to a node failure.

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.

5.2.1 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 the partitions that 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 partitioned units can roam to nodes that are in the majority partition. This calculation of the nodes to roam to during partitions could also be based on some system defined rule. 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 16 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} that comprises of connections to nodes $n_i^a \ni a \leq \ell$ and the set L_D comprising of connections to nodes $n_i^b \ni b > \ell$.

Figure 17 depicts the connectivity graph that is constructed at the node SSC-A.SC-1.c.6 in Figure 16. 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 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, then we decrement the link count associated with the connection and conclude that the connection loss is compensated by other existing connections between the two corresponding units. 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 $x, y \leq \ell$), then 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

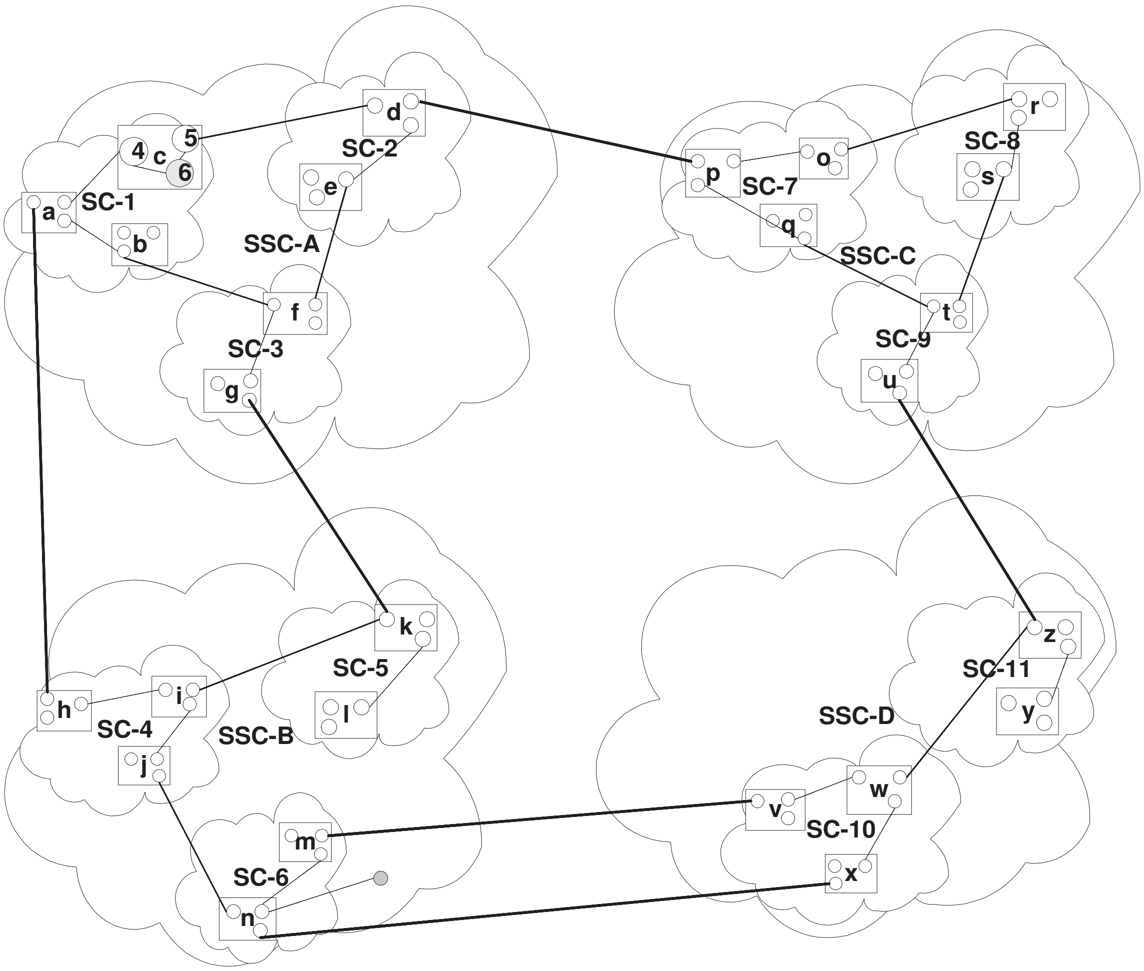

Figure 16: Connectivities between units and the detection of partitions

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 $\#L_{UL} = 0 \bigcap \#L_D = 0$ (where $\#$ is the cardinality of the individual sets), then the unit corresponding to the node is system partitioned. If $\#L_{UL} = 0$, then the unit corresponding to the node is unit partitioned.

Referring to the connectivity graphs in Figure 17, in the case of node **6**, the loss of the connection between clusters **a** and **b**, results in the cluster **b** being unit partitioned within **SC-1** though it is 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 17 concludes that cluster **b** has been system partitioned.

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

5.2.2 Ensuring progress in concurrent partitions

Concurrent partitions may contain clients which issue events and also other clients which are interested in those events. The interested clients should thus be able to receive events which are currently being issued within that partition. All these events would, of course, need to be stored onto a stable

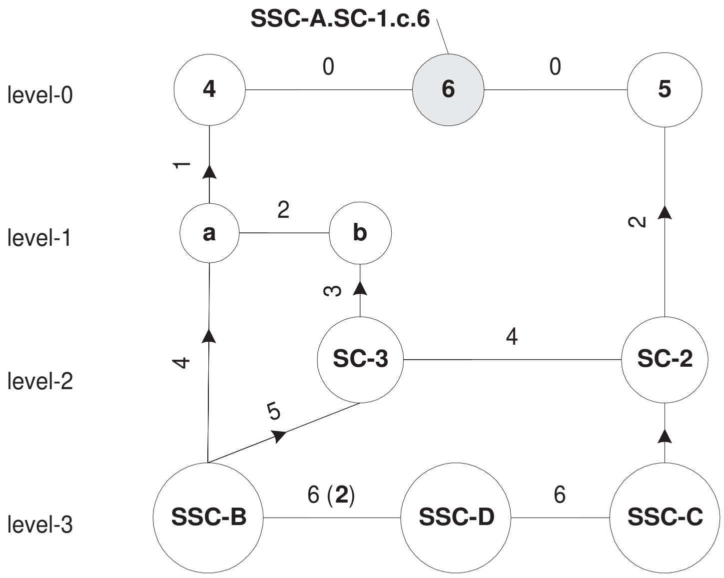

Figure 17: The connectivity graph at node 6 and the detection of partitions.

storage, for rerouting during partition mergers.

5.2.3 Partition Mergers

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

5.3 Stable Storage Issues

Storages exist en route to destinations but decisions need to be made regarding when and where to store an event and also on the number of replications that we intend to have for any given event. Events can be forwarded to clients only after they have been written to stable storage. The greater the number of stable storage hops en route to delivery to a client, the greater the latency in delivering the event to that client. We also need to address the issues pertaining to the control of the replication scheme. In section 5.3.1 we discuss the replication scheme for our system, and the process of adding stable storages within a sub-system. Section 5.4 describes the need for epochs, the assigning of epochs and the storage scheme for events. Section 5.4.5 describes the guaranteed delivery of events to all units within the subsystem. Finally in section 5.4.9 we describe the recovery scheme for roaming clients or clients connecting back after a prolonged disconnect.

5.3.1 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* that exists within a specific unit. For a level-ℓ system, if there is a stable storage set up for servicing all the broker nodes within that unit, then 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 M units at each level, a certain event that would be received by all the clients within the system would be replicated $M \times M \times M$ times. Of course what we are considering here is the extreme case, but nevertheless, it is an exemple of how the replication strategy is a crucial element within the system. We also need a garbage collection scheme, which ensures that the storage space does not 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 throughout the system. Instead, we impose a constraint on the minimum replication scheme for the system. In an N-level system, comprising of level-N units, we require that every node have a replication granularity of at least r_N. Thus in a system comprising of super-super-clusters we require that every broker 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 within the system that have a replication strategy, which is more finely grained. The other constraint, which we impose is that within a level-ℓ unit u_i^ℓ there can be only one stable storage at level ℓ.

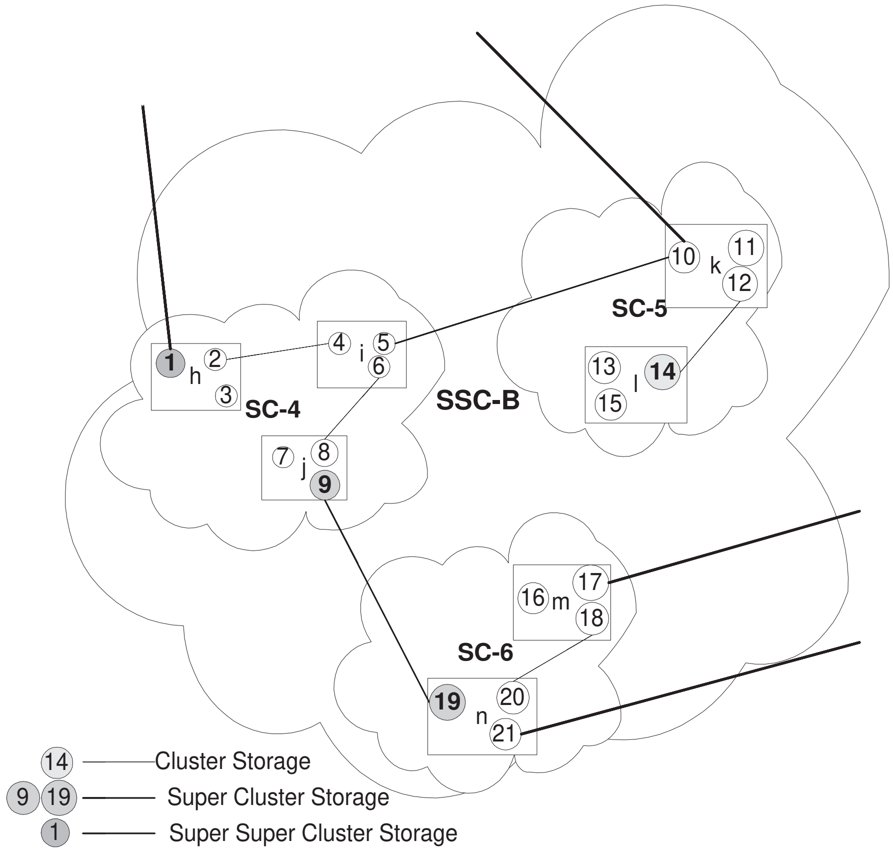

Figure 18: The replication scheme

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 every level-$\ell - 1$ unit within its level-ℓ GES context C_i. Thus, if we fix the replication granularity at ℓ, one of the gatekeepers $g^\ell(C_j^{\ell+1})$ 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; since a unit gatekeeper is aware of the unit's profile.

Figure 18 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** that serves as a stable storage to provide the 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 4 lists the replication granularities available at different nodes within the sub system depicted in figure 18.

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 4: Replication granularity at different nodes within a sub system

Requirements (5.1), (5.2) and (5.3) snapshot the various constraints that we impose on our replication strategy.

Requirement 5.1 *In an N-level system, the replication granularity at each and every node in the system must be at least r_N.*

Requirement 5.2 *A level-ℓ stable storage can only be set up at a node, which serves as a level-ℓ gatekeeper.*

Requirement 5.3 *For a level-ℓ unit u_i^ℓ only one of the gatekeepers g^ℓ can be configured as a level-ℓ stable store.*

5.3.2 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 stable storage 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 refer to the minimum replication granularity and the next highest one respectively. 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 5.4.4).

