CONCURRENCY—PRACTICE AND EXPERIENCE
Concurrency: Pract. Exper. 2002; **01**:1–34 *Prepared using* cpeauth.cls *[Version: 2001/03/05 v2.01]*

An Event Service to Support Grid Computational Environments

Geoffrey Fox[1] and Shrideep Pallickara[2]

[1] *gcf@indiana.edu, Dept. of Computer Science, Indiana University*
[2] *sbpallic@ecs.syr.edu, Dept. of Electrical Engineering & Computer Science, Syracuse University*

SUMMARY

We believe that it is interesting to study the system and software architecture of environments which integrate the evolving ideas of computational grids, distributed objects, web services, peer-to-peer networks and message oriented middleware. Such peer-to-peer (P2P) Grids should seamlessly integrate users to themselves and to resources which are also linked to each other. We can abstract such environments as a distributed system of "clients" which consist either of "users" or "resources" or proxies thereto. These clients must be linked together in a flexible fault tolerant efficient high performance fashion. In this paper, we study the messaging or event system – termed GES or the Grid Event Service – that is appropriate to link the clients (both users and resources of course) together. For our purposes (registering, transporting and discovering information), events are just messages – typically with time stamps. The messaging system GES must scale over a wide variety of devices – from hand held computers at one end to high performance computers and sensors at the other extreme. We have analyzed the requirements of several Grid services that could be built with this model, including computing and education and incorporated constraints of collaboration with a shared event model. We suggest that generalizing the well-known publish-subscribe model is an attractive approach and here we study some of the issues to be addressed if this model is used in the GES.

KEY WORDS: distributed messaging, publish subscribe, guaranteed delivery, grid systems, peer-to-peer infrastructures and event distribution systems.

*Correspondence to: 3-211 CST, 111 College Place, Syracuse University, Syracuse NY-13244, USA

Revised 5 December 2001
Accepted 20 October 2001

1. Introduction

The web in recent years has experienced an explosion in the number of devices users employ to access services. A single user may access a certain service using multiple devices. Most services allow clients to access the service through a broker. The client is then forced to interact with the service via this broker throughout the duration that it is using the service. If the broker fails, the client is denied servicing till such time that the failed broker recovers. In the event that this service is running on a fixed 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 do not always maintain an online presence, and when they do they may the access the service using a different device with different computing and content-handling capabilities. The communication channels employed during every such service interaction may have different bandwidth constraints and communication latencies. Besides this a client accesses services from different geographic locations.

A truly distributed service would allow a client to use services by connecting to a broker nearest to the client's geographical location. By having such local broker, a client does not have to re-connect all the way back to the broker that it was last attached to. If the client is not satisfied with the response times that it experiences or if the broker that it has connected to fails, the client could very well choose to connect to some other local broker. 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. It should not be assumed that a failed broker node 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. Such a 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, usually involving a calculation of state from the 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.

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 is 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 and the departure from the time-bound failure recovery model, leads to a situation 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. The network can thus undulate with the addition and failure/purging of broker node processes.

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) interactions or a be part of 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. 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. Events are *anything* transmitted including updates, objects themselves (file uploads), database updates and audio/video streams. 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 the 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.

We believe that it is interesting to study the system and software architecture of environments which integrate the evolving ideas of computational grids, distributed objects, web services, peer-to-peer networks and message oriented middleware. Such peer-to-peer (P2P) Grids should seamlessly integrate users to themselves and to resources which are also linked to each other. We can abstract such environments as a distributed system of "clients" which consist either of "users" or "resources" or proxies thereto. These clients must be linked together in a flexible fault tolerant efficient high performance fashion. In this paper, we study the messaging or event system – termed GES or the Grid Event Service – that is appropriate to link the clients (both users and resources of course) together. For our purposes (registering, transporting and discovering information), events are just messages – typically with time stamps. The messaging system GES must scale over a wide variety of devices – from hand held computers at one end to high performance computers and sensors at the other extreme. We have analyzed the requirements of several Grid services that could be built with this model, including computing and education and incorporated constraints of collaboration with a shared event model. Grid Services (including GES) being deployed in the context of Earthquake Science can be found in [20]. We suggest that generalizing the well-known publish-subscribe model is an attractive approach and here we study some of the issues to be addressed if this model is used in the GES.

1.1. Messaging Oriented Middleware

Messaging systems based on queuing include products such as Microsoft's MSMQ [28]and IBM's MQSeries [29]. The queuing model 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. A widely used standard in messaging is the Message Passing Interface Standard (MPI) [21]. 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 [35], Java RMI [32] and DCOM [19]. In publish/subscribe systems 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 from the producer to the right consumers. Industrial strength products in the publish subscribe domain include solutions like *TIB/Rendezvous* [17] from TIBCO and *SmartSockets* [16] from Talarian. Other related efforts in the research community include *Gryphon* [4, 1], *Elvin* [45] and *Sienna* [11]. The push by Java to include publish subscribe features into its messaging middleware include efforts like JMS [26] and JINI [2]. One of the goals of JMS is to offer a unified API across publish subscribe implementations. Various JMS implementations include solutions like *SonicMQ* [15] from Progress, *JMQ* [31] from iPlanet, *iBus* [30] from Softwired and *FioranoMQ* [14] from Fiorano. Systems tuned towards large scale P2P systems include *Pastry* [43] from Microsoft, which provides an efficient location and routing substrate for wide-area P2P applications. *Pastry* provides a self-stabilizing infrastructure that adapts to the arrival, departure and failure of nodes. JXTA [33] from Sun Microsystems is another research effort that seeks to provide such large-scale P2P infrastructures.

1.2. Service provided

We have built a "production" system and an advanced research prototype. The production system uses the commercial Java Message Service (SonicMQ) and has been used very successfully to build a synchronous collaboration environment applied to distance education. The publish/subscribe mechanism is powerful but this comes at some performance cost and so it is important that it satisfies the reasonably stringent constraints of synchronous collaboration. We are not advocating replacing all messaging with such a mechanism – this would be quite inappropriate for linking high performance devices such as nodes of a parallel machine linked today by messaging systems like MPI or PVM. Rather we have recommended using a hybrid approach in such cases. Transport of messages concerning the control of such HPCC resources would be the responsibility of the GES but the data transport would be handled by high performance subsystems like MPI. This approach was successfully used by the Gateway computing portal.

