# A Scalable Durable Grid Event Service

Geoffrey C. Fox
*Indiana University, Computer Science and Informatics, Community Grid Labs*

Shrideep Pallickara
*Syracuse University*

# A Scalable Durable Grid Event Service

Geoffrey Fox (gcf@indiana.edu)
Community Grids Laboratory
*Computer Science & Informatics, Indiana University*

Shrideep Pallickara (shrideep@cat.syr.edu )
*Dept. Of Electrical Eng. & Computer Science, Syracuse University*
3-211 CST, 111 College Place
Syracuse, New York 13244
Tel: (315)-443-4884

## Abstract

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.

**Keywords:** Message Oriented Middleware, Grid Systems, publish/subscribe, guaranteed messaging

## 1.0 Introduction

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.

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.

In this paper 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 users 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 the GES. Brokers can run on 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). Further 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). This section 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.

## 2.0 The Broker Topology

One of the reasons why one would use a distributed model is high availability. Having a centralized model would imply a single broker (constituting a single point of failure) hosting multiple clients. 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. The smallest unit of the system is a *broker node* and constitutes a unit at level-0 of the system. Broker nodes grouped together form a *cluster*, the level-1 unit of the system. Clusters could be clusters in the traditional sense, groups of broker nodes connected together by high-speed links. A single broker node could also decide to be part of such traditional clusters, or along with other such broker nodes form a cluster connected together by geographical proximity but not necessarily high-speed links.

Several such clusters grouped together as an entity comprises a level-2 unit of our network and is referred to as *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. Another factor, crucial for optimization of routing algorithms is the *block-limit,* which limits 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. Thus, in a system comprising of super-super-clusters with a block-limit of 32, the total number of broker nodes that can be present in the system is 32 x 32 x 32 x 32 i.e. 1048576.