To sum up our discussions so far, let us consider the topology in figure 18. We then proceed to set up a stable storage at node **SSC-B.SC-6.m.18**. Further, we configure this stable store as a cluster storage with a replication granularity of 1. The replication granularities at nodes **16,17** and **18** are then updated to r_1 from r_2. If however, we were to set up a level-2 stable storage at node **10**, the replication granularities at nodes **10,11** and **12** would be updated from r_3 to r_2. The replication granularity for every broker node in the cluster **SC-5.l** remains unchanged at r_1.

5.3.3 Stability

For an N-level system with a minimum replication granularity of r_N, the responsibility for ensuring stability of messages is delegated to the finer grained stable storages for those sub systems where the replication granularity is less than N. In the case of multiple stable storages, 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 figure 18 stability for nodes **13,14** and **15** is handled by the cluster storage at node **14** while that for nodes **10,11** and **12** is handled by the level-3 stable storage 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 the way an event's destinations are computed at a storage node. When a node is hosting a stable storage with r_ℓ, this node is responsible for computing destinations, comprising of units at level-(ℓ-1), within the level-ℓ unit that the node belongs to. Along 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 the destinations associated with a given event. The acknowledgements include the number of predicates that were serviced at the destinations. The destination associated with the acknowledgement is updated depending on the gateway that the destination is being transmitted over. For acknowledgements issued by a broker node that has a replication granularity of r_ℓ the acknowledgement is never sent over a level-ℓ gateway g^ℓ. Thus, acknowledgements to decrement the 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, this stable storage sends a notification to the coarser grained storage indicating the receipt of the event and also the predicate count that can be decremented for the sub-unit that this storage is servicing. Thus, in figure 18, 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 storage at node **1** accordingly.

5.4 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. Along 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 an event upon receiving acknowledgements from the event's destinations. Consider the following scenario where the predicate count equals the client count for the destination associated with an 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 its 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 and that this client had joined the system for the first time prior to the unit's 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 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 reconnected clients and also to recover 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 the stable storages that exist at different levels within the same GES context.

5.4.1 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, it is the GES context of the broker node that the client 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 clients exist within the context of the finest grained stable storage that the broker node (that it is attached to) is a part of. Thus, if the broker node has a replication granularity of r_2, valid epochs for events received by the client, would be 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 one stable storage configured for servicing a unit u_i^ℓ with a granularity of r_ℓ.

Requirement 5.4 *A persistent client will not receive an event e unless there is an epoch, ξ_e, associated with the event. Also, this epoch should be assigned by the stable storage servicing the broker node that this client is attached to.*

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 from failures and also to service 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 do not 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 subsystem having a replication granularity r_ℓ, it is possible that there are subsystems 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, before forwarding this event to the relevant clients attached to this node. Thus the epoch associated with the same event could be different at different clients in the system. In figure 18 it is possible that by the time an event arrives 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 18, have the same epochs (assigned by the store at node **19**) for events that they would all receive.

5.4.2 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 rejoining after a prolonged disconnect. Using this information these reconnecting clients are still able to ensure that the events they consume can be scheduled for garbage collection from the stable storages, once the reference count for these consumed events reduces to zero. We also maintain information pertaining to the *type* of the event and the length of the serialized representation of the event. Finally we maintain the serialized representation of the event. Thus, the storage format is the tuple – $< \xi^e, (d_0^e, d_1^e, \cdots, d_n^e), (p_0^e, p_1^e, \cdots, p_n^e), e.type, e.length, e.serialize >$.

5.4.3 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 duration this client does not 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 the client rejoins the system at a later time, this client would expect to receive all the events that it missed. This set of missed events would include events, which satisfy the profile change the client 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 waits till it receives the epoch notification, before it can disconnect from the system.

5.4.4 Epochs and the addition of stable storages

In this section we describe the process of adding stable storages. Consider a scenario where a new store is being added within a unit u_i^n. The present replication granularity of this unit is r_m and the new storage for this unit is at level-n. The addition of a stable storage at level n is disseminated only within the unit u_i^n that the hosting node belongs to.

If $n < m$ the new stable storage should access the storage with r_m and retrieve the events which were meant to be disseminated within the unit u_i^n. The predicate count associated with the destinations for each individual event needs to be updated accordingly to reflect the predicate counts associated with the sub-units in u_i^n. The epochs associated with these *retrieved* events should however remain unchanged. This is especially crucial since there are clients, attached to nodes in the unit u_i^n, which have epoch numbers associated with their profiles based on the ones assigned by storage hosting r_m. The epochs associated with the client profiles should remain consistent even if a new stable storage is added. Once this event retrieval process is complete, the newly added stable storage is ready to assign epoch numbers to the events. The first event that this newly added storage is ready to store, after the retrieval process is complete, is the epoch number from which point on the epoch numbers assigned by the old store r_m and the new store r_n can deviate. If $n > m$ at any of the sub-units within u_i^n the replication granularity for nodes in those sub-units will not be updated.

Figure 19 depicts the replication scheme that exists in different parts of the system and also the addition of a new super-cluster storage at node **10**. Prior to the addition of the stable storage at node **10**, the replication granularity at the broker nodes in cluster **k** is r_3 while that in cluster **l** is r_1. When the new level-2 storage is added, this information is disseminated only within the super-cluster **SC-5**. After the dissemination of the storage information, node **10** needs to communicate with the r_3 storage at node **1** and retrieve the events that have **SC-5** in the destination lists. These *retrieved* events when they are stored at node **10** have their epoch numbers unchanged. This is because clients attached to the nodes in cluster **k** have their profile epochs updated by the events

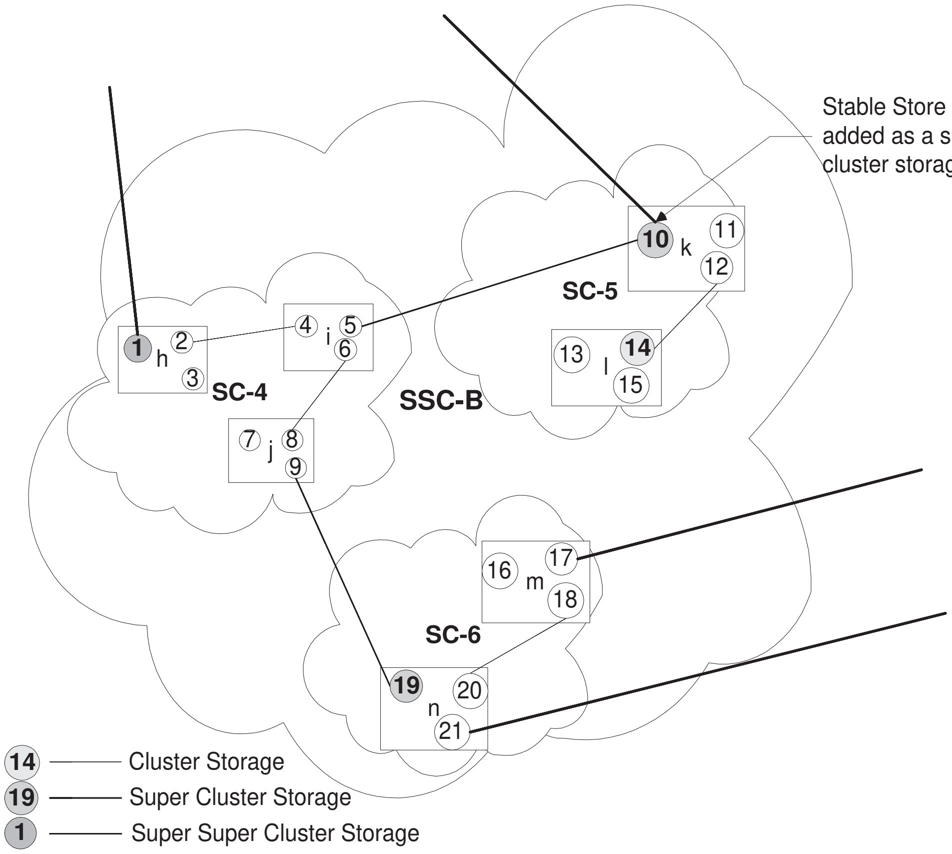

Figure 19: Adding stable stores

with epochs assigned by the r_3 storage at **1**. These epochs thus need to be consistent across the addition of stable storages. Let us say that the epoch associated with the last such retrieved event by node **10** is $\xi_{250}^{node(1)}$, the next time an event arrives at node **10** the epoch assigned to that event would be $\xi_{251}^{node(10)}$. Thus clients which were attached to cluster **SC-5.k** in an earlier incarnation, when they reconnect back, can use their epoch to recover completely. For the stable storage at node **10** the destination list and the corresponding predicate counts associated with the *retrieved* and *new* events should be with respect to the clusters **k** and **l**.

The addition of the level-2 stable storage at node **10** has no effect on the replication granularity of broker nodes in cluster **SC-5.l** since for the broker nodes in this cluster, the storage that is added is a coarser grained stable storage.

5.4.5 Ensuring the guaranteed delivery of events

For a level-N system, the stable storages servicing the individual level-N units are also designated as *system storages*. Figure 20 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 units within the system have received that event. When an event is issued within a super unit u_i^N, the destinations are computed as described in the event routing protocol. However, before the event is allowed to leave unit u_i^N, it must be stored onto the stable storage that provides nodes in u_i^N with the minimum replication granularity of r_N. Thus, in figure 20, for an event issued by a client attached to a node in **SSC-B**, that event must be stored to the system storage in **SSC-B.SC-4.h** before it can be routed to units **SSC-A**,**SSC-C** and **SSC-D**.