Here we study an advanced publish/subscribe mechanism for GES which goes beyond JMS and other operational publish/subscribe systems in many ways. A basic JMS environment has a single server (although by linking multiple JMS invocations you can build a multi-server environment and you can also implement the function of a JMS server on a cluster). We propose that GES be implemented on a network of brokers where we avoid the use of the term servers for two reasons; the publish/subscribe broker service could be implemented on any computer – including a user's desktop machine. Secondly we have included the many application servers needed in a P2P Grid as clients in our abstraction for they are the publishers and subscribers to many of the events to be serviced by GES. Brokers can run either on separate machines or on clients whether these are associated with users or resources. This network of brokers will need to be dynamic for we need to service the needs of dynamic clients. For example suppose one started a distance education session with six distributed classrooms each with around 20 students; then the natural network of brokers would have one for each classroom (created dynamically to service these clusters of clients) combined with static or dynamic brokers associated with the virtual university and perhaps the particular teacher in charge.

Here we study the architecture and characteristics of the broker network. We are using a particular internal structure for the events (defined in XML but currently implemented as a Java object). We assume a sophisticated matching of publishers and subscribers defined as general topic objects (defined by an XML Schema that we have designed). However these are not the central issues to be discussed here. Our study should be useful whether events are defined and transported in Java/RMI or XML/SOAP or other mechanisms; it does not depend on the details of matching publishers and subscribers. Rather, we are interested in the capabilities needed in any implementation a GES in order to abstract the broker system in a scalable hierarchical fashion (section 2); the delivery mechanism (section 3); the guarantees of reliable delivery whether brokers crash or disappear or whether clients leave or (re)join the system (section 4). Section 4 also discusses persistent archiving of the event streams. We have emphasized the importance of dynamic creation of brokers but this was not implemented in our initial prototype. However by looking at the performance of our system with different static broker topologies we can study the impact of dynamic creation and termination of broker services.

1.3. Status

There exists a prototype implementation of GES. This implementation, developed using Java, uses TCP as the transport protocol for communication within the system and is JMS compliant. Support for XML is currently being added to the system. Future work would include work on support for dynamic topologies and security frameworks for authentication, authorization and dissemination of content. The results from our prototype implementation are presented in this paper.

2. Clients and the Broker Topology

In this section we outline the destinations that are associated with an event. We discuss the connection semantics for any client within the system, 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.

2.1. Destination lists and the generation of unique identifiers

Clients in the system specify an interest in the type of events that they are interested in. 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 explicit or implicit 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 scores, 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.

2.2. Client

Events are continuously generated and consumed by clients within the system. Clients have intermittent connection semantics. Clients can be present in the system for a certain duration and be disconnected later on. Clients reconnect at a later time and receive events, which they were supposed to receive in their past incarnations as well as events that they are supposed to receive during their present incarnation. Clients issue/create events while in disconnected mode, these events would be held in a local queue to be released to the system during a reconnect.

Associated with every client is its profile, which keeps track of information pertinent to the client. This includes the application type, events the client is interested in and the broker node the client was attached to in its previous incarnation.

2.3. The Broker Node Topology

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. 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. Additional information pertaining to the need for distributed brokering systems can be found in [22]. 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 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*. 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. In general there would be multiple links connecting a single cluster to several other clusters. This approach provides us with a greater degree of fault-tolerance, by providing us with multiple *routes* to reach nodes within other clusters.

This topology could be extended in a similar fashion to comprise of *super-super-clusters* (level-3 units), *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. This limit, the *block-limit*, is set at 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.

We now delve into the *small world graphs* introduced in [46] and employed for the analysis of real world peer-to-peer systems in [36, 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 to the number of connections that are actually possible between these nodes. In 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 [46] explore graphs where the clustering coefficient is high, and with *long connections* (inter-cluster links in our case). 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.

2.3.1. GES Contexts

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

2.3.2. Gatekeepers

Within the GES context C_i^2 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. 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*.

Depending on the highest level at which there is a difference in the GES contexts of these nodes, the nodes that maintain this active connection are referred to as gatekeepers at the corresponding 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.

Figure 1 shows a system comprising of 78 nodes organized into a system of 4 super-super-clusters, 11 super-clusters and 26 clusters. 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, say C_j^ℓ and C_k^ℓ; the nodes are designated as gatekeepers at $level - \ell$ i.e. $g^\ell(C^{\ell+1})$. Thus, in figure 1 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. The problem of event delivery

The event delivery problem is one of routing events to clients based on the type of events that clients are interested in. Events need to be relayed through the broker network prior to being delivered to clients. The dissemination process should efficiently deliver 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 those events that they are not interested in. This approach places a heavy 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

