Mercury: A Scalable Publish-Subscribe System for Internet Games *

Ashwin R. Bharambe
ashu@cs.cmu.edu

Sanjay Rao
sanjay@cs.cmu.edu

Srinivasan Seshan
srini@cs.cmu.edu

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

ABSTRACT

Today's network games suffer from scalability and performance limitations caused by centralized client-server architectures and/or broadcast communication. In this paper, we argue that the communication between components of a game can be modeled as a *publish-subscribe* system. We present the design of MERCURY, a completely distributed publish-subscribe system, which supports a content-based publish-subscribe model of communication and performs distributed matching using a novel content-based routing protocol. We also present preliminary simulation results identifying key design decisions affecting the scalability and network efficiency of the system.

1. INTRODUCTION

Network games have gained popularity in the last decade, with the online gaming market predicted to increase to $2.3 billion by 2005 [16]. They are rapidly evolving from small 4-8 people games to large scale games involving several thousand participants [6]. We believe that to enable practical and wide-spread deployment of games, two design limitations of the current generation of network games need to be addressed.

First, most network games today are centralized. Players send messages to a central server and these messages are broadcast to all other active players (e.g., Quake I). However, this centralized client-server architecture has resulted in several problems:

- **Scalability**. Games should be able to handle both a large number of players in a single game and a large number of simultaneous instances. Centralized systems create bottlenecks that make such scaling difficult. Current games are unable to support more than 3,000 - 6,000 simultaneous players on a single server [16].

- **Robustness**. A game should be playable as long as the players' computers and their connectivity is operational. Client-server games have central points of failure that prevent such robustness.

- **Response Times**. The performance of a game should only be limited by the connectivity between the players. Since server-based games force all communication to pass through a central point, game performance is limited. This also provides an unfair advantage to users near the central site.

Second, many traditional network games are broadcast-based. That is, messages sent by any participant are delivered to all other participants. Although games such as Mimaze [9] have distributed architectures, all game updates are sent to *all* participants. Thus, communication is a form of broadcast even if it is implemented as IP Multicast. These broadcast-based techniques cannot support a large number of players. A more appropriate design for games is one where a player only receives messages that are relevant to his or her current location in the game.

In this paper, we address these challenges by exploring a new type of communication building block for distributed multiplayer games. Our system, called MERCURY, provides a distributed approach to delivering only messages that a client is interested in. In essence, MERCURY is a *distributed publish-subscribe* system. MERCURY supports subscriptions that are flexible enough to describe player interests in many types of games. Our initial results show that MERCURY provides good scalability by effectively distributing the responsibility of matching game events to player interests.These results also show that the system cannot yet provide the response times needed by network games. However, we believe that there are promising techniques for improving this performance.

While we focus on Internet games in this paper, we believe that MERCURY can greatly help the design and deployment of a wide range of truly distributed applications over the Internet, e.g., chat rooms and instant messengers, which currently use a centralized client-server model.

*This research was sponsored by DARPA under contract F30602-99-1-0518. Additional support was provided by IBM. Views and conclusions contained in this document are those of the authors and should not be interpreted as resprresenting the official policies, either expressed or implied, of DARPA, IBM or the U.S. government.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Netgames2002 April 16-17, 2002 Braunschweig, Germany.
Copyright 2002 ACM 1-58113-493-2/02/0004 ...$5.00.

In the next section, we motivate the use of publish-subscribe systems for building games. In Section 3, related work in the areas of distributed simulation modeling and publish-subscribe systems is reviewed briefly. Section 4 describes the architecture of the MERCURY system. Preliminary evaluation results are presented in Section 5. Finally, Section 6 summarizes and details future work.

2. BUILDING GAMES USING PUBLISH-SUBSCRIBE

In all multiplayer games, multiple users share a single instance of a game "world" and each participating node only needs information about this world relevant to its associated player. Broadcast-based games [9] deliver the changes in state of the entire world to each node. In such systems, the game client software is responsible for filtering and displaying only relevant information. Unfortunately, such a design wastes bandwidth and enables players with modified client software to cheat and view data they should not have access to.

At the other extreme, some recent games filter messages at a central server. For example, the Quakeforge [15] game server sends entity updates only for the visible or audible entities from a player's current location. For this to work, every player needs to send all his movement updates to the server which creates a bottleneck both in terms of computation as well as network bandwidth.