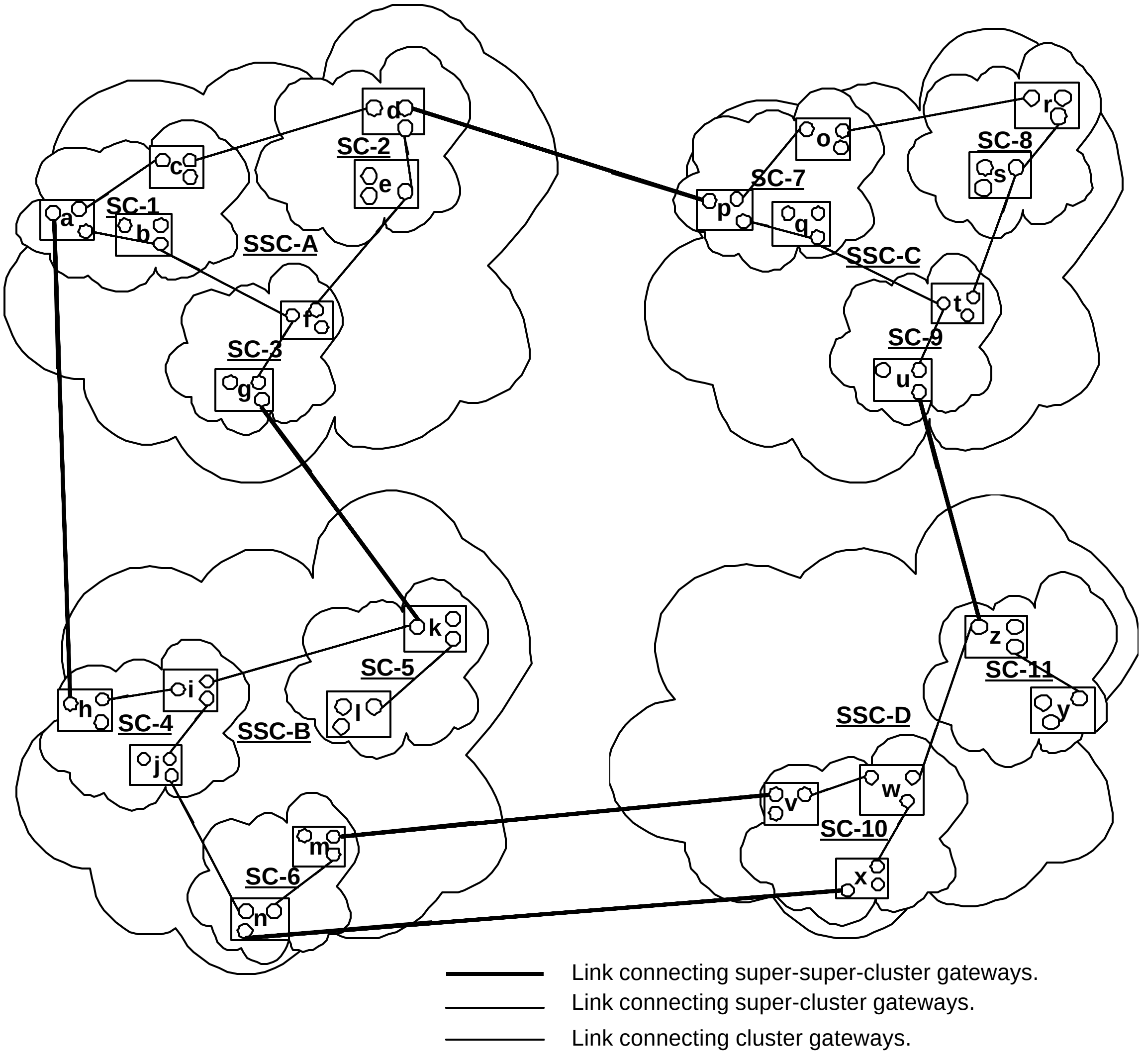
a
b
c
d
e
SC-1
SC-2
SSC-A
f
SC-3
g
o
r
SC-8
s
SC-7
p
q
SSC-C
t
SC-9
u
k
SC-5
h
i
SC-4
SSC-B
l
j
m
SC-6
n
z
SC-11
SSC-D
y
v
w
SC-10
x
Link connecting super-super-cluster gateways.
Link connecting super-cluster gateways.
Link connecting cluster gateways.

Figure 1 : An example network of brokers

The topology we have employed with strongly connected nodes in a cluster, along with multiple connections between different units in the system leads to the creation of *small world* [1] networks. These networks have the advantage that the communication *pathlengths* that a broker node requires to communicate with any other broker node in the system increases logarithmically with geometric increases in the size of the broker network. Clusters have broker nodes at least one of which is connected to at least one of the nodes within some other cluster. In some cases there would be multiple links from a cluster to some other cluster. This architecture provides a greater degree of fault tolerance by providing multiple routes to reach the same cluster. We refer to such nodes as *gatekeepers*. 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. If a broker is connected to a broker in some other cluster within the same super-cluster we refer to those brokers as cluster gatekeepers (gatekeepers at level-1). Similarly if the brokers are in different super-clusters but within the same super-super-cluster, the brokers providing this gateway are referred to as super-cluster gatekeepers (gatekeepers at level-2). A given broker node could be a gatekeeper at several levels. 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**.

## 3.0 Routing events

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 would however place a strain on network resources. Under conditions of high load and increasing selectivity by the clients, the number of events that a client discards would far exceed the number of events it is actually interested in. This scheme also affects the latency associated with the reception of real time events at the client. The increase in latency is due to the cumulation of queuing delays associated with the *uninteresting/flooded* events. The system thus needs to be very selective of the kinds of events that it routes to a client.

