Routing events using the Grid Event Service

Shrideep Pallickara[*] Geoffrey C. Fox[†]

June 25, 2001

[*] Department of Electrical Engineering &Computer Science, Syracuse University
[†] Department of Computer Science, Florida State University

1 Introduction

Distributed messaging systems broadly fall into three different categories. Namely queuing systems, remote procedure call based systems and publish subscribe systems. Message queuing systems with their store-and-forward mechanisms come into play where the sender of the message expects someone to handle the message while imposing asynchronous communication and guaranteed delivery constraints. The two popular products in this area include IBM's MQSeries [23] and Microsoft's MSMQ [22]. MQSeries operates over a host of platforms and covers a much wider gamut of transport protocols (TCP, NETBIOS, SNA among others) while MSMQ is optimized for the Windows platform and operates over TCP and IPX. A widely used standard in messaging is the Message Passing Interface Standard (MPI) [16]. MPI is designed for high performance on both massively parallel machines and workstation clusters. Messaging systems based on the classical remote procedure calls include CORBA [30], Java RMI [26] and DCOM [15]. Publish subscribe systems form the third axis of messaging systems and allow for decoupled communication between clients issuing notifications and clients interested in these notifications.

The decoupling relaxes the constraint that publishers and subscribers be present at the same time, and also the constraint that they be aware of each other. The publisher is also unaware of the number of subscribers that are interested in receiving a message. The publish subscribe model does not require synchronization between publishers and subscribers. By decoupling this relationship between publishers and consumers, security is enhanced considerably. The routing of messages from the publisher to the subscriber is within the purview of the message oriented middleware (MOM) which is responsible for routing the right content to the right consumers. The publish subscribe paradigm can support both *pull* and *push* paradigms. In the case of pull, the subscribers retrieve messages from the MOM by periodic polling. The push model allows for asynchronous operations where there are no periodic pollings. Industrial strength products in the publish subscribe domain include solutions like *TIB/Rendezvous* [14] from TIBCO and *SmartSockets* [13] from Talarian. Variants of publish subscribe include systems based on content based publish subscribe. Content based systems allow subscribers to specify the kind of content that they are interested in. These content based publish subscribe systems include *Gryphon* [5, 2], *Elvin* [33] and *Sienna* [8]. The system we are looking at, the grid event service, is also in the realm of content based publish/subscribe systems with the additional feature of location transparency for clients.

The shift towards pub/sub systems and its advantages can be gauged by the fact that message queuing products like MQSeries have increased the publish subscribe features within them. This intersection of mature messaging products with pub/sub features serves its purpose for a large number of clients. Similarly OMG introduced services that are relevant to the publish subscribe paradigm. These include the Event services [29] and the Notification service [28]. The push by Java to include publish subscribe features into its messaging middleware include efforts like JMS [20] and JINI [3]. One of the goals of JMS is to offer a unified API across publish subscribe implementations. Various JMS implementations include solutions like *SonicMQ* [12] from Progress, *JMQ* [25] from iPlanet, *iBus* [24] from Softwired and *FioranoMQ* [11] from Fiorano.

In the systems we are studying, unlike traditional group multicast systems, *groups* cannot be preallocated. Each message is sent to the system as a whole and then delivered to a subset of recipients. The problem of reliable delivery and ordering[1] in traditional group based systems with process crashes has been extensively studied [19, 7, 6]. These approaches normally have employed the *primary partition* model [32], which allows the system to partition under the assumption that there would be a unique partition which could make decisions on behalf of the system as a whole, without risk of contradictions arising in the other partitions and also during partition mergers. However the delivery requirements are met only within the primary partition [18]. Recipients that are slow or temporarily disconnected may be treated as if they had left the group. This model works well for problems such as propagating

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

updates to replicated sites. This approach doesn't work well in situations where the client connectivity is intermittent, and where the clients can roam around the network. The main differences between the systems being discussed here and traditional group-based systems are:

1. We envision relatively large, widely distributed systems. A typical system would comprise of hundreds of thousands of server nodes, with tens of millions of clients.
2. Events are routed to clients based on their profiles, employing the group approach to routing the interesting events to the appropriate clients would entail an enormous number of groups - potentially 2^n groups for n clients. This number would be larger since a client profile comprises of interests in varying event footprints.

The approach adopted by the OMG [29, 28] is one of establishing channels and registering suppliers and consumers to those event channels. The event service [29] approach has a drawback in that it entails a large number of event channels which clients (consumers) need to be aware of. Also since all events sent to a specific event channel need to be routed to all consumers, a single client could register interest with multiple event channels. The aforementioned feature also forces a supplier to supply events to multiple event channels based on the routing needs of a certain event. On the fault tolerance aspect, there is a lack of transparency since channels could fail and issuing clients would receive exceptions. The most serious drawback in the event service is the lack of filtering mechanisms. These are sought to be addressed in the Notification Service [28] design. However the Notification service attempts to preserve all the semantics specified in the OMG event service, allowing for interoperability between Event service clients and Notification service clients. Thus even in this case the client needs to subscribe to more than one event channel.

2 The distributed model for the servers

One of the reasons why one would use a distributed model is high availability. Having a centralized model would imply a single server hosting multiple clients. While, this is 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 server nodes in the network. Solving issues of consistency while executing operations, in the presence of replication, leads to a model where other server nodes can service a client despite certain server node failures. The underlying network that we consider for our problem is one made up of the nodes that are hooked onto the Internet or Intranets. We assume that the nodes which participate in the event delivery can crash or be slow. Similarly the links connecting these node may fail or get overloaded. These assumptions are drawn based on real life experiences. One of the immediate implications of our delivery guarantees and the system behavior is that profiles are what become persistent, not the client connection or its active presence in the digital world at all times.

2.1 The Server Node Topology

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

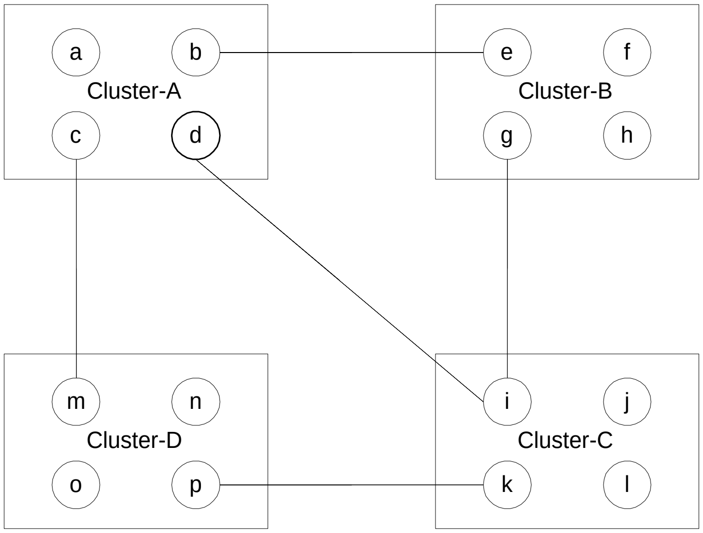


Figure 1: A Super Cluster - Cluster Connections

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

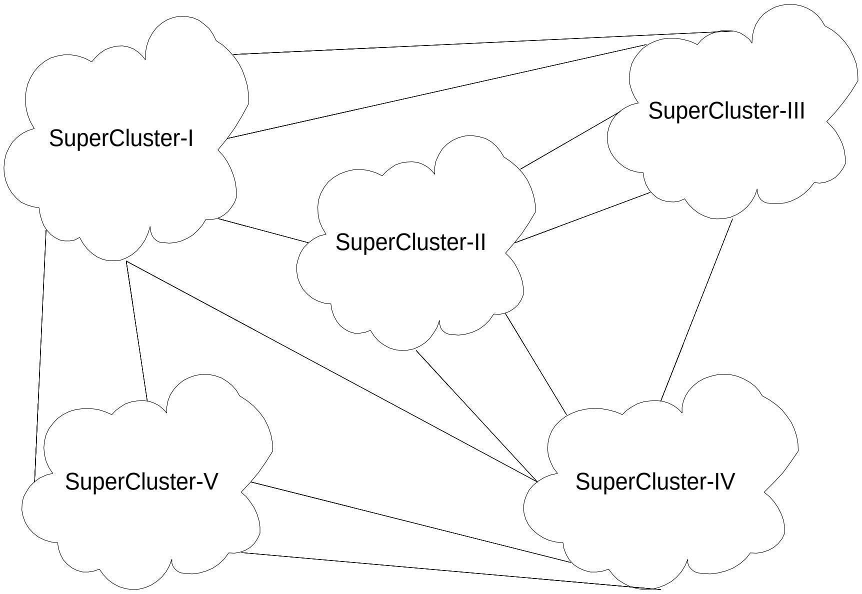


Figure 2: A Super-Super-Cluster - Super Cluster Connections

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

What we essentially have here is a set of strongly connected server nodes comprising a cluster and a set of links connecting a cluster to other clusters. We are interested in the delays that would be involved in connecting from one node in the network to another node in the network. This is proportional to the server node hops that need to be taken en route to the final destination.

We now delve into the *small world graphs* introduced in [35] and employed for the analysis of real world peer-to-peer systems in [31, 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 [35] 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 server node network, the pathlengths will be logarithmic too. Thus in the topology that we have the cluster controllers provide control to local classrooms etc, while the links provide us with *logarithmic* pathlengths and the multiple links, connecting clusters and the nodes within the clusters, provide us with robustness.

2.1.1 GES Contexts

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

$$\begin{aligned}
C_i^0 &= (C_j^1, i) && (1)\\
C_j^1 &= (C_k^2, j) && (2)\\
&\vdots\\
C_p^{N-2} &= (C^{N-1}, p) && (3)\\
C_q^{N-1} &= q && (4)
\end{aligned}$$

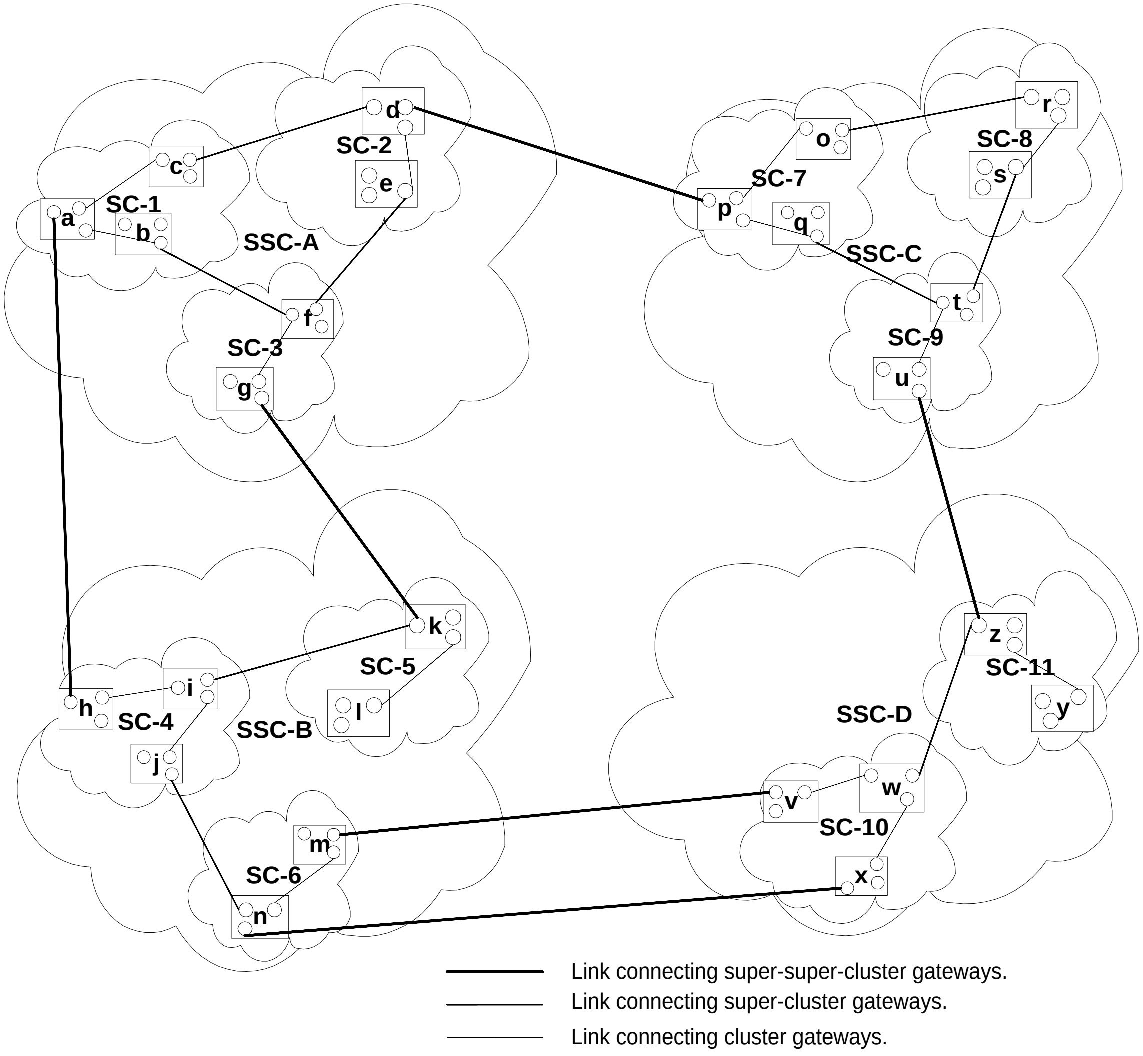
In an N-level system, a unit at level ℓ can be uniquely identified by $(N-\ell)$ GES context identifiers of each of the higher levels. Of course, the units at any level ℓ within a GES context $C_i^{\ell+1}$ should be able to reach any other unit within that same level. If this condition is not satisfied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