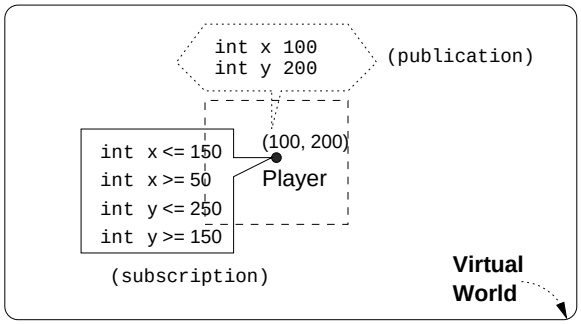

Figure 1: Example of a subscription and a publication

In many ways, this form of message delivery required for games can be modeled as a *publish-subscribe* system. In a publish-subscribe system, publishers inject *publications* into the network, while subscribers register their interests by sending *subscriptions*. The routers in the network deliver a publication only to those subscribers whose subscriptions match the publication. As an example, Figure 1 shows a player expressing his interest in his arena by subscribing to the square $[50, 150] \times [150, 250]$. His current position is advertised as a publication $(100, 200)$ and will be delivered to all other players whose arenas contain these coordinates.

Two important considerations for any publish-subscribe system are what type of subscriptions it supports and how it matches publications against subscriptions. Based on the selective power of the subscription language, publish-subscribe systems can be classified [4] as:

- **Channel or Subject-based**. A subscription can select publications advertising a specific channel or subject chosen from a predefined set of available channels or subjects.
- **Content-based**. A subscription can specify any predicate over the *entire* content of the publication.

The example in Figure 1 motivates our choice of the subscription language. Clearly, range matching, at least on numeric attributes, must be supported by the language. In general, the information that a node needs about the entire game world may be quite different depending on the rules and type of game. Many games display or provide information about the score, condition or location of other players. In first-person shooting (FPS) games, a node needs enough information about the region around a player to render the scene accurately. In a team-play game, a node may also display information about the region near allies or partners. Other games provide information about the terrain but not about the location of other players.

Thus, it would be extremely difficult and unnatural to express these varied filtering requirements in terms of a predefined set of subjects or channels. Hence we chose to support a content-based publish-subscribe model in the MERCURY system.

3. RELATED WORK

While a wide range of publish-subscribe systems have been widely studied in the past, none of this work directly meets the requirements of Internet games. These include both channel-based and content-based publish-subscribe systems. A few of these systems have distributed architectures, but most of them are centralized and use broadcast as the underlying communication mechanism. In this section, we briefly review them, and discuss why they are not suitable for Internet games.

A large number of event notification systems such as Elvin [17], and commercial systems such as iBus ([13], [22]), SmartSockets™ [21], etc. either have a centralised architecture or a federation of servers arranged statically into a hierarchy. Both of these architectures are ill-suited to overload, failure and the dynamically changing Internet environment. The CORBA Notification Service [14] and Sun's Java Messaging Service [20] are mere specifications for the interface of an event notification system and do not advocate any particular architecture for the implementation.

Distributed simulations featuring massive virtual environments have been studied in great detail in the past with resulting standards like DIS[11] and HLA[10]. These architectures are broadcast-based in that each update message is communicated to *all* participants in the simulation. Hence, these protocols cannot scale as the number of entities increased in response to the demand for more and more realism.

Distributed channel based publish-subscribe systems have been studied in the context of IP Multicast. Macedonia et. al. [12] proposed spatial, functional and temporal aggregation depending on the areas of interest of a particular entity. While aggregating information on the basis of spatial locality is useful in capturing rendering-sensitive events in the nearby environment, they mask all organizational relationships between the entities. Singhal and Cheriton[18] use projection aggregations which project organizations onto virtual world "grids". This allows participants to receive different fidelity level information for different entities in the same region. However, both these approaches ultimately use one multicast group for each area of interest or projection ag-

gregation. If the interest areas are not fine-grained enough, many entities might end up receiving updates about a large portion of the virtual world. On the other hand, using too fine-grained interest areas results in an extremely large number of multicast groups, consequently increasing the amount of state each router has to maintain.