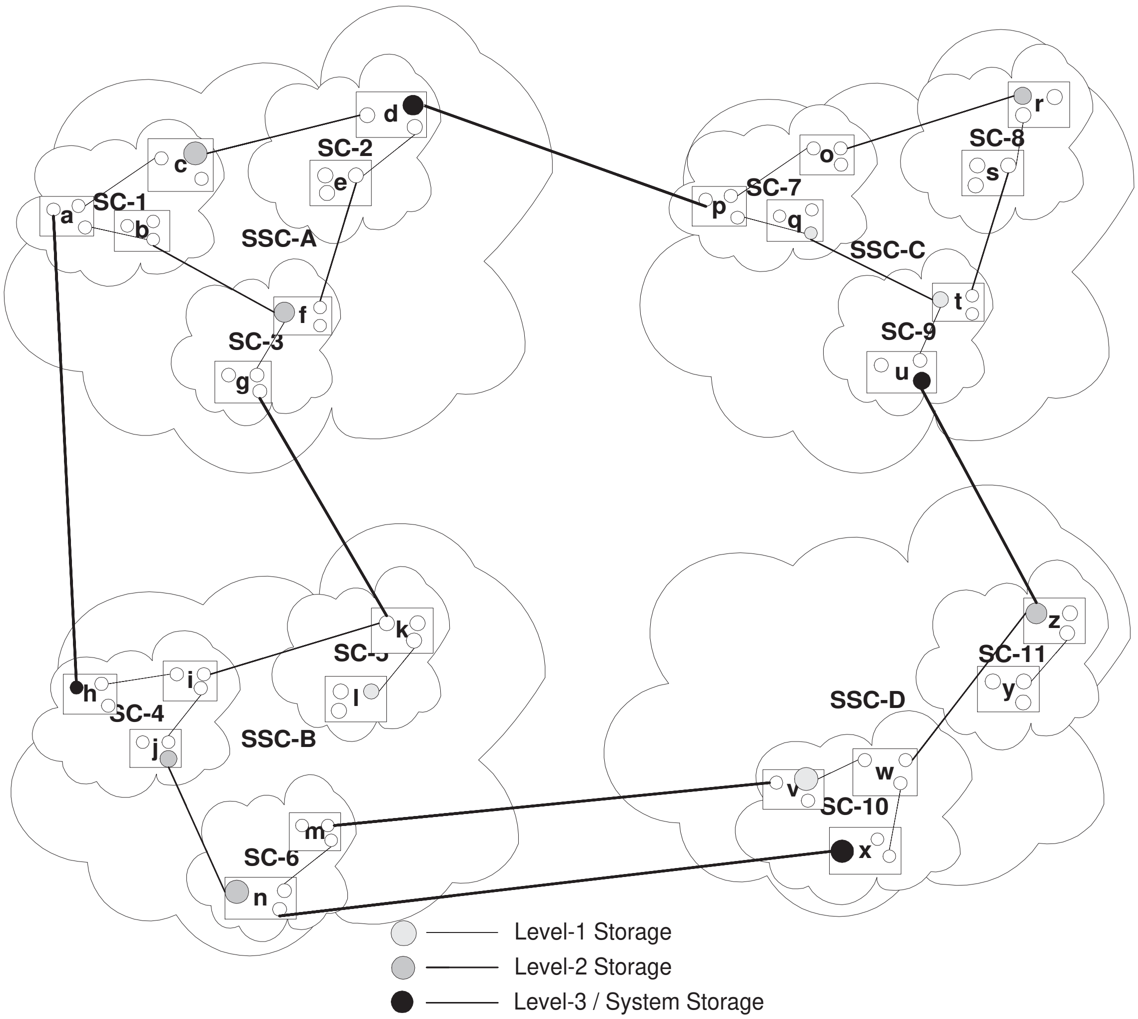

Figure 20: Systems storages and the guaranteed delivery of events

The system storage 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 storage. Associated with these events is a sequence number, which is different from the epoch number associated with the events that clients receive. Further, sequence numbers associated with events are used *only* by the system storages to conjecture the events that they should have received from any other system storage within the system. These sequence numbers are *not* used by the clients or the broker nodes within the system to detect missing events. Once the event is stored to such a system storage, it is ready to be sent across to the 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 units within the system) within u_i^N and not at any other system storages at the other u^N units within the system. When the events are being sent across gateway g^N for dissemination to other u^N units, every event has a sequence number associated with it and also the unit u_i^N in which this 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 also keeps track of the last sequence number that was received from a certain unit u_i^N. Thus the system storage 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**. Each system storage can now keep track of the 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 system storage node then issues a NAK to retrieve those events. When the system storage at **SSC-B** receives this NAK(s_{101}^B, s_{102}^B), it reissues those events to the requesting system store. The system storage at **SSC-A** does not assign an epoch (the system storage can assign an epoch because it is also a level-N storage) or route the event with sequence number s_{103}^B till such time that it receives events with s_{101}^B, s_{102}^B from **SSC-B**. If an event with sequence number s_{98}^B (assigned by **SSC-B**) is received, the system storage 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 $s_{97}^B, s_{96}^B, \cdots s_0^B$.

Upon receipt of an event with an associated sequence number, other system storages issue an *ACK(seqNum)* to facilitate garbage collection. The fact that sequence numbers assigned by any system storage increase 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 (issued by the unit's system storage) 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 $s_1^B, s_2^B, \cdots, s_n^B$ the destination list associated with every event stored by the system storage comprises of **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 events with sequence numbers $s_1^B, s_2^B, \cdots, s_n^B$. When the destination list associated with any of these stored events is reduced to zero that event is garbage collected.

5.4.6 The upward propagation of events

When an event is issued by a client attached to a broker 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 broker nodes. The epoch that is associated with an event should be the epoch that is assigned by the servicing storage for the broker node in question. Events with epochs assigned by replicators r_ℓ are valid only within the unit u_i^ℓ 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_N), the stable storage node is responsible for storing this event and not scheduling it for garbage collection, till such time that it receives a notification from the system storage regarding the receipt of that event. Besides this, for an 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 21 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 reissue 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 storage and also to initiate corresponding reissues 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. This indication is stored in the event via the sequence number assigned by the system storage. An given event will be stored only at one system storage. The system storage 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

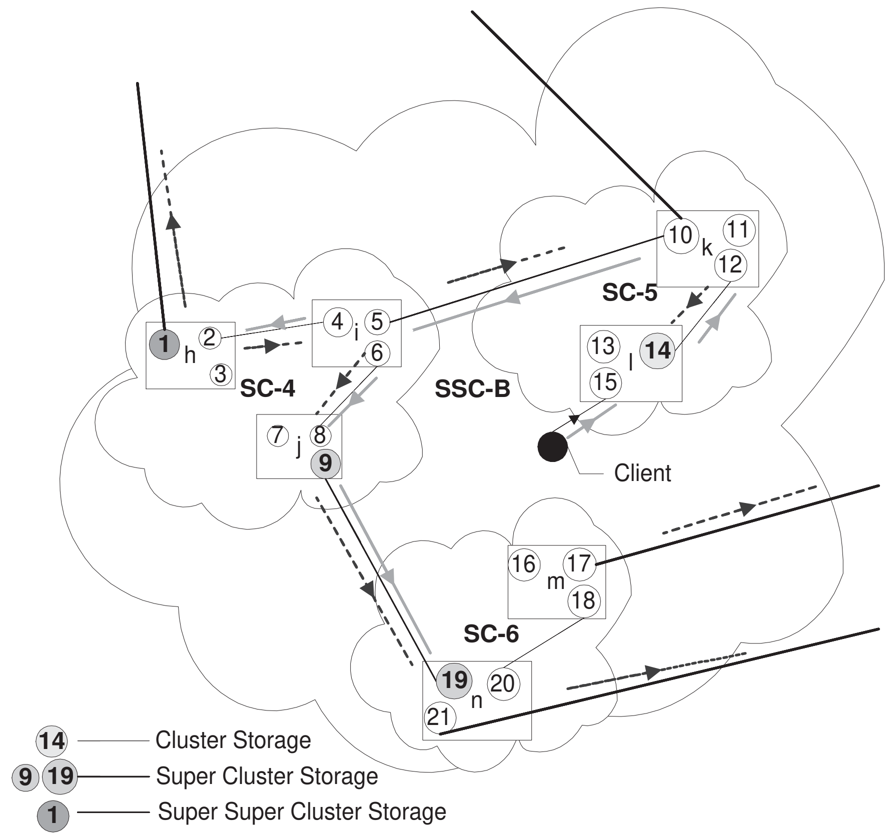

Figure 21: The propagation of events in the system

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**, besides the sequence number assigned by the system storage at node **1**.

5.4.7 Stable storage failures

When a stable storage node fails, the events that it stored would not 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. We require this stable storage to recover within a finite amount of time.

Requirement 5.5 *A stable storage cannot remain failed forever, and must recover within a finite amount of time.*

5.4.8 Handling events for a disconnected client

This problem pertains to one of the most important issues that needs to be addressed by our system. A client node has intermittent connection semantics, and is allowed to leave the system for prolonged durations and still expect to receive all the events that it *missed* in the interim, along with real time events. Events are routed based on a clients persistent profile and the persistent profile is what would be stored at the last broker node that it was connected to. The broker 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 broker 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 is to delegate this responsibility to the broker node that the 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 that this client was being serviced by. The events are thus available for the construction of recovery queues when the client connects back into the system.

5.4.9 Routing events to a reconnected client

The client in question could be both a roaming client or a client which has reconnected after a prolonged disconnect. Associated with every client is the epoch number 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 that 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 lose 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 client's 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 5.1 *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 5.5 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 create and route these 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) The logical address of the broker node that this client was attached to in 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 need not 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) provides us with the stable storage that has stored events for the client. 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.

5.4.10 Locating the stable store

When the client reconnects back into the system, based on the logical address of the broker node that this client was last connected to, the stable storage responsible for assigning epochs to clients attached to that particular broker node is located. This works as follows, the request is first forwarded to the broker 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 broker node is down, the information could be retrieved from the cluster gateway for the cluster that the node is a part of. Since unlike a broker node without a stable store, a storage node is not allowed to remain failed forever – the servicing storage will always be retrieved.

5.4.11 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 5.4.2 an event e is stored in the following format – $< \xi^e, (d_0^e, d_1^e, \cdots, d_n^e), (p_0^e, p_1^e, \cdots, p_n^e), e.type, e.length, e.serialize >$. For any recovering client we first locate the client's servicing storage, based on the logical address of the broker node that this client was attached to in its previous incarnation. 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 *located* stable storage when the recovery request was received. This epoch indicates the point from which point on 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. The set of events at the located stable storage, which form the preliminary set of events to be considered for recovery, are events with epochs greater than ξ_n and less than or equal to ξ_m. The number of events in this preliminary set would be less than or equal to $(m - n)$ since some of these 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 broker node (the client was attached to in its previous incarnation) 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 at the located stable storage.

5.4.12 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, at the subsystem the client was attached to in its previous incarnation. 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, in general, 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 broker node which last hosted this client.

When the client issues an 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 broker node that the client is now attached to. These recovery events are now managed by the stable storage servicing the broker node that the client is now attached to. This stable storage is responsible for issuing acknowledgements in response to the receipt of the recovery events. Upon receipt of acknowledgements from the new storage, the corresponding predicate count associated with the event at the old storage is decremented by one. If this count is reduced to zero, the destination is removed from the destination list. When this destination list is reduced to zero, the event is garbage collected at the old stable storage. 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 this case the broker node that the client was attached to prior to the roam, is now responsible for ensuring that the client does not loose any events that it is interested in. In case of client roam or storage failures during reconnection there is 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 could not 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.

5.4.13 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 storage would have stored events destined for the unit that was being serviced by the now failed stable storage. This higher level stable storage then release these stored events to the recovering lower level stable storage when it recovers after the failure. 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.

5.5 Summary

In this section 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 system storages 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.

6 Pub/Sub Model, Dynamic Topologies & Security

6.1 The GES publish subscribe Model

In the GES publish/subscribe model, clients can attach themselves to any of the nodes comprising the server network. Clients express an interest in the kind of events that they are interested in through their profiles. A client's profile comprises of a number of subscription predicates, each of which specifies a different content that the client is interested in. We place no limit on the number of subscription predicates that a client can specify in its profile. A subscribing client could also be a publishing client, and there are no limit on the different *topics* that a client can publish.

To specify the precise instant of time from which point on a client's profile change is active, we have the notion of active profiles and epochs associated with profile changes. Essentially a client is notified about the system's awareness of the client's profile change. When a profile change is *system active*, events issued by any of the publishers in the system, from that point, will be received at the client, if it matches the client's profile.