2.1.2 Gatekeepers

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

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

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



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

Figure 3: Gatekeepers and the organization of the system

It should be noted that a gatekeeper at level ℓ can be a gatekeeper at any other level. In fact, every node within the system is a gatekeeper at level-0. Figure 3 shows a system comprising of 78 nodes organized into a system of 4 super-super-clusters, 11 super-clusters and 26 clusters. When a node establishes a link to another node in some other cluster, it provides a gateway for the dissemination of events. If the node it connects to is in a different cluster within the same super-cluster GES context C_i^2 both the nodes are designated as cluster gatekeepers. In general, if a node connects to another node, and the nodes are such that they share the same GES context $C_i^{\ell+1}$ but have differing GES contexts C_j^{ℓ}, C_k^{ℓ}, the nodes are designated as gatekeepers at $level - \ell$ i.e. $g^{\ell}(C^{\ell+1})$. Thus, in figure 3 we have 12 super-super-cluster gatekeepers, 18 super-cluster gatekeepers (6 each in **SSC-A** and **SSC-C**, 4 in **SSC-B** and 2 in **SSC-D**) and 4 cluster-gatekeepers in super-cluster **SC-1**.

2.1.3 The addressing scheme

The addressing scheme provides us with a way to uniquely identify each server node within the system. This scheme plays a crucial role in the delivery and dissemination of events to nodes in the system. As discussed earlier, units at each level are defined within the GES context of a unit at the next higher level. In an N-level system the GES context C_j^ℓ is $C_i^\ell = \overbrace{C_j^N(C_k^{N-1}(\cdots(C_m^{\ell+1}(C_i^\ell))\cdots))}^{N-l}$. Thus in a 4-level system, to identify a server node, the addressing scheme specifies the super-super-cluster C_i^3, super-cluster C_j^2 and cluster C_k^1 that the node is a part of, along with the node-identifier within C_k^1. Thus for server node **a**, within cluster **B**, within super-cluster **C** and super-super-cluster **D** the logical address within the system is **D.C.B.a**. This addressing scheme is very similar to the IP addressing scheme.

3 The problem of event delivery

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. A particular event may thus be consumed by zero or more clients registered with the system. Events have explicit or implicit information pertaining to the clients, which are interested in (supposed to receive) 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. In the case of external destination lists, it is the system that computes the clients that should receive a certain 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. The problem of event delivery pertains to the efficient delivery of events to the destinations which could be internal or external to the event. In the latter case the system needs to compute the destination lists pertaining to the event. The system merely acts as a conduit to efficiently route the events from the issuing client to the interested clients. A simple approach would be to route all events to all clients, and have the clients discard the events that they are not interested in. This approach would however place a strain on network resources. Under conditions of high load and increasing selectivity by the clients, the number of events that a client discards would far exceed the number of events it is actually interested in. This scheme also affects the latency associated with the reception of real time events at the client. The increase in latency is due to the cumulation of queuing delays associated with the *uninteresting/flooded* events. The system thus needs to be very selective of the kinds of events that it routes to a client. In this section we describe a suite of protocols that are used to aid the process of efficient dissemination of events in the system.

In section 3.1 we describe the Node Addition Protocol (NAP), which provides for adding a server node or a complete unit to an existing system. The Gateway Propagation Protocol (GPP) discussed in Section 3.2 is responsible for the dissemination of connection information within relevant parts of the sub system to facilitate creation of abbreviated system interconnection graphs. Providing precise information for the routing of events, and the updating of this information in response to the addition, recovery and failure of gateways is in the purview of the GPP. To snapshot the event constraints that need to be satisfied by an event prior to dissemination within a unit and subsequent reception at a client we use the Profile Propagation Protocol (PPP) discussed in Section 3.3.5. PPP is responsible for the propagation of profile information to relevant nodes within the system to facilitate hierarchical dissemination of events. Section 3.4 describes the Event Routing Protocol (ERP) which uses the information provided by PPP to compute hierarchical destinations. Information provided by GPP, such as system inter-connections and shortest paths, are then employed to efficiently disseminate events within the units and to clients subsequently.

Different systems address the problem of event delivery to relevant clients in different ways. In [17] 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 [33] 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 [34]. In [8, 9] 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 [5] each server (broker) maintains a list of all subscriptions within the system in a parallel search tree (PST). The PST is annotated with a trit vector encoding link routing information. These annotations are then used at matching time by a server to determine which of its neighbors should receive that event. [4] describes approaches for exploiting group based multicast for event delivery. These approaches exploit universally available multicast techniques.

The approach adopted by the OMG [30] is one of establishing channels and registering suppliers and consumers to those event channels. The channel approach in the event service [29] 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 [28] design. However the Notification service attempts to preserve all the semantics specified in the OMG event service, allowing for interoperability between Event service clients and Notification service clients. Thus even in this case a client needs to subscribe to more than one event channel. In *TAO* [21], a real-time event service that extends the CORBA event service is available. This provides for rate-based event processing, and efficient filtering and correlation. However even in this case the drawback is the number of channels that a client needs to keep track of.

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

In the case of servers that route static content to clients such as Web pages, software downloads etc. some of these servers have their content mirrored on servers at different geographic locations. Clients then access one of these mirrored sites and retrieve information. This can lead to problems pertaining to bandwidth utilization and servicing of requests, if large concentrations of clients access the wrong mirrored-site. In an approach sometimes referred to as *active mirroring*, websites powered by *EdgeSuite* [10] 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.

3.1 The node organization protocol

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

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

3.1.1 Adding a new node to the system

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

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

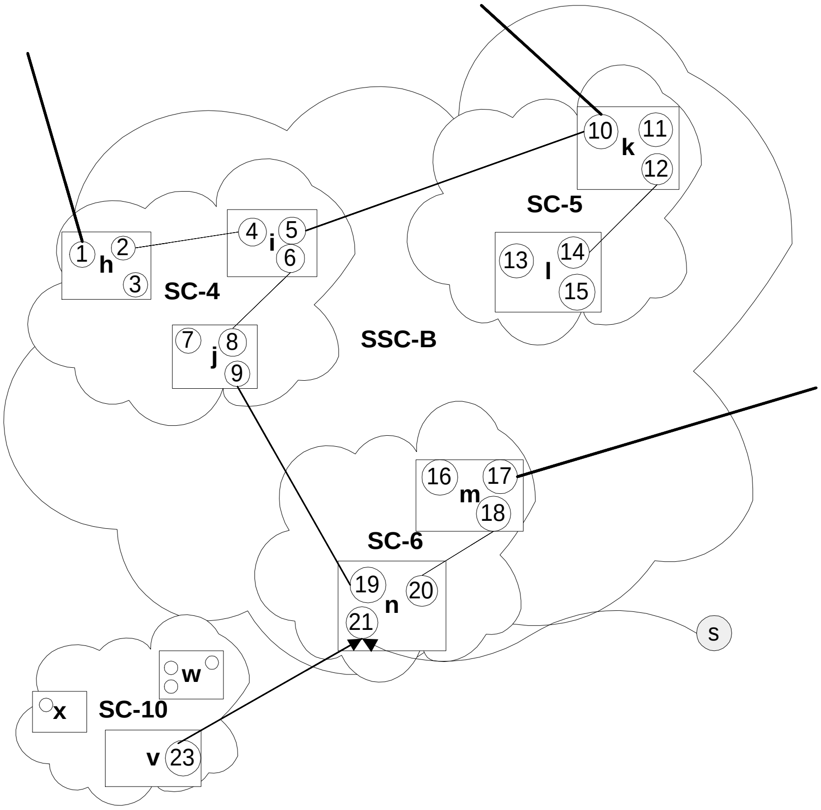


Figure 4: Adding nodes and units to an existing system

Figure 4 depicts a node **s** requesting a connection setup request. If **s** requests to be a level-0 node, then it needs to be part of the cluster **n**. Now, if node **n.21** has not exceeded the connection threshold limit for level-0 connections and also if the node **s** satisfies the IP-discrimination scheme for accepting nodes within the cluster then node **s** is configured as a level-0 node with a connection to node **n.21**. If however, node **n.21** has reached its connection threshold for level-0 connections, but node **s** has satisfied the IP-discrimination requirements for cluster **n**, then **n.21** forwards the request to other nodes within the cluster **n**. If there is a node within the cluster **n**, which has not reached the connection threshold limit, then node **s** is configured as a level-0 gateway to that node in cluster **n**. If however, all the nodes have reached their connection threshold limit, the node responds by providing a list of level-1 gatekeepers that are connected to cluster **n**. Node **s** then proceeds with the same process discussed earlier.

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

3.1.2 Adding a new unit to the system

A unit that can be added to the system could be a cluster, a super-cluster and so on. The process of adding a new unit to the system must follow rules which are consistent with the organization of the system. These rules are simple, a node can be a level-0 gatekeeper of only one cluster. Thus a node in an existing cluster cannot seek to be part of another cluster in the system. In general for a unit at level-ℓ which is being added to the system, any node in the unit being added cannot seek to be a level-$(\ell - i)$ (where $i = 1, 2, \cdots, \ell$) gatekeeper to any sub-system of the existing system.

The process of adding a unit to the system, results in the update of the GES contextual information pertaining to every node within the added unit. This update is only for the highest level of the system, lower level GES contextual information remains the same. Nodes within a cluster have a context with respect to the GES cluster context C_i^1. When this cluster is added to the system, what changes is the GES context C_i^1 while the individual GES contexts C^0 of the nodes with respect to newly assigned GES cluster context C_j^1 remains the same.

Figure 4 depicts the addition of a super cluster **SC-10** to the system. Only one node within the unit that needs to be added can issue the connection setup request. The node which issues this request in figure 4 is the node **SC-10.v.23**. Since this is a level-2 system that is *unit-added*, node **23** or any other node within **SC-10** can not be a level-1 (cluster) gateway to the other nodes within the super-super-cluster SSC-B. Node **23** thus issues a request specifying that it seeks to be a level-3 gateway within super-super-cluster **SSC-B**. Upon a successful connection set up, a new address is assigned for **SC-10** (say **SC-8**), the identifiers for clusters within **SC-10** remain the same. However, the complete address of these clusters change to **SSC-B.SC-8.w** and so on.

