# An Architecture for Large Scale Information Systems

David K. Gifford, Robert W. Baldwin, Stephen T. Berlin, John M. Lucassen

MIT Laboratory for Computer Science
545 Technology Square
Cambridge, MA 02139

A new type of system architecture is described that uses both duplex communication and wide-area simplex communication to implement a single service. A working community information system based on this architecture is discussed from a systems perspective, with an emphasis on the unique way in which processing is distributed among a confederation of shared servers and private personal systems.

In the community information system, each personal system maintains a local, user-defined subset of the databases stored on the shared servers. Database updates are transmitted to the personal systems via a broadcast packet radio system. This design allows many queries to be processed completely at users' personal machines, and thus reduces the reliance on shared servers.

A unifying design principle is that the system is seen as a collection of independent shared and personal databases, as opposed to a single monolithic database. Query routing is used to hide the system's division into component databases from a user.

CR Categories and Subject Descriptors: C.2.4 Computer Communication Networks — *Distributed Systems*, H.3.4 Information Storage and Retrieval — *Systems and Software*, H.4.3 Information Systems Applications — *Communications Applications*, D.4 Operating Systems.

Additional Key Words and Phrases: Community Information System, Multi-Database, Predicate Data Model, Broadcast Packet Radio, Query Routing, Remote Procedure Call

(This work was supported in part by DARPA/ONR contract number N00014-83-K-0125)

## 1. Introduction

This paper introduces a new architecture for large scale information systems. The architecture combines personal computation, broadcast digital communication, two-way communication, and centralized mass storage in a unique way. In this architecture an information system is seen as a collection of independent shared and personal databases, as opposed to a single monolithic database. Query routing is used to hide the division into component databases from the user.

To test our architectural ideas, we have implemented an experimental large scale community information system that is currently operating at the Laboratory for Computer Science at MIT. The system gives our users throughout the Boston area access to a variety of information sources, including the *New York Times*, the *Associated Press News Service*, and several electronic mailing lists.

Our research anticipates a time when most information will be communicated to the home and office by digital means instead of on paper. Digital delivery will make a wider range of information sources available to everyone, and computers will provide the necessary power to filter and process the large volume of information that will be received. These future systems may also be used to contribute information to public databases, as well as for banking, shopping, and other transactions.

Our system design goals include:

- Economy: the system should be able to serve an entire metropolitan area cost-effectively
- Ease of use: the system should have a high-quality user interface that is easily mastered by naive users
- Privacy: the privacy of individual users should be safeguarded
- Protection: the services provided by the system should be available to authorized users only
- Autonomy: users should be able to process information drawn from the system in any way they desire
- Scale: the system should make access to very large databases possible
- Flexibility: it should be easy to add new applications to the system's existing infrastructure.

Our approach to these goals has been to utilize personal computers for most processing tasks, thereby minimizing reliance on centralized servers. Our *Community Information System* consists of two major components: a set of shared servers, operated at a central location, and a large number of personal computers, one for each user. The personal computers help us meet our objectives of economy, ease of use, privacy, protection, and autonomy, while the shared servers provide the traditional advantages of direct access to a large database. To ensure flexibility, the system has been designed with modularity in mind.

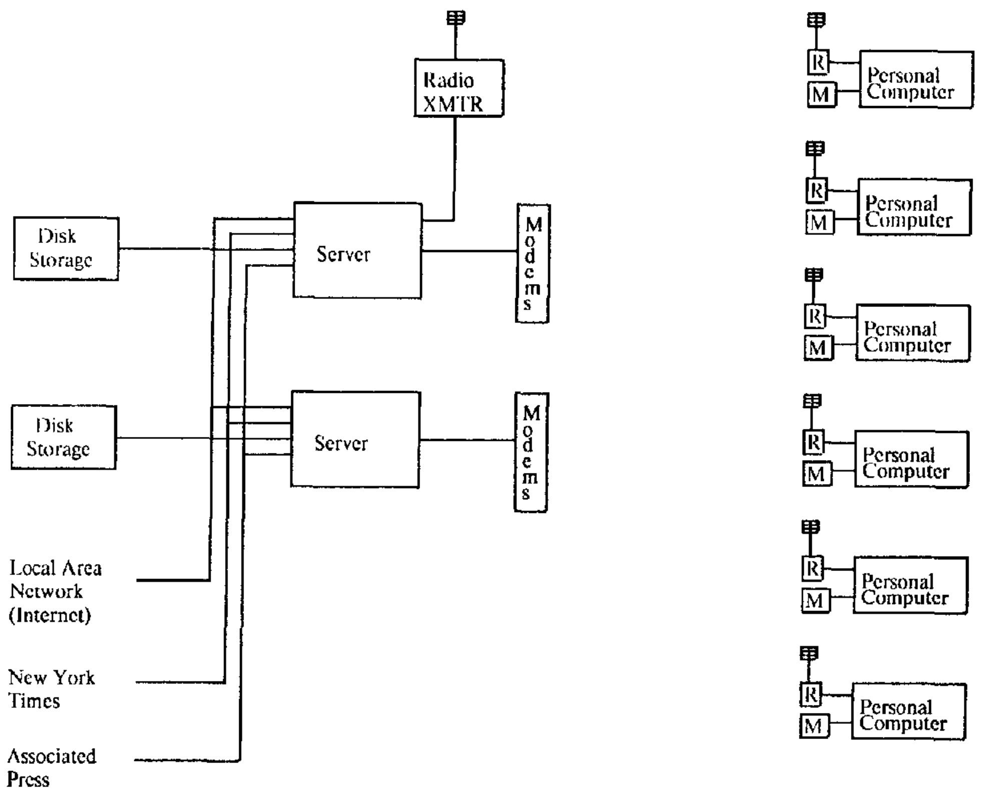
In view of our design goals, we decided that the personal computers should be capable of processing the most common user requests completely on their own, without resort to any shared servers. To this end, we have built a personal database system that runs on a personal computer and is designed to offer a high-quality user interface, while operating autonomously as much as possible. Although a personal computer can not hold all the available information, it may be capable of holding the information that a particular user is most likely to request. To help ensure that most user requests can be processed locally, each user specifies what information should be kept within his personal database system. When a request can not be processed by the personal system alone, shared servers are utilized. This use of shared servers is invisible to the user.