In *Elvin* [2] network traffic reduction is accomplished through the use of *quench* expressions. Quenching prevents clients from sending notifications for which there are no consumers. Strategies to route events to clients by selectively employing links for dissemination do not exist. In *Gryphon* [4] each broker maintains a list of all the subscriptions within the system in a parallel search tree (PST). The PST is annotated with a vector encoding link routing information. At matching time, these annotations are employed by every broker to determine which of its neighbors should receive that event. The obvious disadvantage of this approach is that all subscriptions are maintained at every broker. Thus when a broker is newly added into the system all subscriptions need to be routed to this broker. The approach adopted by the OMG *Event* and *Notification services* [6] is one of establishing channels and registering suppliers/consumers to those channels. This scheme of registering with multiple channels does not scale well increasing number of clients as well as greater selectivity in events. 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. 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.

### 3.1 Connectivity graph

A broker needs to be aware of the network layout to optimize routing to destinations. However, given the potential size (millions) of the broker network, it is impractical for any broker to be aware of the complete network inter-connection scheme. What is required is an abstract view of the broker network, while still being able to ensure the calculation of optimal paths for communication within the system. This information is encapsulated within the connectivity graph. The information encapsulated within the connectivity graph depicts inter-connections between the brokers in the cluster that it is a part of, the interconnections between the clusters within the super-cluster that it belongs to and so on. The connectivity graph at each node would be different, while still providing a consistent view of the system interconnections. Connectivity graphs are updated at each node in response to the creation of connections between brokers. Dissemination constraints are imposed on the propagation of connection information outside a given unit. For example information regarding connections within a cluster should not be propagated outside the cluster. This connection information is also modified as it is being propagated through certain sections of the broker network.

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. Conforming to the dissemination constraints, 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**. 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 [8][9].

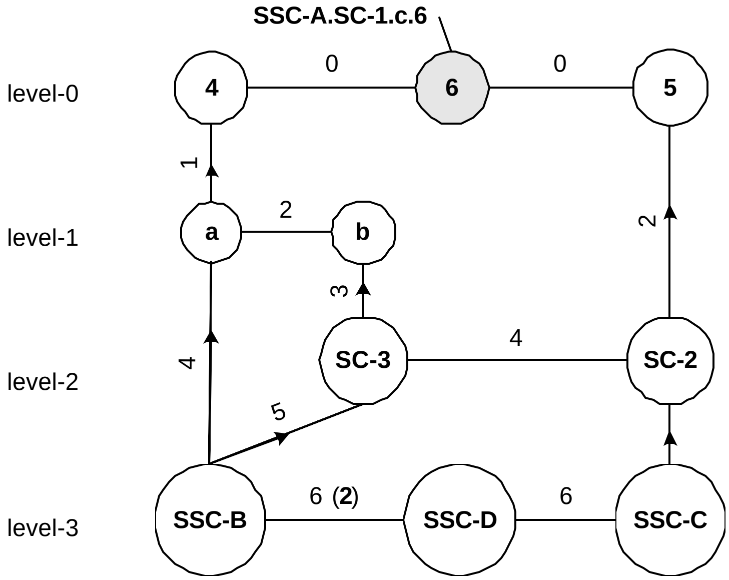

Figure 2 : Connectivity graph at broker node **6**

Figure 2 depicts the connectivity graph depicts the connectivity graph that is constructed at the node **SSC-A.SC-1.c.6** in Figure 1. 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. This cost scheme is encapsulated in the *link cost matrix*, which can be dynamically updated to reflect changes in link behavior with the passage of time.

The process of calculating the shortest path, from the node to the *vertex* (broker hosting the connectivity graph), starts at the node in question. The directional arrows indicate the links, which comprise a valid path from the node in question to the vertex node. Edges with no imposed directional constraints are bi-directional. Hops computed for destinations are then maintained in a routing cache so that events can be disseminated faster throughout the system. The routing cache used in tandem with the event routing information to decide on the next best broker hop for an event ensures efficient dissemination. Sections of the routing cache are invalidated/updated in response to the addition of brokers or initiation of connections between existing brokers.

