SWAN: Small-world Wide Area Networks

(Invited Paper – SSGRR 2003s)

Virgil Bourassa
and Fred B. Holt

***Abstract*— As an alternative to client-server architectures, we have developed the Swan technology to support real-time interactive applications. Providing a balanced multicast capability for ad hoc communities, Swan weaves computer processes into a fabric of TCP/IP connections. Using only local knowledge, these Swan fabrics weave in new arrivals and repair around departures.**

Our Swan solution for ad hoc networks is near-optimal. Swan retains high reliability and logarithmic latency. Latencies scale better than log-base-2 in the number of participants. Swan handles multiple simultaneous arrivals and departures. Swan enforces strong authentication but also provides other opportunities for security.

This patented technology is currently used for collaborative design reviews, meetings, multiplayer games and public safety networks.

I. INTRODUCTION

The need has been increasing in collaborative workflow, in Internet communities, and in distributed applications, for an effective means to allow scalable and reliable sharing of information across multiple processes. To enable world-wide digital collaboration, programmers need a software mechanism allowing dozens, hundreds, or perhaps thousands of participating computer processes to share information easily, quickly, and reliably across the world.

This demand for more interaction among communities on the Internet has been addressed primarily by client-server communications infrastructures. The costs and administration of the resulting server farms indicate the need for a more natural solution. Completely connected peer-networks or rogue-server models provide an alternative for small Internet communities. For low-performance applications like file-sharing, a variety of peer-to-peer solutions have emerged both commercially and academically.

To address the demands of real-time interactive applications, like multiplayer games or emergency networks, we developed Small-world Wide Area Networks (Swan). Swan weaves computer processes into a fabric of TCP/IP connections. These fabrics have the topology of regular graphs, which provides many benefits over trees. For example, Swan fabrics retain high performance, reliability, and survivability, as a session scales from a few to thousands of participating processes.

A. Design Constraints

In 1997 our initial target application was collaborative engineering. At a large manufacturing company the engineering groups wanted to hold design reviews across the Internet. During these reviews there could be a dozen sites participating. Not all sites were guaranteed to arrive on time nor to stay throughout the review. Any participating site would have to be able to lead the review at any time. The 3-D renderer, parts database interface, and assembly-tree viewer would all act as peer applications during the review. Regardless of who initiated a review-event (e.g. calculation, measurement, annotation, new viewpoint, part load/unload), the collaborative performance should be comparable to the single-user performance of these software tools.

This industrial application fixed the design constraints. The solution would have to be secure and reliable. To solve this collaboration problem for the given application-suite (renderer, database, and assembly-tree) on the installed base of users, the solution would have to be easily adopted and compatible with a variety of existing network devices. Anticipating future requirements and alternate uses, we added two more constraints: the solution would be massively scalable, and its participants would be true peers.

The requirement was to deliver a multicast capability in which computer processes were the nodes. The majority of the message traffic would be application-events. The performance time-scales and tolerances were set by applications instead of humans. In mid-1999 we delivered Small-world Wide Area Networks (SWAN) into production use. Swan has all of the following properties.

1) Secure: To be competitively secure against industrial espionage, the solution must support strong authentication and encryption.

2) Self-healing (Reliability I): No human administration is required when a process joins or leaves the session. All the properties are maintained automatically.

3) Robust under node-failures (Reliability II): The communications structure and the message traffic must tolerate graceful and ungraceful departures by nodes (consistent with [1], an *ungraceful* departure occurs when a node leaves without sending the proper control messages).

4) Compatible with existing network devices: No special hardware is presumed to be available. The solution must be widely adoptable today.

5) Easily adopted: The solution must have a simple API that respects the current application architecture.

6) Massively scalable (Scalability I): There is no predetermined ceiling on the number of participants. The session support costs do not become prohibitive with increased participation.

7) High-performance (Scalability II): The latencies for event-message traffic must scale acceptably with the number of participants. The solution must make efficient use of available bandwidth at every node.

