The Grid Event Service: Overview and Future Directions

Geoffrey C. Fox* Shrideep Pallickara†

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 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 are online 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. Also 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's the system as a whole, which should be able to reconstruct the service nuggets that a client missed during the time that it was inactive. Clients just specify the type of events that they are interested in, and the content that the event should at least contain. Clients do not need to maintain an active presence during the time these interesting events are taking place. Once it registers an interest it should be able to recover the missed event from any of the broker nodes in the system. Removing the restriction

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

of clients reconnecting back to the same broker that it was last attached to and the departure from the time-bound recovery failure 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) interaction or a system comprising of agents where discoveries are initiated for services from any of these devices. Finally, some of these devices could also be used as part of a computation. The devices are thus part of a complex distributed system. 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. In this paper we present the Grid Event Service (GES) [23, 21, 22] designed to run on a very large network of broker nodes.

2 Clients and the Broker Topology

Events encapsulate expressiveness at various levels of abstractions - content, dependencies and routing. Where, when and how these events reveal their expressive power is what constitutes information flow within our system. Clients provide services to other clients using events. These events are routed by the system based on the service advertisements that are contained in the messages published by the client. Events routed to a broker are queued and routing decisions are made based on the service advertisements contained in these events and also based on the state of the network fabric.

2.0.1 Destination Lists

Clients in the system specify an interest in the type of events that they are interested in receiving. Some examples of interests specified by clients could be sports events or events sent to a certain discussion group. It is the system which computes the clients that should receive a certain event. A particular event may thus be consumed by zero or more clients registered with the system. Events have implicit or explicit information pertaining to the clients which are interested in the event. In the former case we say that the destination list is *internal* to the event, while in the latter case the destination list is *external* to the event.

An example of an internal destination list is "Mail" where the recipients are clearly stated. Examples of external destination lists include sports score, stock quotes etc. where there is no way for the issuing client to be aware of the destination lists. External destination lists are a function of the system and the types of events that the clients, of the system, have registered their interest in.

2.1 Clients

Clients can generate and consume events in the system. Events in the system are continuously generated and consumed within the system. Clients on the other hand have inherently discrete connection semantics. Clients can be present in the system for a certain duration of time and can be disconnected later on. Clients reconnect at a later time and receive events, which they were supposed to receive as well as events that they were supposed to receive during their respective present incarnation. Clients can issue/create events while in disconnected mode, which would be held in a local queue to be released to the system during a reconnect.

A client profile 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.

2.2 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. Thus all the clients present in the system would be unable to use any of the services provided by the system till a recovery mechanism kicks in. A highly available distributed solution would have data replication at various broker nodes in the network. Solving issues of consistency while executing operations, in the presence of replication, leads to a model where other broker nodes can service a client despite certain broker node failures. The smallest unit of the system is a *broker node* and constitutes a unit at level-0 of the system. Broker nodes grouped together form a *cluster*, the level-1 unit of the system. Clusters could be clusters in the traditional sense, groups of broker nodes connected together by high speed links. A single broker node could also decide to be part of such traditional clusters, or along with other such broker nodes form a cluster connected together by geographical proximity but not necessarily high speed links.

Several such clusters grouped together as an entity comprises a level-2 unit of our network and is referred to as a *super-cluster*. 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 viz. the *block-limit* to 64. In an N-level system this scheme allows for $2^6_N \times 2^6_{N-1} \times \cdots 2^6_0$ i.e $2^{6*(N+1)}$ broker nodes to be present in the system.

We now delve into the *small world graphs* introduced in [28] and employed for the analysis of real world peer-to-peer systems in [20, pages 207 − 241]. In a graph comprising several nodes, *pathlength* signifies the average number of hops that need to be taken to reach from one node to the other. *Clustering coefficient* is the ratio of the number of connections that exist between neighbors of node and the number of connections that are actually possible between these nodes. For a regular graph consisting of n nodes, each of which is connected to its nearest k neighbors − for cases where $n \gg k \gg 1$, the pathlength is approximately $n/2k$. As the number of vertices increases to a large value the clustering coefficient in this case approaches a constant value of 0.75.