The integration of personal computers and shared servers into a single database environment poses a number of interesting research issues. To insulate our users from the distributed nature of the system, the user is presented with a 'single system image': the system appears to the user as a single, large database. At the same time, we decided to make every database autonomous, so that the overall system consists of a large number of independent databases. Database content descriptions are used to route each request to an appropriate database or set of databases.

Our system is designed to be adaptable to technological advances. Thus, our design is not based on specific communication channel characteristics or specific personal computer technology. As technology develops, we expect to be able to upgrade our system accordingly, thereby improving its functionality and performance.

We decided not to rely upon high bandwidth communication within a community. This presented us with the challenge of isolating users from the effects of low to moderate bandwidth communication (1K-10K bits/sec). Our design for the personal database systems seeks to minimize the impact of bandwidth limitations by means of a multi-process structure that enables the system to perform communication tasks in the background while responding to user requests at the same time.

An overview of our prototype system is shown in Figure 1. The shared servers shown in the figure are located at the MIT Laboratory for Computer Science. Each server contains one or more independent databases, unified by a query routing module. Some databases are replicated on more than one server to improve availability and performance. When a shared server receives new information, it is queued for transmission to the personal systems via a broadcast packet radio system. The type of each database item determines both the number of times that the item is transmitted and the key (if any) with which the transmission is encrypted. The broadcasts are received by the personal systems, where a background task uses them to update the local database. The personal database system processes queries from the user; when it does not have the information needed to process a user request, it automatically establishes a connection to the appropriate shared server.



**System Block Diagram**

**Figure 1**

The remainder of this paper is organized as follows. Section 2 introduces the predicate data model and query routing, in which our single system image is based. Section 3 presents the personal database system in detail, and provides a brief overview of its user interface. Section 4 presents the design of the shared servers. The next two sections review related work (Section 5), and system performance and project status (Section 6), and we end with some conclusions about our approach and directions for future work (Section 7).

## 2. The Predicate Data Model

To organize our system, we have developed a new data model which we call the *Predicate Data Model*. The predicate data model is designed for information retrieval applications, and incorporates many ideas from contemporary information retrieval systems. The predicate data model is less powerful than the relational model in many respects (for example it does not support join), but it provides full text searching capabilities that the relational model does not.

A *Predicate Database* consists of a set of records. Each record consists of a number of named fields. There are required fields (such as TYPE and DATE), as well as optional fields. A predicate database can be mutable (*i.e.* records can be altered), immutable (*i.e.* the database can never be altered), or append-only (*i.e.* records can be added to a database but existing records are immutable). To date we have implemented append-only and immutable databases.

The fundamental operation on a predicate database is to restrict attention to a subset of the records in the database. A specification of a subset of the database is called a *query*, and the computation of a database subset is called *query processing*. Once a subset has been selected, the records contained in the subset can be retrieved or deleted.

The predicate data model provides users with a great deal of flexibility in the formulation of queries. This is important in a community information system, where users know certain things about the records they seek, but are unable to produce a single, unique key that identifies the information of interest. The predicate data model permits users to express what they know about the records they are seeking by defining a predicate that matches each such record.

In the predicate data model, *predicates* and *result sets* are defined as follows. A predicate function, or *predicate*, returns TRUE or FALSE when applied to a record, depending on the content of the record. A *query* is a Boolean combination of predicates, and is therefore itself a predicate. A record is said to *match* a query (and vice versa) iff the query predicate returns TRUE when applied to the record. A record **x** is in the *result set* of a query Q iff it matches the query Q. In our present database system, predicates permit selection on the basis of:

- The record type (corresponding roughly to the source of the information);
- The date and time a record was inserted into the database (range queries on dates and times are supported);
- The presence of arbitrary words or phrases in the record or in specific (textual) record fields, such as the SUBJECT, AUTHOR or PRIORITY field.

For example, here are some queries that users of our system might type:

```
(type: times) & (date: [date sep 17])
(subject: movie review) & (author: smith)
(category: financial) & (ibm | apple)
```

The predicate data model has two main advantages over traditional data models. First, it is suitable for dealing with text and other semi-structured information that can not be easily indexed within the framework of more traditional approaches. Research suggests that predicate based approaches may be preferred by novices and experienced users alike [2].

The second advantage is more relevant to this paper: the predicate data model provides a framework for reasoning about the *content* of databases. Most traditional systems make a 'closed world assumption': if a piece of information is not found in a database, then it is assumed not to exist. This assumption is not appropriate in our system, where there are many databases, and where no single database necessarily contains all the available information.