8) True peers: At the communications level, *every* node is equal. The multicast capability does not depend on hidden servers or a preferred class of nodes. Only local knowledge about the communications topology is stored.

B. Swan Solution

Swan provides general wide-area peer-to-peer communications among computer processes. It achieves this with high reliability and low latency, scaling from a single process to thousands of participating processes. The system is completely distributed among the participants, which may join, depart, or even fail, at any time and in any order.

The central innovation in Swan is weaving participating processes together into regular random graphs, using generally available point-to-point network protocols. We call the resulting topology of logical connections *Swan fabrics*. Swan avoids synchronization difficulties by making use of the *small-world effect* [14], [15], using only local knowledge to maintain global properties of reliability and scalability. The Swan fabrics are extended to include new participants and repaired around departing participants with minimal and local disruptions. Swan participants are *true peers*: self-contained members of a completely self-sufficient communication fabric.

As particular families of random regular graphs, the Swan fabrics have near optimal performance. The message latency grows as $O(\log_{r-1} N)$ in the number of participants N, with r being the number of connections at each node [12]. This log-latency makes sessions with hundreds or thousands of participants feasible. Moreover, the fabrics are reliable in the face of multiple simultaneous communication link and process failures.

Several additional innovations are made in Swan to support easy, quick, and reliable large-scale information sharing around the globe. By using existing Internet protocols in non-invasive ways, no special system administration, no operating system modifications, and no special hardware is required. All computers can participate, without requiring root access, daemons, kernel modifications, or the addition of "well-known" port numbers. Joining an information-sharing session is automatic. Processes may join and depart at any time.

Though openly accessible, joining a Swan session is restricted to those processes sharing the Swan code base and aware of the correct channel designation. Swan is data-agnostic and network-agnostic, and so supports additional security features like VPNs and encryption.

Currently, Swan is implemented as a library in which the point-to-point protocol is TCP/IP and each node maintains $r = 4$ connections. These libraries have been compiled on variants of UNIX, including IBM's AIX, Hewlett Packard's HP-UX, Sun's Solaris, and Linux; they have also been compiled on Windows 95/98/2000/Me/NT/XP, and the demos have been used seamlessly across WiFi networks. The publish-and-subscribe API for Swan features six classes, with from three to twelve methods each. Behind this implementation, the core technology is the subject of five US patents pending.

II. Related Work

There are four categories of computer network communication systems that might be applied to the problem of wide-area simultaneous sharing of information for the purpose of digital collaboration. These are: 1) point-to-point networking protocols, 2) client-server middleware, 3) multicast networking protocols, and 4) peer-to-peer middleware.

A. Point-To-Point Networking Protocols

A number of point-to-point networking protocols exist to allow direct one- or two-way communication between two computer processes. Examples include UNIX pipes, TCP/IP, UDP, IBM's SNA, and Xerox' XNS. Of these, only TCP/IP and UDP are universally available for communication between computers connected via the Internet.

Using point-to-point connections directly does not scale easily as the number of participating processes grows. A process is limited in the number of such connections that can be made (roughly 60), and managing even a single connection is a complex task for programmers. Coordinating a communication session involving even a modest number of connections exacerbates the program complexity enormously. For all of these reasons, direct use of a point-to-point networking protocol is not a feasible mechanism for sharing information across a medium- to large-scale collaboration across a wide-area network.

B. Client-Server Middleware

To alleviate the complexity of programming directly at the network protocol level, client-server middleware is available to provide an easier programming abstraction. In client-server middleware, a number of "client" processes find or instantiate a single "server" process, forming a direct network connection between them. The client may then request services from the server, which often is given central authority over a resource, such as a database. Examples include database servers, remote procedure calls (RPC), and CORBA. A variety of commercial offerings (e.g. PreCache, Jabber, Groove, Butterfly.net) offer advanced client-server solutions in which the application programmer should focus on the clients as being peers.

The client-server paradigm provided by this middleware, while providing a mechanism for sequenced resource sharing, is an expensive and limiting approach to digital collaboration. As sessions consume the limited resources of the existing servers, more servers must be added to the farm, resulting in a step-linear cost function for session support. Anticipating session size and balancing loads become prominent areas of applied research in the administration of these server farms.

