September 1985

Report No. STAN-CS-851084
Also numbered CSL-85-284

Taliesin: A Distributed Bulletin Board System

by

Judy L. Edighoffer

Keith A. Lantz

Department of Computer Science

Stanford University
Stanford, CA 94305

Taliesin: A Distributed Bulletin Board System

Judy L. Edighoffer and Keith A. Lantz

Abstract

This paper describes a computer bulletin board facility intended to support replicated bulletin boards on a network that may frequently be in a state of partition. The two major design issues covered are the choice of a name space and the choice of replication algorithms. The impact of the name space on communication costs is explained. A special purpose replication algorithm that provides high availability and response despite network partition is introduced.

This research was supported in part by the Defense Advanced Research Projects Agency under contracts MDA903-80-C-0102 and N00039-83-K-0431. A version of this paper was presented at the Second Interational Symposium on Computer Message Systems, IFIP, September 1985.

1. Introduction

Computer systems offer a variety of facilities to assist communication between people. For example, computer based message systems deliver messages addressed to individuals while computer conferencing systems provide structured access to conversations focusing on particular topics. This paper will focus on computer bulletin board systems, in which messages are organized by topic without imposing a conversational model of user interaction.

The simplest bulletin board implementations support a single copy of each bulletin board, accessible only by users on a single node. Usually, a bulletin board is implemented as a mailbox using existing computer mail facilities [1, 2, 3]. Last read times are either kept in a separate user profile or in the mailbox, using a slightly modified mailbox format. To extend this approach to work on a network, distribution lists, each containing a list of destinations, are created. Computer mail software replicates notices by sending a copy to each destination listed. Another common technique is to use ordinary text files, usually one per topic [5]. File replication schemes can be used to distribute copies over a network.

Typically, current implementations do not support a facility to list the names of existing bulletin boards. Furthermore, their name spaces generally lack the structure to guide such a search. Users are expected to learn about bulletin boards through word of mouth.

With the growth of distributed systems, users frequently work on multiple nodes. The usual, ad hoc replication algorithms produce inconsistent copies of bulletin boards on different nodes. This forces users to choose between missing unreplicated notices and'being inundated by multiple copies.

Nested distribution lists (distribution lists containing references to other distribution lists) cause problems as well. Overlapping distribution lists nested within a single parent list tend to duplicate notices. The number of copies can grow explosively with the number of overlapping lists. Furthermore, circular dependencies among nested distribution lists can cause infinite loops in the propagation of notices.

The final major weakness of existing systems is the growth rate in the costs of communicating, storing, and processing notices [10]. Costs per node have been observed to grow at least linearly with the number of nodes in some large bulletin boards. This effect occurs even in the absence of nested distribution lists.

Taliesin is a bulletin board system designed to overcome these problems. Particular emphasis was placed on handling network partition and on reducing the growth rate of communication costs as a function of network size. The rest of this paper will explain how Taliesin achieves these goals while avoiding other pitfalls common in current implementations. Section 2 sketches the environmental model under which Taliesin is intended to run. Sections 3 through 5 explain the major aspects of the design. Finally, Section 6 discusses a prototype implementation.

2. The Taliesin Architecture

The logical organization of Taliesin follows the standard server/object model used in many operating systems. In this model, the universe consists of objects and agents. An object is a resource, either physical or logical. The primary objects supported by a bulletin board system are *bulletin* boards and notices. Notices are the flyers, advertisements, or like items posted for viewing. A bulletin board is a store of notices characterized by the following properties:

1. The notices submitted to a bulletin board are classified according to their topics.
2. Users choose which notices to view by specifying constraints, primarily upon their topics.
3. The most common operation is to read all notices submitted since the previous read of the same topics.

An agent is a user or any entity capable of manipulating or implementing objects. An agent may act in either of two roles. As a server, an agent supervises access to objects it maintains. As a client, an agent calls upon servers to manipulate other objects. The agents of the prototype implementation are discussed in Section 6.1.