To determine where a query can be processed, we must be able to reason about the content of databases. Figure 2 illustrates the kinds of problems we are trying to solve, with two databases and three queries expressed as a Venn diagram. Query Q1 describes a potential result set that is only partially contained in DB1, but is contained entirely in DB2. Thus Q1 can be processed at DB2. Query Q2 describes a potential result set that is only partially contained in DB2. In this case we can not guarantee that the result of processing Q2 at DB2 will find all information of interest. Our approach is to have the system suggest another query to the user, Q2', which represents the intersection of DB2 and the potential result set of Q2. If the user confirms, this query Q2' will be processed at DB2. The third and final case is represented by query Q3, which is disjoint from both DB1 and DB2. This query can not be processed using only DB1 and DB2, and is therefore rejected. To summarize, there are three cases:

- If $Q \subseteq DB$, the query Q can be processed at a database with predicate DB.
- If $Q \cap DB = \emptyset$ for all database predicates DB, the query Q is rejected.
- Otherwise, the modified query $Q' = Q \cap DB$ (for one or more appropriate database predicates DB) is proposed to the user as an alternative query.

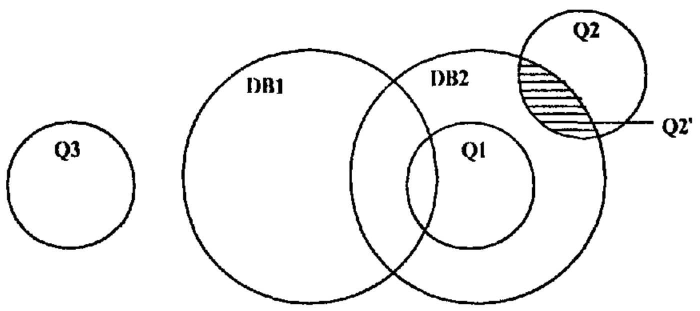
The ability to reason in this manner about the queries presented to our system is central to our strategy for query processing. The query language itself is used to formally describe the content of each database. Content descriptions and queries are formalized in a first order theory that has an efficient decision procedure.

Let DB be the description of a database, phrased in terms of predicates. For example, DB could be a formalization of the statement that a database contains all information from the *New York Times* for the month of November, 1985. In our query language, this would be expressed as:

```
(type: times) & (date: [date nov 1985])
```

To show that a query Q can be processed at a given database **DB**, we must show that $Q \subseteq DB$, where Q denotes the potential result set of the corresponding query, and DB denotes a database. To this end, it is sufficient to show that the statement $Q(x) \Rightarrow DB(x)$ is true for all **x**, *i.e.* that $Q(x)$ implies $DB(x)$ for all x. By checking the truth of similar statements, we can determine when there is an intersection between **DB** and the potential result set of Q, and when they are disjoint. When the potential result set of Q is not contained within DB, we attempt to add the minimum number of terms to Q that will make its potential result set a subset of **DB**.

The Predicate Data Model allows us to view several 'simple' databases as a single 'composite' database. As we described above, the content of



**Database Descriptions and Query Result Sets**

**Figure 2**

a database can be described by a Boolean combination of predicates. This means that the joint content of a *set* of databases is described by the disjunction (logical OR) of the expressions that describe the component databases. Given any query that falls within the scope of a composite database, the result set can be obtained by submitting the query to each of the component databases, and taking the union of the results obtained.

The expressions that describe the component databases can be used to determine which of them contain information needed to process a given query. This gives rise to the concept of *query routing*. Rather than submitting the query to all the component databases, it suffices to submit it to any set of components for which the disjunction of the content descriptions is implied by the query.

A similar procedure can be used when adding records to a composite database. A given append request could be applied to each component database, but it suffices to apply the request to only those components whose description predicates match the record being added.

In our system, each server is implemented as a composite database. Within the server, information is assigned to the component databases based on its type and on the time of acquisition. Every 24 hours the currently active databases are converted from append-only to immutable status, and a new set of databases is created. Because each database has a specific content description, a database can be taken off-line without affecting the system's ability to describe its own content. The composite database manager must note the change, and update the predicate that describes the server's content. Likewise, new databases can be added to the system without causing disruption.

From the outside, a composite database is indistinguishable from a simple database. We will refer to the composite databases stored on the servers simply as 'databases'.

## 3. The Personal Database System

The personal database system is an integral part of the Community Information System. It runs on each user's personal computer, and implements the user interface to the rest of the system. In this section we will first describe the functions of the personal data base system, and second, how these functions are implemented.

### 3.1. Personal Data Base System Functions

The personal database system performs two tasks concurrently: it processes user requests, and it applies database updates received over the digital broadcast channel to the local database. The details of the transmission medium and packet protocol are described in [3].

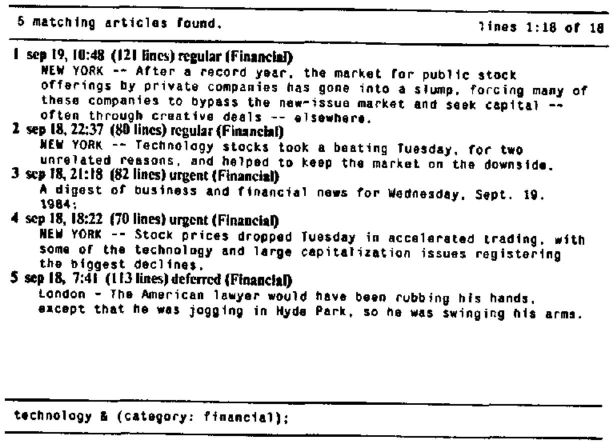
Figures 3 and 4 illustrate how information from the database is presented to the user. Figure 3 shows a set of article summaries resulting from the processing of a query; Figure 4 shows an article that has been selected for display.