Events can either be *persistent* or *transient* events. Transient events exist only within a real-time context. Delivery of these events beyond its self-imposed real time context is not allowed. Transient events could have variations where the system consumes the event after a certain number of server node hops. We refer to such events as *hop-constrained transient events*. At each of the server node hops, the hop associated with the event is incremented by one. When these transient events are received at a server node, if the hop-limit is reached, the server node simply discards the event thus preventing any further routing for that event. The routing characteristics associated with such events can be considered as ripples in a pond, which occur in concentric circles for some distance from the origin of these ripples. Similarly, there are *level-constrained transient events*, which have all the properties of transient events but are constrained by the unit within which they can be disseminated. These events are constrained such that they are not allowed to be routed outside the unit/super-unit that they were issued in. Persistent events need to be stored to stable storage, to account for the system reliability guarantees associated with them. Persistent events also have another flavor, the *time-constrained persistent events*. These events are identical to persistent events except that as soon as these events are stored onto a stable storage, the garbage collection timer, associated with every such time-constrained persistent event, starts ticking. Upon the expiry of the timer, these time-constrained persistent events announce themselves as being ready for garbage collection.

Clients can either be *durable* (persistent) or *non-durable*, the difference being in the reliability of events that are delivered to them. Durable clients can leave the system, fail or roam in response to failure suspicions or need for better response times. The system guarantees that all persistent events issued during this time will be delivered to the client across its various incarnations at different parts of the system. This guarantee holds true even in the presence of failures. In our failure model a unit can fail and remain failed forever. The time-constrained persistent events that are routed to a client are those for which the timer has not expired. The timer's associated with these time-constrained events can vary from a few minutes to up to a few days. All subscribing clients (durable or non-durable) receive transient events, the system can compute alternate routes in response to link or node failures. The system does not guarantee the delivery of these events during failures. All clients receive hop-constrained transient events if they are within the line-of-sight for the publisher of these events. Table 5 outlines our discussion regarding the different kinds of events and clients.

6.2 The need for dynamic topologies

This pertains to the scheme for the dynamic creation of servers, to optimize the routing characteristics for events. The routing characteristics pertain to the bandwidth usage, response times and also on the protocols that would be employed for the dissemination of events. Consider the scenario where there are server nodes at Syracuse and Rochester. A large number of client nodes attached to one of these servers reside in Boston, Houston and Albany. For a set of clients at either of the aforementioned locations this scheme has the obvious disadvantage that messages routed to each of

Published Event	**Non-Durable Client**	**Durable Client**
Transient	at-most-once (missed if inactive)	at-most-once (missed if inactive)
Transient (hop or level constrained)	at-most-once (if within the publisher's line of sight, missed if inactive)	at-most-once (if within the publisher's line of sight, missed if inactive)
Persistent	at-most-once (missed if inactive)	once-and-only-once
Persistent (time constrained)	at-most-once (missed if inactive)	once-and-only-once (missed if the duration of disconnect, for the inactive durable client, is greater than the event's garbage collect timer)

Table 5: The GES publish/subscribe model

the clients utilizes the same bandwidth between the server and client's location. For 10 clients (at the same geographic location) attached to the same server node, for a certain event, the bandwidth could be utilized 10 times for the same event.

The system in response to such a scenario should proceed with the instantiation of server nodes at the client locations. In the present discussion we are referring to locations where a large number of clients reside. Inducing a roam in clients based on their geographic location would then follow this dynamic instantiation of a server node at one of the clients. The induced roam should be towards the newly created server node. Thus in the scheme for routing messages the bandwidth between two locations is utilized only once per message. The long links created between the original server node and the newly created one would normally employ TCP for communication. The newly created server nodes could employ a different approach, e.g. IP Multicast, for disseminating the received events to relevant clients. This when employed with the routing schemes in place would greatly improve system performance, and response times at the clients. Similarly publishing clients could be induced to roam to a location where there is a high concentration of clients interested in receiving the published events.

Other schemes that could be employed include dynamically creating connections between nodes in different units, to create *small world* networks. Further use of schemes to identify *slow* links, removal of these links and the creation of new *fast* links would also greatly improve system performance. Interesting variances of parallel computing algorithms could be employed for this purpose. An analogy resides in hyper cubes where links are created/removed from the 3D mesh of nodes to achieve logarithmic pathlengths.

In our failure model a unit can fail and remain failed forever. The server nodes involved in disseminations compute paths based on the active nodes and traversal times within the system. The routing scheme is thus based on the state of the network at any given time. Thus servers could be dynamically created, connections established or removed, and the events would still be routed to the relevant clients. Any given node in the system, would thus see the server network undulate as the servers are being added and removed.

Connections could also be instantiated dynamically based on the average pathlength for communication with any other node within the system. The connectivity graph maintains abbreviated system views; each node in this graph could also maintain information regarding the average pathlengths for communication with any other node within the unit, which the graph node represents. Connections could be dynamically instantiated to vary clustering coefficients and also to reduce average pathlengths for communications.

6.3 The proposed security framework

The security framework should ensure that the software installed by users of the system is one that has not been tampered with. The system them needs to be able to provide a series of authentications, these include authentication of clients to the system, authentication of servers to the clients and authetication of servers to other servers. Finally we also need to able to secure disseminate information within the system. We now proceed to outline our approaches to each of these issues.

6.3.1 Installing the Software

The software for the package comprises of several files. These files are usually compressed and this compressed file is then made available for download. A hacker could modify some of these files. These modified files are then compressed and made available for download. Users who download these files are not aware if these files have been modified. To ensure that the files have not been modified, we employ message digest functions. We could employ the MD5 message digest algorithm (RFC 1321). MD5 developed by Ron Rivest [67] at MIT takes as input a message of arbitrary length and produces a 128-bit message digest while processing the input in 512-bit blocks. The software is then placed at the download site and users are free to download this software along with the message digest for the software. Users compute the message digest associated with the software, if the value returned matches the value posted on the website then the user concludes that the files have not been modified.

It is however possible that the software has been modified by a hacker. This hacker then proceeds to compute the message digest and places the modified software along with the newly computed message digest at the server. Users would then download this malicious software and when they compute the message digest, they would find that it matches with the one available at the web site. What needs to be ensured is that only the author of the package is allowed to modify the contents in the package, and not someone who has gained access to the server hosting the website. The author of the package needs to be able to state the value of the message digest while ensuring that no one else can change it.

To solve this issue of identity we use digital signatures. The author then encrypts the message digest function with his private key and users employ the author's public key to decrypt the message and retrieve the message digest function. After downloading the software users compute the message digest, if they match it can be assumed that the software was developed by the author/organization whose public key is available for decryption. Since only the author is aware of the private key, only the author could have produced a valid signature. The approach for digital signatures outlined here is the scheme employed by RSA [66] for digital signatures, we could also use the Digital Signature Standard (DSS) approach to providing digital signatures.

It is however possible that someone other than the author has modified the files, computed the message digest functions and then signed it with their own private key and put up the public key for download for subsequent decryption of the message digest. To circumvent this problem we use digital certificates, which provide a binding between the public keys to individuals or organizations. These certificates are issued by certifying authorities (CAs). Of course the CA must be someone whom both the author and the users trust. The CA is not responsible for the creation of the private-public key pair, and merely provides an accessible location for retrieving the public key associated with the author. The CA signs the certificate with its private key, this certificate contains the author's public key. Users then use the author's public key to verify the message digest function.

6.3.2 Authentication

There are several levels of authentication. The first is of course that of authenticating the client. Clients connect to one of the servers for access to services. Any server that a client connects to then needs to authenticate itself to the client. A newly added server needs to be able to authenticate itself to the other server processes that it would connect to.

Authenticating clients

We now very briefly outline the Kerberos [73, 51] Authentication service that we intend to use.

The Kerberos Authentication server (AS) keeps of valid userID's and their passwords. Every server in the system has a unique ID assigned by the system, this information is also maintained by the AS. When a client wishes access to a set of services, the client issues a ticket granting request on behalf of the user along with the ID of the Ticket Granting Server (TGS). The AS responds with a ticket encrypted using a key derived from the user's password. The ticket is recovered only if the user enters the correct password, which would cause the key to be regenerated. The ticket granting ticket that this client receives from the AS is encrypted using a key that is known only to the AS and the TGS. The client then issues a request to the TGS requesting a service-granting ticket. The request comprises of the UserID, the ID of the desired service and the ticket granting ticket. Since the ticket granting ticket is encrypted using a key that only the TGS and the AS are aware of, the TGS decrypts the incoming request and confirms success in the decryption on finding its TGS-ID. It verifies the userID and associated network address to authenticate the user. The TGS also checks to see if the lifetime associated with the ticket-granting ticket has expired. If the lifetime has not expired and if the user is permitted to access the service a ticket is issued for that service.

This service granting ticket issued by the TGS is encrypted using a key known only to the TGS and the server that would provide access to the service, which the client seeks. The client proceeds to use this ticket for access to the server. This sever then authenticates the client based on the contents contained in the service-granting ticket.

Authenticating the server that a client connects to
For authenticating the server to the client, Kerberos also has the notion of a session-key. This session key is securely routed by the AS to both the TGS and the client. When the TGS issues a ticket granting ticket it includes the session key, the server V that should receive this ticket, and a timestamp at which this ticket was issued. Server V is able to decrypt the ticket since it is encrypted based on a key that only TGS and V are aware of. Server V then retrieves the session key and uses this session key to decrypt the authenticating timestamp. It increments the timestamp by one and returns it to the client after encrypting it with the session key. Since it is encrypted with the session key the client is able to confirm that it could have been created only by the right server V, the contents further guarantee that this message is not a replay.

Authenticating a new server process to existing server processes
When a new server process is added to the distributed server network, it is essential to verify if this server process runs the original software. It is conceivable that a user can make changes to the software, after the user has downloaded the original software. This software can then be modified such that it could serve as a black hole within the server network. Events could be routed to this modified server process and no further processing would be done by the process. Of course such a server could also perform other far more harmful activities. Users instantiating new servers would first need to authenticate themselves to the Kerberos authentication server in much the same way as clients do. Furthermore we check to ensure if the message digest, signature and certificates associated with the servers are intact. If these checks succeed we then proceed to share a unique key between the AS and this newly added server.

6.3.3 Communication between servers

We now proceed to outline schemes that could be used for encryption of traffic between the servers. The two possible approaches could involve symmetric key based approaches or asymmetric keys based one. Keys for communication between two servers, over a specific link, could be securely exchanged using the Diffie-Hellman [28, 29] key exchange. We could then encrypt the traffic between the servers using this key. We could use Data Encryption Standard (DES) or triple-DES for encrypting the traffic. In the event that we plan to use triple DES we would need to exchange at least two keys securely.

In the asymmetric key based approach to encryption we could use RSA. Let us say that we need to encrypt the traffic between two servers A and B. Further, servers A and B are aware of the public keys that belong to each other. For data being sent from server A to server B, server A encrypts data to be routed to server B. This encryption is done using server B's public key and server B decrypts this data using its private key.

6.3.4 Communication between server and clients

Communication between servers and clients can be encrypted in much the same as the traffic between two servers is encrypted.

7 Results

In this section we present results pertaining to the performance of the Grid Event Service (GES) protocols. We first proceed with outlining our experimental setups. We use two different topologies with different clustering coefficients. The factors that we measure include latencies in the delivery of events, variance in the latencies and system throughputs among others. We measure these factors under varying publish rates, event sizes, event disseminations and system connectivity. We intend to highlight the benefits of our routing protocols and how these protocols perform under the varying system conditions, which were listed earlier.

8 Experimental Setup

The system comprises of 22 broker node processes organized into the topology shown in the Figure 22. This set up is used so that the effects of queuing delays at higher publish rates, event sizes and matching rates are magnified.