3.2 The gateway propagation protocol - GPP

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

3.2.1 Organization of gateways

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

What is required is an abstract notion of the connectivities that exist between various units (sub-units and super-units alike) within the system. This constitutes the *connectivity graph* of the system. At each node the connectivity graph is different while providing a consistent overall view of the system. The view that is provided by the connectivity graph at a node should be of the connectivities that are relevant to the node in question. Figure 5 depicts the connections that exist between various units of the 4 level system which we would use as an example in further discussions.

3.2.2 Constructing the connectivity graph

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

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

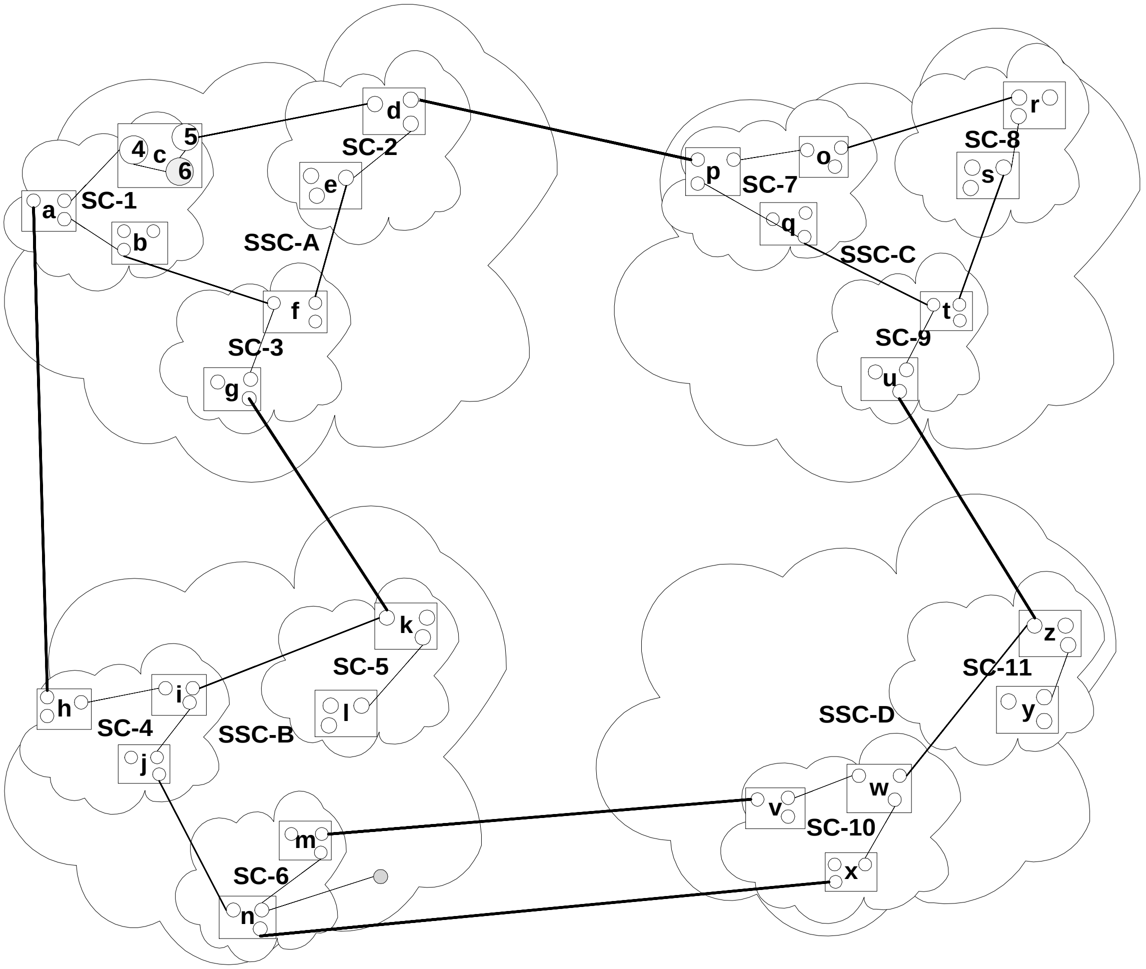


Figure 5: Connectivities between units

to a level-ℓ unit, by all the other clusters within the super cluster that the gatekeeper node is a part of.

3.2.3 The connection

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

A level$-\ell$ connection is the tuple $< n_i^x, n_j^y, \ell >$ where $x \mid y = \ell$ and $x, y \leq \ell$. Units u_i^x and u_j^y share the same level-$(\ell + 1)$ GES context $C_k^{\ell+1}$. For any given node n_i^ℓ in the connectivity graph we are interested only in the level $\ell, \ell + 1, \cdots, N$ connections that exist within the unit and not the $\ell - 1, \ell - 2, \cdots, 0$ connections that exist within that unit. Thus, if a level-ℓ connection is established, the connection information is disseminated only within the higher level GES context $C_i^{\ell+1}$ of the sub-system that the gatekeepers are a part of. This is ensured by never sending a level-ℓ gateway addition information across any gateway $g^{\ell+1}$. Thus, in figure 5 for a super-cluster gateway established within **SSC-A**, the connection information is disseminated only within the super-clusters **SC-1**, **SC-2** and **SC-3**, and subsequently the nodes in super-super-cluster **SSC-A**.

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

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

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

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

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

3.2.4 Link count

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

3.2.5 The link cost matrix

The link cost matrix specifies the cost associated with traversing a link. The cost associated with traversing a level-ℓ link from a unit u^x increases with increasing values of both x and ℓ. Thus the cost of communication between nodes within a cluster is the cheapest, and progressively increases as the level of the unit that it is connected to increases. The cost associated with communication between units at different levels increases as the levels of the units increases. One of the reasons why we have this cost scheme is that the dissemination scheme employed by the system is selective about the links employed for finer grained dissemination. In general a higher level gateway is more overloaded than a lower level gateway. Table 1 depicts the cost associated with communication between units at different levels.

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

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

Table 1: The Link Cost Matrix

3.2.6 Organizing the nodes

The connectivity graph is different at every node, while providing a consistent view of the connections that exist within the system. This section describes the organization of the information contained in connections (section 3.2.3) and super-imposing costs as specified by the link cost matrix (section 3.2.5) resulting in the creation of a weighted graph. The connectivity graph constructed at the node imposes directional constraints on *certain* edges in the graph.

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

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

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

Figure 6 depicts the connectivity graph that is constructed at the node SSC-A.SC-1.c.6 in figure 5. The set L_{UL} at the node **SC-3** in the figure comprises of node **SC-2** at level-2 and node **b** at level-1. The set L_D at **SC-3** comprises of the node **SSC-B** at level-3. The cost associated with traversal over a level-3 gateway between a level-2 unit **b** and a level-3 unit **SC-3** as computed from the linkcost matrix is 3, and is the weight of the connection edge. There are two connections between the super-super-cluster units **SSC-B** and **SSC-D**, this is reflected in the link count associated with the edge connecting the corresponding graph nodes. The directional issues associated with certain edges are imposed by the algorithm for computing the shortest path to reach a node.

3.2.7 Computing the shortest path

To reach the vertex from any given node, a set of links need to be traversed. This set of links constitutes a *path* to the vertex node. In the connectivity graph, the best hop to take to reach a certain unit is computed based on the shortest path that exists between the unit and the vertex. This process of calculating the shortest path, from the node to the vertex, starts at the node in question. The directional

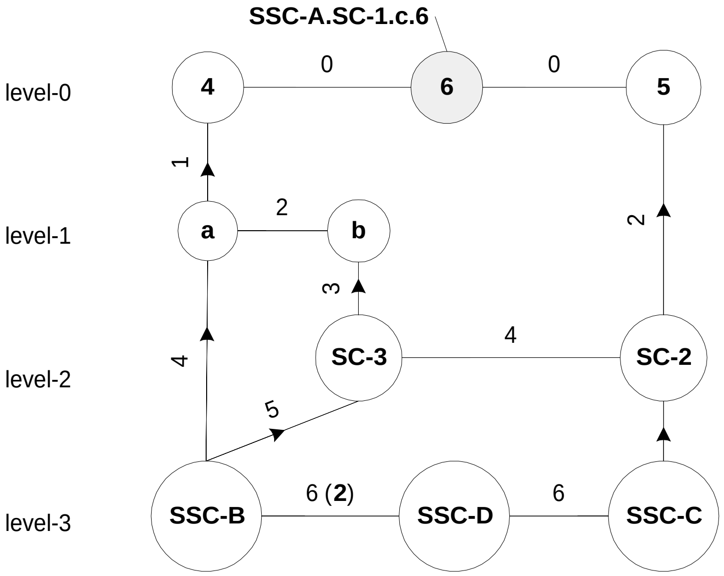


Figure 6: The connectivity graph at node 6.

arrows indicate the links, which comprise a valid path from the node in question to the vertex node. Edges with no imposed directional constraints are bi-directional. For any given node, the only links that come into the picture for computing the shortest path are those that are in the set L_{UL} associated with any of the nodes in a valid path.

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

(a) It allows us to determine if a recursive fork needs to be sent along a certain edge. If we do not keep track of the nodes that were visited, we could end up in an infinite recursion where we revisit the same node over and over again.

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

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

3.2.8 Building and updating the routing cache

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

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

3.2.9 Exchanging information between super-units

When a subsystem u_i^{ℓ} is added to an existing system $u^{\ell+j+1}$; information regarding $g^{\ell+j}, g^{\ell+j-1}, \cdots, g^{\ell}$ connections are exchanged between the system and the newly added sub system. Thus when a super cluster is added to an existing system comprising of super-super-clusters, the existing system routes information regarding super-cluster and super-super-cluster connections to the newly added super-cluster. The way the set of connections, comprising the connectivity graph, is sent over the newly established link is consistent with the rules, which we had set up for sending a connection information over a gateway as discussed in section 3.2.3. Thus, if a new super cluster **SC-4** is added to the **SSC-A** sub-system and a super cluster gateway is established between **SC-4** and node **SC-1.c.6**, then, the connectivity graphs at node **6** would be as depicted in figure 7.(a) while the connectivity graph at the gatekeeper in **SC-4** would comprise of the connections that were sent over the newly established gateway by node **6**.

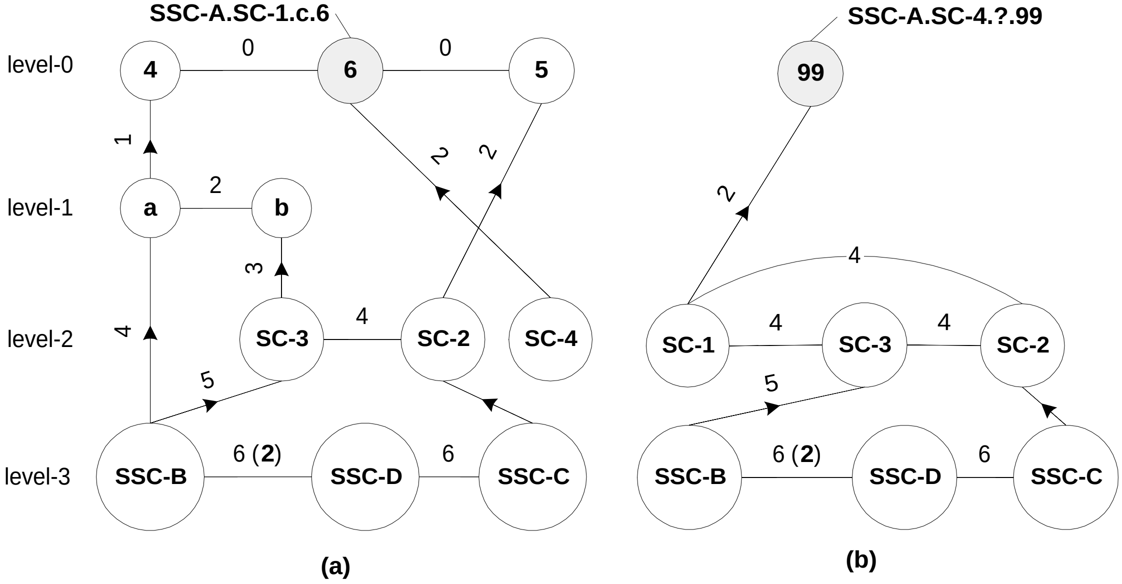


Figure 7: Connectivity graphs after the addition of a new super cluster SC-4.