The systems that come closest to supporting the rich subscription languages that Internet games require are Gryphon [1, 2] and Siena [4]. Like Mercury, both Gryphon and Siena are distributed content-based publish-subscribe systems. However, both these systems flood subscriptions[1] throughout the network thereby significantly limiting the scalability of the system. With our goal of large-scale, wide-area deployment in mind, our system must avoid techniques such as flooding or global knowledge about participants. Some recent systems, such as Herald [3] and Scribe [5] are distributed publish-subscribe systems that have much better scaling properties than Gryphon and Siena. However, these systems achieve their scaling by adopting subject-based subscription languages which are too restrictive for use in games. The challenge then is in designing a distributed publish-subscribe systems that allows flexible query languages like Gryphon and Siena, yet having scaling properties similar to Herald and Scribe.

4. MERCURY ARCHITECTURE

The design of the MERCURY system is motivated by the following requirements:

- **Subscriptions.** The publish-subscribe system must support content-based subscriptions. The content description language must be rich enough to describe game-related subscriptions for team play, visibility, etc.
- **Distributed Matching.** The publish-subscribe system must distribute the responsibility of matching of publications to subscriptions across the participating nodes.
- **Scalable & Network Efficient.** The matching of publications and subscriptions must scale with both the number of subscriptions and publications. The routing of publications to subscribers should match the real-time requirements of game play.

The key to our design is avoiding the flooding of subscriptions or publications and distributing the overall traffic uniformly through the network. The core idea is based upon the rendezvous point concept first described in Scribe [5]. At a high level, the system is composed of three components:

- **Routing.** Subscriptions and publications are routed to one or more rendezvous points such that publications are guaranteed to meet all relevant subscriptions.
- **Matching.** Publications are matched with the subscriptions at the rendezvous points.
- **Delivery.** Once the interested subscribers for a particular publication is known, the message is delivered to them.

[1]Siena uses containment relations for reducing flooding, but their effectiveness remains unclear.

The above approach reduces flooding to a large degree at the cost of some increase in the latency of event delivery. The main challenge in applying the rendezvous point idea to content-based publish-subscribe systems is in mapping subscriptions and publications to rendezvous points and distributing these uniformly throughout the network. In the rest of the section, we describe the subscription language used and details of the routing mechanism employed.

4.1 Subscription Language

A good subscription language should be flexible enough for network games, and at the same time, should not hinder the scalability of the system. The subscription language we chose for MERCURY is a subset of a relational query language like SQL.

In MERCURY, a publication is represented as a list of typed attribute-value pairs, very similar to a record in a relational database. Each field is a tuple of the form: (type, attribute, value). The following types are recognized: `int, char, float,` and `string`. More structured types are not supported in the core since we believe they can be easily composed from these as per individual application requirements.

A subscription is a conjunction of predicates which are tuples of the form: (type, attribute, operator, value). Numeric types permit the following operators: $<, >, \leq, \geq$ and $=$. The `string` type permits *equality, prefix and postfix* operators. The numeric operators support range matching which would be typically useful for modeling game arenas. The string operators are useful in expressing interests in various game statistics based on players' or teams' names.

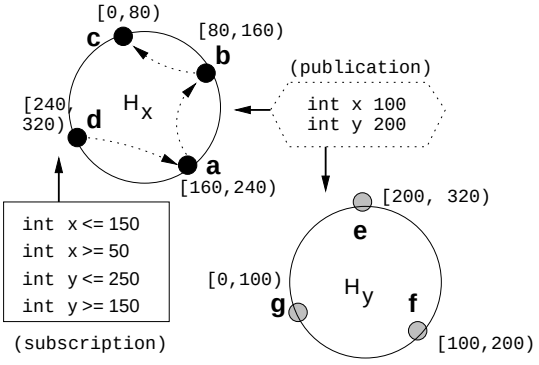

Figure 2: Routing of publications and subscriptions. A subscription gets routed to any *one* of the attribute hubs H_x or H_y. In the figure, the subscription gets stored at nodes `b` and `c`. The publication gets routed to *both* hubs since any one of them could have stored the relevant subscriptions.

4.2 Routing Mechanism

The scalability of the MERCURY system hinges crucially on the design of the publication and subscription routing mechanism. This routing must be efficient in terms of performance and at the same time, it must distribute responsibility among all participants so that no single node in the system becomes a 'hot spot'.

Attribute Hubs

MERCURY divides responsibility among nodes by partitioning them into groups called *attribute hubs*, each in-charge of

a specific attribute in the overall schema. Let H_a denote the attribute hub for an attribute a.