Taliesin is intended to be distributed over a computer network. The network is viewed as consisting of nodes, hardware units that provide data processing and possibly long term data storage, and *communication links* that allow pairs of nodes to exchange data. A state of *network partition* exists when some nodes are unable to communicate with other nodes, directly or indirectly. Taliesin is designed to cope well with network partition.

The Taliesin architecture does not assume any particular network configuration or low-level network protocols. For example, a node can be anything from a large mainframe down to a personal computer or workstation. Similarly, a message, the logical unit of data transmitted between a pair of agents, is whatever the agents find convenient. It can be implemented on top of datagrams or virtual circuits. A single Taliesin message may consist of multiple “messages” as defined by the underlying operating system.

3. Name Space Design

Some of the flaws of current systems stem from the name spaces used. For example, if a name space that agrees with users’ intuition were chosen, users would find it easier to locate desired bulletin boards. Ease of location in turn encourages users to follow only the topics they are interested in and to send notices to appropriate, specialized bulletin boards. With the resulting increased locality of use, based on geography or common interest, the growth rate of costs is reduced.

3.1. Flat Name Space

In some ways. a flat name space is attractive for its simplicity. However, the lack of structure combined with human laziness contributes to performance problems. Because names do not reflect relationships between topics, it is difficult to specify collections of topics in a way that will help automate the process of finding appropriate bulletin boards to read. Furthermore, the pain of locating the bulletin board for a topic, particularly in the absence of a directory listing facility, discourages users from sending to the most appropriate topic. Both of these factors discourage the subdivision of a bulletin board into subtopics when the volume of notices posted to it becomes excessive. This behavior causes the *Universal Bulletin Board Problem.*

Definition 1: A *universal bulletin board* has the property that the fraction of users at any node reading it is a constant, ρ, independent of the network size.

To obtain a rough estimate of the traffic to a universal bulletin board, two assumptions are made concerning the behavior of users:

1. Every reader posts P notices per time unit.
2. There are a users per node. For convenience, α is assumed to be sufficiently large that pa $>$ 1.

The following pair of propositions demonstrates the undesirable behavior.

Proposition 2: The number of notices posted to a universal bulletin board is $\alpha\rho NP$, a quantity proportional to N, the number of nodes.

Proposition 3: Every node receives at least one copy of the bulletin board.

Proof: The proof of Proposition 2 is trivial. To prove Proposition 3, note that each reader must see a copy of the notice. This means the reader’s node must have gotten a copy by an explicit read, a submission, or a pre-fetch for a local cache. So every node with a reader incurs the communication cost of obtaining a copy of every notice on the bulletin board. Since $\rho\alpha > 1$, every node has a reader and receives a full copy.

Since a flat name space can be expected to exhibit an excessive growth rate in communication costs, it was not adopted for Taliesin. Another potential choice is a general purpose database

3.2. General Purpose Database

Using a general purpose database, users can most precisely specify which notices are worth reading. The question is if too much performance is traded off for expressive power. This section discusses the matter in the context of a relational database [12].

3.2.1. Wasted Power

A bulletin board system requires only two relations: one used to store notices and one used to keep a record of last read times for each user. The simplicity of these relations limits the usefulness of the operators in relational algebra, especially join. Join is used to express queries that combine data from different relations. In a bulletin board system, it would primarily be employed to select notices based upon a last-read-time. This could easily be done in a user agent by reading a profile, then composing a selection clause. The other application for join is to express complex relationships between tuples, such as asking for all notices submitted by anyone who has posted a notice about the federal budget. No need for such complex queries was anticipated.

Selection, on the other hand, is essential in picking out notices relating to specified topic keywords. Intersection, set difference, and union can be transformed into selection except when combining results from an application of join. Projection has limited usefulness. Its two uses are to suppress unwanted fields in a relation or to simplify the intermediate tuples resulting from the application of the join operator. Since notices are relatively simple and join need not be used, it would suffice to support only a small number of predefined projections.