Generally, resources are not well-used in client-server solutions. Client resources tend toward underutilization, while the resources of the servers are in excessive demand. Each client conveys information directly to the server, and possibly after some preprocessing and filtering, the server disseminates the information either through routing lists or by being polled by the clients. This creates a performance bottleneck as the number of participants increases, adds undue latencies, and wastes time with polls and heartbeats. The demands placed on the servers to provide the communications infrastructure drive the overhead in costs and administration for the resulting server farms.

C. Multicast Networking Protocols

Multicast networking protocols allow selective broadcast of messages to multiple recipients. It retains the complexity of direct network communication mentioned above, but is a natural choice for digital collaboration. Currently, multicast is available for UDP messages, but virtually all UDP multicast traffic is limited to a single local-area network or, at most, a small set of connected local-area networks. UDP multicast, in its current implementation, could easily swamp the Internet otherwise, as it would have to saturate the Internet with each message to find all possible participants.

Several wide-area multicast networking protocols have been proposed [2], and some, such as IP Multicast, are in limited commercial and/or research deployment. Most of these solutions require special router hardware and/or software to achieve data sharing without overwhelming the participating networks. Even if a standard solution were selected today, it would take years, or possibly decades, before the entire Internet infrastructure could be completely retrofitted with the new technology.

Additionally, the solutions proposed in this area, in an attempt to conserve bandwidth, are not constructed with reliability as a concern. By using spanning trees among the routers involved, any node failure can temporarily partition the collaborative session.

D. Peer-to-peer Middleware

Peer-to-peer middleware provides the programmer with a software library that is intended to provide an easy-to-use abstraction, such as "publish-and-subscribe" or "shared objects," for immediately sharing information among a set of collaborating processes. Hidden from the programmer is how the actual communication takes place.

The underlying communication infrastructure may make use of a multicast network protocol, or a graph of point-to-point network protocols, or a combination of the two. The infrastructure in commercial use today, in products such as IBM's Sametime, Data Connection's DC-Share, and Microsoft's NetMeeting, is the T.120 Internet standard. Given its commercial prominence and its influence on other solutions, we consider the limitations of T.120 as a solution for digital collaboration.

1) T.120 Internet Standard: When first connecting to a T.120 communication session on a given host computer, the local application spawns both a proxy process (the MCU) and a daemon process. The daemon process is a resident process which instantiates the MCU and listens for additional such requests. The MCU forms a direct connection to the MCU of another host designated by the application user, or is designated as the root of the session. The requesting process and all additional processes on the host wishing to join the session form a direct connection to the MCU process on that host. To share information, a process sends a message to its MCU, which is sent up the tree of MCUs to the root, then down the tree of MCUs and disseminated among their attached processes.

The responsibility of determining the topology of the connection graph lies with the end-users. In addition to being a nuisance to the users, it is not likely to result in an efficient structure for performance. The most common kind of connection scheme seen in practice is for all host MCUs to connect directly to the root MCU, a star topology.

The MCUs create worst-case performance, reliability, and scalability. All messages must be serialized through each MCU to a potentially large number of processes on the host. Loss of an MCU not only removes all of the processes on the host, but also prunes the subtree attached to it from the session. The need to coordinate all messages through the root MCU makes that process a performance bottleneck and a single point of failure for the session. This serialized messaging limits the speed to the slowest host and/or communication link in the tree.

Finally, the T.120 daemon must be installed on each host participating in a session. This requires additional administration and maintenance, and limits the set of hosts that can join in a session. It also requires an additional "well-known" port number, which must be coordinated globally among all computers on the Internet.

2) Application-layer multicast: In response to the momentum toward online collaboration and the lack of acceptable infrastructures, there is a growing community of researchers working on application-layer multicast [1]. This work builds on earlier forays [3]–[6] into providing a platform for digital collaboration. These technologies [3], [4] initially addressed group communications among distributed devices, but the growing markets for shared media, chat, online games, and collaborative work have inspired a new wave of research [7]–[11] into application-layer multicast. Commercial offerings in these markets take sophisticated client-server approaches and downplay the presence of the servers.