To meet the goals we outlined in Section 1, we have designed the system so that a user's most frequent requests can be answered from the user's personal database. To this end a user compiles a list of routine queries into what is known as the *filter list*. The queries in the filter list are disjunctively combined (OR'ed together) to create a predicate called FL (for filter list) that describes the information that will be retained at the user's personal computer. The personal database that results is precisely the set of records necessary to process any query in FL.

The predicate FL may describe more information than can be stored on the user's personal system. When this occurs, the system must make a choice among the records that match FL, deciding which records to keep and which to discard. To deal with this, we let the user list the component queries of FL in order of importance, and ask him to specify a 'budget' for each query in FL. This 'budget' indicates how many records matching the query should be retained. Each record in the personal database is associated with the most important query in FL that matches it. If a new record arrives that matches FL and the personal database system has insufficient resources to store it (such as disk storage or main memory), the personal database system will try to make room by deleting records from queries in FL that are "over budget". If there still is insufficient room for the new record, the personal database system will delete records from queries in FL that are less important than the most important query that matches the new record.

Because the system does not necessarily keep all records that match FL, and because a user's personal computer may miss certain updates (when it is turned off or when there are uncorrectable channels errors), FL does not accurately describe the contents of the local database. A solution to this problem, which we have not implemented, would be to maintain a separate predicate, PDB, that describes the local database exactly. PDB could be obtained by conjoining each query of FL with an additional predicate that describes a set of time intervals for which the local database has a complete set of database updates.

At the user's request, the personal database system will display the filter list. The user can easily change the filter list, or instruct the system to process one of the queries in the filter list. By allowing the users to select preplanned queries, the system inherits many of the advantages

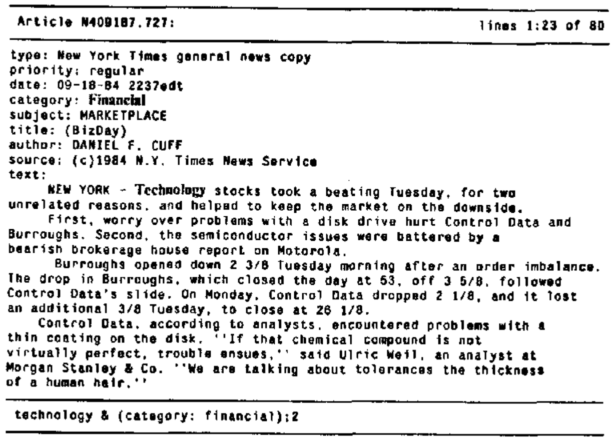


```
5 matching articles found.                                  lines 1:18 of 18

1 sep 19, 10:48 (121 lines) regular (Financial)
   NEW YORK -- After a record year, the market for public stock
   offerings by private companies has gone into a slump, forcing many of
   these companies to bypass the new-issue market and seek capital --
   often through creative deals -- elsewhere.
2 sep 18, 22:37 (80 lines) regular (Financial)
   NEW YORK -- Technology stocks took a beating Tuesday, for two
   unrelated reasons, and helped to keep the market on the downside.
3 sep 18, 21:18 (82 lines) urgent (Financial)
   A digest of business and financial news for Wednesday, Sept. 19,
   1984;
4 sep 18, 18:22 (70 lines) urgent (Financial)
   NEW YORK -- Stock prices dropped Tuesday in accelerated trading, with
   some of the technology and large capitalization issues registering
   the biggest declines.
5 sep 18, 7:41 (113 lines) deferred (Financial)
   London - The American lawyer would have been rubbing his hands,
   except that he was jogging in Hyde Park, so he was swinging his arms.

technology & (category: financial);
```

Typical Article Summary Display
Figure 3



```
Article N409187.727:                                        lines 1:23 of 80

type: New York Times general news copy
priority: regular
date: 09-18-84 2237edt
category: Financial
subject: MARKETPLACE
title: (BizDay)
author: DANIEL F. CUFF
source: (c)1984 N.Y. Times News Service
text:
     NEW YORK - Technology stocks took a beating Tuesday, for two
unrelated reasons, and helped to keep the market on the downside.
     First, worry over problems with a disk drive hurt Control Data and
Burroughs. Second, the semiconductor issues were battered by a
bearish brokerage house report on Motorola.
      Burroughs opened down 2 3/8 Tuesday morning after an order imbalance.
The drop in Burroughs, which closed the day at 53, off 3 5/8, followed
Control Data's slide. On Monday, Control Data dropped 2 1/8, and it lost
an additional 3/8 Tuesday, to close at 26 1/8.
    Control Data, according to analysts, encountered problems with a
thin coating on the disk. ''If that chemical compound is not
virtually perfect, trouble ensues,'' said Ulric Weil, an analyst at
Morgan Stanley & Co. ''We are talking about tolerances the thickness
of a human hair.''

technology & (category: financial);2
```

Typical Article Display
Figure 4

of menu oriented systems.

When a user selects a query from the filter list, this query is placed in the query input line as if it had been typed by the user. The user is free to edit the query input line before submitting the query. The user may also submit arbitrary queries that do not appear in the filter list.

The task of deciding how and where to process a query is called query routing. To perform query routing, the personal system needs the descriptions of all available databases. Given a query, query routing determines which databases could process the query. From this set, we may select, for example, the database that has the lowest estimated communication cost. If the selected database is unavailable, the next lowest cost database may be selected.