Each broker node process is hosted on 1 physical Sun SPARC Ultra-5 machine (128 MB RAM, 333 MHz), with no SPARC Ultra-5 machine hosting two or more broker node processes. For the purpose of gathering performance numbers we have one publisher in the system and one *measuring subscriber* (the client where we do our measurements). The publisher and the *measuring* subscriber reside on the same SPARC Ultra-5 machine and are attached to nodes **22** and **10** respectively in the topology outlined in figure 22. In addition to this there are 100 subscribing client processes, with 5 client processes attached to every other broker node (nodes **22** and **10** do not have any other clients besides the publisher and measuring subscriber respectively) within the system. The 100 client node processes all reside on a SPARC Ultra-60 (512 MB RAM, 360 MHz) machine. The publisher is responsible for issuing events, while the subscribers are responsible for registering their interest in receiving events. The run-time environment for all the broker node and client processes is Solaris JVM (JDK 1.2.1, native threads, JIT).

9 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*, *event sizes* and *matching rates*. Publish rate corresponds to the rate at which events are being issued by the publisher. Event size corresponds to the size of the individual events being published by the publisher. 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 events received at a client we calculate the *mean latency* for the sample of received events, the *variance* in the sample of these events and the *system throughput* measured in terms of the number of events received per second at the measuring subscriber. We also measure the highest and lowest event latencies within the sample of events that have been received. Another very important factor that needs to be measured is the change in latencies as the connectivity between the nodes in a broker network is increased. This increase in connectivity has the effect of reducing the number of broker hops that an event has to take prior to being received at a client. The effects of change in latencies with decreasing broker hops is discussed in section 10.5.

9.1 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

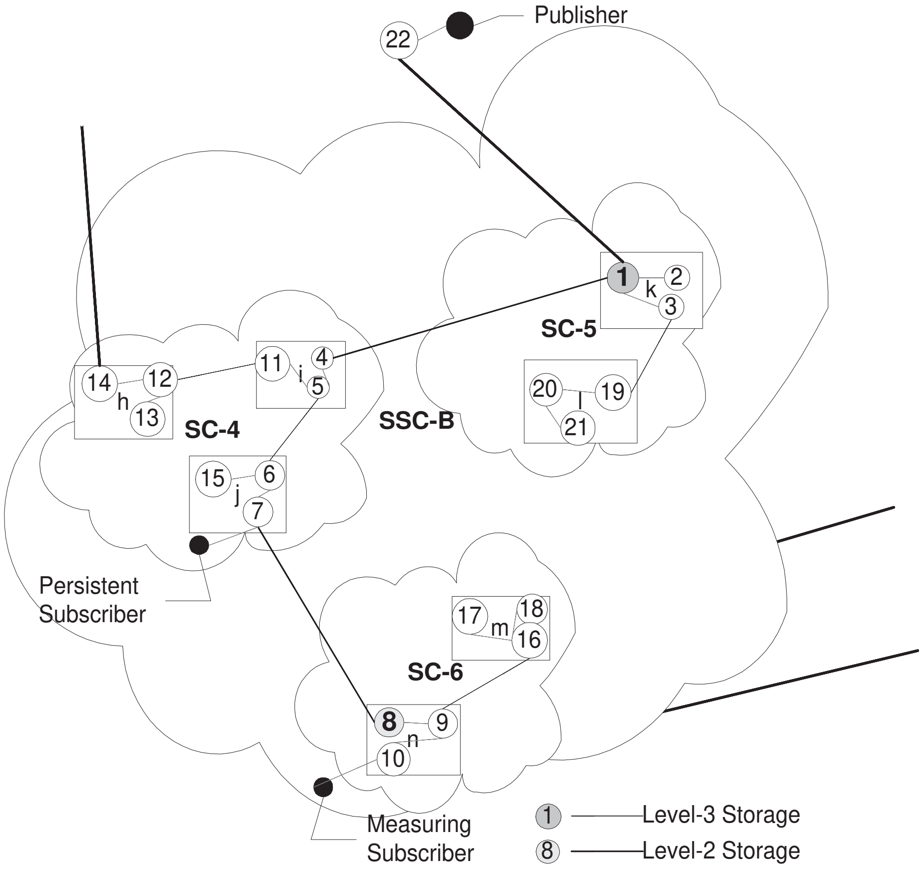

Figure 22: Testing Topology - (I)

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 broker 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 the events 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 subscribing client within the system 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 events 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 events that are published at a time have been fixed (for the publisher and the broker node that it is attached to), we proceed to compute the real publish rates for the sample of events that we send. This is the publish rate that we report in our results.

For each matching rate we vary the size of the events from 30 to 500 bytes, and vary the publish rates at the publisher from 1 Event/Sec to around 1000 Events/second. For each of these cases we measure the latencies in the reception of events. To compute latencies we have the publishing client and the *measuring* subscriber residing on the same machine. Events issued by the publisher are *timestamped* and when they are received at the subscribing client the difference between the *present* *time* and the *timestamp* contained in the received event constitutes the latency in the dissemination of the event at the subscriber via the broker network. In case the publisher and the subscriber are on two different machines, with access 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 event 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 broker nodes within the broker network, as depicted in figure 22. In fact it takes 9 broker hops for an event issued by the publisher to be received at the measuring subscriber.

10 Discussion of Results

In this section we discuss the latencies gathered for varying values of publish rates, event sizes and matching rates. We then proceed to include a small discussion on system throughputs at the clients. We also discuss the trends in the variance of the latencies, associated with the sample of events 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.

10.1 Latencies for the routing of events to clients

At high publish rates and increasing event sizes, the effects of queuing delays come into the picture. This queuing delay is a result of the events being added to the queue faster than they can be processed. In general, the mean latency associated with the delivery of events to a client is directly proportional to the size of the events and the rate at which these events were published. The latencies are the lowest for smaller events 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 event 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 events of a larger size at higher publish rates.

Figures 23 and 24, depict the pattern of decreasing latencies with decreasing matching rates. Table 6 outlines the reduction in latencies and the variance associated with the latencies corresponding to the sample of events received at a client. 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. The routing algorithms are very selective about the links that are employed for event dissemination. Thus, events are queued only at those broker nodes which –

- Have attached clients interested in those events
- Are en route to broker nodes which are interested in these events. These broker 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.

10.2 System Throughput

We also depict the system throughputs at the client under conditions of varying event sizes and publish rates. We choose to depict the system throughputs at a matching rate of 100%. At matching rates other than 100% only the relevant events are being routed to the clients. The events received do not reveal the true throughputs that can be achieved at a client. Figure 25 depicts the system throughputs achieved at a client under conditions of different publish rates and event sizes. The maximum throughput achieved was 480.76 *events/Sec* at a publish rate of 492 *events/Sec* with the sample of events being of size 75 bytes.