### 3.2 Organization of Profiles

A profile comprises of *constraints* on successive attributes in an event's signature. Constraints from multiple profiles are organized in the *profile graph*. We use the general matching algorithm the Gryphon [4] system to organize profiles and match the events. A constraint is the specification of a value that a particular attribute can take. We however also allow for the weakest constraint, denoted *****, which encompasses all the values within the range permitted by the attribute's type. Figure 3 depicts the profile graph constructed from three different profiles. To ensure reliable delivery and subsequent garbage collection of events from stable storage 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* (individual filters in a client profile) within that unit that are interested in the traversal of that edge. The predicate information also allows us to remove certain edges and nodes from the profile graph, when clients are not interested in those constraints any more.

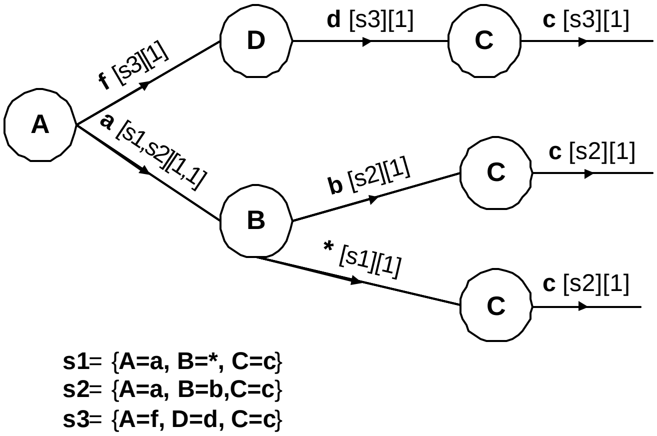

Figure 3 : Profile Graph example

When an event arrives we first check to see if the profile graph contains the first attribute contained in the event. If that is the case we can proceed with the matching process. When an event's content is being matched, the traversal is allowed to proceed only if –

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

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

### 3.3 Propagation of profiles

In our scheme no broker maintains all subscriptions that exist within the system. Information regarding profiles is propagated hierarchically; a broker maintains profiles of all attached clients, cluster gatekeepers maintain profiles all brokers within that cluster and so on. In the case of multiple unit-gatekeepers within the same unit, profile information needs to be propagated to every such gatekeeper. This organization of profiles ensures that –

(a) When an event is routed to a unit, there is at least one destination within that unit which is interested in the content contained in that event.

(b) There are no events that are not routed to that unit, which would have at least one valid destination within that unit.

To ensure these properties, a 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. This information regarding the nodes to propagate profile changes (at different unit levels) is computed from the connectivity graph. In the connectivity graph depicted in Figure 2 profile changes at any of the broker nodes within cluster **c**, need to be routed to node **4**. Any profile changes at cluster gatekeeper node **4** needs to be routed to node **5**, and also to a node in cluster **b**. Similarly level-2 changes at node **5** needs to be routed to the level-3 gatekeeper in cluster **a** and super-clusters **SC-3**, **SC-2**. When such propagations reach any 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*, associated with edges in the profile graph, does not fail due to delivery of the same profile change multiple times within the same unit.

### 3.4 Routing events

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 are made on the basis of this information, which is added by the system to ensure the fastest dissemination. As the event flows through the system, via gateways, the routing information is modified to snapshot its dissemination within the broker network.

Before an event is sent over a gateway, the broker analyses the routing information to ensure that the event is not routed to a unit, which had already received the event. Further, information pertaining to lower level disseminations are discarded prior to routing an event over a higher level gateway. A gatekeeper at a higher level, when it is presented with an event, computes the lower level destinations e.g. a cluster gatekeeper computes the broker destinations associated with a newly arrived event. This calculation is based on the profiles available at the gatekeeper. At every node the best hops to reach the destinations are computed. Nodes and links that have not been failure suspected are the only entities that can be part of the shortest path. Thus, at every node the best decision is taken. The event routing protocol, along with the profile propagation protocol and the gateway information ensure the near *optimal routing* scheme for the dissemination of events in the existing topology.