At the other end of the spectrum of graphs is the *random graph*, which is the opposite of a regular graph. In the random graph case the pathlength is approximately $\log n/\log k$, with a clustering coefficient of k/n. The authors in [28] explore graphs where the clustering coefficient is high, and with *long connections* (inter-cluster links in our case). They go on to describe how these graphs have pathlengths approaching that of the random graph, though the clustering coefficient looks essentially like a regular graph. The authors refer to such graphs as *small world graphs*. This result is consistent with our conjecture that for our broker node network, the pathlengths will be logarithmic too. Thus in the topology that we have the cluster controllers provide control to local classrooms etc, while the links provide us with *logarithmic* pathlengths and the multiple links, connecting clusters and the nodes within the clusters, provide us with robustness.

2.2.1 Gatekeepers

Within a super-cluster, clusters have broker nodes at least one of which is connected to at least one of the nodes existing within some other cluster. In some cases there would be multiple links from a cluster to some other cluster within the same super-cluster. This architecture provides a greater degree of fault tolerance by providing multiple routes to reach the same cluster. Some of the nodes in the cluster thus maintain connections to the nodes in other clusters. Similarly, some nodes in a cluster could be connected to nodes in some other super-cluster. We refer to such nodes as *gatekeepers*. 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 C_j^{ℓ}, C_k^{ℓ}, the nodes are designated as gatekeepers at $level - \ell$ i.e. $g^{\ell}(C^{\ell+1})$. Thus, in figure 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**.