3.2.2. Costs to Avoid

A bulletin board system does not require the full expressive power of a database, but might benefit from it if the costs are not too high. Replication is needed to ensure fast response and high availability. Replicating entire relations inflicts selected nodes with storage costs proportional to the size of the network. Replicating or distributing fragments either results in network wide propagation of queries or introduces the problem of matching fragment contents to query topics. The latter is just a disguised way of creating and naming bulletin boards, not a solution to the naming problem.

A related cost is the reduction in response time resulting from copy synchronization. At first glance, posting a notice does not require any synchronization. However, a last-read-time must be reported with each read. Since the read-time must guarantee that the operation included every notice up to the reported time, every copy must be synchronized. This can produce a very poor response time on a large, frequently partitioned network.

3.3. Chosen Name Space

The structure of the chosen name space resembles a tree, but extensions to remove ordering constraints produce a directed acyclic graph. A bulletin board name consists of a set of unordered keywords. Each keyword has the syntactic form of a relative UNIX file name not starting with '.' or '$'. The topic of a bulletin board is given by the intersection of the keyword categories. To guide the efficient placement of copies, the keywords are tagged as specifying a location, an organization, or a topic.

Bboard 1

SITE :	USA
ORGANIZATION:	Stanford
TOPIC :	**pascal**, UNIX

BBoard 2

SITE:	USA
ORGANIZATION:	
TOPIC:	Pascal

BBoard 3

SITE:	USA
ORGANIZATION:	Stanford
TOPIC:	

The graphical nature of the name space can be seen by equating the bulletin boards with vertices. Edges are defined by the relation *DescendantOf*. DescendantOf(α, β) is true for a pair of bulletin boards, α and β, if the name of α is the same as the name of β but with the addition of keywords or path name components to the ends of existing keywords. In the example below, both DescendantOf(Bboard 1, Bboard 2) and DescendantOf(Bboard 1, Bboard 3) are true. Figure 3-1 depicts a sample naming graph, but with edges drawn only to immediate descendants.