A user's personal database system maintains a description of available remote databases. At present, the descriptions of the available databases are fixed. However, we plan to include database descriptors in the database itself. The user's filter list predicate FL would then implicitly contain a term matching any record that contains a database description. Thus, descriptions of remote databases would be kept up to date automatically.

The current implementation of the personal database system is limited to information retrieval. We plan to add facilities that will allow users to update remote databases interactively.

## 3.2. Implementation of the Personal Database System

Figure 5 shows the internal organization of the personal database system as implemented for the IBM-PC family of computers. The modules shown are organized into two processes. One process monitors the keyboard and the mouse, processes user requests, and writes to the display. A second process receives the incoming stream of database updates from the packet radio system, and applies them to the local database as necessary. A non-preemptive scheduling discipline is used: the receiver process must yield at regular intervals to give the user interface process a chance to run, and vice versa.

We will not describe the system's modules in detail, but we will discuss their interrelations. The window manager reads from the keyboard and mouse, passes completed commands to the database user interface, and updates the display as requested by the database user interface. The database user interface is responsible for implementing all commands, and for formatting database records for display. The query routing module is in charge of processing queries. As described earlier, query routing uses the filter list and descriptions of remote databases to decide where a query should be processed.

The local database is stored on disk, and an index is maintained in primary memory. The amount of data that can be kept in the local database depends on the amount of storage available. An average news article requires 5K bytes of disk storage, and approximately 470 bytes of main memory. This results in an approximate capacity of 40 news articles for a system with a 320 KB floppy disk, and 250 news articles for a system with 512K bytes of memory and a hard disk.

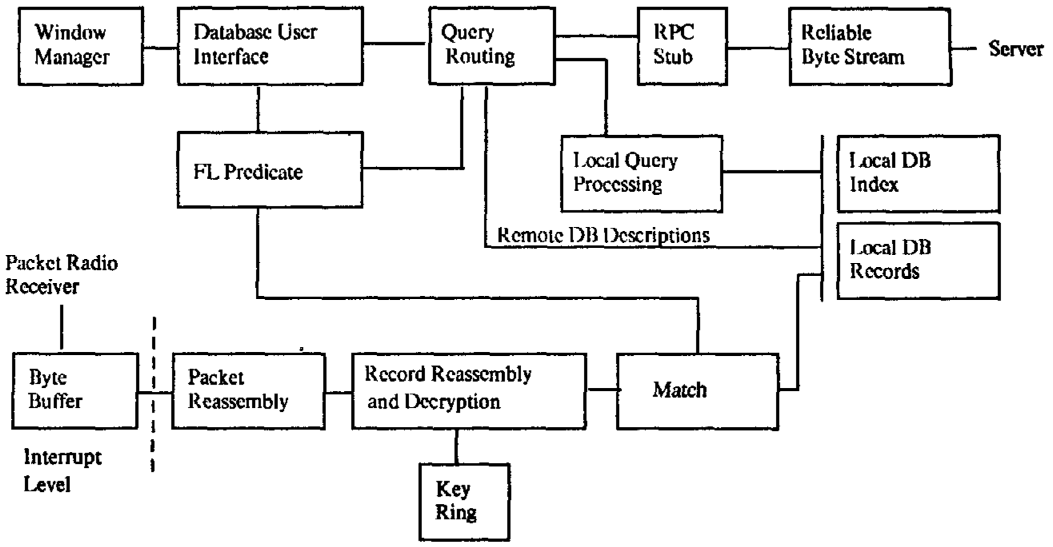
Our protocol for access to shared database servers is based on remote procedure calls. Table I shows the interface to a shared server. The interface is designed to be implemented using remote procedure calls; however our current implementation does not actually export the procedures shown in the interface. We plan to explore how we could automatically generate a true RPC stub for the procedures in Table I.

The operations in Table I fall into two classes:

1. Connection Management: **Connect** is used to establish a connection with a server. **Connect** is passed the name of a server, and it returns a completion status. The interface only supports a single connection at a time. **Disconnect** closes the currently open connection. **Abort** can be called while an RPC in progress; the RPC in progress will be immediately terminated and return **aborted** as its status.
2. Database Requests: The server interface is designed to permit query processing at a server to occur concurrently with other client processing. The procedure **EstablishQuery** initiates processing of a query at the server, and then returns immediately to the client. **CountMatchingRecords** returns the number of records that have matched a query so far, and a flag indicating if this count is final. **FetchSummaries** is used to return summaries of specified records, while **FetchRecord** is used to obtain specified lines from a specific record.

There are three innovations in our model of remote procedure call. They are:

1. Backcalls for incremental results: In our RPC model clients can pass procedures to servers. When a server applies such a value, a remote



**Internal Organization of the Personal Database System**

**Figure 5**

procedure call from the server to the client will result. These backward remote procedure calls are named *backcalls*. Backcalls are used in our application to provide results to a user as soon they are computed. For example, FetchSummaries and FetchRecord will return data to a client incrementally, by calling the procedure passed as the deliver-to formal parameter. In addition to providing results as they are computed, backcalls also permit bulk data transfer between a server and a client within a remote procedure call framework.

2. Explicit Flow Control: Clients usually have limited buffering capability. To permit explicit flow control, FetchSummaries and FetchRecord have parameters that permit the client to specify how much data should be returned.
3. RPC Abort: We permit an RPC in progress to be aborted. In our application, the personal database system will abort a FetchSummaries or FetchRecord request when a user decides to issue a new command without waiting for the previous request to finish. If Abort did not exist, the client would have to wait for each request to finish before issuing a new request. Because RPC requests can require a long time to finish, waiting for completion would be unacceptable.