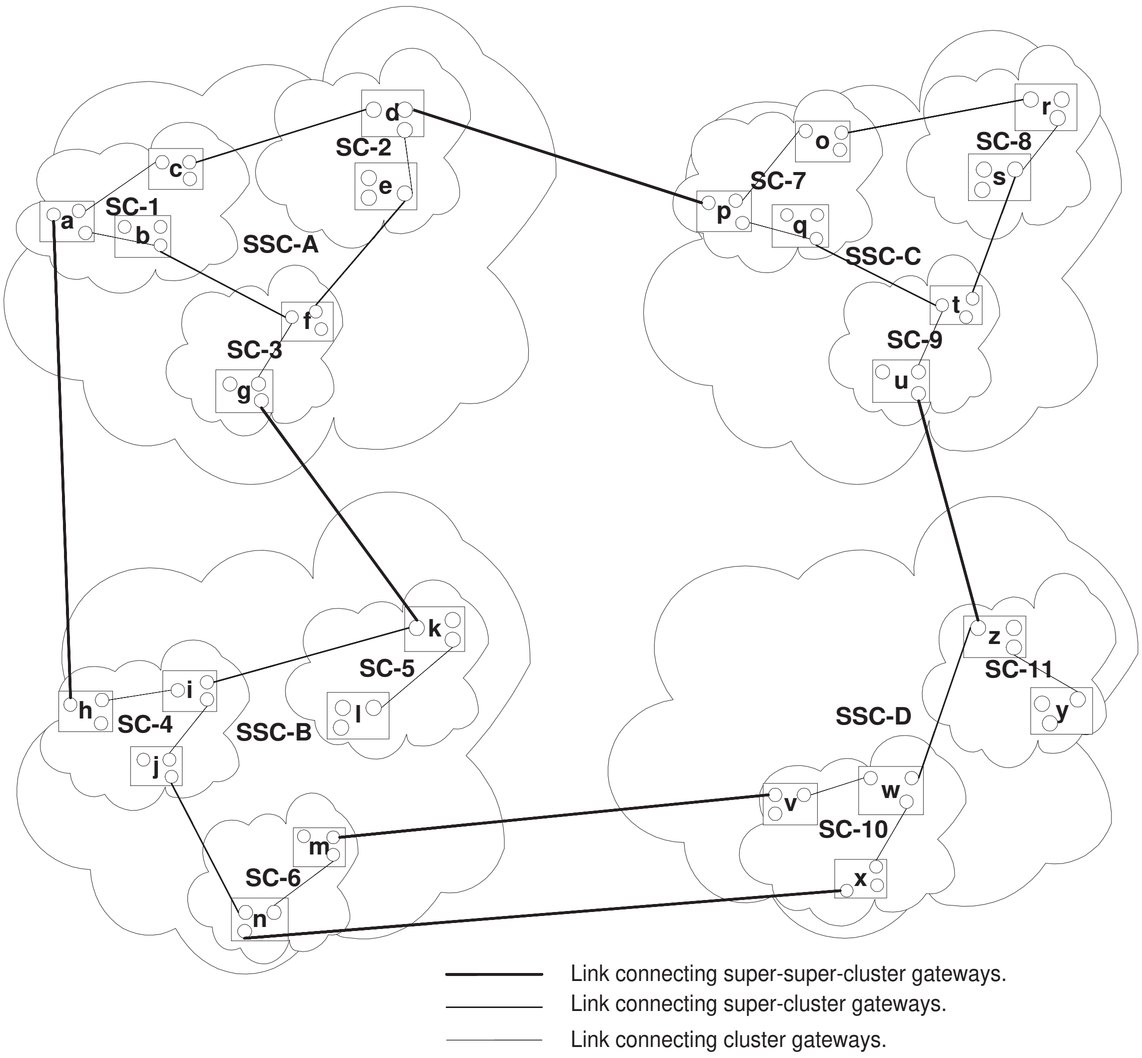

Figure 1: Gatekeepers and the organization of the system

3 The event Delivery problem

The problem of event delivery pertains to the efficient delivery of events to the destinations, which could be internal or external to the event. In the latter case the system needs to compute the destination lists pertaining to the event. The system merely acts as a conduit to efficiently route the events from the issuing client to the interested clients. A simple approach would be to route all events to all clients, and have the clients discard the events that they are not interested in. This approach would however place a strain on network resources. Under conditions of high load and increasing selectivity by the clients, the number of events that a client discards would far exceed the number of events it is actually interested in. This scheme also affects the latency associated with the reception of real time events at the client. The increase in latency is due to the cumulation of queuing delays associated with the *uninteresting/flooded* events. The system thus needs to be very selective of the kinds of events that it routes to a client.

Different systems address the problem of event delivery to relevant clients in different ways. In [12] each subscription is converted into a deterministic finite state automaton. This conversion and the matching solutions nevertheless can lead to an explosion in the number of states. In [25] network traffic reduction is accomplished through the use of *quench* expressions. Quenching prevents clients from sending notifications for which there are no consumers. Approaches to content based routing in *Elvin* are discussed in [26]. In [3, 4] 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 [2] 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 the neighbors that should receive the event. [1] describes approaches for exploiting group based multicast for event delivery. These approaches exploit universally available multicast techniques.