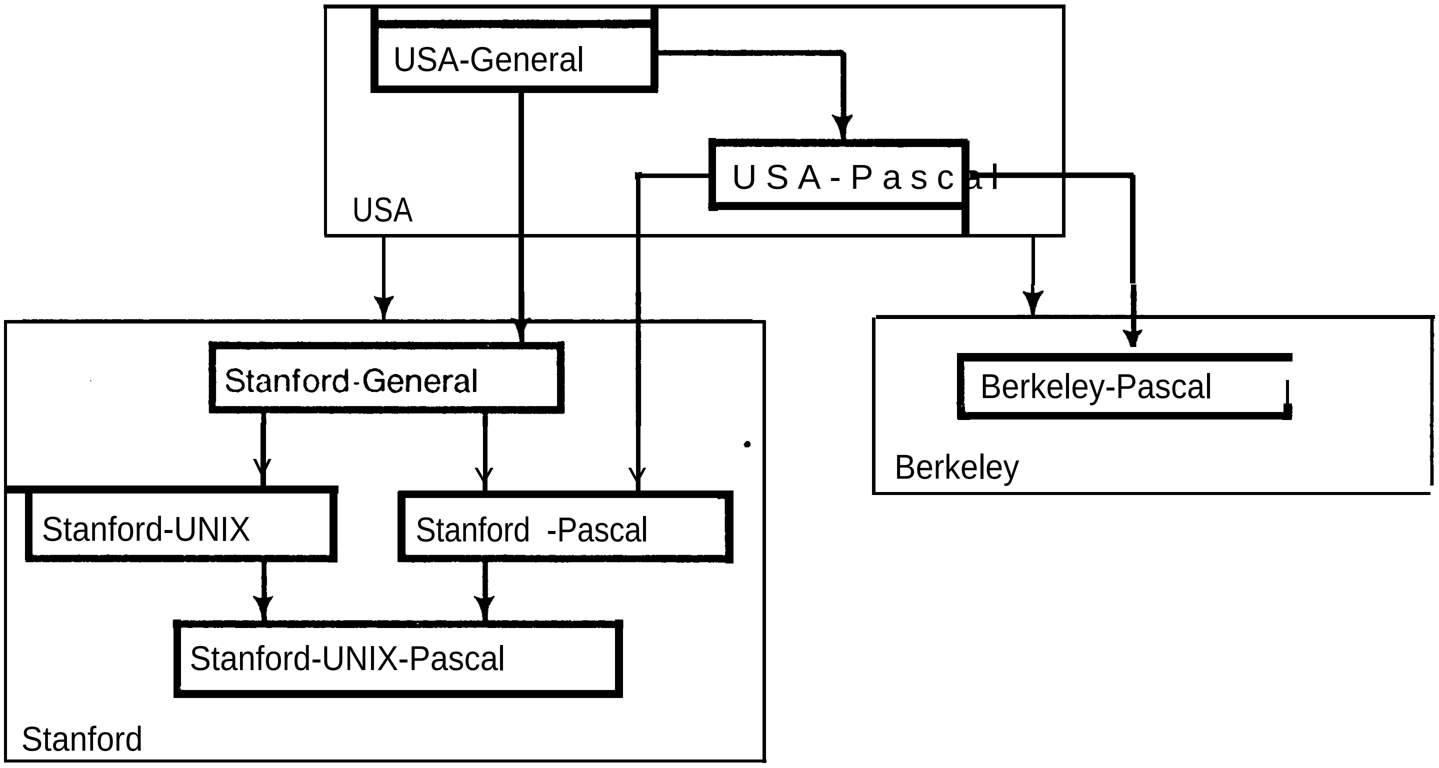

Figure 3-1: Example of a name space graph.

3.4. Costs of the Chosen Name Space

Unlike the flat name space, which offers scant hope of containing the growth of costs, the underlying tree-like nature of this name-space suggests the possibility of a growth rate roughly proportional to the log of the network size. A simplified analysis shows that this is indeed the case.

The first simplification is to group all bulletin boards having a common set of location keywords. Figure 3-1 shows a sample name space graph. The groupings are indicated by fine lines drawn about the original bulletin boards. If a user reads any bulletin board in a collection, he will be treated as reading all. Hence, the true costs will at worst be overestimated. To further simplify the analysis, it is assumed that the resulting tree is a complete, balanced, binary tree.

Grouping of bulletin boards in this manner sets up a correspondence between the content of the bulletin board and the interests of users in a geographical region. A leaf bulletin board might contain notices pertaining to scheduled downtime for the node or discussions about programs available just on that node. If the node were run by a particular department, the leaf bulletin board might also contain departmental announcements. The bulletin board at the next level up would cover a small collection of nodes, perhaps all in a small research group. It could then contain notices about seminars for the group or discussing the group's current project. At the root, the most general bulletin board would contain notices of potential interest to users at all nodes. These notices might discuss anything from new features of the bulletin board system to world politics.

The next step is to make some assumptions about user behavior. Users can have accounts on multiple nodes, but it is assumed that they are only interested in news applying to their primary login node. That is, they are interested in the corresponding leaf vertex and all its ancestors. Furthermore, it is assumed that users tend to submit notices applicable to their login node. Therefore:

1. Users follow one leaf plus its ancestors.
2. Users are equally divided among the leaf bulletin boards: α users each.
3. Users post P notices per time unit.
4. The number of notices a user posts to a bulletin board k levels removed from a leaf falls off as k increases: the number of notices sent to a bulletin board k levels removed is c 2^{-k} P. Actually, C, the "constant" of proportionality, is a decreasing function of the number of users.