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

3.3 Organization of Profiles and the calculation of destinations

Every event conforms to a signature which comprises of an ordered set of attributes $\{a_1, a_2, \cdots, a_n\}$. The values these attributes can take are dictated and constrained by the *type* of the attribute. Clients within the system that issues these events, assign values to these attributes. The values these attributes take comprise the content of the event. All clients are not interested in all the content, and are allowed to specify a filter on the content that is being disseminated within the system. Thus a filter allows a client to register its interest in a certain type of content. Of course one can employ multiple filters to signify interest in different types of content. These filters specified by the client constitutes its *profile*. The organization of these profiles, dictates the efficiency of matching content. A level-ℓ gatekeeper snapshots the profiles of all the level-(ℓ-1) units that share the same GES context C_i^ℓ with it.

3.3.1 The problem of computing destinations

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

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

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

(a) Organization of profiles in a profile graph

(b) Propagation of profiles to the nodes that are responsible for the calculation of hierarchical destination lists.

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

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

3.3.2 Constructing a profile graph

As mentioned earlier, events encapsulate content in an ordered set of $<$ *attribute*, *value* $>$ tuples. The constraints specified in the profiles should maintain this order contained within the event's signature. Thus to specify a constraint on the second attribute (a_2) a constraint should have been specified on the first attribute (a_1). What we mean by constraints, is the specification of the value that a particular attribute can take. We however also allow for the weakest constraint, denoted $*$, on any of the attributes. The $*$ signifies that the filtered events can take any of the valid values within the range permitted by the attribute's type. By successively specifying constraints on the event's attributes, a client narrows the content type that it is interested in. It is not necessary that a constraint be specified on all the attributes $\{a_1, a_2, \cdots, a_n\}$. What is necessary is that if a constraint is imposed on an attribute a_i constraints for attributes $a_1, a_2, \cdots, a_{i-1}$ must be in place, even if some or all of these constraints is the weakest constraint $*$. Thus if a constraint is specified till attribute a_i and no constraints are imposed on some of the attributes $a_1, a_2, \cdots, a_{i-1}$, the system assigns these attributes the weakest constraint $*$. It makes more sense imposing the constraint $*$ on higher order attributes $a_{i+1} \cdots a_n$ than on the lower order attributes $a_1, a_2, \cdots a_{i-1}$. Such a scheme has the effect of narrowing content down to the ones which are very closely related to each other. For every event type we maintain a profile chain. Different profile chains when added up constitute the profile graph.

We use the general matching algorithm, presented in [2], of the Gryphon system to organize profiles and compute the destinations associated with the events. Constraints from multiple profiles are organized in the *profile graph*. Every attribute on which a constraint is specified constitutes a node in the profile graph. When a constraint is specified on an attribute a_i, the attributes $a_1, a_2, \cdots, a_{i-1}$ appear in the profile graph. A profile comprises of constraints on successive attributes in an event's signature. The nodes in the profile graph are linked in the order that the constraints have been specified. Any two successive constraints in a profile result in an edge connecting the nodes in the profile graph. Depending on the kinds of profiles that have been specified by clients, there could be multiple edges, *originating from* a node. Following the scheme in [2] we do not allow multiple edges *terminating at* a node since it results in a situation where the event matching results in an invalid destination, due to that event having satisfied partial constraints of different profiles from within the same unit.

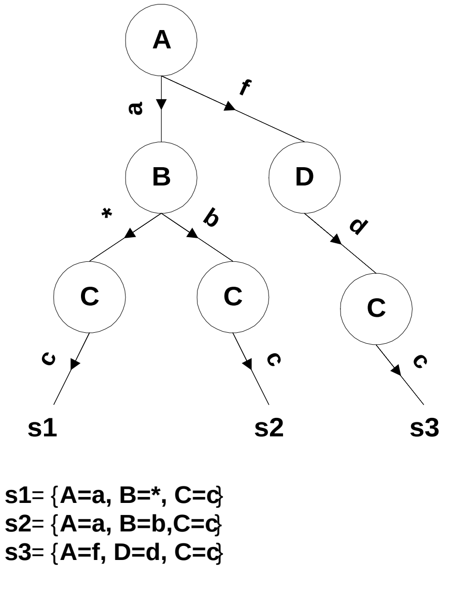


Figure 8: The profile graph - An example.

Figure 8 depicts the profile graph constructed from three different profiles. The example depicts how some of the profiles share partial constraints between them, some of which result in profiles sharing edges in the profile graph. A certain edge is marked as traversed by an event if the two successive constraints that created the edge, have been satisfied by that event. The presence of an edge signifies the existence of at least one client, which is interested in the content satisfying at least two of the constraints contained in that edge. An event's traversal along an edge simply indicates that the event's content has satisfied some partial constraint imposed by one or more of the clients. As we traverse further down the profile chain, the events we are looking for get more fine grained. The final constraint on an attribute leads to the creation of a destination edge. The edges arising out of node **C** in figure 8 are destination edges.

3.3.3 Information along the edges

To support hierarchical disseminations and also to keep track of the addition and removal of edges, besides the basic organization of constraints, the graph needs to maintain additional information along its edges. This additional information also plays a very important role in the reliable delivery of events to clients (we discuss this in a later section). Along every edge we maintain information regarding the units that are interested in its traversal. For each of these units we also maintain the number of predicates $\delta\omega$ within that unit that are interested in the traversal of that edge. The first time an edge is created between two constraints as a result of the profile specified by a unit, we add the unit to the route information maintained along the edge. For a new profile ω_{new} added by a unit, if two of its successive constraints already exist in the profile graph, we simply add the unit to the existing routing information associated with the edge. If the unit already exists in the routing information, we increment the predicate count associated with that destination.

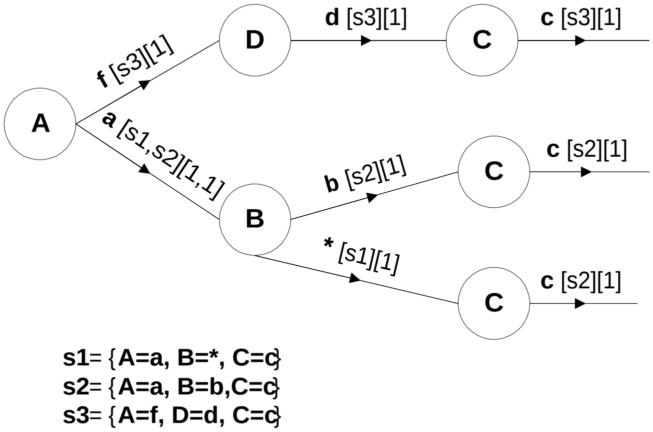


Figure 9: The complete profile graph with information along edges.

The information regarding the number of predicates $\delta\omega$ per unit that are interested in two successive constraints allows us to remove certain edges and nodes from the profile graph, when no clients are interested in the constraints any more. Figure 9 provides a simple example of the information maintained along the edges. We discuss how the profiles are propagated, where they are propagated and how this information along the edges is modified and updated in section 3.3.5.

3.3.4 Computing destinations from the profile graph

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

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

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

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

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

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

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

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

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

When we send the profile graph information over to the higher level gatekeeper g^{ℓ}, the information contained along the edges in the graph needs to be updated to reflect the nodes logical address at that level. Thus when a node propagates the clients profile to the cluster gatekeeper, it propagates the edges created/removed with the server as the destination associated with the profile predicate. Similarly, when this is being propagated to a super-cluster gatekeeper the profile change is sent across as a profile change in the cluster. Any change in the client's profile is propagated to gatekeepers at higher levels, that the server node in its abbreviated view of the system is aware of. What we are trying to do is to maintain information in the profile graph, in a manner which is consistent with the dissemination constraints imposed by properties (a) and (b). The reason we maintain destination information the way we do is that this model ties in very well with our topology and the routing algorithms that are in place. The connectivity graph provides us with an overall view of the interconnections between units at different levels. The organization and calculation of destinations from the profiles comprising the profile graph, feeds right into our routing algorithms that compute the shortest path to reach the units (destinations) where an event needs to be routed. In general for a level-N system, if there is a subscribing client with GES context C_j^N and the issuing client has GES context C_i^N the destinations are computed (N+1) times. Thus, in a system comprising of super-super-clusters, the destinations are computed four times prior to reception at the client.

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

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

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

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

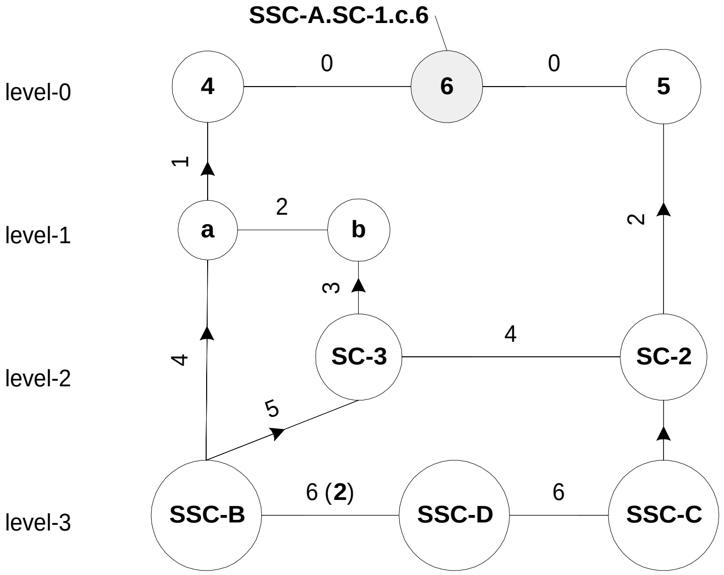


Figure 10: The connectivity graph at node 6.

In the figure 10, any $\delta\omega^0$ changes at any of the nodes within cluster **c**, are need to be routed to node **4**. Any $\delta\omega^1$ changes at node **4** need to be routed to node **5**, and also to a node in cluster **b**. Similarly $\delta\omega^2$ changes at node **5** needs to be routed to the level-3 gatekeeper in cluster **a** and superclusters **SC-3**, **SC-2**. When such propagations reach any unit/super-unit the process is repeated till such time that the gateway that the node seeks to reach is reached. Every profile change has a unique-id associated it, which aids in ensuring that the *reference count scheme* does not fail due to delivery of the same profile change multiple times within the same unit.

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

3.3.6 Unit additions and the propagation of profiles

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

3.3.7 Active profiles

The profile propagation protocol aids in the creation of destination lists at units within different levels. These destination lists are then employed at each level for finer grained disseminations. Since the profile add/change propagates through the system to higher level gateways, it is possible that a gateway at a higher level has not yet been notified about the profile add/change. Thus though it may receive an event which would match the profile change, the destination list may not include the lower level unit. It is possible that a client may receive events issued by clients within a certain unit, though it may not receive similar events from clients published by units within a different GES context.

What interests us is the precise instant of time from which point on we can say that all events that satisfy the client's profile will be delivered to the client. To address this issue we introduce the concept of *active profiles*, which provides guarantees in the routing of events within a unit. The active profile approach provides us with a unit-based incremental approach towards achieving system guarantees during a profile add/change. If a profile is *super-cluster active* all events issued by clients attached to any of the server nodes within a super-cluster C_i^2 will be routed to the interested client. Thus the first event that is received by the client is an indication that all subsequent events routed to that unit, matching the same profile would also be received by the client. When we say that a profile is *unit-active*[2] what we mean is that for every event that is being routed within that unit the destination lists calculated would include information to facilitate routing to the client. Since a client profile is unit active, all events, issued within the unit, will be routed to the client if it satisfies the client profile.

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

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

by the client is said to be *system active*. When a profile change initiated by a client is system active, events issued by any other clients within the system will be routed to this client, if those events match the system active profile change that was initiated by this client.