The approach adopted by the OMG [19] is one of establishing channels and registering suppliers and consumers to those event channels. The channel approach in the event service [18] approach could entail clients (consumers) to be aware of a large number of event channels. The two serious limitations of event channels are the lack of event filtering capability and the inability to configure support for different qualities of service. These are sought to be addressed in the Notification Service [17] design. However the Notification service attempts to preserve all the semantics specified in the OMG event service, allowing for interoperability between Event service clients and Notification service clients and thus suffers from similar drawbacks. In *TAO* [13], 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 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* [5] from Akamai, redirect their users to specialized Akamized URLs. *EdgeSuite* then accurately identifies the geographic location from which the clients have accessed the website. This identification is done based on the IP addresses associated with the clients. Each client is then directed to the server farm that is closest to the client's network point of origin. As the network load and server loads change clients could be redirected to other servers. We employ schemes, which ensure that each broker node maintains abbreviated views of system inter-connectivities. This abbreviated system view is maintained in the *connectivity graph*. The connectivity graph has imposed directional constraints on graph traversal and also dynamic costs associated with the same based on link type and links connecting two system units (brokers, clusters, cluster of clusters etc). This is then used to provide us with the fastest hops to employ to reach any given destination. It is ensured that this graph maintains the *true* state of the system, so that only active nodes and fast links are employed for the routing at every broker node where such decisions are made. To ensure that a client misses no interesting event and also to ensure that uninteresting events are not routed to parts of the system not interested in receiving the events, we employ an intelligent dissemination scheme. This dissemination scheme is hierarchical, as is the calculation of destinations and the propagation of profiles. The profile changes are routed to relevant nodes in the system. A client would thus route profile changes to the broker it is attached to, while the broker propagates its profile changes to its cluster controllers (there could be more than one for a cluster) and so on. The hierarchical destinations computed for an event ensure that only the relevant parts of the sub-system receive the event. This scheme is capable of handling dense and sparse interests in different parts of the system equally well. The logarithmic pathlengths achieved by the organization scheme for the broker nodes, combined with the calculation of fastest routes to reach destinations at every broker node hop and the exact sub-systems to route an event provides a near optimal routing scheme for the events.

In our system the node organization protocol could be used in the creation of *small world* [28, 20] networks. This organization, which comprises of strongly connected broker nodes in clusters connected by *long links* ensures that the pathlength increases logarithmically for geometric increases in the size of the broker node network. The feature of having multiple links between two units/super-units ensures a greater degree of fault tolerance. Links could fail, and the routing to those units could still be performed using the alternate links. The organization of connection information ensures that connection losses (or additions) are incorporated into the connectivity graph hosted at relevant nodes. Certain sections of the routing cache are invalidated in response to this addition (or loss) of connections. This invalidation and subsequent calculation of best *hops* to reach units (at different levels) ensure that the paths computed are consistent with the state of the network, and include only valid/active links. The ability to compute routes to reach destinations at different levels lends the scheme very useful for hierarchical disseminations.

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

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

These approaches result in only the relevant links and functioning nodes being employed for disseminations. The *small world* behavior that would exist in broker network, when appropriately organized, ensures that the pathlengths for these disseminations would only increase logarithmically with the number of broker nodes. Details pertaining to the protocols employed for routing of events can be found in [23, 22]

4 Reliable Delivery Problem

Systems such as *Sienna* [4, 3] and *Elvin* [26, 12, 25] focus on efficiently disseminating events, and do not sufficiently address the reliable delivery problem in the presence of failures. In *Gryphon* the approach to dealing with broker failures is one of reconstructing the broker state from its neighboring brokers. This approach requires a failed broker to recover within a finite amount of time, and recover its state from the brokers that it was attached to prior to its failure. *SmartSockets* [8] provides high availability/reliability through the use of software redundancies. Mirror processes receiving the same data and performing the same sequence of actions as the primary process, allows for the mirror process to take over in the case of process failures. The *mirror process* approach runs into scaling problems as the number of processes increase, since each process needs to have a mirror process. Since there is an entire server network that would be mirrored in this approach the network cycles expended for dissemination also increases as the number of server nodes increases. *SmartSockets* also allows for routing tables to be updated in real time in response to link failures and process failures. What is not clear though, is how the system is affected if both the process and its mirror counterpart fail. *TIB/Rendezvous* [9] integrates fault tolerance through delegation to another software *TIB/Hawk* which provides it with immediate recovery from unexpected failures or application outages. This is achieved through the distributed *TIB/Hawk* micro-agents, which support autonomous network behavior, while continuing to perform local tasks even in the event of network failures.

Message queuing products are statically pre-configured to forward messages from one queue to another. This leads to the situation where they generally do not handle changes to the network (node/link failures) very well. They also require these queues to recover within a finite amount of time to resume operations. To achieve guaranteed delivery, JMS provides two modes: persistent for sender and durable for subscriber. When messages are marked persistent, it is the responsibility of the JMS provider [7, 15, 14, 6] to utilize a store-and-forward mechanism to fulfill its contract with the sender (producer).