U will denote the number of users and $P(U)$ the number of notices presented to a user's agent. As was explained in the discussion about bulletin boards of universal interest, $P(U)$ is a useful estimate of the communication cost.

$P(U)$ can be approximated based upon the assumptions above. To this end, a claim as to the size of bulletin boards will be proven. That claim will then be used to prove that the desired growth rate is achieved.

Proposition 4: The number of notices posted to each bulletin board is αcP, independent of U.

Proof: Define $D(U)$ to be the depth of the tree. There are α users per leaf node of the tree, implying the tree has U/α leaf nodes. Since the tree is a balanced, complete, binary tree, $D(U)$ is $1 + lg\ (U/\alpha)$, counting a tree with just one node as having depth 1. Consider a level k bulletin board (leaves are at level 0). Its subtree has 2^k leaves and hence is followed by $\alpha 2^k$ users. By assumption, each of these users posts $c2^{-k}P$ notices. The total number of notices sent to the bulletin board is then $\alpha 2^k * c2^{-k}P$, a constant.

Proposition 5: $P(U)$ is proportional to $1 + lg$ (U/a).

Proof: Each user only deals with the $D(U)$ bulletin boards in the leaf to root path traced starting with the local bulletin board. So, $P(U)$ for a user must be the sum of the sizes of these bulletin boards. $P(U)$ is thus proportional to $1 + lg\ (U/\alpha)$.

4. Replication Mechanism Design

In a distributed bulletin board system, several objects must be replicated. These are notices, the bulletin board descriptors containing protection and replication information, user profiles retaining last-read-time information, and the name-to-location mappings. Each of these object classes has different performance needs.

The performance savings due to a special purpose replication mechanism determines whether or not the design performs better than a general database in terms of response time and coping with network partition. A good replication mechanism will also resolve several other problems. By clearly defining a method to make consistent copies, it will eliminate the frustration of dealing with inconsistent copies. The mechanism will also lessen the need for nested distribution lists with their attendant problems.

4.1. Performance Goals

Since users must wait for the bulletin board system to respond to any read operation, the replication mechanism must respond quickly to read requests. On the other hand, updates in general do not require fast response. Lapses on the order of a day or more will probably be tolerable for bulletin board descriptors and name mappings. Updates to user profiles should be visible the next time a user reads the bulletin board system. Posting notices is the most time-critical update. Ideally, newly posted notices should become visible immediately at a local copy of a bulletin board.

Another issue is availability. User profiles must have very high availability since they are the key to doing everything else. Fortunately, profiles will typically be accessed only from a handful of nodes corresponding to users' normal login sites. Name mappings also require very high availability, but they are used at a large number of sites, not just at the locations of the corresponding objects. Bulletin boards proper have the least critical need for high availability. Still, network partition should not interfere with reading or posting to a bulletin board as long as a copy is available.

Because of the performance demands associated with notice operations, a special replication algorithm was designed for notices. The other objects are kept in a consistent state by means of a slightly modified version of the majority vote scheme described by Thomas [11]. Only the notice replication algorithm is described in this paper.

4.2. Read-Time Representation

A factor complicating the replication of notices is the need to save a last-read-time. The last-read-time must identify what has been seen using a modest amount of storage. There are three basic approaches. One is to save an identifier for every notice presented to each user. This does provide an exact record of what has been seen, but the space requirement grows with time. Another commonly used method is to impose an ordering on notices based on their posting times. To allow users to read different copies of a bulletin board, the ordering must be consistent across copies. This forces synchronization on every read, a potentially expensive operation. The third alternative is to save one quantity per bulletin board which will change when a new notice is submitted but which does not represent an ordering on notices. This uses little space, but poorly represents what has been seen. The entire contents must be read every time the bulletin board is changed.