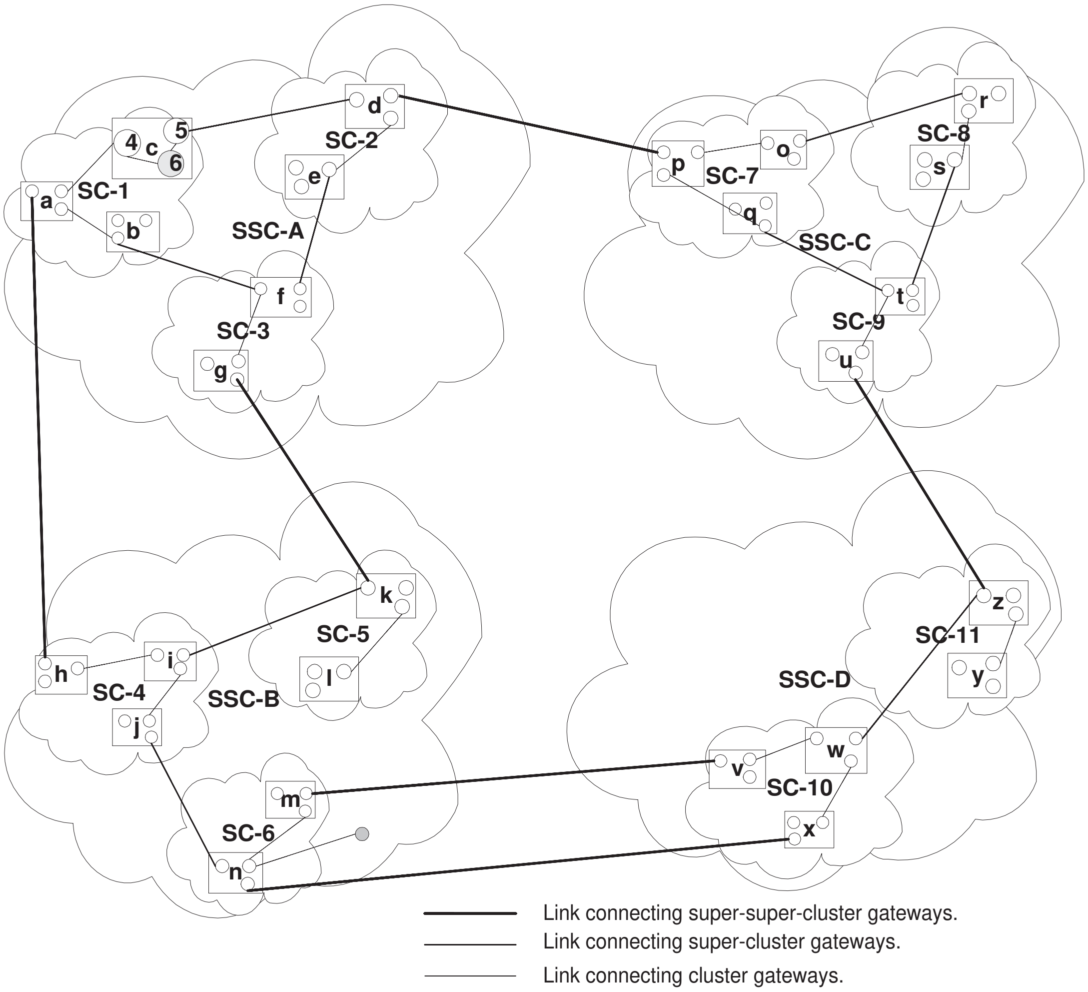

Figure 1. Gatekeepers and the organization of the system

scheme increases the latency associated with the reception of real time events at the client 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.

Prior Art

Different systems address the problem of event delivery to relevant clients in different ways. In *Elvin* [23] each subscription is converted into a deterministic finite state automaton which can lead to an explosion in the number of states. Network traffic reduction [45] is accomplished through the use of *quench* expressions that prevent clients from sending notifications for which there are no consumers. In *Sienna* [11, 12] 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 *Gryphon* [4] 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. A related Gryphon effort for exploiting universally available multicast techniques for event delivery can be found in [3].

The approach adopted by the OMG is one of establishing channels and registering suppliers and consumers to those event channels. The channel approach in the event service [34] 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. In *TAO* [27], 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* [13] from Akamai, redirect their users to specialized Akamized URLs. *EdgeSuite* identifies the geographic location from which the clients have accessed the website and then re-direct clients to the broker farm that is closest to their network point of origin. As the network load and broker loads change clients could be redirected to other brokers. Active mirroring entails all serviced content to be cached at all (or most) of the broker farms. The scheme is very effective when the data is being accessed by very large number of clients and also when the rate of content change (and subsequent cache updates) is relatively low. This need for caching and the propagation of content updates constricts the amount of data that can be cached besides requiring cached data to be constantly updated. This approach is not suited for data that is transient with a real-time context associated with it. Furthermore in most services the interaction model tends to be far more complex than the traditional client server model that the EdgeSuite model best services.

GES Solution Highlights

Our solution to the problem of event delivery handles the dissemination problem in a near optimal fashion. An event is routed only to those units that have at least one client that is interested in the event. Furthermore, the links employed during the routing ensures the fastest dissemination since each broker makes routing decisions, which ensure that the path from that broker to the intended recipients is the fastest (usually the shortest path). The routing decisions are made based on the current state of the network. A broker or islands of brokers could fail and the routes computed would avoid these failed sections of the broker network while routing to recipients. Solutions to the delivery problem, involve a matching step being performed at every broker. In our solution for a broker network, organized as an N-level system, the matching step is not performed at every broker as the event is being relayed through the broker network to its intended recipients. In fact this matching step is performed at most (N+1) times prior to delivery at a given recipient. The solution to the event delivery problem handles dense and sparse interests in events equally well. The solution for delivery of events to clients experiencing service interruptions due to single/multiple broker failures is discussed in the next section.

3.1. 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. 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. 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 those connectivities that are relevant to the node in question. Figure 1 depicts the connections that exist between the various units of the system that we would be using as an example in further discussions.

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. Restrictions imposed on the dissemination of connection information ensure update propagation only to affected sub-systems within the system.

3.1.1. 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. 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. Thus, connections established between broker nodes in a cluster are *not* disseminated outside that cluster.

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. Details regarding the information encapsulated in a connection, the update of this information during disseminations and the enforcement of dissemination constraints can be found in [40, 38, 39]. Thus, in figure 1 the connection between **SC-2** and **SC-1** in **SSC-A**, is disseminated as one between node **5** and **SC-2**. When this information is received at **4**, it is sent over as a connection between the cluster **c** and **SC-2**. When the connection between cluster **c** and **SC-2** is sent over the cluster gateway to cluster **b**, the information is not updated. As was previously mentioned, the super cluster connection **(SC-1,SC-2)** information is disseminated only within the super-super-cluster **SSC-A** and is not sent over the super-super-cluster gateway available within the cluster **a** in **SC-1** and cluster **g** in **SC-3**.

Every edge created due to the dissemination of connection information 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. Further, associated with every edge is the cost of traversal. In general the cost associated with traversing a level-ℓ link from a unit u^x increases with increasing values of both x and ℓ.

This cost scheme is encapsulated in the link cost matrix, which 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 could be done every few seconds.

3.1.2. Organizing the nodes

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. Figure 2 depicts the connectivity graph that is constructed at the node **SSC-A.SC-1.c.6** in figure 1. The cost associated with traversal over a level-3 gateway between a level-2 unit **b** and a level-3 unit **SC-3** as computed from the linkcost matrix is 3, and is the weight of the connection edge. 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 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.

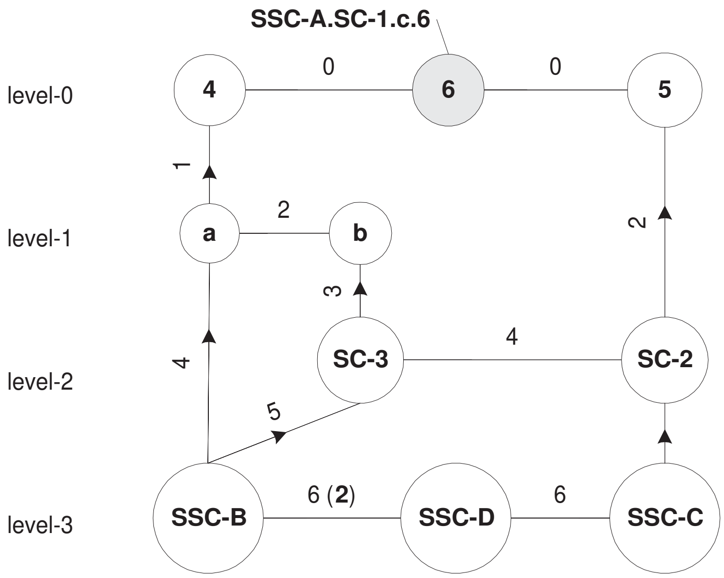

Figure 2. The connectivity graph at node 6.