## 4.0 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 network partitions. 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.

Systems such as *Sienna* [3] and *Elvin* [2] focus on efficiently disseminating events, and do not sufficiently address the reliable delivery problem in the presence of failures. In *Gryphon* the approach to dealing with broker failures is one of reconstructing the broker state from its neighboring brokers. This approach requires a failed broker to recover within a finite amount of time, and recover its state from the brokers that it was attached to prior to its failure. What is not clear is the state recovery problem in the presence of multiple broker failures. *SmartSockets* [5] provides high availability/reliability through the use of software redundancies. Mirror processes receive the same data and perform identical sequence of actions as the primary process, and can thus take over in the event of process failures. This approach runs into scaling problems since each process needs to have a mirror counterpart. Also, since entire networks would be mirrored in this approach, network cycles expended for dissemination increase with network size. Recovery mechanisms during process/mirror pair failures are however not addressed. *TIB/Rendezvous* [10] in tandem with *TIB/Hawk* allow uninterrupted local operations while recovering from application outages. Most of these systems nevertheless require failed brokers to recover within a finite amount of time. Statically pre-configured message queuing products such as IBM's MQSeries and Microsoft's MSMQ employ the *store-and-forward* approach and 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.

### 4.1 Stable storage and data replication issues

Storages exist en route to destinations but decisions need to be made regarding when and where to store an event and also on the number of replications that we intend to have for any given event. Events can be forwarded to clients only after they have been written to stable storage. Since events arrive at a unit via gatekeepers, the stable storages are configured at these brokers. These stable storages are then responsible for ensuring the guaranteed delivery of events for clients attached to brokers within the unit that it is servicing. The level of the gatekeeper, which hosts the stable storage dictates the *replication granularity* of the brokers being serviced by that stable storage. Also if a unit has more than one unit-gatekeepers, only one of them can be configured to have access to stable storage.