III. Swan Fabrics

Swan fabrics use random r-regular graphs for the communication topology, with r even and $r \geq 4$. Recall that a graph is r-regular if every node is incident to exactly r edges. Each node participating in the Swan session uses $r + 1$ sockets, r internal to the fabric and 1 *shingle*, a socket for handling

control messages external to the fabric. Currently, Swan is implemented with the parameter $r = 4$.

Each node in the graph represents a *Swan Herald*, an instantiation of the top-level class in this publish-and-subscribe paradigm. Participating Swan processes instantiate Swan Herald objects to act as their communication agents. Swan Heralds share item announcements, to which they subscribe, with other Swan Heralds on the same logical "channel." These peer Swan Heralds then publish these announcements within their own processes.

Each edge in the graph represents a point-to-point connection between Swan Heralds. Currently Swan employs TCP/IP network connections, an accepted Internet standard that provides reliable, ordered delivery along each connection. The details of TCP/IP are encapsulated in a telephone abstraction, allowing for easier maintenance and modular replacement by other point-to-point protocols.

Swan Heralds are very flexible. They can co-exist in great numbers on a single device. They can co-exist in the same application. Swan Heralds are invisible to each other unless they share the same logical channel. The Swan Heralds are woven into the same fabric only if their logical channels are identical; otherwise they belong to different fabrics and are entirely independent of each other.

A Swan Herald channel is designated by two 32-bit keys: a *channel type*, used to distinguish the application, such as a particular CAD package, from other applications; and a *channel instance*, which separates concurrent exercises of the application, such as two distinct sessions of that CAD package. Thus a single application can support multiple fabrics for different communities or different levels of control, and multiple applications of various types can run orthogonally to each other on a single device.

A. Growing in general

In the general case, we have an r-regular graph on N nodes, and an additional node arrives. Swan chooses $r/2$ disjoint edges at random and interposes the newcomer on each of them. The result is an r-regular graph on $N + 1$ nodes. Visually it looks like the newcomer has clothespinned the $r/2$ edges together. This is illustrated in Fig. 1.

In fact, each half-edge corresponds to one of the r internal sockets maintained by some Swan Herald. This Swan Herald disconnects this socket from its current neighbor and reconnects to the newcomer. Extending the fabric in this way involves only local operations, and it is minimally disruptive – introducing a newcomer into an r-regular graph requires at least r disconnections and r reconnections. Topologically the Swan fabrics form certain families of r-regular graphs across varying numbers of nodes.

While individual TCP/IP connections are designed to be reliable, maintaining the reliability of the graph as a whole is a different problem. At the graph level, what we are most concerned about is avoiding a partition, where the graph becomes separated, with two or more groups of participants talking only amongst themselves, unaware that the other groups exist.

A graph is said to be m-connected if it requires the removal of m nodes to cause it to become partitioned. An r-regular graph can at best be r-connected, since removal of one node's r neighboring nodes would isolate that node from the rest of the graph. With high probability Swan fabrics are r-connected. Swan can handle at least $r - 1$ near-simultaneous dropouts without partitioning. In experiments with $r = 4$, we have simultaneously killed a random sample of 40 out of 200 participating processes without warning, and the fabric has repaired itself without losing any messages. These experiments demonstrate that the particular sets of r nodes which would partition the graph are well-hidden against random attacks. Note also the specific construction in introducing a newcomer enables Swan to maintain better-than-expected connectivity for random graphs [13].