To avoid the flaws in each of these approaches, a slightly different tack was taken. Time is measured by the passage of epochs. Each copy has two pointers into the epoch time scale. The first, the *posting-epoch*, represents the "current" time at the copy. The other, the read-epoch, marks the beginning of potentially inconsistent data. Last-read-times are represented by an epoch identifying the stable portion of the copy read plus a list of identifiers for notices falling in the unstable zone.

4.3. The Algorithm

A read operation begins when a user submits a request containing last-read-time information. The bulletin board service locates a copy of the bulletin board and selects notices dated after the supplied read-epoch. Any not seen before are reported to the user. Finally, the read-epoch of the copy is reported and the copy enters a new epoch by incrementing its posting-epoch.

To post a notice, it is submitted to any bulletin board server, which assigns it a unique identifier. The notice is forwarded to a server with a copy of the bulletin board. At this point, the notice is labeled with the copy's posting-epoch and stored. Later, the reconciliation operation will do the replication of the notice.

Reconciliation is performed in the following steps:

1. Check agreement on descriptor.
2. Exchange notices dated after the read-epochs.
3. Exchange current posting-epochs and advance them to 1 + maximum of the old values.
4. Exchange lower bounds on the posting-epoch for all other copies.
5. Advance the read-epoch to be the latest epoch known to be less than every copy's posting-epoch.

This algorithm ensures that any two copies agree on what notices belong to any epoch up through the minimum of their read-epochs. It does not require synchronization between copies at the time of reading or posting. Notice operations are not disabled by network partition so long as one copy is available. While the read-epoch and posting-epoch of every copy remain close, the storage for the last.-read-information remains bounded. Thus, the algorithm does a good job of meeting the performance goals. However, advancing the read-epoch requires a consensus of all copies. Network partition will halt the advance of the read-epoch but not the posting-epoch. A method for alleviating this problem will be discussed in Section 4.5.

4.4. Algorithm Interactions

Notice replication uses knowledge of existing copies that is changed by the descriptor replication scheme. This requires coordination between changing the set of copies and the advancement of the read-epoch. Both replication algorithms must undergo some minor modifications.

Whenever a new copy is created, an effective creation-epoch must be assigned. Correct behavior is obtained by setting the creation-epoch to be the posting-epoch of the copy kept by the server accepting responsibility for the request. The new copy starts with a posting-epoch equal to its creation-epoch. Propagating outstanding votes as part of the reconciliation process ensures that no copy will claim to have a consistent version of the creation-epoch without knowing about the new copy.

Destroying a copy is done in a similar fashion, although the destruction-epoch does not need to be the posting-epoch of the initiating copy. The copy that is being destroyed does not necessarily know to stop assigning its epoch numbers to notices after the destruction-epoch. So, every such notice must be reassigned to another epoch. The reassignment can be handled in a manner similar to that of posting notices, but requires that notices carry the identity of the copy at which they were assigned an epoch number.

4.5. Coping with Partition

The notice replication mechanism copes very well with partition except that if one copy remains unavailable, the read-epochs of the remaining copies will fall far behind their posting-epochs. A solution is to to demote unavailable copies to the status of caches: i.e., copies with 0 votes. Such copies can still be used for reading and to help propagate postings and updates. However, they cannot be allowed to assign epoch numbers to notices or copy creation requests. Demotion from a positive number of votes to zero is handled much like destruction while promotion from zero votes to a positive number is handled like creation. The majority vote scheme provides a sufficient degree of synchronization to ensure that partitioned copies cannot mutually demote one another.

5. Other Facilities

The design of a bulletin board service clearly encompasses more than just a name space and a replication algorithm. A variety of other facilities, such as protection and notice deletion, were developed but will not be presented here. An extension to the read operator is sufficiently interesting to warrant inclusion, however.

To partially automate the task of reading a group of related bulletin boards, the read operator was augmented to allow a client to indicate a collection in terms of the previously discussed DescendantOf relation. In the final design, there are six types of searches. For each of the three categories of keywords, one may chase links either forward or backward. This extension implies that an array of last-read-times must be returned, one entry per bulletin board scanned.