3.4 The event routing protocol - ERP

Event routing is the process of disseminating events to relevant clients. This includes matching the content, computing the destinations and routing the content along to its relevant destinations by determining the next node that the event must be relayed to. Every event has routing information associated with it, which could be used by the system to determine the route the event would take next. This routing information is not added by the client issuing this event but by the system to ensure faster dissemination and recovery from failures. When an event is first issued by the client, the server node that the client is attached to adds the routing information to the event. This routing information is the GES contextual information (see Section 2.1.1) pertaining to this particular node in the system. As the event flows through the system, via gateways the routing information is modified to snapshot its dissemination within the system. This information is then used to avoid routing the event to the same unit twice. What a node also needs to decide is when it would be futile to try and find a higher order gateway, and also when all the higher level units that could possibly be covered have been covered. Of course it should also know if there is a higher order gateway that needs to be reached. This decision is based on the event routing information and the information pertaining to gateways that's available at a node. If there are no such units that need to be reached, the event routing would proceed with lower order disseminations. However if there is a unit that needs to be reached, gateways would have to be employed to reach this unit as fast as possible. The event routing information contained with an event simply indicates the units, which were present en route to reception at the node.

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

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

  There could be multiple links connecting a unit to some other unit. This scheme provides us with a greater degree of fault-tolerance. This also leads to the situation[3] where the event could be routed to the same unit over multiple links. In this case the duplicate detection algorithm detects this duplicate event and halts any further routing for this event.

- In case the gateway decides to send the event over the gateway, all routing information pertaining to lower level disseminations are stripped from the event routing information.

  This is because the routing information pertaining to the lower level disseminations is within the GES context of a specific level-ℓ unit and will not be valid within other level-ℓ units. Also, in

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

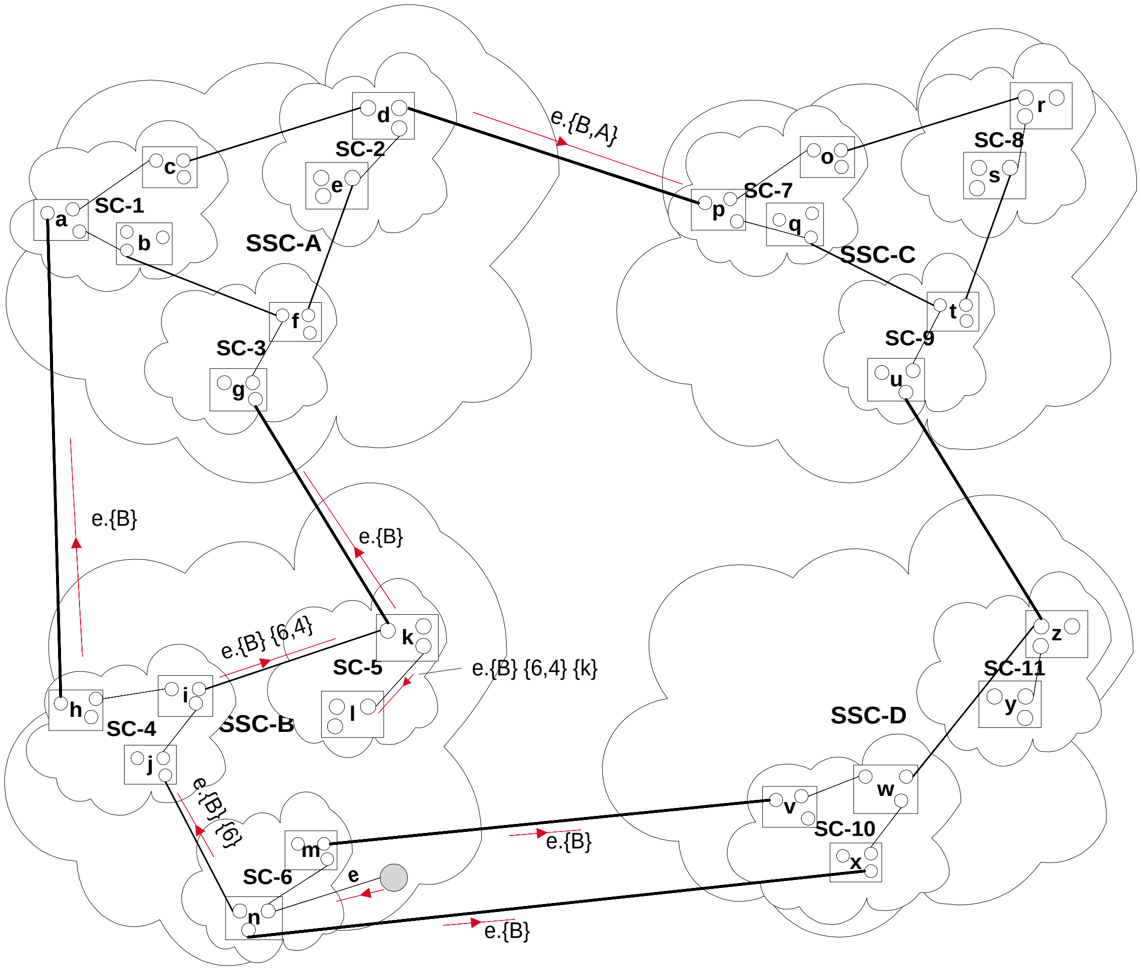


Figure 11: Routing events

general a higher order gateway would be more overloaded[4] compared to a lower order gateway. Reducing the amount of information being transferred over the gateway helps conserve bandwidth.

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

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

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

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

(b) Units which have already received the event.

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

When a gatekeeper g^ℓ with GES context C_i^ℓ is presented with an event it computes the $u^{\ell-1}$'s within C_i^ℓ that the event must be routed to. At every node the best hops to reach the destinations are computed. Thus, at every node the best decision is taken. Nodes and links that have not been failure suspected are the only entities that can be part of the shortest path. The event routing protocol, along with the profile propagation protocol and the gateway information ensure the optimal routing scheme for the dissemination of events in the existing topology.

3.5 Routing real-time events

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

3.5.1 Events with External Destination lists

When an event arrives at a gatekeeper g^ℓ, the gatekeeper checks to see if the event satisfies its profile. The profile maintained at g^ℓ snapshots the profile of the level-ℓ unit that the gatekeeper belongs to. This check is necessary to confirm if the event needs to be disseminated within the level-ℓ unit. Routing events based on the gatekeeper profile is the process which calculates the destination lists. This is a recursive process in which each higher order gatekeeper performs this check before disseminating the event to lower order gatekeepers.

When an event doesn't match the gatekeeper g^ℓ's profile, g^ℓ decides upon the next route that event would take based on the routing information encoded into the event by the event routing protocol.

- The gatekeeper $g_j^\ell(C_i^{\ell+1})$ checks the routing information provided by ERP to see if it needs to relay the event to other gatekeepers g^ℓ within the GES context $C_i^{\ell+1}$.
- The gatekeeper also uses the information provided by ERP to check if it could route the event to a higher order gateway which has not received the event.

In the event that these steps lead to no actions on part of the gatekeeper g^ℓ the gatekeeper takes no further actions to route this event. If the gatekeeper decides to route this event to other level-ℓ and higher order gatekeepers, the system can employ lower order gateways within the GES context $C_i^{\ell+1}$ to relay this event.

3.5.2 Events with Internal Destination lists

These are events which require the system to be able to route the event to a specific client in the system. Clients which are interested in receiving point-to-point events thus need to include their identifier in their profile. The sequence of steps that are needed to route the event are similar to the steps we take to route events with external destination lists as discussed in section 3.5.1.

3.6 Unique Events - Generation of unique identifiers

Associated with every event e sent by client nodes in the system is an event-ID, denoted *e.id*, which uniquely determines the event e, from any other event e' in the system. These ID's thus have the requirement that they be unique in both space and time. Clients in the system are assigned Ids, ClientID, based on the type of information issued and other factors such as location, application domain etc. To sum it up clients use pre-assigned Ids while sending events. This reduces the uniqueness problem, alluded earlier to a point in space. The discussion further down implies that the problem has been reduced to this point in space.

Associating a timestamp, *e.timeStamp*, with every event e issued restricts the rate (for uniquely identifiable[5] events) of events sent by the client to one event per granularity of the clock of the underlying system. Resorting to sending events without a timestamp, but with increasing sequence numbers, $e.sequenceNumber$, being assigned to every sent event results in the ability to send events at a rate independent of the underlying clock. However, such an approach results in the following drawbacks

a) If the client node issues an infinite number of events, and also since the sequence numbers are monotonically increasing, the sequence number assigned to events could get arbitrarily large i.e. $e.sequenceNumber \rightarrow \infty$.

b) Also, if the client node were to recover from a crash failure it would need to issue events starting from the sequence number of the last event prior to the failure, since the event would be deemed a duplicate otherwise.

A combination of timestamp and sequence numbers solves these problems. The timestamp is calculated the first time a client node starts up, and is also calculated after sending a certain number of events $sequenceNumber.MAX$. In this case the maximum sending rate is related to both $sequenceNumber.MAX$ and the granularity of the clock of the underlying system. Thus the event ID comprises of a tuple of the following named data fields : $e.PubID$, *e.timeStamp* and $e.sequenceNumber$. Events issued with different times t1 and t2 indicate which event was issued earlier, for events with the same timestamp the greater the timestamp the later the event was issued.

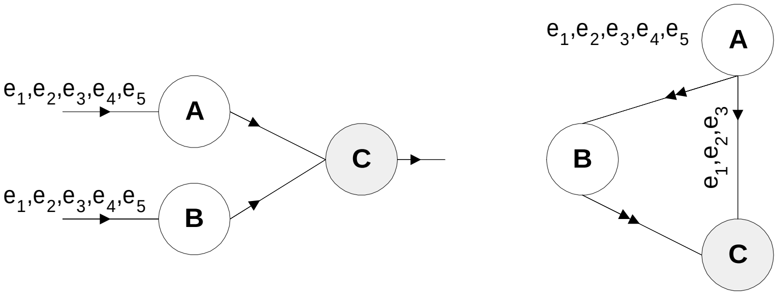
Systems such as *Gnutella* [1] propagate events through the network without duplication, using the IETF UUID [27] which gives a unique 128-bit identifier on demand. The authors guarantee the uniqueness until 3040 A.D. for the ID's generated using their algorithm. Such a scheme of unique ID's could also be very conveniently incorporated into the Grid Event Service for a unique identifier for every event.

3.7 Duplicate detection of events

Multiple copies of an event can exist in the system. This occurs due to multiple gateways existing between units and also due to events taking multiple routes to the reach destinations in response to

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

failure suspicions. Events need to be duplicate detected because for any event e that is a duplicate event, the path taken by the event as dictated by ERP is exactly the same as that taken by the event e which was previously received. In section 3.6 we discussed the generation of unique identifiers for events. This scheme of unique ID generation provides us with information pertaining to *unrelated events* (events issued by different clients) and in the case of *related events* (events issued by the same client) the order of their occurrence. In our scheme of duplicate event detection we use this unique ID generation as the basis for our duplicate event detection scheme.



(a)

(b)

Figure 12: Duplicate detection of events

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

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

Consider the case in figure 12.(a), at nodes A and B events e_1, e_2, e_3, e_4 and e_5 are all events issued by the same client. Node C maintains the last event that was received. The links we assume in the system are unreliable and unordered. Since these links allow the events to overtake each other, if node C receives e_3 first node C could errantly conclude that it had received e_1 and e_2. To resolve this we impose the requirement that the events be received in order (this is more so in the case of events issued by the same client), i.e. we do not let events overtake each other in the reception sequence at any node within the system.

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

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

Table 2: Reception of events at C

Consider the case in figure 12.(b), node A has sent events e_1, e_2 and e_3 over link l_{AC} at time t. At time $t + \delta$ node A suspects a node C failure which could either be due to an overcrowded link l_{AC} or a slow process at C. Now if A were to compute the alternate route to C that goes via B; if it doesn't send

e_1, e_2, e_3 prior to sending e_4 and e_5, the events e_1, e_2, e_3 would be duplicate detected if e_4 arrives before e_1. Once we make this minor change of resending unacknowledged events across the alternate route in response to suspicions it simply reduces to the case depicted in figure 12.(a). As an optimization feature we could also send *anti-events* down the failed/slow link whenever we resort to computing an alternate route.