The RPC stub uses a byte stream module to communicate with a shared server. The byte stream module uses an autodial modem to establish a server connection without user intervention.

The reception of database updates via the packet radio system is handled by the set of modules at the bottom of Figure 5. Bytes arrive from the packet radio receiver at 4.8K bits/sec, and are placed in a 5K byte ring buffer at interrupt level. The size of the buffer accommodates a service latency of up to 10 seconds. Such latency can be caused by activity in the user interface process.

The receiver process polls the byte buffer and reassembles records out of individual packets. Each packet contains information for error detection and error correction. To mask channel errors, packets are transmitted more than once, and these transmissions are separated in time. When a previously unseen packet arrives, it is copied into its proper place in the record reassembly buffer. A bit map of received packets is maintained so record reassembly can determine when an entire record has arrived.

Once an entire record has arrived, its key number is looked up on the key ring. If a matching key is available, the contents of the record are decrypted. If the decrypted record matches the filter list predicate, it is presented to the local database for insertion. The record may or may not be retained, depending on resource availability and on the record's priority relative to the information already present. The match module has been designed so that the time required to match an incoming record is essentially independent of the complexity of the filter list.

## 4. The Shared Database Servers

The shared database servers form the second half of our system. We will first discuss the functions of the shared servers, and second, how these functions are implemented.

### 4.1. Shared Server Functions

The shared database servers in our system perform three major functions: they

1. accept data from information sources and add the data to their own databases,
2. transmit database updates to personal systems via a broadcast digital packet radio system, and
3. implement the remote procedure call interface described in Table 1 to allow remote access by personal systems.

The organization of these functions within a single server is described below.

```
Status: type = {completed, aborted, error}

Connect: proc (name: string)
  returns (s: Status)

Disconnect: proc ()
  returns (s: Status)

Abort: proc ()
  returns (s: Status)

EstablishQuery: proc (query: string)
  returns (s: Status)

CountMatchingRecords: proc ()
  returns (final: bool, count: int, s: Status)

FetchSummaries:
  proc (first-record, last-record: int,
        deliver-to: proc (s: Summary) )
  returns (s: Status)

FetchRecord:
  proc (record, first-line, last-line: int,
        deliver-to: proc (l: Line) )
  returns (s: Status)
```

**Table I. RPC Interface to the Shared Servers**

The conceptual organization of the data in each server borrows an important idea from Section 2, where the personal and server systems are modeled as a collection of databases, rather than one large database. Similarly, the information in each server is organized as a collection of databases. These databases may even reside on distinct storage units, so that any database can be physically removed from a server without affecting the remaining databases. The existence of multiple internal databases is not observable at the interface to the server. Query routing is used inside a server to forward each request to the proper set of databases.

Since we view a server as a collection of databases, the databases themselves can be regarded as the basic units of information that are stored on a server. The ability to view predicate databases as the basic *unit of configuration* provides a number of conceptual and practical advantages:

- Databases can be *relocated* from server to server, and the descriptive predicates of each server can easily be updated accordingly.
- Databases can be *replicated* to achieve performance and availability goals. Replication is most naturally done at the level of a database. Immutable databases can be easily replicated, either within a server or on several distinct servers. Replication of append-only and mutable databases is a more complicated issue.
- Each database can have its own representation and implementation, provided that all databases implement the same interface. This is useful when the databases store different kinds of data. In our application, most databases begin their lives as append-only databases, and then become immutable. They can then be consolidated to improve performance and reduce storage demands.

Thus far, we have discussed how data is organized in our database system, but we have not examined how the data actually gets there. The bulk of our data consists of news articles, which arrive in a continuous stream over dedicated telephone circuits. Each input stream is converted into a stream of database records, and the resulting records are inserted, based on their types, into appropriate databases. When a database is replicated on more than one server, the incoming telephone circuit is connected to each server so that all server copies of the database are kept up to date.

In addition to these continuous streams, there are sources that must be polled periodically for new information. In our present system, electronic bulletin board entries are received in this manner. New messages are formatted into database records and inserted into the appropriate databases.

As new information arrives at the system, it is queued to be sent out over the digital broadcast channel. Each record is transmitted more than once, with transmissions separated in time. The transmitted records are encrypted to protect all information that is broadcast from unauthorized use. The encryption key used depends on the type of the record (*New York Times*, *Associated Press*, etc). This scheme logically divides the broadcast transmissions into several independently protected streams. Users are given keys for only the streams they are authorized to receive. These keys are placed on the "key ring" of the personal database system.

The remote procedure call interface shown in Table I does not have any provision for the authentication of users that communicate with the system. We plan to use the key rings of the personal database systems for authentication in this two-way setting.