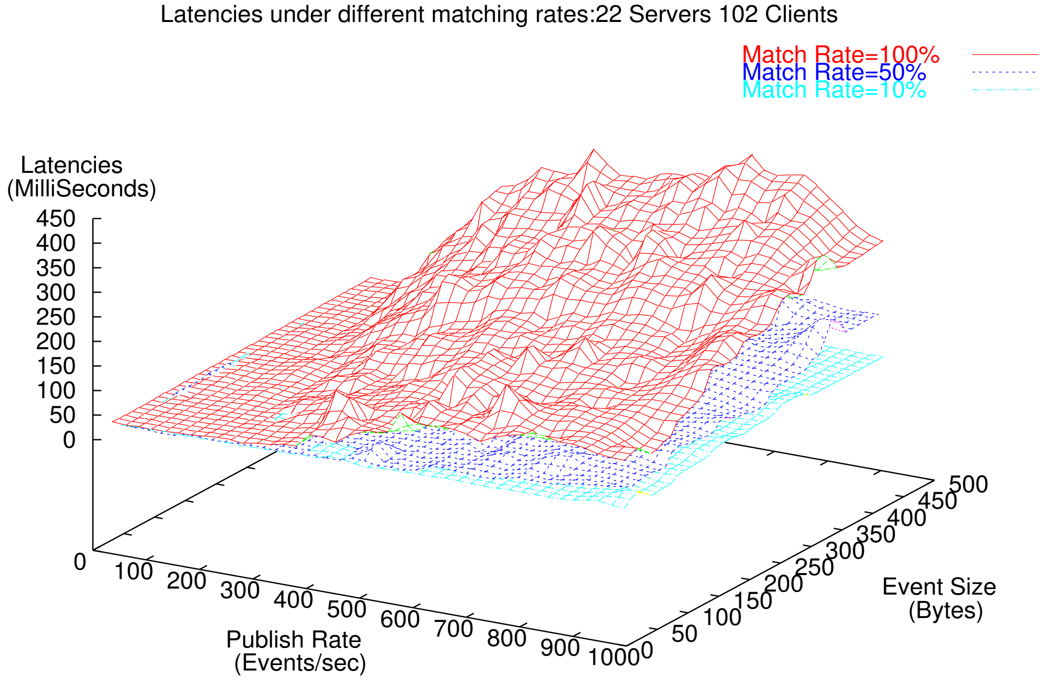

Figure 23: Latencies for match rates of 100%, 50% and 10%

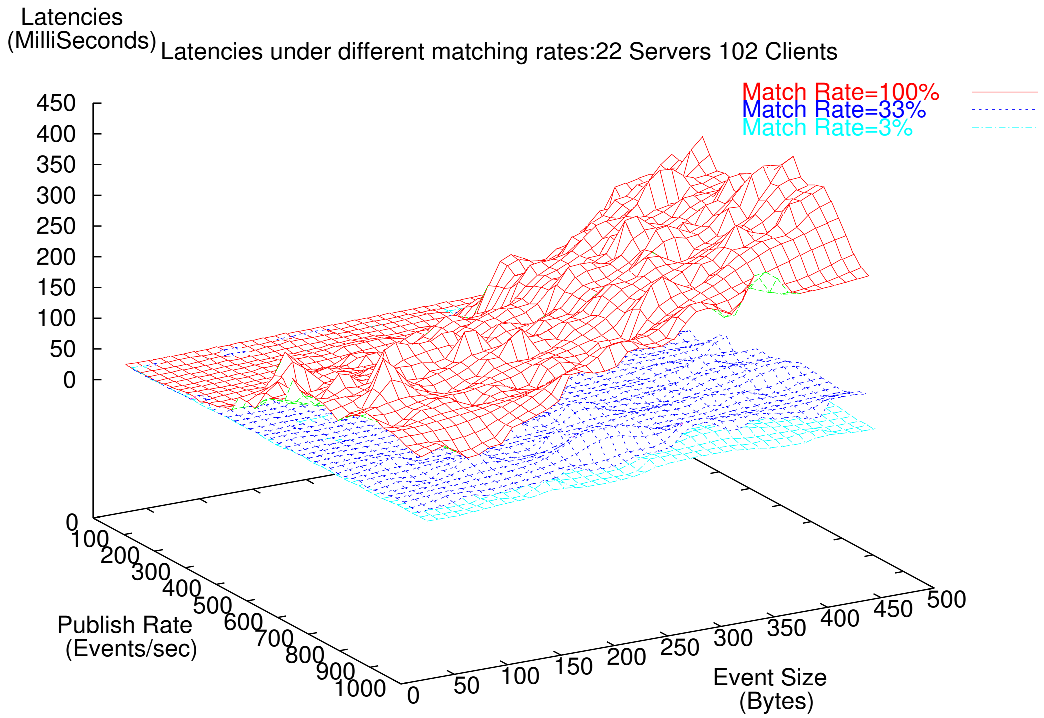

Figure 24: Latencies for match rates of 100%, 33% and 3%

Match Rate	Latency Range @ < publish Rate, event Size >		Variance Range	
	Latency $\mathcal{L}_a$	Latency $\mathcal{L}_a$	Variance $\mathcal{V}_a$ @ $\mathcal{L}_a$	Variance $\mathcal{V}_b$ @ $\mathcal{L}_b$
100%	15.54 $mSec$ @ $<1evt/Sec$, $50B>$	391.85 $mSec$ @ $<952evts/Sec$, $450B>$	2.3684 $mSec^2$	69,713.93 $mSec^2$
50%	13.02 $mSec$ @ $<20evts/Sec$, $50B>$	178.66 $mSec$ @ $<952evts/Sec$, $350B>$	56.8196 $mSec^2$	14,634 $mSec^2$
33%	15.18 $mSec$ @ $<20evts/Sec$, $50B>$	95.969 $mSec$ @ $<952evts/Sec$, 425 $B>$	26.57 $mSec^2$	1,263 $mSec^2$
25%	14.40 $mSec$ @ $<20evts/Sec$, $50B>$	66.6 $mSec$ @ $<961evts/Sec$, $400B>$	0.24 $mSec^2$	587.04 $mSec^2$
20%	15.35 $mSec$ @ $<20evts/Sec$, $50B>$	62.35 $mSec$ $<952evts/Sec$, $400B>$	12.02 $mSec^2$	312 $mSec^2$
10%	14.40 $mSec$ @ <20 $evts/Sec$, $50B>$	52.0 $mSec$ @ $<952evts/Sec$, $400B>$	0.44 $mSec^2$	103 $mSec^2$
5 %	14.0 $mSec$ @ $<20evts/Sec$, $50B>$	47.6 $mSec$ @ $<961evts/Sec$, $425B>$	0.44 $mSec^2$	87.44 $mSec^2$

Table 6: Latency and Variance Range Table

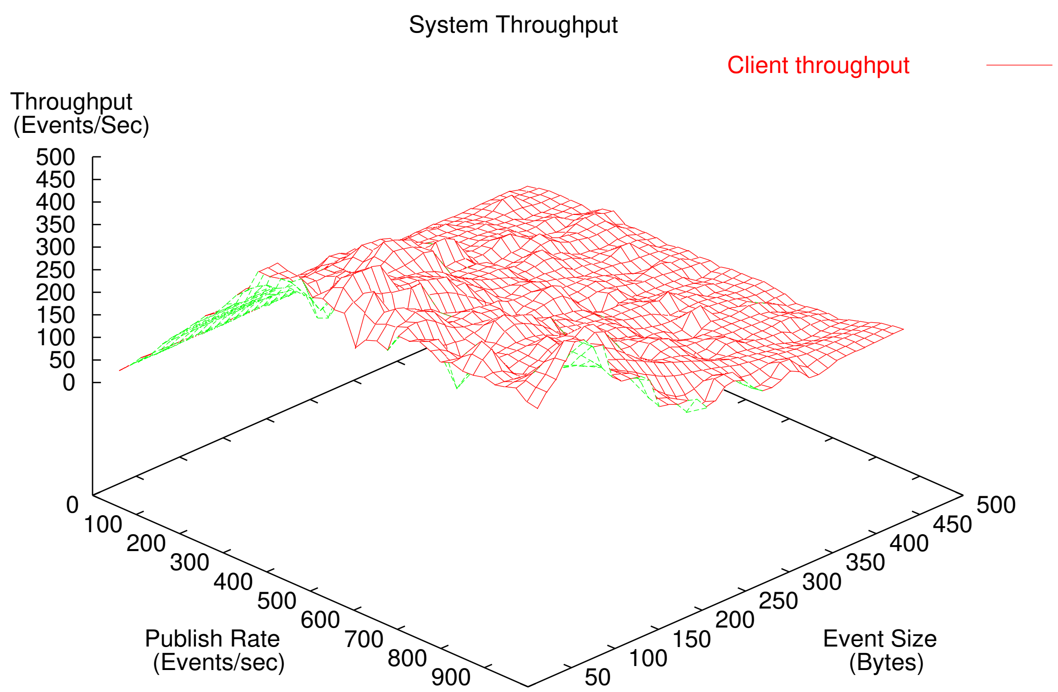

Figure 25: System Throughput

10.3 Variance

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

10.4 Persistent Clients

In figure 22 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 any of the nodes in super-cluster **SC-6** have a replication granularity of r_2. When the events issued by the publisher in the test topology of figure 22 are being disseminated, when clients attached to nodes in **SC-6** receive the event, that event would have been replicated twice (once at node **1** and once at node **8**). 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 (with the exception of nodes **7** and **10**) in the system has 5 persistent subscribing clients attached to it, for a total of 102 persistent subscribing clients. The publisher and the 2 *measuring* subscribers are all hosted on the same machine for reasons discussed earlier. Figure 26 depicts the latencies in delivery of events at persistent clients, with singular and double replications for a matching rate of 50%.

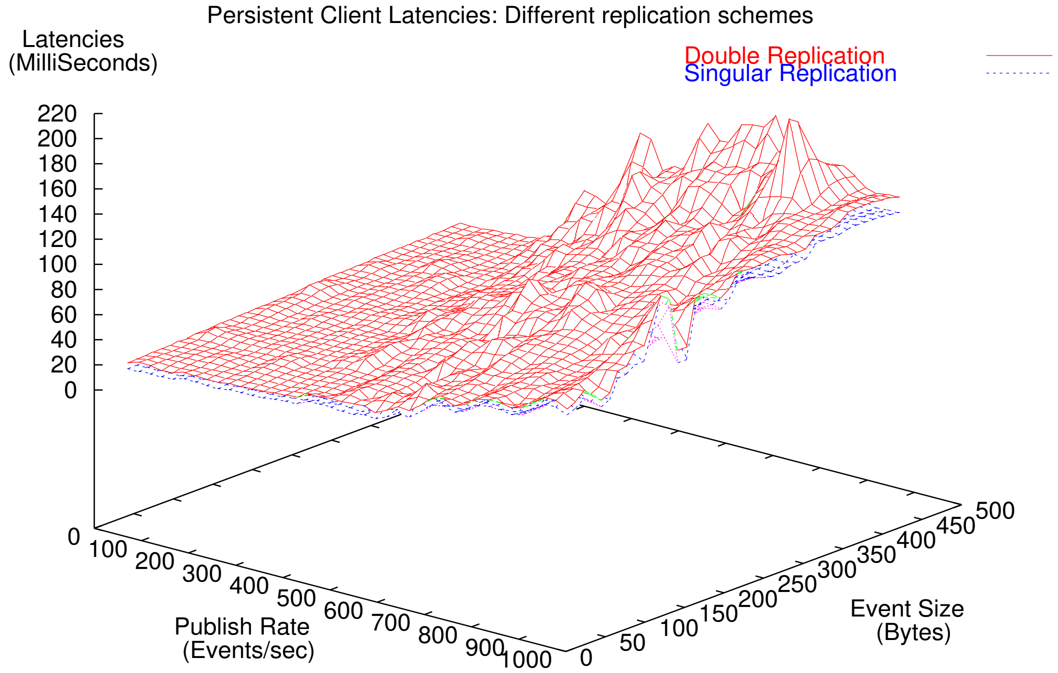

Figure 26: Match Rates of 50% - Persistent Clients (singular and double replication)

10.5 Pathlengths and Latencies

The topology in figure 22 allows us to magnify the latencies, which occur by having the queuing delays at individual broker hops add up. In that topology the number of broker hops taken by an event prior to delivery at the measuring subscriber is 9. We now proceed with testing for the topology outlined in figure 27. The layout of the broker nodes is essentially identical to the earlier one, with the addition of links between nodes resulting in a strongly connected network. We have 5 subscribing clients at each of the broker nodes. The mapping of broker nodes and subscribing client nodes to the physical machines is also identical to the earlier topology. As can be seen the addition of super-cluster link between super-clusters **SC-5** and **SC-6**, and level-0 links between nodes **8** and **10** in cluster **SC-6.n** reduces the number of broker hops, for the shortest path from the publisher to the measuring subscriber at node **10**, from 9 to 4.

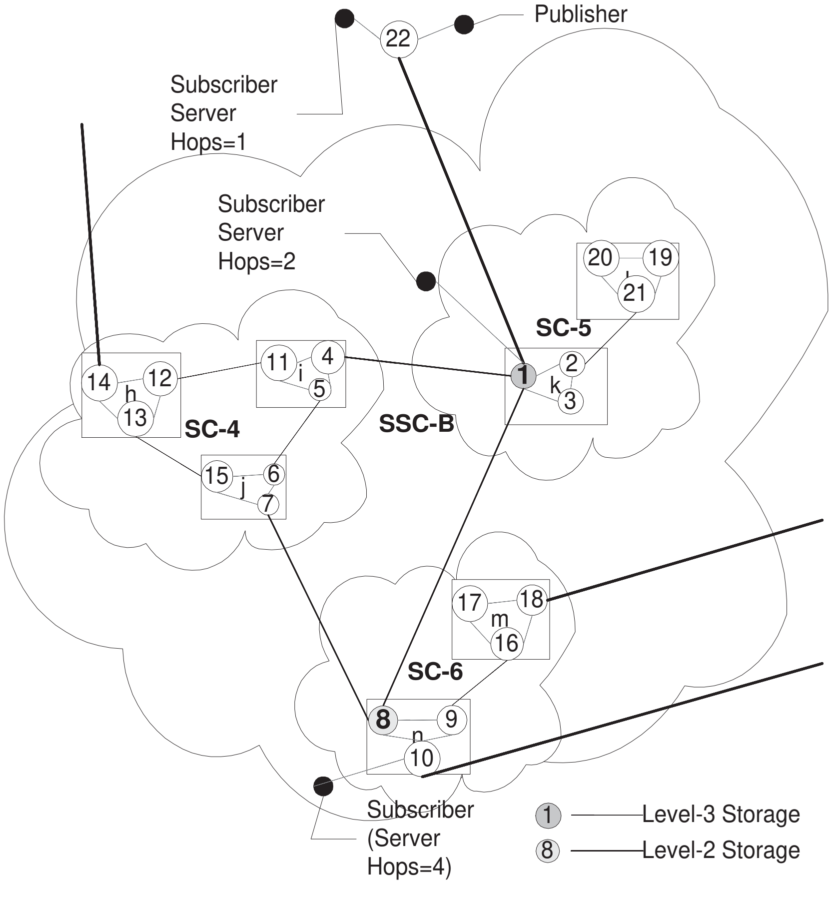

Figure 27: Testing Topology - Latencies versus broker hops

In this setting we are interested in the changes in latencies as the number of broker hops vary. We measure the latencies at three different locations, the measuring subscriber at node **10** has a broker hop of 4 while the measuring subscribers at nodes **1** and **22** have broker hops of 2 and 1 respectively for events published by the publisher at node **22**.

In general, as the number of broker hops reduce the latencies also reduce. The patterns for changes in latency as the event size and publish rates increase is however similar to our earlier cases. We depict our results, gathered at the three measuring subscribers for matching rates of 50%, 25% and 10%. The pattern of decreasing latencies with a decrease in the number of broker hops is clear by looking at figures 28 through 30.