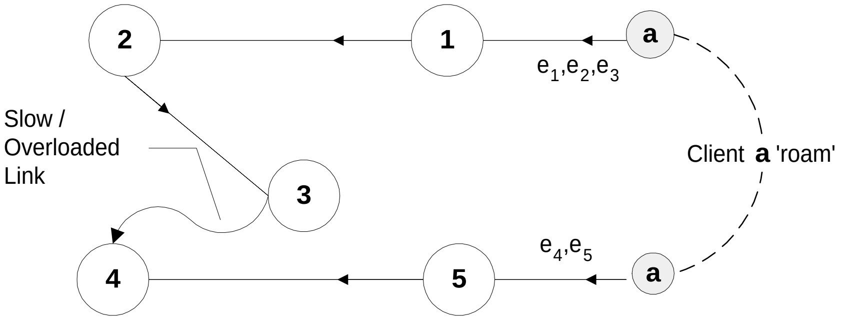


Figure 13: Duplicate detection of events during a client roam

Figure 13 depicts the scenario where a client roam could lead to duplicate detection of events which are not truly duplicate events. The case in which our duplicate detection scheme breaks down, is detailed in table 3. To account for such a scenario we include the incarnation number in our duplicate detection scheme. Incarnation numbers would be incremented for every roam and reconnection of the issuing client. The events would then be treated as events with a different *clientID* thus preventing the duplicate detection of events which should not have been duplicate detected in the first place.

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

Table 3: Reception of events at 4: Client roam

3.8 Interaction between the protocols and performance gains

In our system the node organization protocol could be used in the creation of *small world* [35, 31] networks. This organization, which comprises of strongly connected server nodes in clusters connected by *long links* ensures that the pathlength increases logarithmically for geometric increases in the size of the server 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 server 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 server 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 server network, when appropriately organized, ensures that the pathlengths for these disseminations would only increase logarithmically with the number of server nodes.

4 Results

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

4.1 Experimental Setup

The system comprises of 22 server node processes organized into the topology shown in the Figure 14. This set up is used so that the effects of queuing delays at higher publish rates, event sizes and matching rates are magnified.

Each server node process is hosted on 1 physical Sun SPARC Ultra-5 machine (128 MB RAM, 333 MHz), with no SPARC Ultra-5 machine hosting two or more server node processes. For the purpose of gathering performance numbers we have one publisher in the system and one *measuring subscriber* (the client where we do our measurements). The publisher and the *measuring* subscriber reside on the same SPARC Ultra-5 machine and are attached to nodes **22** and **10** respectively in the topology outlined in figure 14. In addition to this there are 100 subscribing client processes, with 5 client processes attached to every other server node (nodes **22** and **10** do not have any other clients besides the publisher and measuring subscriber respectively) within the system. The 100 client node processes all reside on a SPARC Ultra-60 (512 MB RAM, 360 MHz) machine. The publisher is responsible for issuing events, while the subscribers are responsible for registering their interest in receiving events. The run-time environment for all the server node and client processes is Solaris JVM (JDK 1.2.1, native threads, JIT).

4.2 Factors to be measured

Once the publisher starts issuing events the factor that we are most interested in is the *latency* in the reception of events. This latency corresponds to the response times experienced at each of the clients. We measure the latencies at the client under varying conditions of *publish rates*, *event sizes* and *matching rates*. Publish rate corresponds to the rate at which events are being issued by the publisher. Event size corresponds to the size of the individual events being published by the publisher. Matching rate is the percentage of events that are actually supposed to be receieved at a client. In most publish subscribe systems, at any given time for a certain number of events being present in the system, any given client is generally interested in a very small subset of these events. Varying the matching rates allows us to simulate such a scenario, and perform measurements under conditions of varying selectivity. For a sample of events received at a client we calculate the *mean latency* for the sample of received events, the *variance* in the sample of these events and the *system throughput* measured in terms of the number of events received per second at the measuring subscriber. We also measure the highest and lowest event latencies within the sample of events that have been received. Another very important factor that needs to be measured is the change in latencies as the connectivity between the nodes in a server network is increased. This increase in connectivity has the effect of reducing the number of server hops that an event has to take prior to being received at a client. The effects of change in latencies with decreasing server hops is discussed in section 4.3.4.

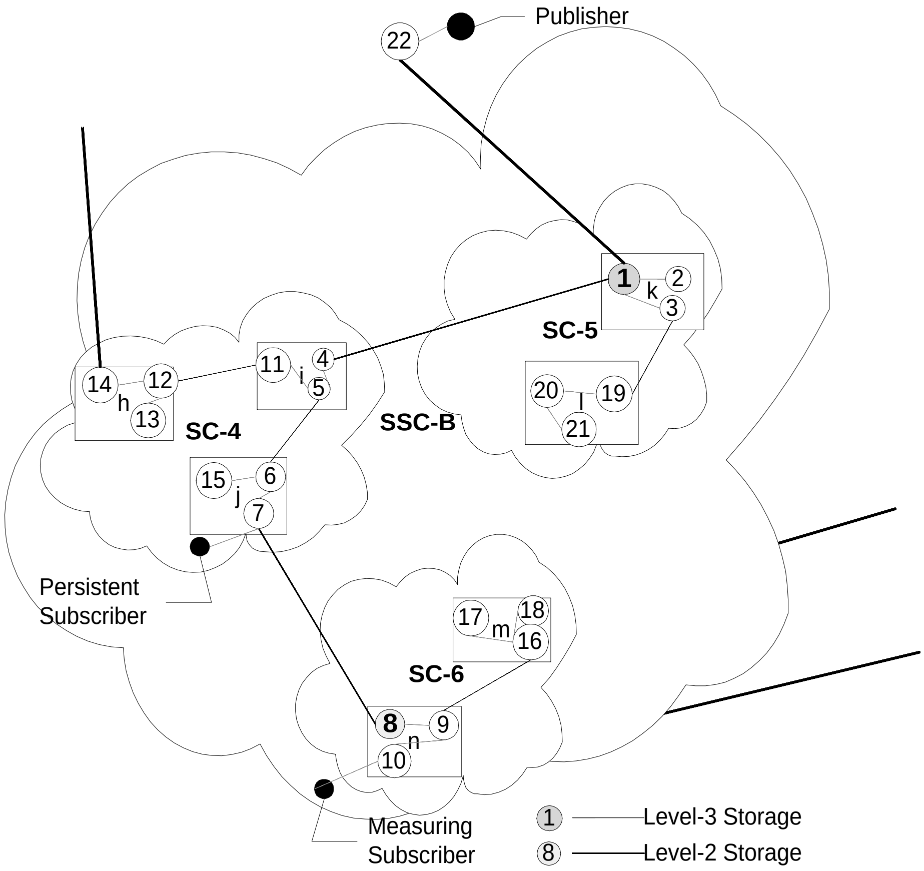


Figure 14: Testing Topology - (I)

4.2.1 Measuring the factors

For events published by the publisher the number of tag-value pairs contained in every event is 6, with the matching being determined by varying the value contained in the fourth tag. The profile for all the clients in the system, thus have their first 3 <*tag=value*> pairs identical to the first 3 pairs contained in every published event. This scheme also ensures that for every event for which destinations are being computed there is some amount of processing being done. Clients attached to different server nodes specify an interest in the type of events that they are interested in. This matching rate is controlled by the publisher, which publishes events with different footprints. Since we are aware of the footprints for the events published by the publisher, we can accordingly specify profiles, which will allow us to control the dissemination within the system. When we vary the matching rate we are varying the percentage of events published by the publisher that are actually being received by clients within the system. Thus, when we say that the matching rate is set at 50%, any given subscribing client within the system will receive only 50% of the events published by the publisher. To vary the publish rates, we control the *sleep time* associated with the publisher thread, and also the number of events that it publishes at a time, once the publisher thread *wakes up*. This requires some preliminary tuning. Once the values for the *sleep time* and the number of events that are published at a time have been fixed (for the publisher and the server node that it is attached to), we proceed to compute the real publish rates for the sample of events that we send. This is the publish rate that we report in our results.

For each matching rate we vary the size of the events from 30 to 500 bytes, and vary the publish rates at the publisher from 1 Event/Sec to around 1000 Events/second. For each of these cases we measure the latencies in the reception of events. To compute latencies we have the publishing client and the *measuring* subscriber residing on the same machine. Events issued by the publisher are *timestamped* and when they are received at the subscribing client the difference between the *present time* and the *timestamp* contained in the received event constitutes the latency in the dissemination of the event at the subscriber via the server network. In case the publisher and the subscriber are on two different machines, with access to different underlying system clocks, we would need to synchronize the clocks and also account for the drift in clock rates prior to computing the latencies in event reception. Having the publisher and one of the subscribers on the same physical machine with access to the same underlying clock, obviates this need for clock synchronization and also accounts for clock drifts. It should be noted that though the publisher and the *measuring* subscriber are on the same machine, they are connected to two different server nodes within the server network, as depicted in figure 14. In fact it takes 9 server hops for an event issued by the publisher to be received at the measuring subscriber.

4.3 Discussion of Results

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

4.3.1 Latencies for the routing of events to clients

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

Figures 15 through 18, depict the pattern of decreasing latencies with decreasing matching rates. The latencies vary from 391.85 *mSecs* to 52.0 *mSecs*, with the <*publish rate, event size*> varying from <952 *events/Sec* , 450 *Bytes*> for a matching rate of 100% to <952 *events/Sec*, 400 *Bytes*> for a matching rate of 10%. This reduction in the latencies for decreasing matching rates, is a result of the routing algorithms that we have in place. These routing algorithms ensure that events are routed only to those parts of the system where there are clients, which are interested in the receipt of those events. The routing algorithms are very selective about the links that are employed for event dissemination. Thus, events are queued only at those server nodes which –

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

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

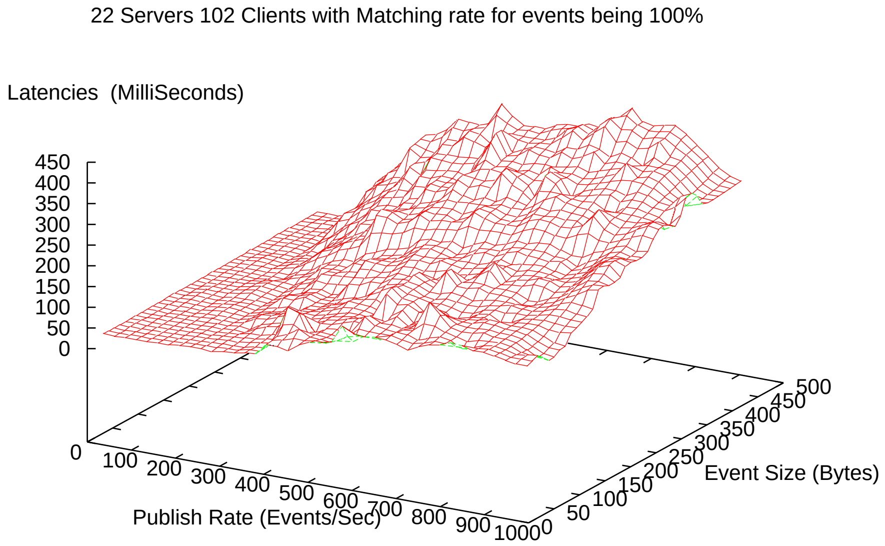


Figure 15: Match Rates of 100

Figure 15 depicts the case for matching rates of 100%. In this case the mean latency for the sample of events varies from 15.54 *mSec* for <1 *event/Sec*, 50 *Bytes*> at a throughput of 1 event/Sec to 391.85 *mSec* for <952 *events/Sec*, 450 *Bytes*> with a throughput of 78 *events/Sec* at the client. The variance in the sample of events varies from 2.3684 $mSec^2$ to 69,713.93 $mSec^2$ for the 2 cases respectively. The maximum throughput achieved was 480.76 *events/Sec* at publish rates of 492 *events/Sec* with events of size 75 bytes.