To account for failure scenarios, recovery from such failures and the reliable delivery of events to clients in the presence of such failures we need a storage scheme. The replication scheme that we have designed allows different replication strategies to exist in different parts of the system. We employ a scheme that allows us to detect partitions in response to unit (brokers, clusters, cluster of clusters etc.) failures or link failures and take appropriate actions to initiate recovery. The GES failure model allows a unit to fail and remain failed forever. Clients attached to affected units can roam the network, attach themselves to a different broker node and still receive all the events that they were supposed to receive. The GES model allows a stable storage to fail, the only constraint imposed is that these stable storages do not remain failed forever and recover within a finite amount of time. During stable storage failures only certain sections of the subsystem are affected. Similarly a stable storage could be added at different parts of the system and be configured as a finer or coarser grained stable storage at the subsystem that it was added to. Clients need not be notified about the addition of stable storages, the system manages reliable delivery of events to clients transparently. The addition of stable storage is disseminated only within certain parts of the system. Details regarding the precise recovery mechanism for clients can be found in [23, 21].

5 Pub/Sub Model, Dynamic Topologies & Security

5.1 The GES publish subscribe Model

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

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

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

Clients can either be *durable* (persistent) or *non-durable*, the difference being in the reliability of events that are delivered to them. Durable clients can leave the system, fail or roam in response to failure suspicions or need for better response times. The system guarantees that all persistent events issued during this time will be delivered to the client across its various incarnations at different parts of the system. This guarantee holds true even in the presence of failures. In our failure model a unit can fail and remain failed forever. The time-constrained persistent events that are routed to a client are those for which the timer has not expired. The timer's associated with these time-constrained events can vary from a few minutes to up to a few days. All subscribing clients (durable or non-durable) receive transient events, the system can compute alternate routes in response to link or node failures. The system does not guarantee the delivery of these events during failures. All clients receive hop-constrained transient events if they are within the line-of-sight for the publisher of these events. Table 1 outlines our discussion regarding the different kinds of events and clients.

5.2 The need for dynamic topologies

This pertains to the scheme for the dynamic creation of servers, to optimize the routing characteristics for events. The routing characteristics pertain to the bandwidth usage, response times and also on the

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

Table 1: The GES publish/subscribe model

protocols that would be employed for the dissemination of events. Consider the scenario where there are server nodes at Syracuse and Rochester. A large number of client nodes attached to one of these servers reside in Boston, Houston and Albany. For a set of clients at either of the aforementioned locations this scheme has the obvious disadvantage that messages routed to each of the clients utilizes the same bandwidth between the server and client's location. For 10 clients (at the same geographic location) attached to the same server node, for a certain event, the bandwidth could be utilized 10 times for the same event.

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

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

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

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

5.3 The proposed security framework

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

5.3.1 Installing the Software

The software for the package comprises of several files. Users who download these files are not aware if these files have been modified. To ensure that the files have not been modified, we employ message digest functions. We could employ the MD5 message digest algorithm (RFC 1321). The software is then placed at the download site and users are free to download this software along with the message digest for the software. Users compute the message digest associated with the software, if the value returned matches the value posted on the website then the user concludes that the files have not been modified.

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

To solve this issue of identity we use digital signatures. The author then encrypts the message digest function with his private key and users employ the author's public key to decrypt the message and retrieve the message digest function. After downloading the software users compute the message digest, if they match it can be assumed that the software was developed by the author/organization whose public key is available for decryption. Since only the author is aware of the private key, only the author could have produced a valid signature. The approach for digital signatures outlined here is the scheme employed by RSA [24] for digital signatures, we could also use the Digital Signature Standard (DSS) approach to providing digital signatures.