Nodes inside each attribute hub are logically arranged in a circle, with every node keeping track of its successor and predecessor pointers. Each node is reponsible for a *range* of attribute values. For numeric types, the mapping is straightforward since they have a bounded range - at worst, it would be bounded by the size of the data type. Thus, for an attribute a of integer type, each node would be responsible for an interval of length $2^{32}/|H_a|$. In practice, the dynamic range would be much smaller. For example, in Figure 2, the maximum value of either coordinate is dictated by the size of the virtual world.

String attributes, on the other hand, can have arbitrary length - so, there is no clear way to partition them efficiently. By partitioning them on the basis of first few or last few characters, we can support prefix and postfix operators efficiently. For example, a node can be in-charge of all strings starting with '`d-f`'. This is precisely the reason why the *substring* operator cannot be efficiently supported by the MERCURY system. It demonstrates the trade-off between scalability and expressiveness of the selectivity mechanism.

Routing of Publications and Subscriptions

Let $\mathcal{A}$ denote the set of attributes in the namespace. Let $n = |\mathcal{A}|$ be the cardinality of the set. For example, in a FPS game, $\mathcal{A}$ could be { `x-coord`, `y-coord`, `z-coord`, `event-type`, `player`, `team` }. Suppose that $\mathcal{A}_S$ denotes the set of attributes in a subscription S. Similarly, the set of attributes present in a publication P be denoted by $\mathcal{A}_P$.

The routing of subscriptions and publications is done in the following manner: A subscription S is routed to H_a, where a is *any* attribute chosen from $\mathcal{A}_S$. As discussed later in the section, the choice of the hub critically affects the degree of flooding of a subscription.

Given a publication P, the set of subscriptions which could match P can reside in any of the attribute hubs H_b where $b \in \mathcal{A}_P$. Hence the publication P is sent to *all* such H_bs. Thus, it is possible that a publication could be sent to all the n attribute hubs. However, we believe that in a typical application, only a few attributes will be popular, i.e., most subscriptions will contain reference to at least one of the popular attributes. In this case, hubs for these attributes will suffice. In the rare case of a subscription not containing any of these attributes, we can send it to all the hubs.

Content gets routed along the circle using successor and predecessor pointers. A node in H_a compares the value of attribute a in a publication or a subscription to the range it is responsible for. Depending on the results of the comparison, it stores and/or forwards the message appropriately.

Figure 2 illustrates the routing of subscriptions and publications. It depicts two hubs H_x and H_y corresponding to the X and Y coordinates of a player. The minimum and maximum values for the x and y attributes are 0 and 320 respectively. Accordingly, the ranges are distributed to various nodes. The subscription enters H_x at node `d` and gets stored at nodes `b` and `c`. The publication is sent to both H_x and H_y. In H_x, it gets routed to node `b` and matches the earlier subscription, while in H_y, it is thrown away after getting routed to node `e`.

In this scheme a publication gets routed to *exactly* one rendezvous point since it carries a *single* attribute value. A subscription, however, can contain ranges and hence, can be stored at any number of rendezvous points depending on its selectivity. Also, note that a subscription can have different selectivities in different attributes. Thus, sending the subscription to a hub of a high selectivity attribute will reduce flooding.

We note that our routing protocol has been motivated by the Chord [19] system. Like Chord, MERCURY too organizes members into a logical ring. However, the key difference is that while Chord identifies the node in charge of a resource (attribute value) by a hash on the resource ID (attribute value), MERCURY requires that consecutive nodes in the ring handle contiguous attribute value ranges. Thus, while Chord can handle exact matches, MERCURY can support a more flexible query language, including queries on a range of attribute values. However, while use of hashing in Chord can ensure better load balancing in terms of responsibilites assigned to nodes, load balancing achieved in MERCURY depends on the distribution of attribute values. We plan to explore these issues further as discussed in Section 6.

We are currently working on several optimizations to enhance the performance of the basic system. In the above routing scheme, a publication would take $n/2$ hops, on an average, along the circle to get routed to its rendezvous point. By implementing approximate *finger pointers* as in Chord, we believe we can reduce the average number of hops to $\mathcal{O}(\log n)$. Another possible optimization we can make is utilizing 'locality' in published events. In a game, publications correspond to movement events of a player. Since location attributes for a player change gradually, attribute values in these publications will be close to each other. We use this attribute locality to implement effective caching. A publisher starts its search for a rendezvous point with the rendezvous point used for the previous publication. This reduces the number of hops needed to route a publication.

Throughout the discussion, we have glossed over how a new node is added to the system. A new node must obtain information about the list of hubs and its position in the routing system. This can be done by querying any random node in the system. We assume some simple out-of-band scheme for discovering this random node.