6. The Prototype Implementation

Taliesin exists as more than just a design. A prototype has been implemented on a collection of Sun workstations interconnected via an Ethernet. It runs under the V-System, a message-based distributed operating system [4, 7].

6.1. Agents of the Taliesin System

Taliesin is organized as a distributed collection of agents. User agents provide the user interface. *Postmasters* provide mail storage and transport facilities. This separation of user agent and postmaster is common in mail system design [3, 6].

A third type of agent, the *universal directory* server (UDS), provides name service. Name service was split out of the postmaster into a distinct server for several reasons. First, the replication policies of name mappings, particularly for copy placement, differ from those for notices. Isolating the naming facilities promised to ease the impact of the inevitable design modifications. Second, the UDS provided a common mechanism to handle not just the naming of bulletin boards, but also of user profiles and of users independent of their accounts on particular nodes. Finally, the authors simply were interested in exploring some of the problems in writing a general purpose name server. Other designs have for similar reasons separated name service from mail service [3, 9]. An example configuration of the three types of agents discussed thus far is shown in Figure 6-1, where each agent could be running on a different node.

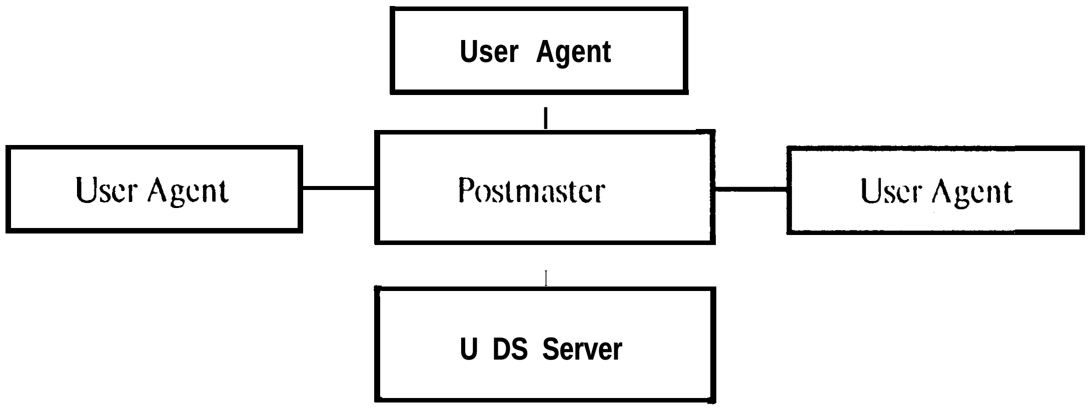

Figure 6- 1: A sample set of agents.

In addition, for ease of debugging and testing, a fourth agent, the Nexus, was introduced. It is capable of monitoring the message traffic between agents and of simulating multiple nodes on one (real) node. The latter facility has allowed us to simulate network partition without resorting to disconnecting Ethernet interfaces and the like.

6.2. Name Service Interactions

While a full description of the UDS is beyond the scope of this paper (see [8]), a few points are of interest here. Like all name services, the UDS maps object *names* to object *identifiers.* The string names used for objects in the UDS are like those of UNIX file names, except that they are prefixed by '%', not an initial '/'. The object identifiers that the names map into are *(manager-id, manager-specific-id)* pairs. The manager-id uniquely identifies the server that implements the object. The manager-specific-id is a variable length sequence of bytes, uninterpreted by the UDS, that identifies the object to the server.