3.1.3. 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 events can be disseminated faster throughout the system. The routing cache should be used in tandem with the routing information contained within a routed event 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.

3.2. 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 issue these events, assign values to these attributes – the content identifier. 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. 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.

Prepared using cpeauth.cls

3.2.1. Constructing a profile graph

Events encapsulate content identifiers in an ordered set of $<$ *attribute*, *value* $>$ tuples. The constraints specified in the profiles should maintain this order contained within the event's content identifier. 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.

We use the general matching algorithm, presented in [1], of the Gryphon system to organize profiles and match 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.

Figure 3 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. 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. Figure 3 provides a simple example of the information maintained along the edges.

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 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 attribute 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.

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

In the hierarchical dissemination scheme that we have, unit gatekeepers compute destination lists for the sub-units that they service. A cluster gatekeeper would thus maintain profiles of the broker nodes contained with that cluster. Profile graphs at a super-cluster gatekeeper maintain information pertaining to profiles stored at cluster gatekeepers within the super-cluster that

Prepared using cpeauth.cls

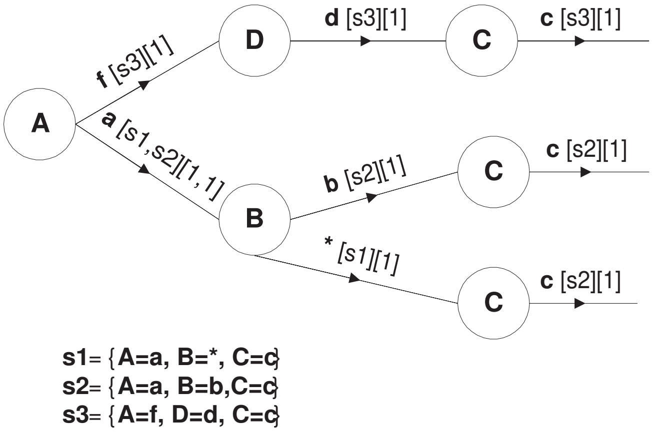

Figure 3. The complete profile graph with information along edges.

it is a part of. Profile changes thus need to be propagated to relevant nodes, and should be done such that when an event arrives at a unit gatekeeper –

(a) The events that are routed to sub-units, are those with content such that at least one destination exists within those sub-units.
(b) There are no events, which were not routed to a sub-unit, with content such that there would have been a valid destination within that sub-unit.

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.

For profile changes within sub-units 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. 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. A profile change in broker (as a result of a change in an attached client's profile) needs to be propagated to the unit (cluster, super-cluster, etc) gatekeeper within the unit that the broker is a part of.

In the connectivity graph depicted in figure 2 any profile changes at any of the broker nodes within cluster **c**, needs to be routed to node **4**. Any such broker profile changes that result in a change of the cluster profile at node **4** needs to be routed to node **5**, and also to a node in cluster **b**. Similarly any changes in the super-cluster profile 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.

3.3. 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 broker node that the event must be relayed to. Events have routing information associated with them, which indicate its dissemination within various parts of the broker network. The dissemination information at each level can be accessed to verify disseminations in various sections of the broker network. Routing decisions and the brokers that an event must be relayed to are made on the basis of this information. This routing information is not added by the client issuing this event but by the system to ensure fastest possible 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. As the event flows through the system, via gateways, the routing information is modified to snapshot its dissemination within the system.

A cluster gatekeeper, when it receives an event, computes the broker destinations associated with that event. This calculation is based on the profiles available at the gatekeeper as outlined in the profile propagation protocol. 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. The Reliable Delivery Of Events

Reliable delivery involves the guaranteed delivery of events to intended recipients. The delivery guarantees need to be satisfied even in the presence of single or multiple broker failures, link failures and any associated network partitions as a result of these failures. In GES clients need not maintain an active online presence and can also roam the network attaching themselves to any of the nodes in the broker network. Events missed by clients in the interim need to be delivered to these clients irrespective of the failures that have already taken place or are currently present in the system.

Prior Art

The problem of reliable delivery [25, 7] and ordering [9, 8] in traditional group based systems with process crashes has been extensively studied. These approaches normally have employed the *primary partition* model [42], 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 the risk of contradictions arising in the other partitions and also during partition mergers. However the delivery requirements are met only within the primary partition

[24]. Recipients that are slow or temporarily disconnected may be treated as if they had left the group. This model, adopted in Isis [6], 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 [41] and Transis [18] manage *minority partitions*, and can handle concurrent views in different partitions. The overheads to guarantee consistency are however too strong for our case. DACE [10] 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. This is achieved through a self-stabilizing exchange of views through the Topic Membership protocol. In [5], the effect of link failures on the solvability of problems (which are solved with reliable links) in asynchronous systems, has been rigorously studied. [44] describes approaches to building fault-tolerant services using the state machine approach.

Systems such as *Sienna* [12, 11] and *Elvin* [23, 45] 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 maintained active connections to prior to its failure. *SmartSockets* [16] 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. The system state when both the process and its mirror counterpart fail is not addressed. *TIB/Rendezvous* [17] 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 such as IBM's MQSeries [29] and Microsoft's MSMQ [28] 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 [15, 31, 30, 14] to utilize a store-and-forward mechanism to fulfill its contract with the sender (producer).