5. PRELIMINARY EVALUATION

We have performed an initial evaluation of MERCURY using network simulation. Our primary goal is to explore the scalability and performance aspects of MERCURY. To quantify the scalability of the system, we use the following metrics:

- *Publication-routing load*, defined as the average number of publications routed by a node, and
- *Matching load*, defined as the average number of subscriptions stored by a node.

These metrics reflect the distribution of traffic through the network and the computational burden placed by the system on each node. We measure the perceived performance of game players using *publication delivery delay,* defined as the average delay for a publication to reach the interested subscribers. We also evaluate performance using the number of MERCURY level and network level hops required for publications to reach the subscribers.

5.1 Simulation Setup and Methodology

In our simulation, we implement the routing and matching mechanisms for MERCURY as an extension of the `ns-2` [7] network simulator. The simulations currently lack any realistic cross-traffic. We use the *transit-stub* topologies generated by the Georgia Tech. Internet Topology Model generator [23] for our experiments.

Each experiment is conducted in the following manner: Nodes in the topology are randomly divided into attribute hubs. Each node inside each hub is assigned a value-range it is resposible for. The simulation is run for 200 seconds. During the simulation, no node leaves or joins the system. Also, routing is done using statically computed routes.

The input to the simulations is a synthetic workload modeling players in a FPS game like Quake. The attribute space consists of two attributes: `x-coord` and `y-coord`. The virtual world is modeled as a square. Player movements are modeled using the mobility models [8] in `ns-2`. If a player is currently at location (x, y) in the virtual world, he subscribes to a square of side-length $2 * radius$ centered at (x, y). The subscription is changed when the player moves by a distance of $\alpha * radius$ where α is a tunable parameter. Publications are sent when a player's location changes by 2.0 units.

5.2 Results

We performed our simulations on network topologies of different sizes. Table 1 presents relevant characteristics of the topologies used in the simulation.

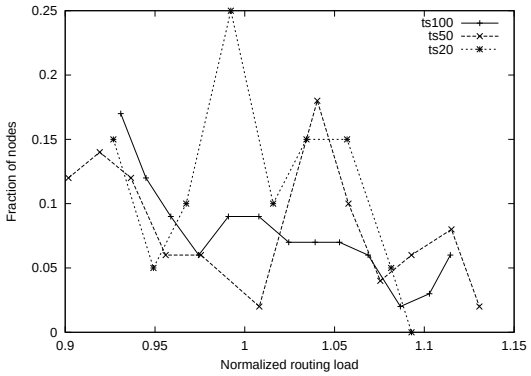

Figure 3: Distribution of the publication-routing load. Load is normalized with respect to the average load.

The current results for scalability are shown in Figures 3 and 4. Figure 3 depicts the distribution of publication routing load over all the nodes. The horizontal axis represents load normalized with respect to the average load. Thus, if the system were to balance load perfectly, every node would have a normalized load of 1, i.e., there would be a sharp spike in the graph. The plot shows that the maximum and minimum loads are within 10-12% of the average load. This result appears promising; however, the system needs to be evaluated under different workloads.

Figure 4 shows a similar situation with respect to the matching load. In this case, load is more unevenly distributed. The maximum and minimum loads are within 60-70% of the average load. This is an artifact of the workload we have used. The central region in the virtual world is covered by more subscription 'squares' as compared to peripheral regions. Hence, the nodes responsible for the central region receive more subscriptions.

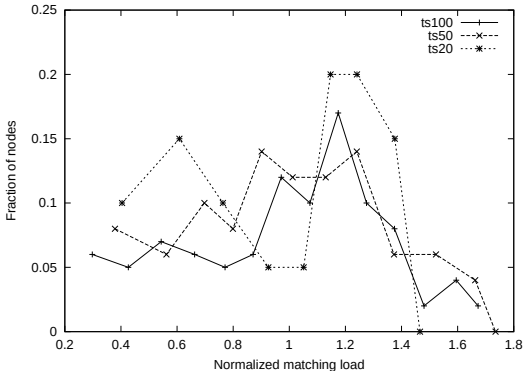

Figure 4: Distribution of the matching load. Load is normalized with respect to the average load.

Table 2 also shows the number of subscription copies, defined as the number of nodes storing a particular subscription. The low values of this metric indicate that the system reduces subscription flooding to a large degree.