## 4.2. Implementation of the Shared Database Servers

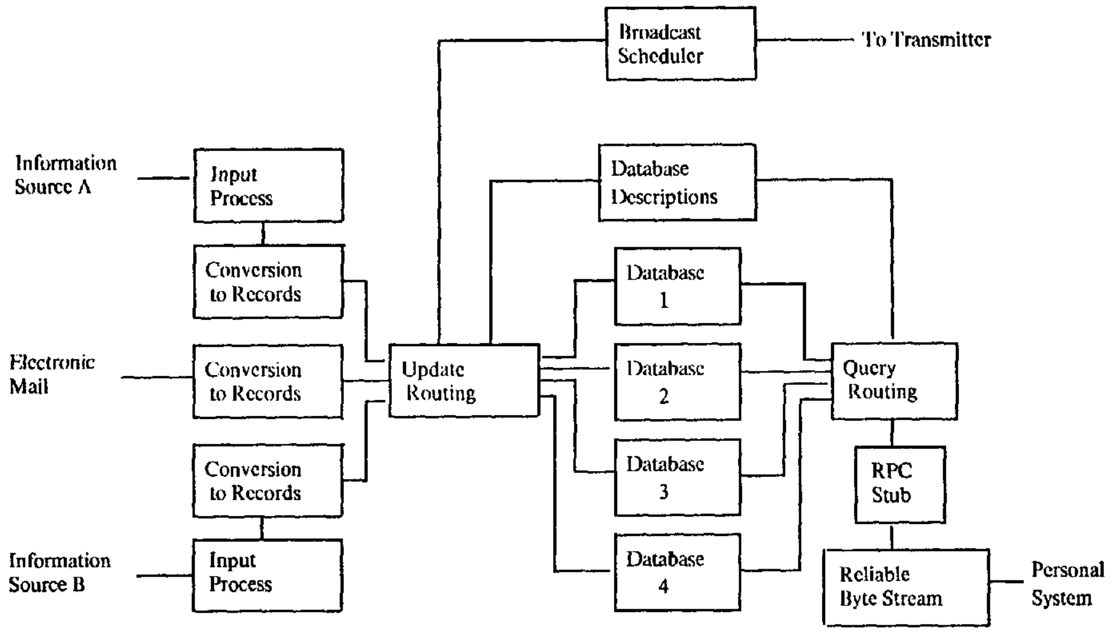
Figure 6 is a block diagram of a typical server as implemented under 4.2 BSD UNIX$^{TM}$. The exact configuration of each server depends on the databases stored on the server, the information sources to which it is connected, and the communication channels through which the personal systems can access the server. Only one server is used to drive the digital broadcast channel. To ensure continuity of service, a second server can take over the broadcast channel if the server driving the channel fails.

To maximize concurrent activity and failure isolation, each major function in a server is performed by a separate process. Processes communicate via messages that are placed in the file system, in an approximation of recoverable message queues. If a process fails, its input messages will accumulate until the process is restarted. Input messages are not deleted until they have been completed processed: if a process fails while processing an input message, this message will be processed again later. Processes that modify non-volatile state information must operate properly when a failure occurs. Transactions could be used to impose additional structure on the failure recovery procedure [4].

In Figure 6, information flows from the left side of the figure to the right side. Information from the various sources is constantly being read by *input daemons* that simply record incoming data items. The incoming items are stored in separate files. To reduce the chance that the input daemons would fail and lose information, they were designed to be extremely simple. To date, we have observed no input daemon failures.

Each input daemon forwards incoming data items to a source-specific *format conversion* process. Each information source has its own format. The source conversion processes take received data items, and convert them into standard format database records. The resulting records are forwarded to the broadcast scheduler and to the database insertion process.

Figure 6 also shows an input process for electronic bulletin board input. This process periodically polls a set of mailboxes, and when it finds mail, converts it to standard format database records, which are then



**Internal Organization of a Shared Server**

**Figure 6**

forwarded to the broadcast scheduler and to the database insertion process.

The *broadcast scheduler* process receives database records, along with an expiration time for each record, an indication of how many times the record should be transmitted, and the key with which the record should be encrypted when it is transmitted. The information broadcasts are scheduled on the basis of this information. In addition to driving the digital broadcast channel, the scheduler maintains a status display for the system administrator.

The *database insertion* process adds new records to the appropriate databases, based on the content of the record and the descriptive predicate of each database. In our present system configuration, a new set of such databases is created every night at midnight, and the databases with the information that arrived during the preceding 24 hour period are converted to immutable status. Each database collects information from a specific source. Thus, there is a separate database for each type of information for each day. These databases are kept on line for a period of time which varies, depending on the source of the database. Eventually, a database may be either archived or deleted.

The final component of a server is the *remote procedure call interface* to the personal systems. When a personal system connects to a server, a process is assigned to manage the connection and process requests from the personal system. Figure 6 shows one such process. Data from the personal system is first processed by a module that implements a byte stream interface. This module is connected to the server's RPC stub module. The RPC stub module calls the appropriate procedures in the server's query routing module.

The query routing module uses the descriptions of the available databases to determine a strategy for processing the request. It then applies the query to the appropriate database(s), and forwards the results to the user's system as they become available. If a query spans multiple databases of a given type but with different dates, the query is applied to these databases in reverse chronological order. This ensures that the most recent records in the result set are produced first. This approach often allows the personal system to display a complete screen of results well before query processing has been completed.

This concludes our discussion of the implementation of the server system. The next section discusses work that is related to our approach.

## 5. Related Work

This section compares our system with other systems that have similar goals. We will consider two broad categories: systems that share our goal of serving a large user community, and systems that share our goal of distributed query processing.

### 5.1. Community Information Systems

*Teletex* [7] and *Videotex* [1] are two contemporary technologies for community information systems. Teletex involves the broadcast transmission of limited amounts of information to specially equipped television sets that can display preformatted pages of information. All available information is transmitted in an endless cycle, which usually lasts no more than a few minutes. In the teletex approach the user selects a page to be displayed, and when the page is next received it is captured by the receiver in a local buffer and displayed on the screen. Most systems limit their teletex transmissions to a total of a few hundred pages to keep response time acceptable.