We extend the fabrics randomly since these random graphs greatly outperformed all of the deterministic local rules we could think of. The deterministic rules tended to be more disruptive, and they led to much higher latencies. The latencies in the Swan fabrics fall toward the middle of the range given for r-regular random graphs [12]. The maximum number of logical connections required to deliver a message corresponds to the diameter d of the graph. For 4-regular random graphs on N nodes we have with high probability the following bounds on the diameter:

$$\lfloor \log_3 N \rfloor + \lfloor \log_3 \ln N - \log_3 12 \rfloor + 1 \le d$$

and

$$d \le \lceil \log_3 N + \log_3 \ln N + \log_3 8 \rceil + 1.$$

Representative values for these bounds are tabulated here.

nodes $N =$	100	1K	10K	100K	1M	10M
upper bnd	9	11	14	16	18	21
lower bnd	4	6	8	10	13	15

In simulations and tests up to several hundred nodes, the diameters for the Swan fabrics have stayed near the average of these two bounds. As examples, Swan fabrics of diameter 6 have accommodated 200 nodes, and a Swan fabric of diameter 7 accomodated 450 nodes.

How can Swan find random edges with only local knowledge? The newcomer has contacted one Swan Herald on its shingle (see Section III-D below). This Swan Herald initiates $r/2$ drunk walks through the existing graph. If the fixed lengths of these walks are long enough, then the nodes at the ends of these walks are almost uniformly distributed across the graph. Each of these $r/2$ nodes so selected picks one of its edges at random and offers it to the newcomer, whose address was part of the message sent on the drunk walk. The newcomer checks that the edges are disjoint.

B. Small Regime

The graph cannot become r-regular until it has at least $r+1$ members. While the graph has fewer than $r+1$ members, we refer to it as being in the *small regime*. In the small regime, we completely connect all participants. Swan Heralds make

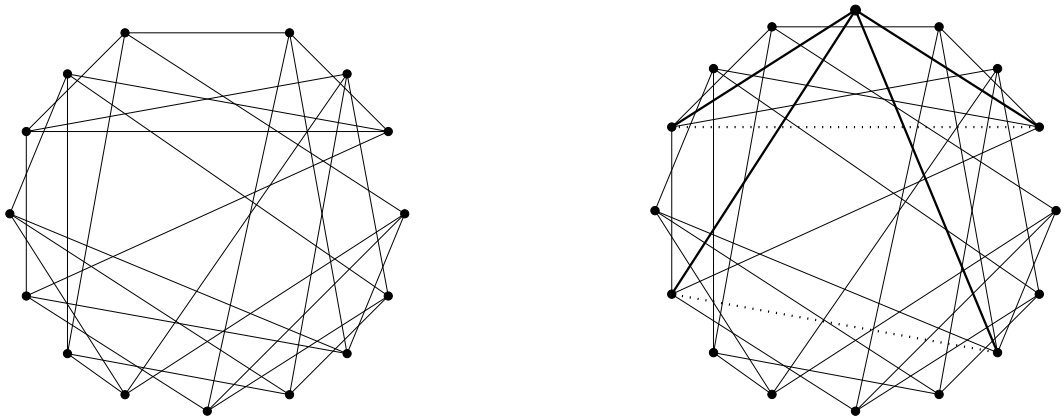

Fig. 1. Extending a Swan fabric. This example illustrates a sixteenth node joining a Swan fabric. The two dotted edges are selected at random and pinned together at the new node, shown in bold. This modification uses only local knowledge and is minimally disruptive. The resulting fabric remains 4-regular, its diameter grows as $\log N$ in the number of participants, and it tends to maintain 4-connectedness.

the transition seamlessly back and forth between the small and large regimes.

C. Departures

Swan accommodates both graceful and ungraceful departures. In a graceful departure, a Swan Herald is instructed by its application to disconnect from the fabric. In an ungraceful departure, a Swan Herald dies for any number of reasons but without sending any control messages. In either case, the remaining Swan Heralds work to repair the fabric in such a way as to maintain high connectivity and low diameter.

When disconnecting gracefully, a Swan Herald sends a message to each of its neighbors alerting them to its departure. This message lists the departing Herald's neighbors. To facilitate smooth reconnection, the first pair of Swan Heralds in the list attempt to connect directly to one another, while the second pair do the same. Barring successful reconnection, the first and third, and second and fourth, attempt direct connection.