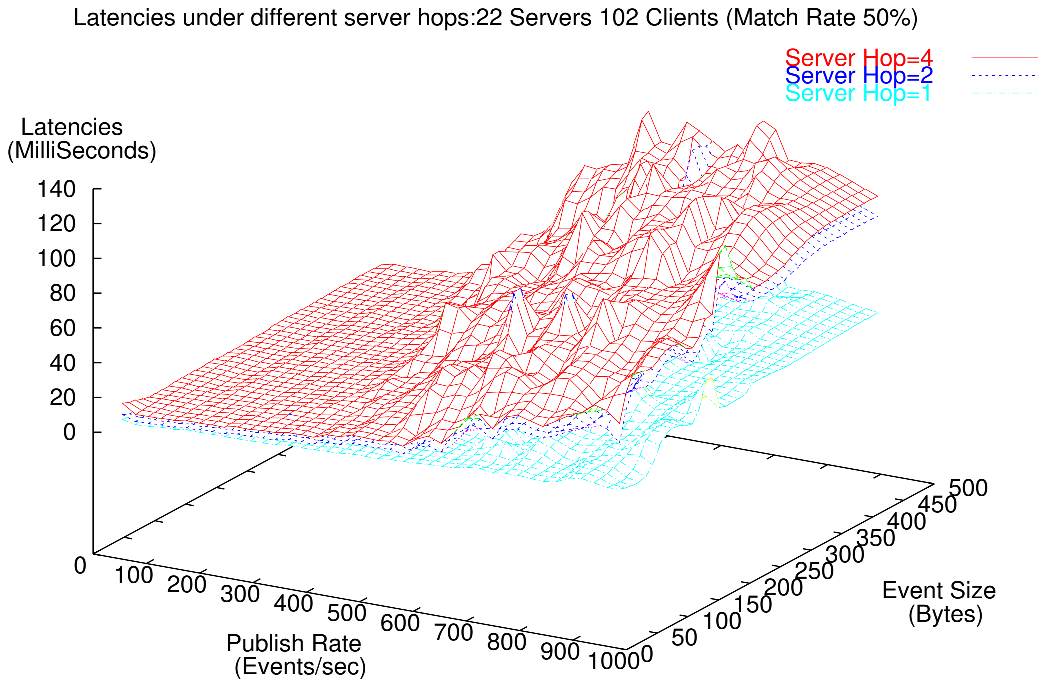

Figure 28: Broker Hops - Match Rate 50%

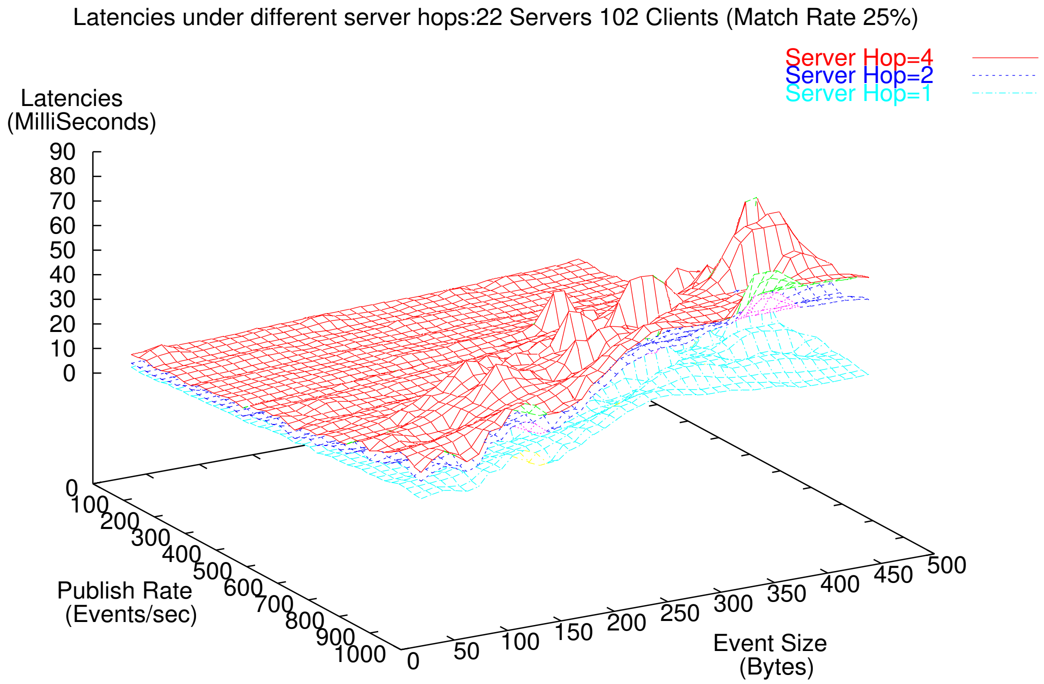

Figure 29: Broker Hops - Match Rate 25%

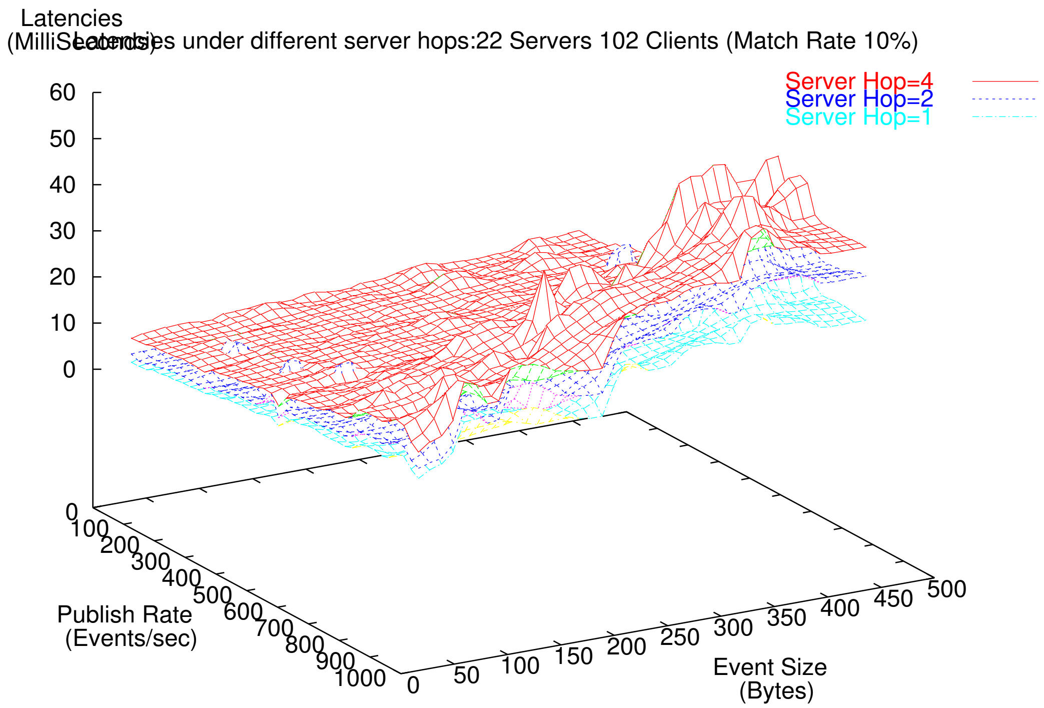

Figure 30: Broker Hops - Match Rate 10%

11 Conclusion

In this paper we have seen how the latencies vary with event sizes, matching rates, publish rates and connectivities. In general latencies decrease with increase in system connectivity, this being a result of the decrease in average pathlengths as the connectivity increases. On the other hand, increase in event sizes and publish rates result in an increase in the latency associated with event delivery. With decreasing matching rates, the latencies in event delivery decreases.

12 Future Directions

In this section we present extensions to GES. Some of these extensions are already being implemented or in the process of being implemented. We also include a brief discussion of the application domains that GES could be extended into.

12.1 Dynamic reshuffling

As mentioned in section 6.2, based on the system's load the dynamic instantiation of servers at clients can be very useful in the utilization of bandwidth, and generally would lead to better response times at the clients. Dynamic reshuffling of the system could be done based on the addition of powerful server nodes into the system. Such a node could be configured as a gatekeeper at a much higher level. Other nodes in the subsystem where the new node was added could also have their addresses reassigned based on how the system is being reorganized. It is conceivable that server nodes would have connections added or removed based on the addition of this node. Identification of slow nodes could enable us to then induce a failure in such slow server nodes, and subsequent reconfiguration of the connections.

12.2 Automatic configuration of nodes/units

In this case when a new node is being added that server node is simply slingshot into the GES server network. The system is then responsible for configuring it as a gatekeeper based on the hosting machine's processing power, IP address and the number of concurrent connections allowed. The system should also be able to initiate connections between other server nodes in the system, in a manner, which would maintain the small world properties for the server network.

12.3 GMS software architecture

Grid Message Service (GMS) is being developed at Florida State University (FSU) as the message service to support a collaborative portal to be used in education and computing. GMS uses a publish/subscribe infrastructure identical to GES with some additions. A client's profile comprises of a set of predicates which the client mandates that a certain event satisfy prior to the client being targeted 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 targeted for a client. This scheme is depicted in figure 31.

12.3.1 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 not only on the profiles but also on 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 the more powerful matching. Published topics, subscriber profiles and device properties are defined in an XML syntax GXOS (General XML Object Specification), developed by Geoffrey Fox, which is then 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 on 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

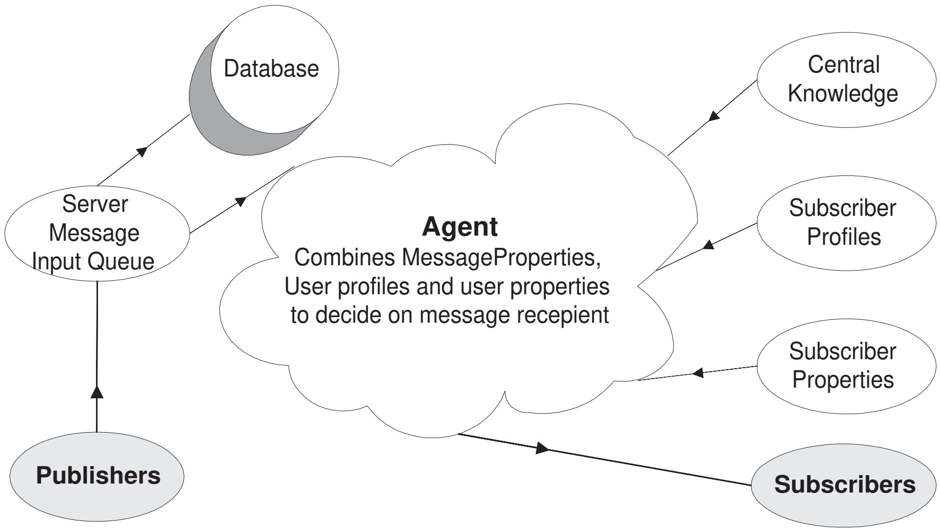

Figure 31: An agent based approach

we will narrow the destination lists associated with the event based on the client's content handling capabilities.