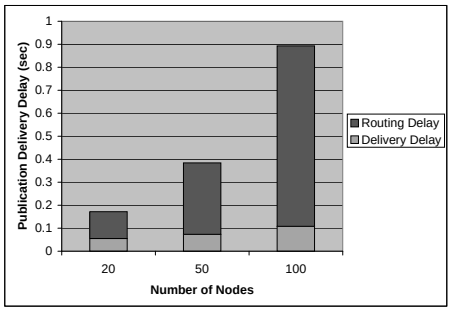

Figure 5: Publication Delivery Delay

Statistic	ts20	ts50	ts100
Hopwise diameter	7	8	11
Delay diameter	139 ms	166 ms	254 ms

Table 1: Topology characteristics

Figure 5 shows the results for end-user perceived performance of the system. Routing delay refers to the average time taken by a publication to reach the rendezvous point. Delivery delay refers to the average time required to deliver the publication to relevant subscribers from the rendezvous point. These results indicate that the overall delay scales linearly with the number of nodes. It is reasonable (about 400ms) for upto 50 nodes, but beyond 100 nodes, it is very large (about 900ms) as compared to the diameter of the network. The graph also shows that routing delay accounts for most of publication-delivery delay, as expected.

The propagation delays for a centralized system would be roughly equal to twice the delivery delays shown in Figure 5. However, a large part of end-user perceived delay in

Statistic	ts20	ts50	ts100
Subscription copies	2.18	3.97	6.95
Mercury-level hops	3.89	9.03	21.00

Table 2: Statistics

a centralized system stems from the excessive queueing delays and packet loss at the server under high load conditions. These factors have not been modeled in current simulations. Hence, using these simulation results, it would not be fair to compare a centralized system against MERCURY directly.

In any case, the absolute magnitude of the delay is very large and not suitable for real-time game play. A more worrisome aspect is the fact that this value is about 4 times the network diameter. As Table 2 shows, this is almost entirely a result of the large number of routing hops at the MERCURY level. As our on-going work, we are exploring several promising approaches to reduce the delays to acceptable levels.

6. SUMMARY AND FUTURE WORK

In this paper, we have made a case for using a publish-subscribe based architecture for Internet games. We have presented the design of MERCURY, a distributed publish-subscribe system. The system supports a fairly rich subscription language and utilizes a novel content-based routing protocol for achieving fully distributed matching. We have presented a preliminary evaluation of the scalability and performance aspects of MERCURY. We believe MERCURY achieves its scalability requirements to a large degree by preventing subscription flooding and partitioning matching and routing load effectively throughout the system. We also note that there are a number of ways in which performance of the system can be improved. The rest of the section discusses the principal areas we have identified for future work based on our preliminary simulation results.

Reducing publication-delivery delay

Figure 5 illustrated that the overall publication-delivery delay is undesirably high and that it is mainly a consequence of high number of routing hops at the MERCURY level. This in turn is a result of the routing algorithm taking $\mathcal{O}(n)$ hops, on average. We are currently working on utilizing finger pointers [19] to reduce it to $\mathcal{O}(\log n)$ hops.

Also, the logical structure of each attribute hub is independent of the underlying network topology. This implies that with each MERCURY level hop, we can end up crossing the entire network, in the worst case. By taking actual network distances into account while constructing the attribute hub, we can reduce the routing delay to about the diameter of the network. With these modifications, the total delay would be comparable to the propagation delay for a centralized system.

Load balancing

As mentioned in Section 4, the distribution of load, i.e., number of publications or subscriptions handled by each node, is dependent on the distribution of attribute values in publications. For example, if a large number of players concentrate in a relatively small area of the virtual world, all updates would have to be routed only to the nodes responsible for that specific area. We are currently working on using approximate histogram based techniques to effectively distribute load dynamically.

Game state maintenance and robustness

An essential component of a game is the notion of highly consistent and persistent state which records the effects of all actions taken by players. Currently, MERCURY just acts as a filter for publications. However, it can easily support a persistent state by considering publications as write events on the underlying distributed database. The writes can actually be performed at the rendezvous points.

Also, the system needs to be robust to node failures, i.e., if a node fails or leaves the system, either, it should not affect the correct operation of other nodes or the recovery should be relatively quick.