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

5.3.2 Authentication

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

Authenticating clients

We now very briefly outline the Kerberos [27, 16] Authentication service that we intend to use. The Kerberos Authentication server (AS) keeps of valid userID's and their passwords. Every server in the system has a unique ID assigned by the system, this information is also maintained by the AS. When a client wishes access to a set of services, the client issues a ticket granting request on behalf of the user along with the ID of the Ticket Granting Server (TGS). The AS responds with a ticket encrypted using a key derived from the user's password. The ticket is recovered only if the user enters the correct password, which would cause the key to be regenerated. The ticket granting ticket that this client receives from the AS is encrypted using a key that is known only to the AS and the TGS. The client then issues a request to the TGS requesting a service-granting ticket. The request comprises of the UserID, the ID of the desired service and the ticket granting ticket. Since the ticket granting ticket is encrypted using a key that only the TGS and the AS are aware of, the TGS decrypts the incoming request and confirms success in the decryption on finding its TGS-ID. It verifies the userID and associated network address to authenticate the user. The TGS also checks to see if the lifetime associated with the ticket-granting ticket has expired. If the lifetime has not expired and if the user is permitted to access the service a ticket is issued for that service.

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

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

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

5.3.3 Communication between servers

We now proceed to outline schemes that could be used for encryption of traffic between the servers. The two possible approaches could involve symmetric key based approaches or asymmetric keys based one. Keys for communication between two servers, over a specific link, could be securely exchanged using the Diffie-Hellman [10, 11] key exchange. We could then encrypt the traffic between the servers using this key. We could use Data Encryption Standard (DES) or triple-DES for encrypting the traffic. In the event that we plan to use triple DES we would need to exchange at least two keys securely.

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

5.3.4 Communication between server and clients

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

6 Conclusions

In this paper, we have described the Grid Event Service (GES), a distributed event service designed to run on a very large network of server nodes. GES comprises of a suite of protocols, which are responsible for the organization of nodes, creation of abbreviated *system views*, management of profiles and the hierarchical dissemination of content based on these profiles. Creating small world networks, using the node organization protocol ensures that the pathlengths would only increase logarithmically with geometric increases in the size of the server network. The feature of having multiple links between two units/super-units ensures a greater degree of fault tolerance. The protocols in GES ensure that the routing is intelligent and can handle sparse/dense interest in certain sections of the system. GES's ability to handle the complete spectrum of interests equally well, lends itself as a very scalable solution under conditions of varying publish rates, matching rates and message sizes.

The paper outlined a scheme for the delivery of events in the presence of server node failures. In our scheme a unit could fail and remain failed forever. The only requirement that we impose is that if a stable storage fails, it should recover within a finite amount of time. The replication strategy, that we adopted allows us to add stable storages and also to withstand stable storage failures. The replication strategy, epochs associated with received events and profile ID's associated with client profiles allowed us to account for a very precise recovery of events for clients with prolonged disconnects or those which have roamed the network.

GES could be extended very easily to support dynamic topologies. Based on the concentration of clients at specific locations, bandwidth utilization can be optimized with the creation of dynamic servers at some of the clients. This scheme fits very well with our failure model for system units, where they can remain failed forever. Detection schemes can be employed to detect slow nodes, which serve as performance bottlenecks. Forcing these affected nodes to fail then reconfigures the system. GES immediately employs newly added units in the routing of events and responds to unit failures equally well. Similarly links can be added and removed in a similar fashion to optimize the routing for events. The routing decisions at each server node are based on the current state of the system. GES thus provides the base infrastructures for dynamic topologies.

References

[1] 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.

[2] 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.

[3] 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.

[4] 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.

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

[6] 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.

[7] 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.

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

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

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

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

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

[13] 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.

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

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

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

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

[18] 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.

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

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

[21] Shrideep Pallickara and Geoffrey Fox. The grid event service (ges) framework: Research direction & issues. Technical report, IPCRES Grid Computing Laboratory, 2001.

[22] Shrideep Pallickara and Geoffrey Fox. Routing events in the grid event service. Technical report, IPCRES Grid Computing Laboratory, 2001.

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

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

[25] 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.

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

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

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