12.3.2 The execution Model - GXOS, MyXoS & RDF

All objects in the GMS system are self defining. The objects and meta-data describing them are separately managed. This allows the use of powerful technologies for managing the meta-objects while using classic high performance computing approaches for the raw objects. This provides a combination of 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, My eXtensible Web Operating System (MyXoS) , that manages these objects and meta-objects is driven by XML commands - initially in an RDF (Resource Description Framework) syntax. Information regarding RDF schema and syntax could be found in [53, 16]. GMS is used for both synchronous and asynchronous collaboration – it handles asynchronous information channels, the updates for shared display or shared event collaborations and the Jabber [68] instant messenger built into our collaborative framework.

Research is currently underway into the 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 [32] but efficiency requires that this be combined with *lazy* parsing so that we expand GXOS trees only as needed to refine our access. This is particularly challenging and has important programming style implications as we look at models where data structures are defined in XML and not directly as C++ or Java classes.

12.4 The XML 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[7] and cached at the server node. Thus we circumvent the network operation.

An event exists within the context of a stream, thus the specification of an event includes the stream that this event is a part of; this is specified by the StreamId. Every event needs to have an Id, `Mspaces:EventId`, that is unique in space and time. Events also should be able to specify the linkages that exist between them and events within other streams, this constitutes the `Mspaces:EventLinkage`. Resolution of the event linkage is a precursor to the creation of merged streams. We also need an indication of the type of event that this event is, i.e. live or recovery and the security constraints contained within the event. This is included in `Mspaces:EventType`. Events could also possibly specify zero or more 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

[7]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.

to be updated to reflect this change. This notification of the change in the DTD of the event that a client is interested in 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 12.4.1 and element analysis in table 7 assumes the `?` occurrence operator.

12.4.1 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)>
```

Tables 7 and 8 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 within the event element hierarchy.

12.5 Application Domains For GES

In this section we discuss the possible application domains for the Grid Event Service (GES). In section 12.5.1 we discuss employing GES for developing collaboratory applications. Section 12.5.2 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 12.5.3 we describe a possible extension to the GES the Grid Event Service Micro Edition (GESME) which handles messages and events for hand held devices.

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···N 0/1	Overall root element of the Event
`Mspaces:StreamId`	None	0	Stream the event belongs to
`Mspaces:EventId`	Mspaces:ClientID Mspaces:TimeStamp Mspaces:SequenceNum Mspaces:Incarnation	1 1 1 1	Unique event ID.
`Mspaces:EventType`	Mspaces:VersionNum Mspaces:LatestVersion	1 0/1	Indicates the version/liveness of events.
`Mspaces:EventLinkage`	Mspaces:BundleId Mspaces:BundleNumber Mspaces:BundleLinkage Mspaces:BundleOrder	0/1 0/1 1 1	Specifies linkage of events in multiple streams.
`Mspaces:BundleLinkage` *(Enumeration)*	Mspaces:TimeConstraint Mspaces:MergeScheme	0/1 0/1	Specifies methods for the merger of streams/bundles.
`Mspaces:BundleOrder` *(Enumeration)*	Mspaces:StreamId Mspaces:Bundle	1···N 1···N	Ordering for concurrent events

Table 7: Mspaces:Event Hierarchy – (I)

Element	Nested Elements	Num	Purpose of the Element
`Mspaces:ClientID`	None	0	The issuing Client ID.
`Mspaces:TimeStamp`	None	0	Time Stamp in mSecs
`Mspaces:SequenceNum`	None	0	Issue events at higher rates.
`Mspaces:Incarnation`	None	0	Allows for DD during a issuing client *roam*.
`Mspaces:Version`	None	0	The version number of the DTD.
`Mspaces:LatestVersion`	None	0	Inform clients about the version change to DTD.
`Mspaces:BundleId`	None	0	Identifies a specific bundle within the stream.
`Mspaces:BundleNumber`	None	0	Specifies the numbering within the bundle of a stream.
`Mspaces:TimeConstraint`	None	0	Specifies merging based on time.
`Mspaces:MergeScheme`	None	0	Specifies the scheme for mergers.

Table 8: Mspaces:Event Hierarchy – (II)

12.5.1 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 [17], ISAAC [52] and TANGO [35] 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* [36, 38] 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 [61] 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 [37, 27]. However these systems though distributed all maintain the notion of a Session object, which could serve as a single point of failure for clients that 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. This notion 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.

12.5.2 P2P Systems

Another important trend is peer-to-peer computing (P2P) [2] with recent work typified by the JXTA [50] initiative by 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.

12.5.3 Grid Event Service Micro Edition

One extension of importance GESME (GES Micro Edition) would handle 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. This would be a very important extension to GES.

12.6 Summary

In this section we outlined the future directions for GES. This included a discussion on the dynamic reshuffling of the system and reconfiguration of the nodes within the system. We also outlined an extension to GES, which would be based on object based matching to account for the content handling capabilities that are available at clients. We presented a discussion for the XML encapsulation of content. Finally, we presented the application domains into which GES can be extended, namely collaboration systems, P2P systems and GESME.

13 Conclusion

In this paper, we have presented the Grid Event Service (GES), a distributed event service designed to run on a very large network of server nodes. We outlined a formal specification for the reliable delivery of events to clients with intermittent connection semantics. The delivery guarantee needs to be met across client roams and also in the presence of server failures. We also outlined a specification for the delivery of merged streams to interested clients. 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. Creating small world networks, using the node organization protocol ensures that the pathlengths would only increase logarithmically with geometric increases in the size of the server network. The feature of having multiple links between two units/super-units ensures a greater degree of fault tolerance. Links could fail, and the routing to the affected units is performed using the alternate links. The protocols in the GES protocol suite 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 then 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 units within the system. The organization of connection information ensures that connection losses (or additions) are incorporated into the connectivity graph hosted at the server nodes. Certain sections of the routing cache are invalidated in response to this addition (or loss) of connections. This invalidation and subsequent calculation of best *hops* to reach units (at different levels) ensure that the paths computed are consistent with the state of the network, and include only valid/active links. The event routing protocol uses the profile information available at the unit gatekeepers to compute the sub-units that the event should be routed to. To reach these destinations every node, at which this event is received, employs the *best hops* to reach those destinations. This *best hop* is computed based on the cost of traversal and also the number of links connecting the different units. Thus, in our system, based on the organization of profiles and subsequent matching of events, the only units to which an event is routed to are those that have clients interested in that event. 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 under conditions of varying publish rates, matching rates and message sizes.

The paper outlined a scheme for the delivery of events in the presence of server node failures. In our scheme a unit could fail and remain failed forever. The only requirement that we impose is that if a stable storage fails, it should recover within a finite amount of time. The replication strategy, that we adopted allows us to add stable storages and also to withstand stable storage 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 could be extended very easily to support dynamic topologies. Based on the concentration of clients at specific locations, bandwidth utilization can be optimized with the creation of dynamic servers at some of the clients. This scheme fits very well with our failure model for system units, where they can remain failed forever. Detection schemes can be employed to detect slow nodes, which serve as performance bottlenecks. Forcing these affected nodes to fail then reconfigures the system. GES immediately employs newly added units in the routing of events and responds to unit failures equally well. Similarly links can be added and removed in a similar fashion to optimize the routing for events. The routing decisions at each server node are based on the current state of the system. GES thus provides the base infrastructures for dynamic topologies.

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 of the nodes within the system and still receive all the events it was supposed to receive is a significant contribution. Clients can connect to local servers instead of reconnecting all the way back to the remote server that it was last connected to. This scheme optimizes bandwidth utilization. This optimization is very pronounced when there is a high concentration of clients accessing the remote server. 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. This model ensures that clients need not wait for a server to recover after this server has failed. During failures clients do not experience a denial of service in this model. The service, as mentioned earlier, extends very naturally into dynamic topologies allowing for the dynamic instantiation and purging of servers and connections. Changes in network fabric are incorporated by the routing algorithms, which ensure that the routing decisions made at a node are based on the current state of the system. The replication strategy presented in this paper, 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 additional robustness and reduce the time required to recover from stable storage failures.

The results in section 7 demonstrated the efficiency of the routing algorithms and confirmed the advantages of our 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 in our framework and the failure model that we impose on server nodes provides for very easy 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. Another important feature is the ability to access another part of this distributed service for better response times or convenience. This paper makes significant contributions towards providing solutions to these issue.

References

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

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

[3] 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.

[4] R. Albert, H. Jeong, and A. Barabasi. Diameter of the World-Wide Web. *Nature*, 401:130, 1999.

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

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

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

[12] 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.

[13] 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.

[14] 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.

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

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

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

[18] 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.

[19] 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.

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

[21] Akamai Corporation. EdgeSuite: Content Delivery Services . Technical report, URL: http://www.akamai.com/html/en/sv/edgesuite_over.html, 2000.

[22] 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.

[23] Progress Software Corporation. SonicMQ: The Role of Java Messaging and XML in Enterprise Application Integration. Technical report, URL: http://www.progress.com/sonicmq, October 1999.

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

[25] TIBCO Corporation. TIB/Rendezvous White Paper. Technical report, URL: http://www.rv.tibco.com/whitepaper.html, 1999.

[26] 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.

[27] 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.

[28] W. Diffie and M. Hellman. Multiuser Cryptographic Techniques. In *Proccedings of the AFIPS National Computer Conference*, June 1976.

[29] W. Diffie and M. Hellman. New Directions in Cryptography. In *IEEE Transactions on Information Theory*, November 1976.

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

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

[32] exolab.org. Castor: The Source Code Generator. Technical report, URL: http://castor.exolab.org/sourcegen.html, 2001.

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

[34] 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.

[35] 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.

[36] 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.

[37] 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.

[38] 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.

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

[40] 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.

[41] 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.

[42] 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.

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

[44] 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.

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

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

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

[48] iPlanet. Java Message Queue (JMQ) Documentation. Technical report, URL: http://docs.iplanet.com/docs/manuals/javamq.html, 2000.

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

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

[51] John Kohl, Clifford Neuman, and Theodore T'so. The Evolution of the Kerberos Authentication System. In *Distributed Open Systems*, pages 78–94. IEEE Computer Society Press, 1994.

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

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

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

[55] Nancy Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[56] The Object Management Group (OMG). CORBA Notification Service. URL: http://www.omg.org/technology/documents/formal/notificationservice.htm, June 2000. Version 1.0.

[57] The Object Management Group (OMG). OMG's CORBA Event Service. URL: http://www.omg.org/technology/documents/formal/eventservice.htm, June 2000. Version 1.0.

[58] The Object Management Group (OMG). OMG's CORBA Services. URL: http://www.omg.org/technology/documents/, June 2000. Version 3.0.

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

[60] 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.

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

[62] Shrideep B. Pallickara. *A Grid Event Service*. PhD thesis, Syracuse University, June 2001.

[63] 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.

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

[65] 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.

[66] R. Rivest, A. Shamir, and L. Adleman. A Method for Obtaining Digital Signatures and Public Key Cryptosystems. In *Communications of the ACM*. February 1978.

[67] Ron Rivest. The MD4 Message Digest Algorithm. In *Proceedings of Crypto'90*, August 1990.

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

[69] 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.

[70] 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.

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

[72] William Stallings. *Cryptography and Network Security.* Prentice Hall, second edition, 1999.

[73] J. Steiner, C. Neuman, and J. Schiller. Kerberos: An Authentication Service For Open Networked Systems. In *Proceedings of the Winter 1988 USENIX Conference*, February 1988.

[74] D.J. Watts and S.H. Strogatz. Collective Dynamics of 'Small-World' Networks. *Nature*, 393:440, 1998.