The postmaster and the UDS must cooperate to name bulletin boards. Postmasters use the UDS to name bulletin boards and user profiles. These object types are not hardwired into the UDS. Rather, the implementation uses the type independence of the UDS. However, a mapping from the graph-structured bulletin board names to the tree-structured UDS names must be defined. First, the canonical form of the bulletin board name is produced by sorting the keywords alphabetically within category. For each category, a single string is constructed by starting with a token for the category, then the keywords are concatenated in order prefixed by '/.'. Finally, the UDS name is constructed by starting with a '%' then concatenating the strings for each category in a predefined order separated by '/'. A sample translation is shown below. The mapping can be reversed to obtain the canonical form of the bulletin board name.

BBoard Name:

SITE:	
ORGANIZATION:	**stanford/dsg**
TOPIC:	vgts, graphics

UDS Name:

```
%$G/$P/.stanford/dsg/$T/.graphics/.vgts
```

Finally, the UDS must provide the facilities to enable postmasters to identify the ancestors and descendants of given bulletin boards. Names of potential ancestors can be enumerated, but those of descendants can only be found by searching the name space. Initially, the UDS recognized a token that allowed wild-carding within a directory. Finding all the names of descendants required repeated queries and so yielded a very poor response time. The UDS was modified to recognize a new token that allowed wild-carding across directory boundaries. This produced a major improvement in response time.

6.3. Status

At the present time, postmaster and UDS servers offer nearly the full range of functionality described in their designs. However, neither replication nor forwarding between servers has been implemented. Two user agents are available, one that offers access to all the functions of the Nexus, UDS, and postmaster, but it is almost user hostile, and another that is considerably more user-friendly.

7. Concluding Remarks

One of the most interesting observations made in the process of designing Taliesin is that the choice of a name space is the key to controlling the growth rate of costs. So far, design looks feasible, but speed is a potential problem, especially in the wild-card searches. Taliesin has not yet been released to a user community, so no statistics on its actual behavior have been gathered as yet. Nevertheless, the analyses presented here promise improvements in the growth rate of costs and minimal degradation of service in the event of network partition.

Acknowledgments

Joseph Pallas implemented the "friendlier" user agent.

References

1. Anonymous. *MM User's Manual.* 1981.

2. Anonymous. *UNIX Programmer's Manual.* 1985. On-line documentation for readnews, checknews, and postnews.

3. A. Birrell, R. Levin, R. Needham, and M. Schroeder. "Grapevine: An exercise in distributed computing". *Comm. ACM* **25, 4** (April **1982), 260-274.** Presented at the 8th Symposium on Operating Systems Principles, ACM, December 1981..

4. D.R. Cheriton. "The V Kernel: A software base for distributed systems". *IEEE Software* **1, 2** (April **1984),** 19-42.

5. C. Daney. The VMSHARE computer conferencing facility. **Proc.** International Symposium on Computer Message Systems, **IFIP,** 1981, pp. 329343.

6. D.P. Deutsch. Design of a message format standard. **Proc.** International Symposium on Computer Message Systems, **IFIP,** 1981.

, , 7. Distributed Systems Group. *V-System Reference Manual.* Distributed Systems Group, Department of Computer Science, Stanford University, 1985.

8. K.A. Lantz, J.L. Edighoffer, and B.L. Hitson. Towards a universal directory service. **Proc.** 4th Symposium on Principles of Distributed Computing, ACM, August, 1985, pp. **250-260.**

9. P. Mockapetris. Domain names: Concepts and facilities. RFC 882, Network Information Center, SRI International, September, 1983.

10. M.D. Schroeder, A.D. Birrell, and R.M. Needham. "Experience with Grapevine: The growth of a distributed system". *ACM Transactions on Computer Systems* 2,1 (February **1984),** 3-23. Presented at the 9th Symposium on Operating Systems Principles, ACM, October 1983..

11. R.H. Thomas. "A majority consensus approach to concurrency control for multiple copy data bases". *ACM Transactions on Database Systems* **4, 2** (June **1979), 180-209.** Earlier version appeared as Technical Report 3733, Bolt Beranek and Newman, December 1977..

12. J.D. Ullman. *Principles of Database Systems.* Computer Science Press, 1982.