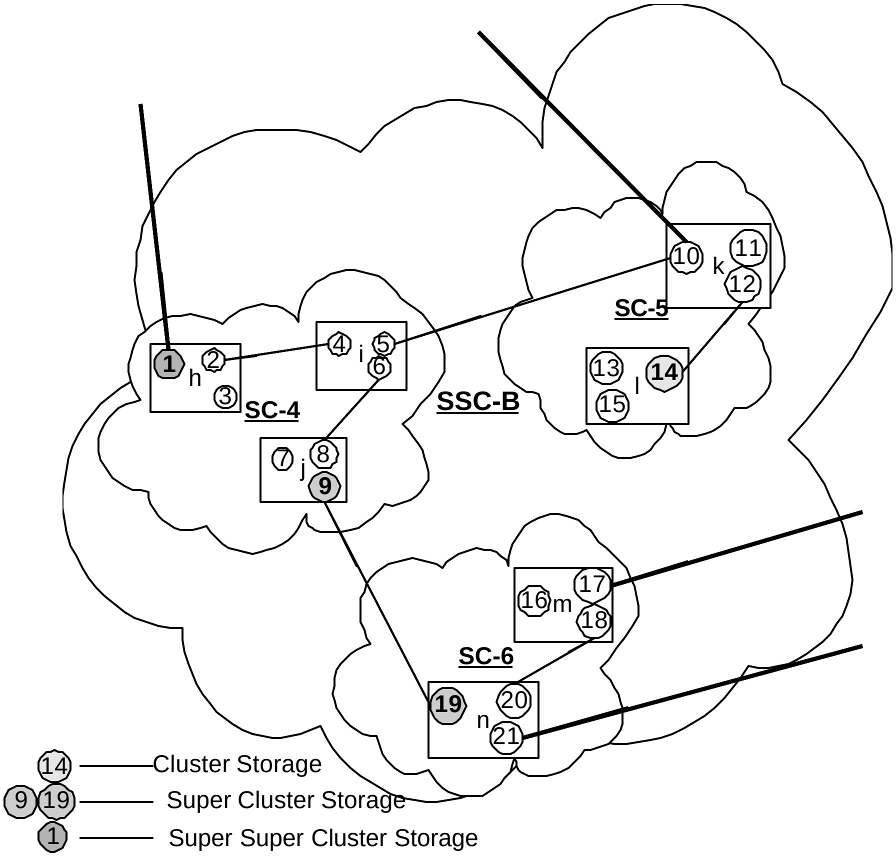

Figure 4 : The replication scheme

Figure 4 depicts the different replication granularities that can exist within different parts of a sub system. Table 1 outlines the stable storages that service the nodes within the system. 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 3. Similarly there could be no other stable storages, which try to service units **SC-4** and **SC-6** with a replication granularity of 2.

| Nodes | Granularity | Servicing Storage |
|---|---|---|
| 10,11,12 | 3 | 1 |
| 1,2,3,4,5,6,7,8,9 | 2 | 9 |
| 16,17,18,19,20,21 | 2 | 19 |
| 13,14,15 | 1 | 14 |

Table 1: Replication granularity at different nodes

With the addition of stable storages to service smaller units, brokers within those units that were being serviced by higher-level stable storages, will now have their replication granularities updated. In the topology in Figure 4 if we were to set up a level-2 stable storage at node **10**, the replication granularities at nodes **10,11** and **12** would be updated from 3 to 2. The replication granularity for every broker node in the cluster **SC-5.l** remains unchanged at 1.

### 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, with the system expending 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.

Epochs are truly determined by the replication schemes 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.
(c) For every client profile there is an epoch associated with it.

To aid in precise recoveries of clients every event received at a client needs to have an epoch associated with it. The arrival of such an epoch bearing event results in an advancement of the epoch associated with the client's profile.

When a node is hosting a stable storage, destinations pertaining to lower level disseminations are computed. The node also computes the *predicate count per destination* for every computed destination, a feature that aids in the garbage collection scheme. These destinations, along with the predicate count for individual destinations, the epoch and event itself is what is stored at the stable storage.

### 4.3 System storages and guaranteed delivery of events

We refer to units at the highest level comprising the broker network as *super-units*. The constraint imposed by the system is that there should be a stable storage, which is responsible for ensuring the stability of events being delivered to clients attached to any of the brokers within the super-unit. Stable storages servicing these individual super units are also additional responsibility of functioning as *system storages*. For events issued by clients attached to nodes within these units, the system storage nodes have the responsibility to maintain events in stable storage till such time that they are sure that all the other system storages within the system have received that event. When an event is issued within a unit, the destinations are computed as described in the event routing protocol. However, before the event is allowed to leave the super-unit in which it was issued, it must be stored onto the system storage within that super-unit. The system storage node maintains the list of all known super-unit destinations within the system. This destination list is associated with every event that is stored by the system storage. Also associated with these events is a sequence number, which is different from the epoch number associated with the events that clients receive. Further, sequence numbers associated with events are used *only* by system storages to conjecture the events that they should have received from any other system storage within the system.

Once the event is stored at the system storage, it is ready to be sent across to the other super-unit destinations within the system. Also, for an event that is issued by a client within a given super-unit, the event is stored to only at the system storage within the super-unit in which the event was issued, and not at any other system storages within the system. Positive acknowledgements from other system storages aid in the garbage collection of events that have been stored by a system storage. Negative acknowledgements are used to retrieve events from a given system storage once holes have been detected in the sequence of events that should have been received.

### 4.4 Routing events to a reconnected client

When a client is not present in the system, the event is not acknowledged and thus cannot be garbage collected by the storage 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, who 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.

For a profile associated with a client, when a disconnected client joins the system it presents the node that 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 received from the stable storage of the sub system that it was formerly attached to.
(c) The list of the profile ID's associated with client's profile.

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 [8][9].