Reducing network traffic

Reducing the flooding of subscriptions is one of the important objectives of MERCURY for achieving better scalability. Recall that a subscription is routed to *any* H_a where $a \in \mathcal{A}_S$. Consider a subscription of the form `int x any`, `int y < 20`, `int y > 5`. If this subscription is sent to H_x, it will be stored at *all* nodes in H_x since it matches every possible range. On the other hand, if it is sent to H_y, with a very high chance, it will be stored at a few nodes only. Thus, if the selectivity of a subscription for each attribute can be determined inexpensively, we can send it to the attribute hub of highest selectivity. This will further reduce the flooding of subscriptions.

Also, note that a publication needs to be delivered to multiple recipients once it has been matched at a rendezvous point. Currently, we use unicast for doing this. We would like to leverage existing end system multicast solutions to provide efficient delivery.

Workload modeling

For gaining better intuition into the working of the system, we need to more accurately model the traffic patterns of games for driving our simulations.

7. REFERENCES

[1] M. Aguilera, R. Strom, D. Sturman, M. Astley, and T. Chandra. Matching events in a content-based subscription system. *ACM Symposium on Principles of Distributed Computing (PODC '99)*, pages 53–61, May 1999.

[2] G. Banavar, T. Chandra, B. Mukherjee, J. Nagarajarao, R. Strom, and D. Sturman. An efficient multicast protocol for content-based publish-subscribe systems. *The 19th IEEE International Conference on Distributed Computing Systems (ICDCS '99)*, pages 262–272, May 1999.

[3] L. F. Cabrera, M. B. Jones, and M. Theimer. Herald: Achieving a Global Event Notification Service. In *Proceedings of the Eight Workshop on Hot Topics in Operating Systems*, Elmau, Germany, May 2001.

[4] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Design and Evaluation of a Wide-Area Event Notification Service. *ACM Transactions on Computer Systems*, 19(3):332–383, Aug. 2001.

[5] M. Castro, P. Druschel, A.-M. Kermarrec, and A. Rowstron. SCRIBE: A large-scale and

decentralized publish-subscribe infrastructure. In *Third International Workshop on Networked Group Communication (NGC2001)*, London, UK, Nov. 2001.

[6] Everquest. `http://www.everquest.com`.

[7] Network Simulator. `http://www.isi.edu/nsnam/ns/`.

[8] K. Fall and K. Varadhan. NS Manual.

[9] L. Gautier and C. Diot. Design and Evaluation of MiMaze, a Multi-player Game on the Internet. In *IEEE Multimedia Systems Conference*, Austin, TX, July 1998.

[10] Institute of Electrical and Electronics Engineers. *High Level Architecture,* `http://www.dmso.mil/public/transition/hla/index_html`.

[11] Institute of Electrical and Electronics Engineers. Standard for Information Technology, Protocols for Distributed Interactive Simulation. Technical report, Mar. 1993.

[12] M. R. Macedonia, M. J. Zyda, D. R. Pratt, D. P. Brutzman, and P. T. Braham. Exploting reality with multicast groups: A network architecture for large-scale virtual environments. In *Proc. of the 1995 IEEE Virtual Reality Symposium (VRAIS95)*, Mar. 1995.

[13] S. Maffeis. iBus: The Java Intranet Software Bus. Technical report, SoftWired AG, Zurich, Switzerland, Feb. 1997.

[14] Object Management Group. Notification Service. Technical report, Aug. 1999.

[15] QuakeForge. `http://www.quakeforge.net`.

[16] Red Herring. `http://www.redherring.com/insider/2002/0117/724.html`.

[17] B. Segall and D. Arnold. Elvin has left the building: A publish/subscribe notification service with quenching. In *Proceedings of AUUG '97*, Brisbane, Australia, Sept. 1997.

[18] S. Singhal and D. Cheriton. Using projection aggregations to support scalability in distributed simulation. In *Proc. of 1996 International Conference on Distributed Computing (ICDCS'96)*, 1996.

[19] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *Proceedings of the ACM SIGCOMM '01 Conference*, San Diego, California, August 2001.

[20] Sun Microsystems. Java Message Service, 1999.

[21] Talarian Inc. *Talarian: Everything You Need To Know About Middleware,* `http://www.talarian.com/industry/middleware/whitepaper.pdf`.

[22] TIBCO Inc. *Rendezvous Information Bus,* `http://www.rv.tibco.com/datasheet.html`.

[23] E. W. Zegura, K. Calvert, and S. Bhattacharjee. How to Model an Internetwork. In *IEEE Infocom 1996*, San Francisco, CA.