GES Solution Highlights

Our solution to the reliable delivery problem eliminates the constraint that a failed broker recover within a finite amount of time. Also, we do not rely on state reconstructions of broker since these solutions lead to problems during multiple broker failures. In our solution we allow for a broker to fail and remain failed forever. The reliable delivery scheme also incorporates a replication scheme that provides for different replication strategies to exist at different parts of the system. The scheme also allows for this replication strategy to change at different parts of the system as time progresses. Changes to the replication scheme at a certain section of the broker network results in updates at the corresponding affected subsystems within the broker network. Roaming, disconnected and connected clients are not affected by these changes in replication schemes within the broker network. Clients reconnecting after a failure or prolonged disconnect make a complete recovery of the events that were published in the interim. The recovery is precise and only those events that were missed by a reconnecting client are routed through the system en route to the recovering client as recovery events. The persistence model in most systems involves the persistent store subscribing to all events that are published in the system. This could lead to exponential rise in storage requirements at different stable storages in the system. With increasing selectivity in the type of events a client is interested in and also in the number of clients that are present in the system, the events stored at storages servicing subsystems would far exceed the number of events that clients in those subsystems are actually interested in. In the GES replication scheme a stable storage servicing a subsystem subscribes to only those events that the brokers (and in turn the clients) it services are interested in. We also have a garbage collection scheme, which ensures that the storage space does not increase exponentially with time. The epochs and reference counting scheme ensures that no client is starved of the events that it was supposed to receive while ensuring that the event is garbage collected once all clients interested in receiving those events have received that event.

In our failure model a unit can fail and remain failed forever. The broker 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. Brokers 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 broker network undulate, as the brokers 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. The routing algorithms and the failure model allow support for such dynamic reconfiguration of networks.

In section 4.1 we discuss the replication scheme for our system, and issues pertaining to event stability. Section 4.2 describes the need for epochs, the assigning of epochs and the storage scheme for events. Section 4.2.3 describes the guaranteed delivery of events to all units within the subsystem. Finally in section 4.2.5 we describe the recovery scheme for roaming clients or clients connecting back after a prolonged disconnect.

4.1. 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

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 I. Replication granularity at different nodes within a sub system

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.

4.1.1. 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. If there is a stable storage set up for servicing all the broker nodes within a level-ℓ unit, then we denote the replication granularity for nodes within that part of the sub system as r_ℓ. 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 fine 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 4 depicts the different replication granularities 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 I lists the replication granularities available at different nodes within the sub system depicted in figure 4.

4.1.2. Stability

For an N-level system with a minimum replication granularity of r_N, the responsibility for ensuring stability of events 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,

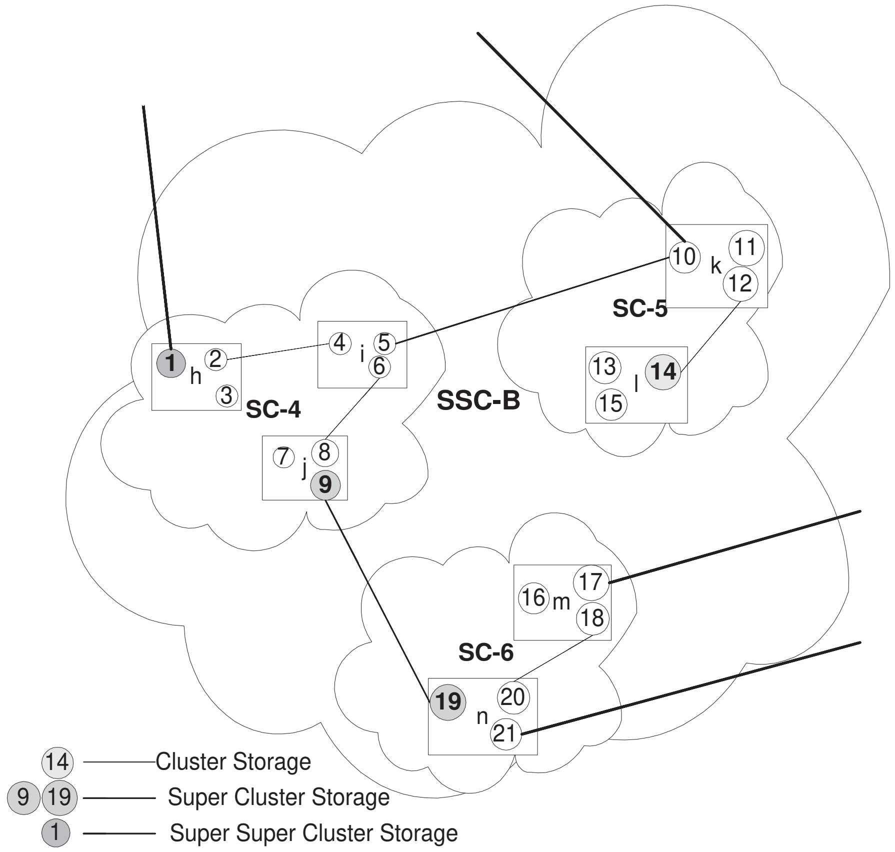

Figure 4. The replication scheme

at different levels within a single super-unit, the stability requirements for individual nodes are delegated to the finest grained store servicing the corresponding node. Thus, in figure 4 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 the event in case of failures or roams initiated by clients. When a node is hosting a stable storage with r_ℓ, this node is responsible for computing destinations, comprising units at level-(ℓ-1), within the level-ℓ unit that the node belongs to. Along with these destinations the node also computes the *predicate count per destination* that are interested in receiving the event, this feature allows us to garbage collect events upon receiving acknowledgements.

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 4, 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.

4.2. Epochs

Epochs are used to aid the reconnected clients and also to recover from failures. We use epochs to ensure that the recovery queues constructed for clients would not comprise of events that a client was not originally interested in. Failure to ensure this could lead to starvation of some of the clients. We also need epochs to provide us with a precise indication of the time from which point on a client should receive events. Not having this precise indication (during recoveries) leads to client starvations and also would also cause the system to expend precious network cycles in routing these events. We also have an epoch associated with every profile change and require that the client wait till it receives the epoch notification, before it can disconnect from the system.

4.2.1. Epoch generation

Epochs, denoted ξ, are truly determined by the replication granularities that exist in different parts of the system. 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 an epoch ξ^ω associated with it.

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. 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.

4.2.2. The storage format

When an event is written to a stable storage, there is an epoch number associated with it. Since all events are not routed to all destinations we maintain the destinations associated with the event. For each destination we also store the predicate count associated with that destination. 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 $- < \xi^e, (d_0^e, d_1^e, \cdots, d_n^e), (p_0^e, p_1^e, \cdots, p_n^e), e.type, e.length, e.serialize >$.

4.2.3. System storage and guaranteed delivery of events

For a level-N system, the stable storages servicing the individual level-N units are also designated as *system storages*. 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.

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. Furthermore, 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.

4.2.4. 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 4.1. *A stable storage cannot remain failed forever, and must recover within a finite amount of time.*

4.2.5. Routing events to a reconnected client

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. The recovering 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.