### 4.5 GES Solution Highlights and extension to dynamic topologies

Our solution to the reliable delivery problem eliminates the finite-time-recovery constraint for individual brokers, while not relying on state reconstructions from neighboring brokers. The solution allows the replication strategy to evolve over time while ensuring that no client is affected by these changes. Clients reconnecting after system failures or prolonged disconnects make a complete and precise recovery. The stable storage persistence model is very selective about the events that it subscribes to, and incorporates a garbage collection scheme that performs even in the presence of roaming/disconnected clients and multiple broker failures. The solution allows lends itself very well to dynamic topologies. Brokers could be dynamically created, connections established or removed, and the events would still be routed to the relevant clients based on the current network fabric. Any given node in the system would thus see the broker network undulate, as the brokers are being added and removed. Average pathlengths for communication could be reduced by instantiating connections to optimize clustering coefficients within the network.

## 5.0 Results:

The system comprises of 22 broker node processes organized into the topology shown in the Figure 5. Each broker node process is hosted on 1 physical Sun SPARC Ultra-5 machine (128 MB RAM, 333 MHz), with no 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) residing on the same Ultra-5 machine and attached to nodes **22** and **10** respectively. In addition to this there are 100 subscribing client processes, with 5 client processes attached to every broker node (except nodes **22** and **10)** within the system. The 100 client node processes all reside on a SPARC Ultra-60 (512 MB RAM, 360 MHz) machine. The run-time environment for all the broker node and client processes is Solaris JVM (JDK 1.2.1, native threads, JIT).

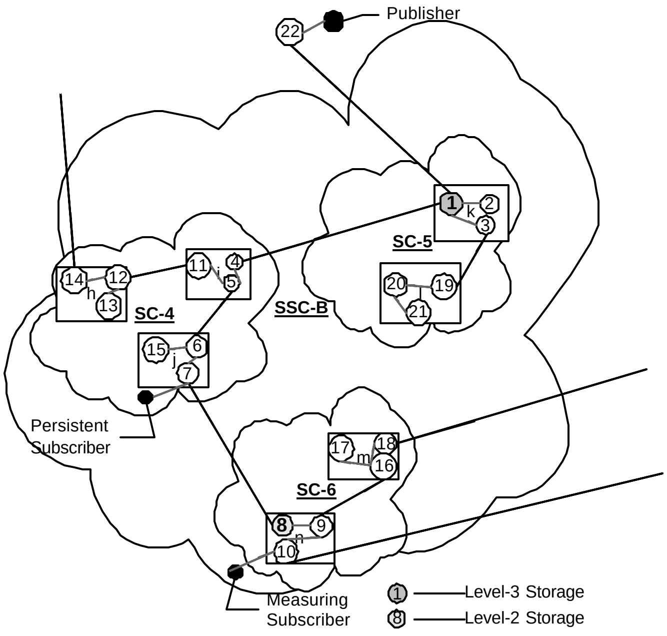

Figure 5: Testing topology

Clients attached to different broker nodes specify an interest in the type of events that they are interested in. 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.

### 5.1 Latencies for routing events to clients

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 the broker hop latencies by a factor of 20. Latencies would thus be in the range of tens of microseconds. 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.