Figure 16 depicts the case for matching rates of 50%. In this case the mean latency for the sample of events varies from 13.02 *mSec* for <20 *events/Sec*, 50 *Bytes*> to 178.66 *mSec* for <952 *events/Sec*, 350 *Bytes*>. The variance in the sample of events varies from 56.8196 $mSec^2$ to 14,634 $mSec^2$ for the 2 cases respectively.

Figure 17 depicts the case for matching rates of 25%. In this case the mean latency for the sample of events varies from 14.40 *mSec* for <20 *events/Sec*, 50 *Bytes*> to 66.6 *mSec* for <961 *events/Sec*, 400 *Bytes*>. The variance in the sample of events varies from 0.24 $mSec^2$ to 587.04 $mSec^2$ for the 2 cases respectively.

Figure 18 depicts the case for matching rates of 10%. In this case the mean latency for the sample of events varies from 14.40 *mSec* for <20 *events/Sec*, 50 *Bytes*> to 52.0 *mSec* for <952 *events/Sec*, 400 *Bytes*>. The variance in the sample of events varies from 0.44 $mSec^2$ to 103 $mSec^2$ for the 2 cases respectively.

4.3.2 System Throughput

We also depict the system throughputs at the client under conditions of varying event sizes and publish rates. We choose to depict the system throughputs at a matching rate of 100%. At matching rates other

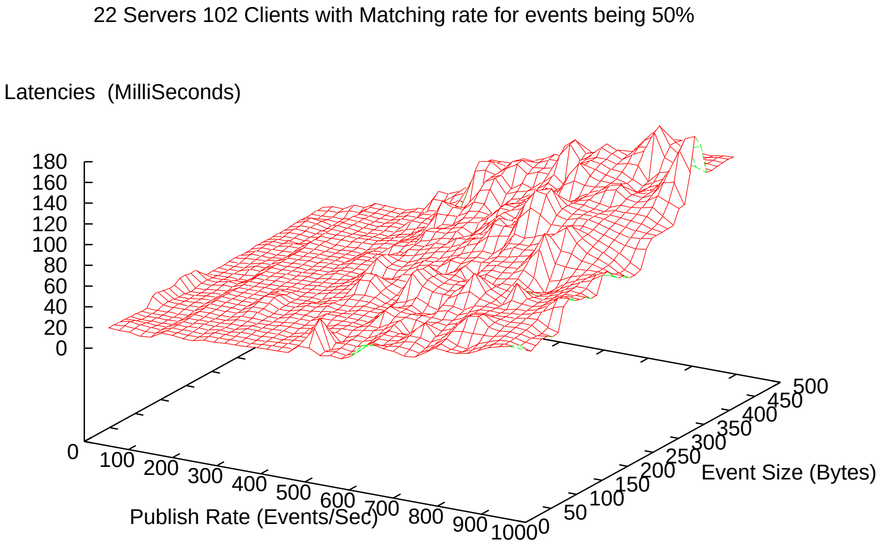


Figure 16: Match Rates of 50

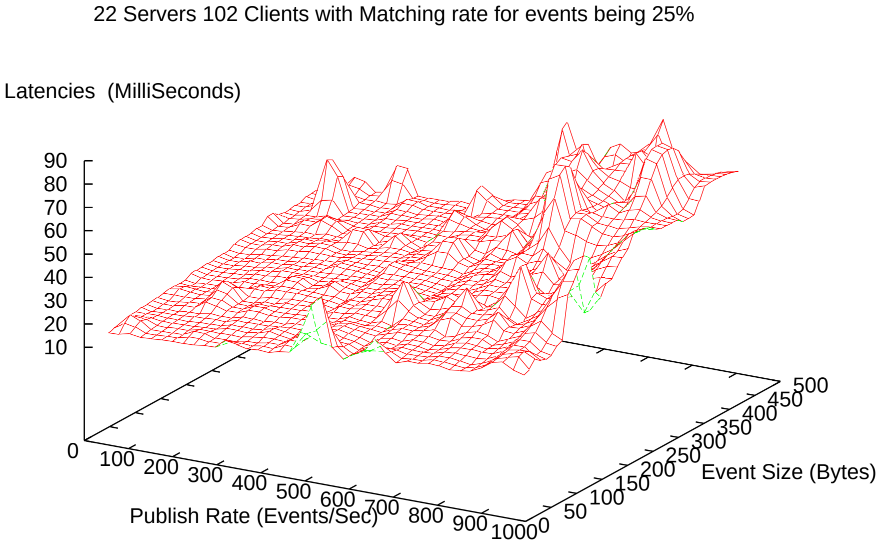


Figure 17: Match Rates of 25

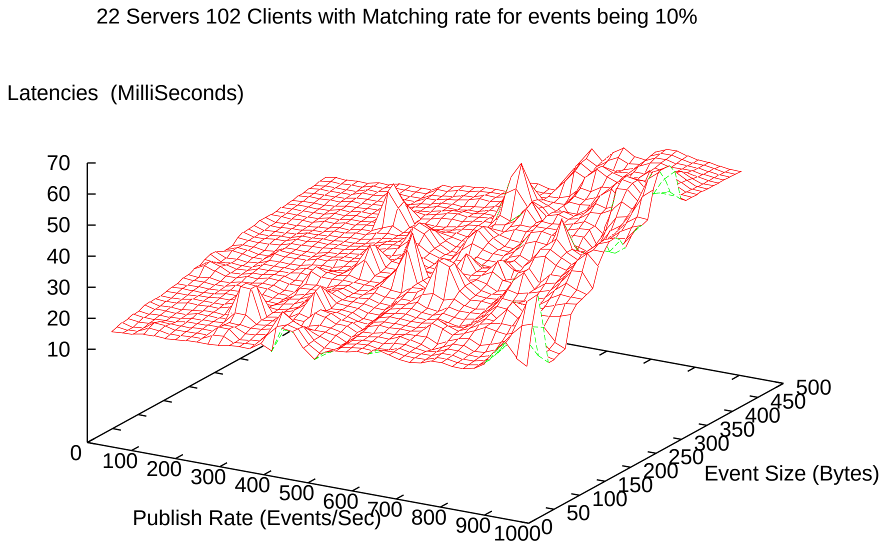


Figure 18: Match Rates of 10

than 100% only the relevant events are being routed to the clients. The events received do not reveal the true throughputs that can be achieved at a client. Figure 19 depicts the system throughputs achieved at a client under conditions of different publish rates and event sizes. The maximum throughput achieved was 480.76 *events/Sec* at a publish rate of 492 *events/Sec* with the sample of events being of size 75 bytes.

4.3.3 Variance

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

4.3.4 Pathlengths and Latencies

The topology in figure 14 allows us to magnify the latencies, which occur by having the queuing delays at individual server hops add up. In that topology the number of server hops taken by an event prior to delivery at the measuring subscriber is 9. We now proceed with testing for the topology outlined in figure 20. The layout of the server nodes is essentially identical to the earlier one, with the addition of links between nodes resulting in a strongly connected network. We have 5 subscribing clients at each of the server nodes. The mapping of server nodes and subscribing client nodes to the physical machines is also identical to the earlier topology. As can be seen the addition of super-cluster link between super-clusters

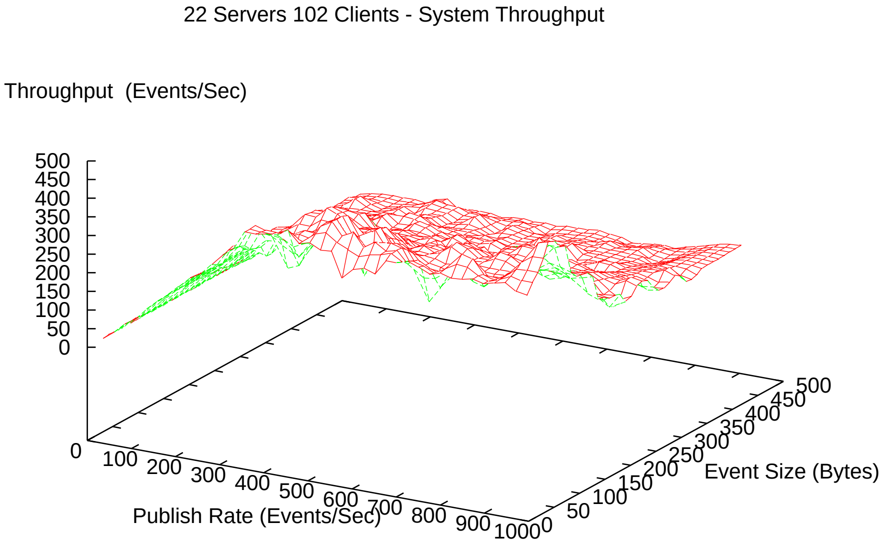


Figure 19: System Throughput

SC-5 and **SC-6**, and level-0 links between nodes **8** and **10** in cluster **SC-6.n** reduces the number of server hops, for the shortest path from the publisher to the measuring subscriber at node **10**, from 9 to 4.

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

In general, as the number of server hops reduce the latencies also reduce. The patterns for changes in latency as the event size and publish rates increase is however similar to our earlier cases. We depict our results, gathered at the three measuring subscribers for matching rates of 50% and 10%. The pattern of decreasing latencies with a decrease in the number of server hops is clear by looking at figures 21 through 26. We had also made measurements for a matching rate of 25%, and the pattern is the same in those results too. However, we have not included the figures for that case.