Stable storages at higher levels are aware of the finer grained replication schemes that exist within its unit. If a coarser grained stable storage is servicing the broker the client was last attached to, 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 finer grained stable storage, which now services the client's lower level GES context.

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.
(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. Item (c) provides for the precise recovery of the disconnected client . Details regarding the precise recovery mechanism can be found in [40, 39].

5. 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 and variance in the latencies. 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.

5.1. Experimental Setup

The system comprises of 22 broker node processes organized into the topology shown in the Figure 5. 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 5. 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).

5.2. Factors to be measured

Once the publisher starts issuing events the factor that we are most interested in is the *latency* in the reception of events. This latency corresponds to the response times experienced at each of the clients. We measure the latencies at the client under varying conditions of *publish rates*, *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 and the *variance* in the sample of these events. 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 5.3.3.

5.2.1. Measuring the factors

For events published by the publisher the number of tag-value pairs contained in every event is 6, with the matching being determined by varying the value contained in the fourth tag. The profile for all the clients in the system, thus have their first 3 <*tag=value*> pairs identical to the first 3 pairs contained in every published event. This scheme also ensures that for every

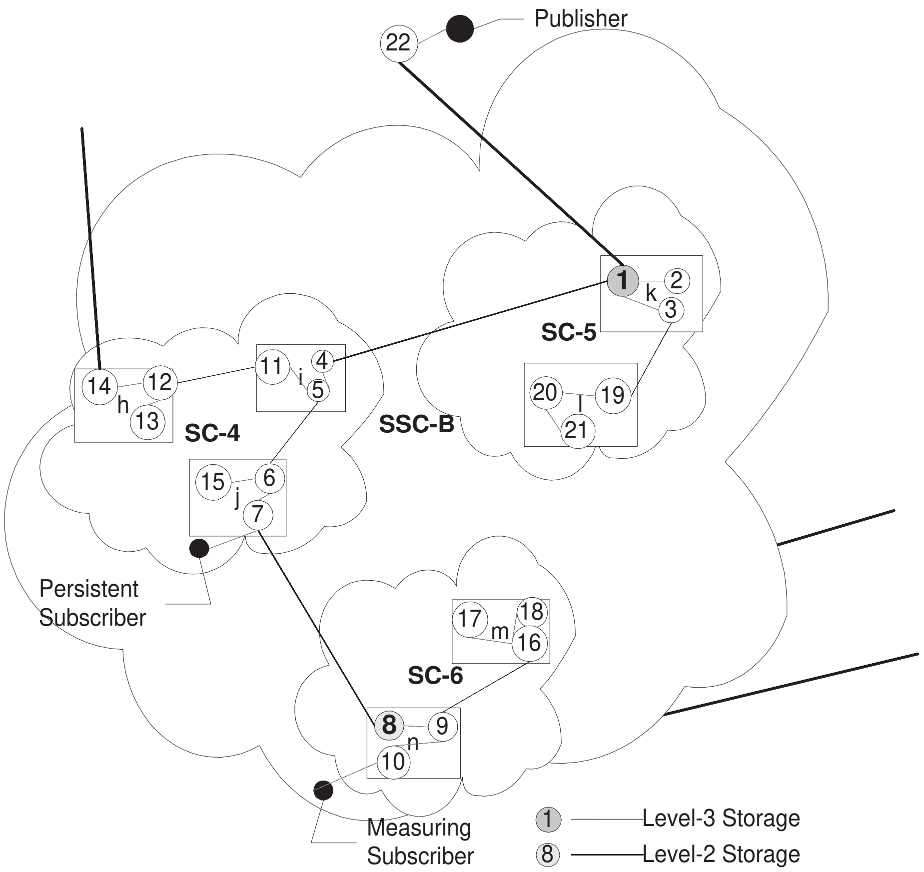

Figure 5. Testing Topology - (I)

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.

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. Having the publisher and one of the subscribers on the same physical machine with access to the same underlying clock, obviates the need for clock synchronization and also accounts for clock drifts.

5.3. 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. The delays are in the range of 1-2 mSec for every broker hop. Implementing certain sections of the networking code in C/C++ and then employing JNI to provide access to these native routines could improve upon this delay significantly. The only disadvantage that would result is that we would need to compile programs separately for different platforms. However the brokers written in Java and Java/JNI could still continue to inter-operate with each other. Thus users can decide which parts of the broker network would run the JNI-optimized implementation and which would not.

5.3.1. 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.

Figure 6 depicts the pattern of decreasing latencies with decreasing matching rates. Table II 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. In the flooding approach, all events would still have been routed to all clients irrespective of the matching rates. Additional results

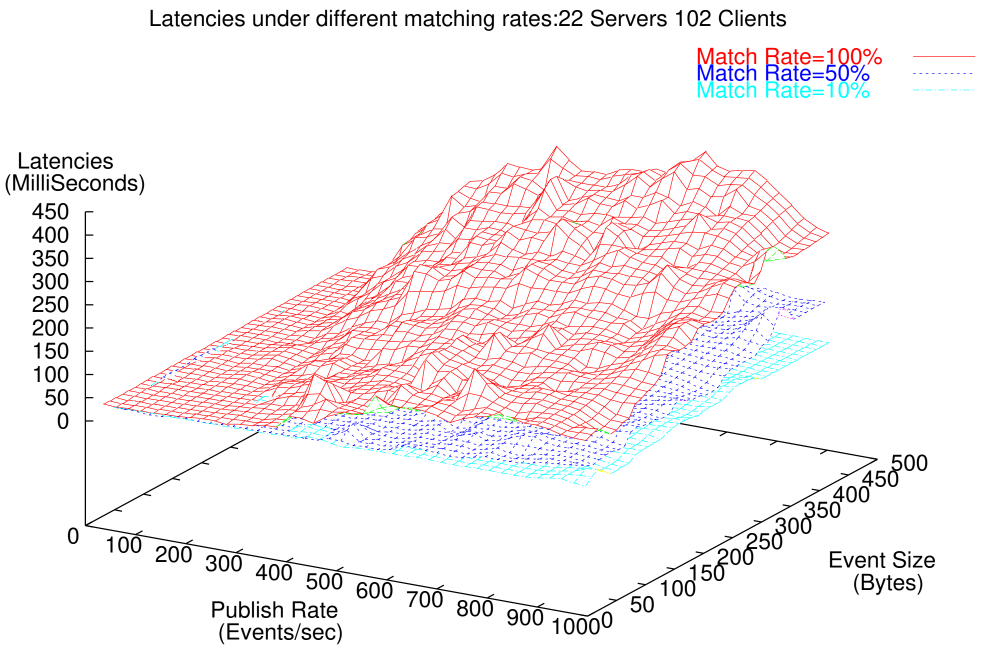

Figure 6. Latencies for match rates of 100%, 50% and 10%

for latencies at different matching rates, system throughput measurements and changes in variance in the latency samples can be found in [40, 37, 39].