When a Swan Herald departs ungracefully, it fails to alert its neighbors. Instead, the loss is discovered by its neighbors either immediately or, at the latest, when they next attempt to send information to the dead Herald. Upon discovering the loss of their neighbor, each of these Swan Heralds will broadcast a general connection request through the graph. Any Heralds which also lack a connection will respond and form a direct connection with the message originator. While it requires more effort than the disconnect mechanism, the undervalent neighbors will quickly find each other and reconnect properly.

During either disconnection or lost neighbor recovery, it is unlikely but possible to arrive in a state in which two nodes which already share an edge are also in need of an additional connection. These two Swan Heralds are in need of a connection to repair the graph, but the only connections available are to each other. A second connection to each other would not restore the integrity of the graph. The general connection request mentioned above is one of Swan's safety nets. Through the general connection requests, each will discover that their neighbor needs a connection, just as they do. This is a first hint that a pathological condition has occurred. However, this situation can also arise when making the transition from the large to the small regime. Swan includes the capability to detect and resolve these pathological conditions, distinguishing them from the small regime.

D. Rendevous Points

Unlike the systems surveyed in [1], Swan allows every node to act as a Rendevous Point (RP) for the session. RPs are necessary to provide newcomers a reasonable means for finding an active session. With Swan, the RPs are provided as an external list, which can be arbitrarily long and can also be edited. The length of the RP list is less important than the probability that a device on the list has a Swan Herald currently participating on the right channel.

TCP/IP provides each device with a port space shared among all processes. The higher numbered ports are designated as the user port space. When a process wishes to connect to another, it must contact the destination process' device at the particular port number on which it is "listening." This listening device will then establish a link to a socket within the destination process. It isn't possible to ask the destination device which ports have listening processes. Instead, an attempted contact will either succeed or fail depending on the existence of a listener on the port.

The TCP/IP port space implementation brings up a number of challenges. Not only must Swan Heralds find each other on some selected port, but they must also be able to tolerate conflicts from other processes using the shared port space. Multiple Swan Heralds must be able to coexist on the same host using the same port space as well. Since each attempt to connect takes time, we must also limit how many port numbers we're willing to check.

Swan introduces an innovative scheme to meet these challenges without resorting to daemons or centralized session managers. It uses the logical channel type and instance numbers to determine a sequence of ports to search. This method

minimizes possible conflicts with other, non-participating processes. If a conflict arises either with another Swan Herald or with any other process, the search is repeated until an available port number is found.

When the Swan Herald object is created, it attaches a listening socket, its shingle, to the first port available. It then searches through its list of RPs for other Swan Heralds sharing its channel, to find another participant's shingle.

To prevent searching indefinitely for other participants, the programmer also specifies the length of the sequence of ports. This depth must not only be sufficient to seek past conflicts with non-participants, but also past holes that can occur when ports are released.

Initially, the first Swan Herald in a session searches all specified RPs, searching to the maximum depth until it is satisfied that no other parties exist in the session. If the Swan Herald is on one of the RPs, and thus can be found by others, it has succeeded in connecting to the fabric, currently consisting of itself.

All subsequent Swan Heralds in the session will quickly find a Swan Herald in this fabric, and make contact through them. To join the session, the Swan Herald must successfully perform a "secret handshake" to ensure that it is, in fact, a Swan Herald, and that it belongs to the shared channel.

While searching, it is possible that two newcomers may find each other. To prevent them from forming their own separate session from one that may already exist at a greater search depth, newcomers cannot use each other as contacts for joining. Only after it has been woven into the fabric can a Swan Herald be used as an RP.

E. Broadcasting

The Swan fabric acts as both the control topology and the data topology. The main work of the fabric is to share information among the attached Swan Heralds. Since the fabrics are r-regular, r-connected, and of low latency, this broadcasting of information can be done reliably and quickly. Any Swan Herald with a message to share sends that message to each of its neighbors. Each recipient of the message notes the neighbor from whom it first received this message and forwards it in turn to the other $r-1$ neighbors.