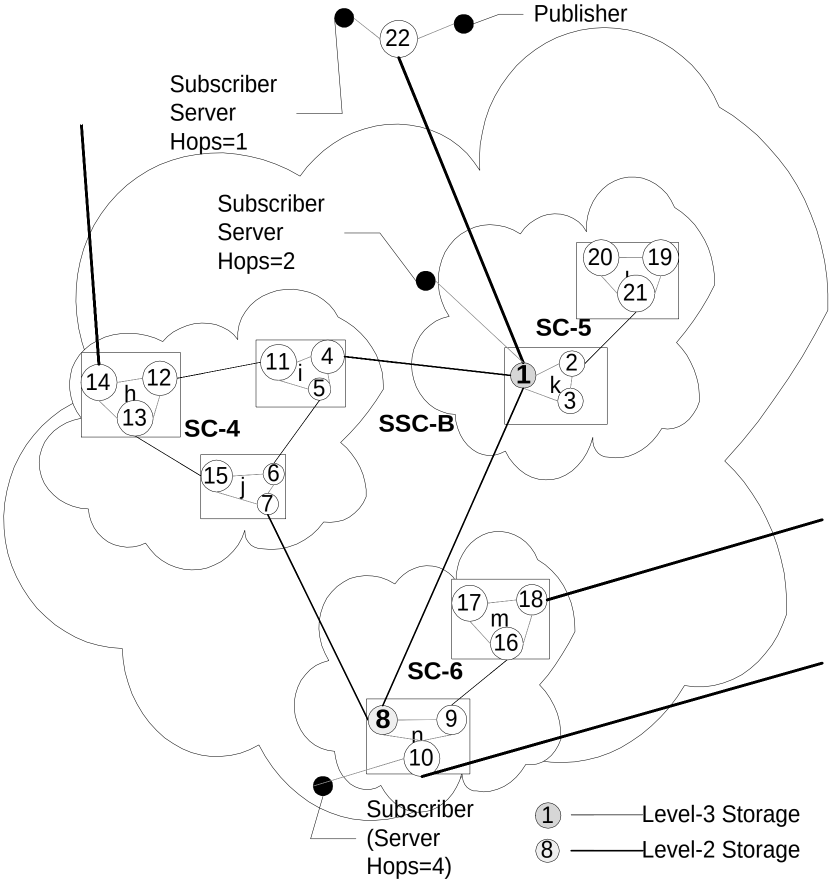


Figure 20: Testing Topology - Latencies versus server hops

4.4 Summary of results

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

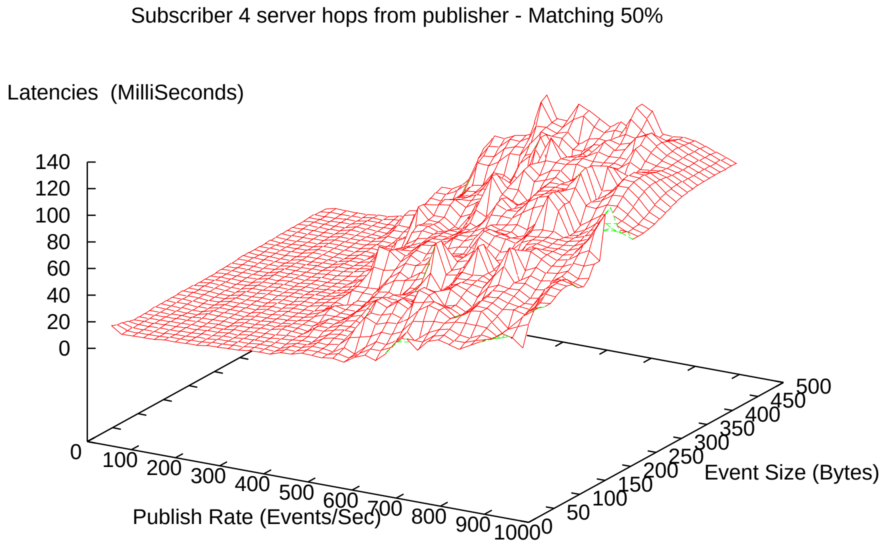


Figure 21: Match Rates of 50% - Server Hop of 4

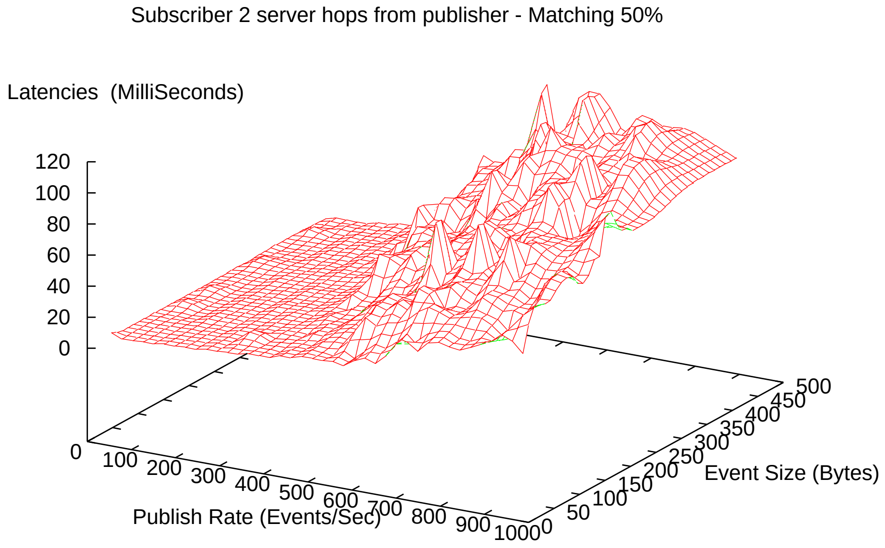


Figure 22: Match Rates of 50% - Server Hop of 2

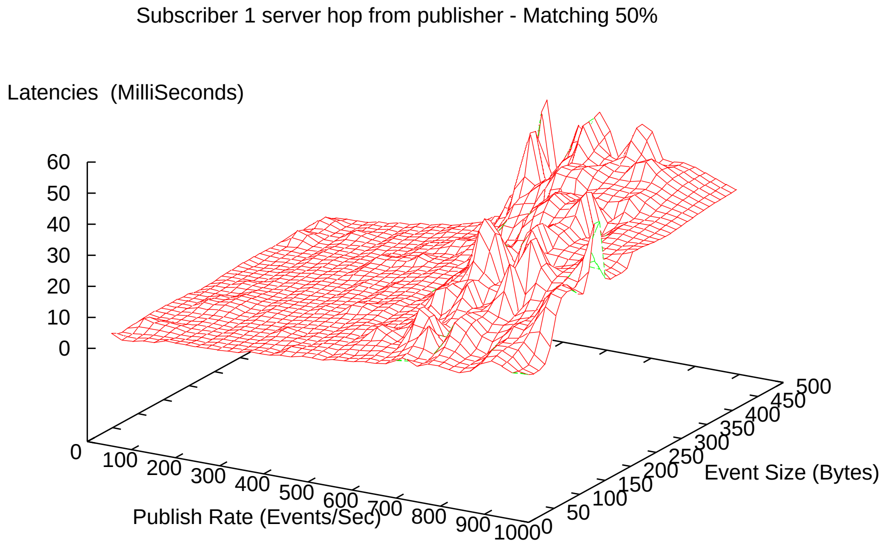


Figure 23: Match Rates of 50% - Server Hop of 1

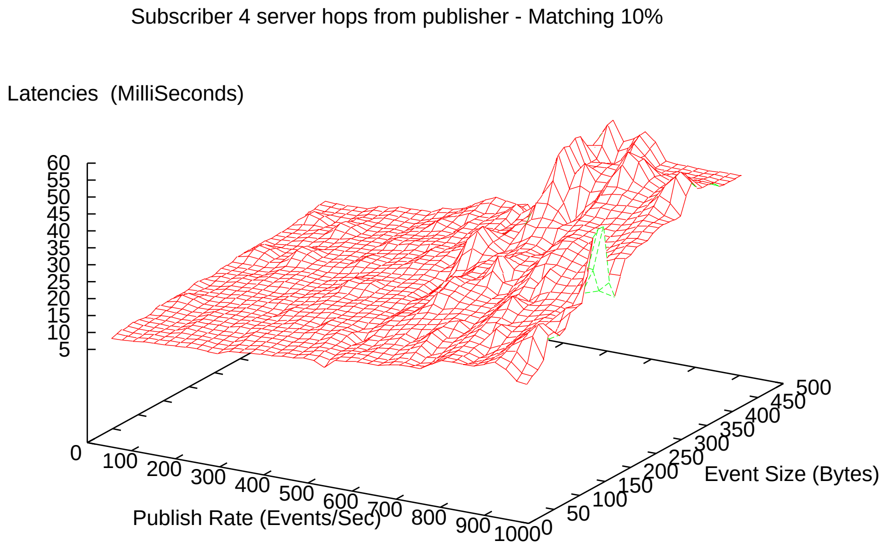


Figure 24: Match Rates of 10% - Server Hop of 4

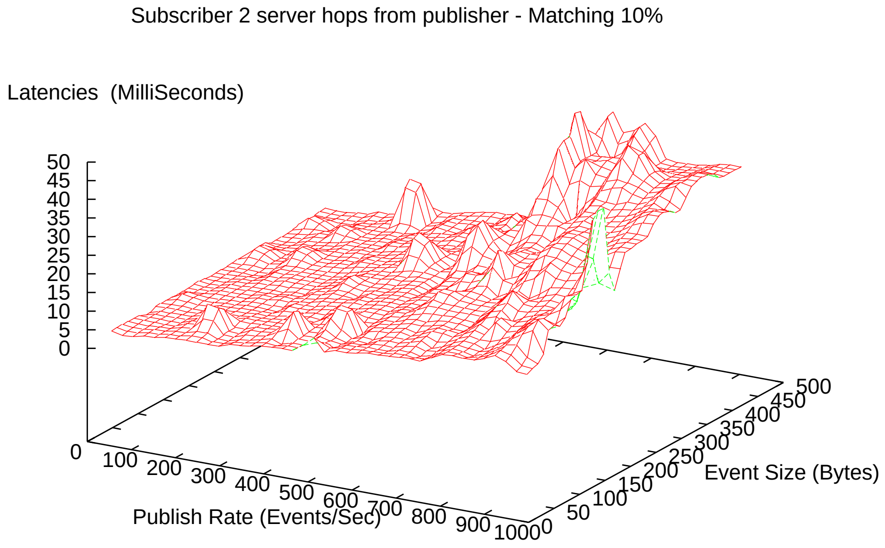


Figure 25: Match Rates of 10% - Server Hop of 2

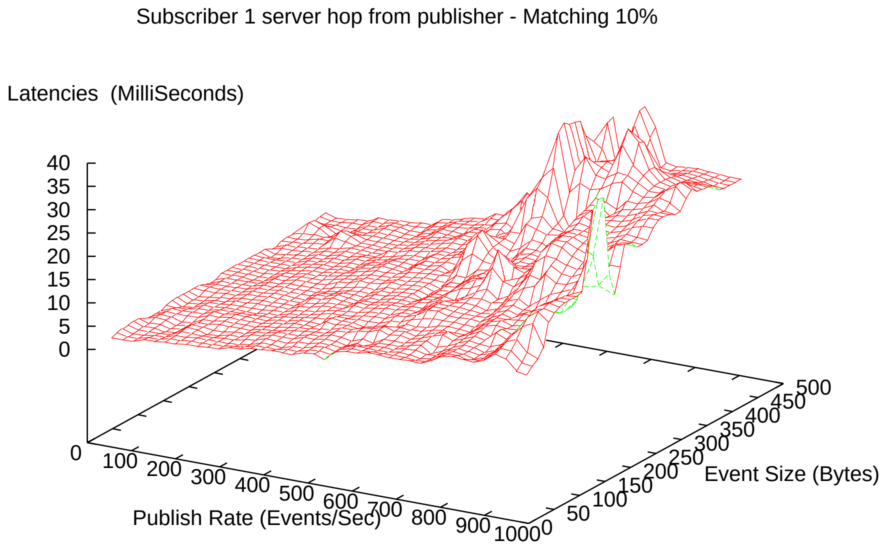


Figure 26: Match Rates of 10% - Server Hop of 1

5 Future Directions: The need for dynamic topologies

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

6 Conclusion

In this paper, we have presented the Grid Event Service (GES), a distributed event service designed to run on a very large network of 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. Links could fail, and the routing to the affected units is performed using the alternate links. The protocols in the GES protocol suite exchange information collected and processed by the other protocols. Thus when a new connection is added the information is used to update the connectivity graph, which is used to identify the relevant nodes for the propagation of profiles to. This information contained in the profile graphs is then used for the hierarchical dissemination of content. All these protocols can run concurrently, adding a lot of flexibility to the overall system.

The *system views* at each of the server nodes respond to changes in system inter-connections, aiding in the detection of partitions and the calculation of new routes to reach units within the system. The organization of connection information ensures that connection losses (or additions) are incorporated into the connectivity graph hosted at the server nodes. Certain sections of the routing cache are invalidated in response to this addition (or loss) of connections. This invalidation and subsequent calculation of best *hops* to reach units (at different levels) ensure that the paths computed are consistent with the state of the network, and include only valid/active links. The event routing protocol uses the profile information available at the unit gatekeepers to compute the sub-units that the event should be routed to. To reach these destinations every node, at which this event is received, employs the *best hops* to reach those destinations. This *best hop* is computed based on the cost of traversal and also the number of links connecting the different units. Thus, in our system, based on the organization of profiles and subsequent matching of events, the only units to which an event is routed to are those that have clients interested in that event. The protocols in GES ensure that the routing is intelligent and can handle sparse/dense interest in certain sections of the system. GES's ability to handle the complete spectrum of interests equally well, lends itself as a very scalable solution under conditions of varying publish rates, matching rates and message sizes.

The results in section 4 demonstrated the efficiency of the routing algorithms and confirmed the advantages of our dissemination scheme, which intelligently routes messages. Industrial strength JMS solutions, which support the publish subscribe paradigm generally are optimized for a small network of servers. The seamless integration of multiple server nodes in our framework provides for very easy maintenance of the server network.

References

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

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

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

[4] 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.

[5] 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.

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

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

[8] 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.

[9] 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.

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

[11] 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.

[12] 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.

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

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

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

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

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

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

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

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

[21] 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.

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

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

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

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

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

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

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

[29] 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.

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

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

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

[33] 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.

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

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