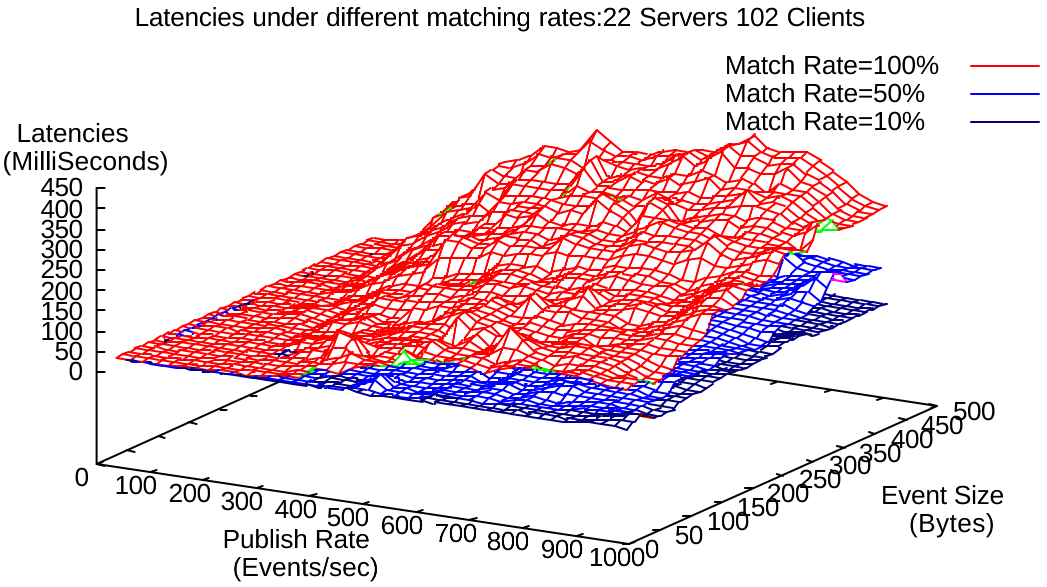

Figure 6 : Latencies under different selectivity rates for clients

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 results in
Figure 6 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; an affect that is more pronounced at higher publish rates and increasing event size. 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. Additional results for different matching rates, varying broker interconnectivity clustering coefficients and also under replication strategies can be found in [8][9].

## 6.0 Conclusion

This paper outlined a scheme for the guaranteed delivery of events to roaming/disconnected clients in the presence of broker and link failures. In section 2 we discussed how the small world behavior of our broker layout leads to greater resilience and better pathlength characteristics. Section 3 outlined the hierarchical dissemination of profiles and content, which along with the routing strategy employed at each broker hop leads to a near optimal solution. In section 4 we described a strategy which allows the replication scheme to evolve with the passage of time. The reliable delivery solution eliminated the finite recovery constraint for brokers and accommodated stable storage failures, while accounting for precise recoveries of disconnected clients in the presence of failures. This model provides an excellent foundation in the development of dynamic topology strategies. This paper addressed broker organization issues, dealt with routing strategies needed for extremely large broker networks, a solution for the reliable delivery problem and shown how these problems can be solved while ensuring good performance. The GES system currently available could be used to test the dynamic topology strategies that we outlined. In the GES protocol layer, a strategy to initiate dynamic servers and connections could very easily be employed. This will be addressed in our future production system and is not the focus of our future research. 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 are generally optimized for a small network of brokers. The seamless integration of multiple broker nodes and the failure models in our framework provide for very easy maintenance of large broker networks.

## References

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

[2] 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.

[3] Antonio Carzaniga, David S. Rosenblum and Alexander L. Wolf. Achieving Scalability and Expressiveness in an Internet-Scale Event Notification Service. *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 219-227, Portland OR, USA, July 2000.

[4] Marcos Aguilera, Rob Strom, Daniel Sturman, Mark Astley and Tushar Chandra. Matching Events in a Content-based Subscription System. *Proceedings of the 18th ACM Symposium on Principles of Distributed Computing*. May 1999.

[5] Talarian Corporation. *SmartSockets: Everything you need to know about middleware: Mission Critical Interprocess Communication*. Technical Report: URL: http://www.talarian.com/products/smartsockets

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

[7] Shrideep Pallickara and Geoffrey Fox. The Grid Event Service (GES) Framework: Research Directions & Issues. Technical Report. IPCRES Grid Computing Laboratory, Indiana University.

[8] Shrideep Pallickara and Geoffrey Fox. Initial Results from an Early Prototype of the Grid Event Service. Technical Report. IPCRES Grid Computing Laboratory, Indiana University.

[9] Geoffrey Fox and Shrideep Pallickara. An Event Service to Support Grid Computational Environments Under Review *Concurrency and Computation: Practice & Experience*.

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