The total number of individual messages sent between Swan Heralds to share one announcement is $(r-1)N+1$. This redundancy provides both reliability and performance. If a Swan Herald fails in the course of a message broadcast, the information has multiple other paths for reaching all other Swan Heralds. These multiple paths also provide routes around slow processes. A slow computer or communication link will not significantly delay the sharing of information, as the information will arrive at each Swan Herald by the fastest path available.

This redundancy comes at a cost of increased message traffic. This $(r-1)$-fold duplication of each message at each Swan Herald is a motivation for keeping r as low as possible.

As the message is forwarded, it is queued by each Swan Herald to be announced within its process. Since each Swan Herald will receive duplicates of each message, only the first is forwarded and queued, then duplicates are dropped. Swan uses an $O(1)$ algorithm to detect duplicates instantly. In this way Swan maintains redundancy at the communications level that is hidden from the application above.

Messages are ordered from each author before being announced. Because of the reliable nature of both the Swan graph and the underlying TCP/IP connections, lost messages in the stream are rare in the extreme. However, it is possible that as a newcomer is joining the fabric, it arrives too late to receive one message in a sequence, but happens to receive a prior message that is still passing through the fabric.

While a remote possibility, we guard against acquiring such holes in the message stream at the time of joining. Existing Swan Heralds connecting to a newcomer do not forward messages directly to the newcomer. Instead the messages are queued according to their author until they can be delivered in order, without holes. Thus, all of a newcomer's neighbors cooperate to prevent this potential problem from occurring.

IV. Performance

In an effort to provide a solid distribution of Swan, we have developed stress tests for it that challenge its construction, destruction, and broadcasting functionality to ensure reliability and scalability. These tests have shown Swan to be very robust. By using the novel techniques we've described above for the construction and repair of the Swan fabrics, and paying attention to deadlock avoidance and other potential interactions, in actual use a Swan session can accommodate multiple simultaneous arrivals and departures of Swan Heralds without difficulty. Sequenced arrivals and departures present no difficulty, regardless of their number.

Swan has seen production use for collaborative design reviews and for distributed computing. It has been demonstrated in collaborative meeting software, auctions, and multiplayer games. The Swan libraries can be compiled with minor modifications for any platform that supports TCP/IP.

Swan is a catalyst for modular design of application suites. Once the message format is agreed upon, any application which traffics in this message format and has authorization (including the correct channel type and instance numbers) can participate in a Swan fabric. One common suite of these complementary applications includes host processes, interacting clients, and data stores. The host process handles metrics and administration for a session: rosters, subscription, billing, network monitoring. The interacting clients generate most of the messages; these are the game clients, the collaborative work applications, the CAD systems. The data stores provide heavier data transfers. If the data files are large enough, they will be passed along an overlay network; Swan acts as the control topology to generate an efficient data topology [1].

If we assume uniformity among processors and communication links, the primary determinants of latency are the maximum number of edges connected to a node (its valence) and the maximum number of nodes between a source and its destination. For example, the former would dominate in

a star topology, while the later would dominate in a a ring topology. Swan fabrics have a fixed number of edges at each node, and its diameter grows logarithmically in the number of participants (Section III-A). The number of nodes must more than double before the latency increases incrementally.

A. Bandwidth

Bandwidth is the limiting resource for a Swan fabric. The relevant constraint at the i^{th} node is

$$(r-1)\mu N \leq B_i$$

in which μ is the average volume of message traffic per node per second, and B_i is the bandwidth available to this node. This constraint softens as more bandwidth becomes available. It also indicates a trade-off between the size of a session and the level of interactivity provided by the application. This constraint reiterates the cost of redundancy (Section III-E) and encourages us to keep $r = 1$.

On the whole, a Swan fabric exhibits a bandwidth which is some average of the bandwidths of all the participants. Nodes which fall behind can be helped by their neighbors through message culling, against priority flags or expiration tags.