5.3.2. Persistent Clients

In figure 5 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 5 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

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 II. Latency and Variance Range Table

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 7 depicts the latencies in delivery of events at persistent clients, with singular and double replications for a matching rate of 50%.

5.3.3. Pathlengths and Latencies

The topology in figure 5 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 8. 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.

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

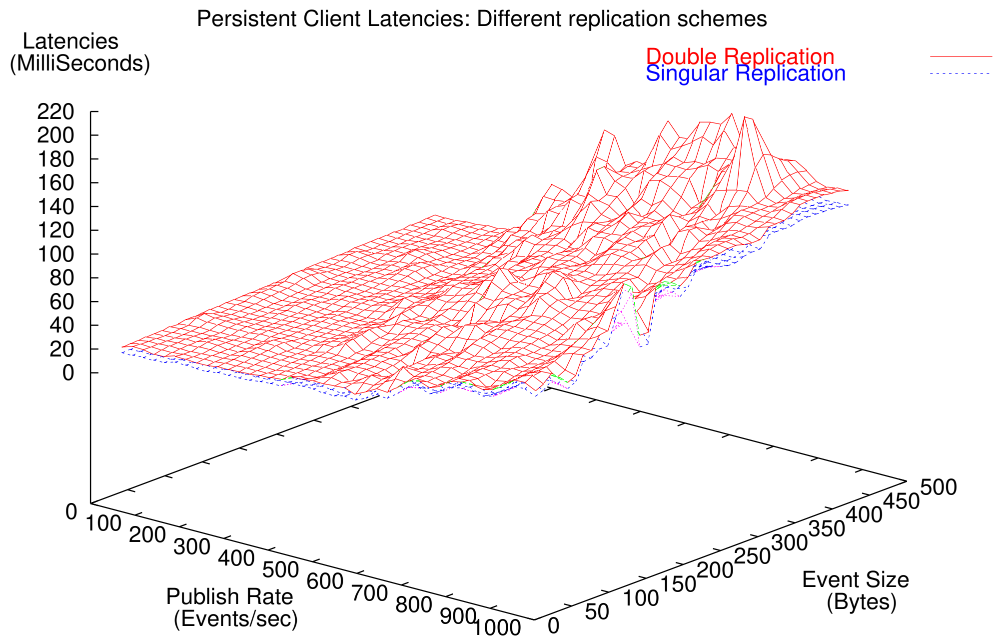

Figure 7. Match Rates of 50% - Persistent Clients (singular and double replication)

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 a matching rate of 50%. The pattern of decreasing latencies with a decrease in the number of broker hops is clear by looking at figure 9. Additional results at different matching rates can be found in [37].

6. 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 broker nodes. The delivery guarantee needs to be met across client roams and also in the presence of broker failures. 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. The broker topology ensures that pathlengths would only increase logarithmically

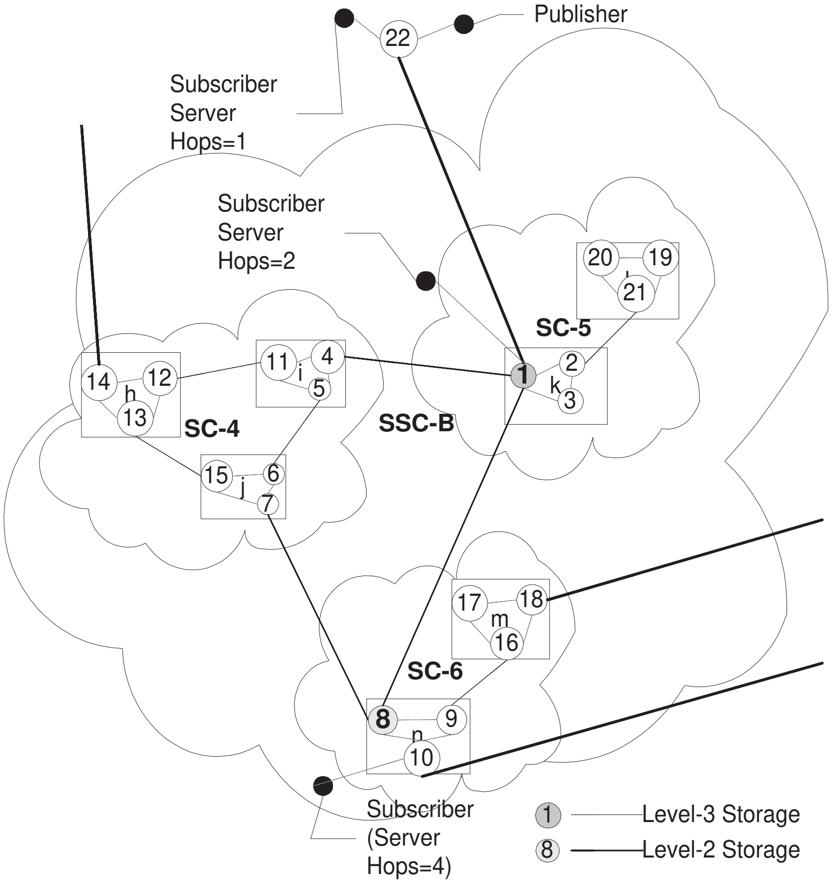

Figure 8. Testing Topology - Latencies versus broker hops

with geometric increases in the size of the broker network. The feature of having multiple links between two units/super-units ensures a greater degree of fault tolerance.