B. Load Balancing

In situations in which some devices are known to have greater bandwidth than the majority of participants, Swan does support load balancing. To make fair use of this greater bandwidth, the process on this device can instantiate more than one Swan Herald in the same fabric. A simple overhead routine manages these duplicated Swan Heralds, clearing out duplicate messages for the application above and accelerating the delivery of messages through this set of Swan Heralds. A message sent by another node will arrive at one of this set of Swan Heralds by the fastest route. While the message continues to propagate through the Swan fabric, this Swan Herald passes the message to the rest of the set, which forward it from their locations in the logical fabric. The one process can be present at several nodes in the same Swan fabric.

V. Conclusion

Swan is a significant advance in the infrastructure for digital collaboration. Swan provides reliable and high-performance sharing of information, that scales to hundreds or thousands of asynchronous cooperating processes. It does this using generally available standard Internet protocols and requiring no system modifications or administration. Currently implemented to grow random 4-regular graphs of TCP/IP connections, Swan provides latencies that grow as $\log_4 N$ in the number of participants. It uses redundant messaging to accomodate processes with slow connections, and to provide reliability for the message stream when nodes depart. Swan has been in production use since 1999 and is the subject of five US patents pending.

Swan represents an enabling technology. With such an infrastructure one can envision a new generation of applications for digital collaboration, for work or play. An airplane could be designed and developed by twelve hundred engineers in a massive, long-term shared CAD session. Command, Control, Communication, and Intelligence applications could coordinate hundreds of independent software agents. Multiplayer games, with participants constantly joining and leaving, could run indefinitely and without the expense of server farms. Swan goes beyond the "pull" and "push" technologies currently seen on the Internet, giving us the ability to share and collaborate interactively.

References

[1] S. Banerjee and B. Bhattacharjee, A comparative study of application layer multicast protocols, manuscript, Dept. of Computer Science, U. of Maryland, April 2003.

[2] K. Obraczka, Multicast transport protocols: a survey and taxonomy, *IEEE Communications Magazine*, Jan. 1998, pp. 94–102.

[3] L. Rogrigues and P. Verissimo, xAMp: a multi-primitive group communications service, *Proc. of 11th Symposium on Reliable Distributed Systems*, IEEE, 1992.

[4] L.A. Ciscon, *et al.*, A distributed data sharing environment for teleorobotics, *Presence*, 3, no. 4, Fall 1994, pp. 321–340.

[5] M. Parsa and J.J. Garcia-Luna-Aceves, Scalable Internet Multicast Routing, *ICCCN'95*, IEEE, pp.162–166.

[6] J. Liebeherr and B.S. Sethi, A scalable control topology for multicast communications, *IEEE Infocomm*, 1998, pp. 1197–1204.

[7] R. Zhang and Y.C. Hu, Borg: a hybrid protocol for scalable application-level multicast in peer-to-peer networks, *NOSSDAV'03*, Monterey, CA, June 1-3, 2003.

[8] S.Q. Zhuang, *et al.*, Bayeux: an architecture for scalable and fault-tolerant wide-area data dissemination, *NOSSDAV'01*, Port Jefferson, NY, June 25-26, 2001, pp. 11–20.

[9] I. Stoica, *et al.*, Chord: a scalable peer-to-peer lookup service for Internet applications, *Proc. of SIGCOMM'01*, ACM, 2001, pp. 149–160.

[10] A. Rowstron, *et al.*, SCRIBE: the design of a large-scale event notification infrastructure, *NGC'01*, 2001.

[11] M. Castro, *et al.*, Scalable application-level anycast for highly dynamic groups, submitted.

[12] B. Bollobas and W. Fernandez de la Vega, The diameter of random regular graphs, *Combinatorica*, 2, no.2, 1982, pp. 125–134.

[13] B. Bollobas, *Random Graphs*, Academic, London, 1985.

[14] D.J. Watts and S.H. Strogatz, Collective dynamics of 'small-world' networks, *Nature*, 393, 4 June 98, pp. 440–442.

[15] G. Korniss, *et al.*, Suppressing roughness of virtual times in parallel discrete-event simulations, *Science*, 299, 31 Jan 03, pp. 677–679.