The *system views* at each of the broker 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 broker nodes. 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

Prepared using cpeauth.cls

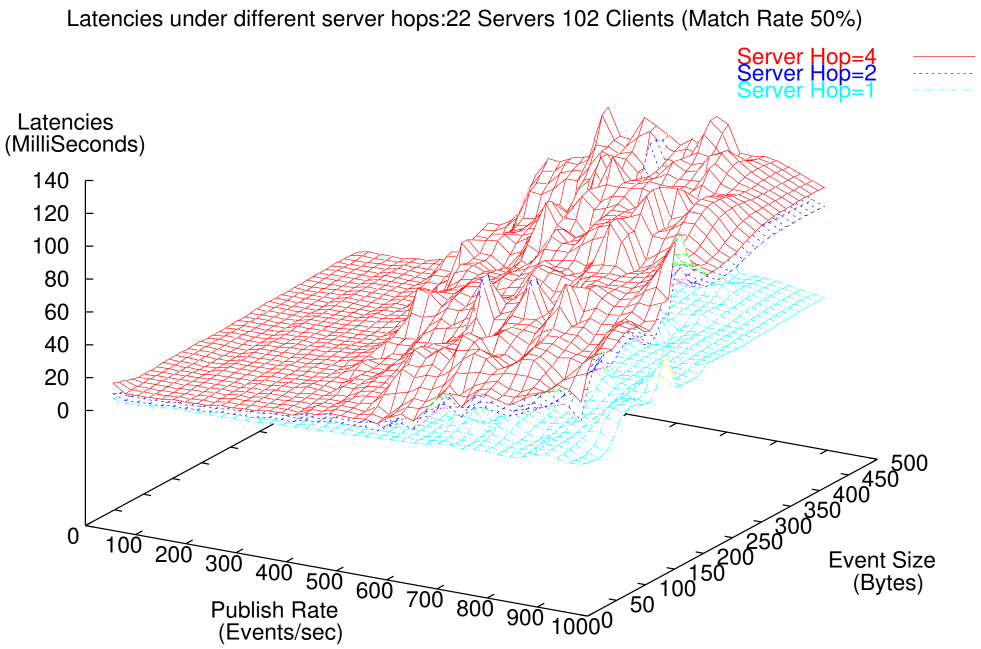

Figure 9. Broker Hops - Match Rate 50%

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 broker 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 brokers 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.

Clients can connect to local brokers instead of reconnecting all the way back to the remote broker 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 broker. The failure model of the system, which allows a broker node or a unit/super-unit of broker nodes to fail and remain failed forever and still satisfy delivery guarantees is another significant feature, which also allows the system to be easily extensible. This model ensures that clients need not wait for a broker to recover after this broker 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 brokers 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 5 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 brokers. The seamless integration of multiple broker nodes in our framework and the failure model that we impose on broker nodes provides for very easy maintenance of the broker network.

REFERENCES

1. 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.
2. Ken Arnold, Bryan O'Sullivan, Robert W. Scheifler, Jim Waldo, and Ann Wollrath. *The Jini Specification*. Addison-Wesley, June 1999.
3. 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.
4. 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.
5. 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.
6. 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.
7. Kenneth Birman. The process group approach to reliable distributed computing. *Communications of the ACM*, 36(12):36–53, 1993.
8. 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.
9. 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.

10. 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.
11. 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.
12. 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.
13. Akamai Corporation. EdgeSuite: Content Delivery Services . Technical report, URL: http://www.akamai.com/html/en/sv/edgesuite_over.html, 2000.
14. 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.
15. 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.
16. Talarian Corporation. Smartsockets: Everything you need to know about middleware: Mission critical interprocess communication. Technical report, URL: http://www.talarian.com/products/smartsockets, 2000.
17. TIBCO Corporation. TIB/Rendezvous White Paper. Technical report, URL: http://www.rv.tibco.com/whitepaper.html, 1999.
18. D Dolev and D Malki. The transis approach to high-availability cluster communication. In *Communications of the ACM*, volume 39(4). April 1996.
19. Guy Eddon and Henry Eddon. Understanding the DCOM Wire Protocol by Analyzing Network Data Packets. *Microsoft Systems Journal*, March 1998.
20. Geoffrey Fox et al. Grid Services and Earthquake Science. *Concurrency and Computation: Practice and Experience*, 2002.
21. Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. Technical report, Message Passing Interface Forum, May 1994.
22. Geoffrey Fox and Shrideep Pallickara. An Approach to High Performance Distributed Web Brokering. To appear in ACM Ubiquity.
23. John Gough and Glenn Smith. Efficient recognition of events in a distributed system. In *Proceedings 18th Australian Computer Science Conference (ACSC18)*, Adelaide, Australia, 1995.
24. 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.
25. 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.
26. Mark Happner, Rich Burridge, and Rahul Sharma. Java message service. Technical report, Sun Microsystems, November 1999.
27. 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.
28. Peter Houston. Building distributed applications with message queuing middleware - white paper. Technical report, Microsoft Corporation, 1998.
29. IBM. IBM Message Queuing Series. http://www.ibm.com/software/mqseries, 2000.
30. Softwired Inc. iBus Technology. http://www.softwired-inc.com, 2000.
31. iPlanet. Java Message Queue (JMQ) Documentation. Technical report, URL: http://docs.iplanet.com/docs/manuals/javamq.html, 2000.
32. Javasoft. Java Remote Method Invocation - Distributed Computing for Java (White Paper). URL: http://java.sun.com/marketing/collateral/javarmi.html, 1999.
33. Kammie Kayl. JOY POSES JXTA INITIATIVE: Pushing the Boundaries of Distributed Computing. Technical report, Sun Microsystems, February 2001.
34. 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.
35. The Object Management Group (OMG). OMG's CORBA Services. URL: http://www.omg.org/technology/documents/, June 2000. Version 3.0.
36. Andy Oram, editor. *Peer-To-Peer - Harnessing the Benefits of a Disruptive Technology.* O'Reilly & Associates, Inc., 1.0 edition, March 2001.

37. Shrideep Pallickara and Geoffrey Fox. Initial Results from an Early Prototype of the Grid Event Service. Technical report, IPCRES Grid Computing Laboratory, 2001.
38. Shrideep Pallickara and Geoffrey Fox. Routing Events in The Grid Event Service. Technical report, IPCRES Grid Computing Laboratory, 2001.
39. Shrideep Pallickara and Geoffrey Fox. The Grid Event Service (GES) Framework: Research Direction & Issues. Technical report, IPCRES Grid Computing Laboratory, 2001.
40. Shrideep B. Pallickara. *A Grid Event Service*. PhD thesis, Syracuse University, June 2001.
41. R Renesse, K Birman, and S Maffeis. Horus: A flexible group communication system. In *Communications of the ACM*, volume 39(4). April 1996.
42. 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.
43. Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *Middleware*, 2001.
44. 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.
45. 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.
46. D.J. Watts and S.H. Strogatz. Collective Dynamics of 'Small-World' Networks. *Nature*, 393:440, 1998.