Videotex [1] is essentially time-sharing on a very large scale. Each videotex user has an inexpensive terminal that communicates with the central system, either over telephone lines or via a digital two-way cable system. Since videotex is a two-way system, it can also be used for home banking, shopping, and other transactional services.

In a sense, our system represents an attempt to combine the best aspects of both teletex and videotex. By combining personal computation, broadcast communication, two-way communication and centralized mass storage, our design achieves the following advantages:

- The economies of scale of broadcast communication are combined with the capability to access large databases and to perform transactional services;
- A high-quality user interface is possible because of the processing power and storage available at each personal database system;
- Performance is improved because the answers to many queries can be determined using the personal database only;
- The system provides a great deal of flexibility because data can be integrated with a user's other computational tools;
- Because of the open system architecture, a user can access a wide variety of databases, potentially offered by a number of vendors;
- Privacy can be safeguarded by ensuring that certain requests are processed entirely within the user's personal system;
- Users can operate autonomously from the central system.

## 5.2. Distributed Database Systems

Distributed database systems, as the name implies, permit the distribution of data over a collection of cooperating computers; the resulting system can be viewed by a user as a cohesive whole. The relational database system R* [8] is an example of a contemporary distributed database system.

In R*, relations (collections of records) can be distributed horizontally or vertically, or replicated at multiple sites. R* also provides a naming scheme whereby information at remote sites can be accessed directly. The query compilation and query processing components of R* automatically route queries to appropriate database nodes for processing. Systems similar to R* include distributed Ingres [6] and the SDD-1 system [5].

The model we have proposed for query routing is sufficiently powerful for immutable and append-only databases. For more comprehensive systems that include shared, replicated, mutable data, our approach would need to be combined with a mechanism to maintain integrity and serializability.

Our approach differs from the traditional distributed database approach in one important respect. In traditional distributed database systems, dependencies can develop between databases, while in our approach databases are always considered to be strictly independent. Dependencies of this kind are central to many applications. However, the scheme that we have outlined is suitable under many circumstances, and has the following advantages:

- Databases can be freely added to and deleted from the system without affecting other databases;
- A single, simple mechanism allows our system to adjust to changes in the set of available databases;
- Because of query routing, queries do not have to indicate where the requested data is to be found.

## 6. Implementation Status and Performance

Our community information system began regular operation in April of 1984. Every day we receive and distribute approximately 150 news articles from the *New York Times* totalling 600K bytes, 1300 news articles from the *Associated Press News Service* totalling 6M bytes, and 70K bytes from electronic mailing lists.

Currently, 15 users outside of our own research group use a broadcast-only version of the system. These users have been very helpful in suggesting user interface enhancements (one user added mouse support himself) and in motivating us to catalog the content of our databases. In addition, the users let us know immediately if there is a problem with our database transmissions!

A research version of the system, incorporating both query routing and server access, is operational and is currently being tested within our laboratory. The performance measurements shown below apply to this version of the personal database system. A user acceptance test of the full personal database system, including query routing and server access, is planned for this fall.

The performance of the system is excellent for requests that can be locally processed. On a PC AT, less than 300 milliseconds are required to either process a query or display a record from the local database.

To compare the performance of local and remote queries, we performed a set of experiments. Each experiment consisted of making a request that caused the personal database system to access a server. The observed response times were directly related to the bandwidth (1.2K bits/sec) of the channel linking the personal database system and the server. The results of our experiments are shown in Table II.

One shortcoming of the current implementation is that the personal database system can not collect database updates from the digital broadcast channel while a user is running other applications. We believe that this problem will be solved by the next generation of personal computer operating systems, which will support true multitasking.

| | |
|---|---|
| Time from query entry to display of the first record summary | 2.7 sec. |
| Time from query entry to display of a complete summary page | 11.6 sec. |
| Time from request for a record to display of the first line | 4.1 sec. |
| Time from request for a record to display of a complete page | 11.9 sec. |

**Table II:**
**Typical Performance of Remote Query Processing**
**(1.2 KBit/Sec Channel)**

## 7. Conclusions

We are pleased with the initial user acceptance of our system. Users have been able to grasp the idea of a local database defined by logical predicates, and we have found that users actively modify the definition of their local database to match their interests. Our experience demonstrates that broadcast technology is an effective way to update local databases. The natural advantage of the broadcast portion of our system is that it can scale to a user population of any size.

Database content descriptions and query routing can help integrate personal databases into a comprehensive information service. Experience with our system has shown that users sometimes wish to retrieve information that is not available on their personal system. Our system will automatically route such queries for non-local information to an appropriate server database. We believe that our decision to structure the system as a collection of independent databases has simplified our design, and has resulted in both a clear conceptual framework and a number of practical benefits.

Based on our research results, we feel that the architecture that we have proposed is well adapted to a variety of information system applications. In addition, a number of its component ideas — including our model of remote procedure call and recoverable message queues — will find application in other contexts.

## Acknowledgments

The authors would like to thank Barbara H. Liskov and Heidi R. Wyle for their comments.

## References

[1] Bright, R.D., *Prestel - The World's First Public Viewdata Service*, IEEE Trans. on Consumer Electronics CE-25, 3 (July 1979), pp. 251-255.

[2] Geller, V.J., and Lesk, M.E., *How Users Search: A Comparison of Menu and Attribute Retrieval Systems on a Library Catalog*, Bell Laboratories, Murray Hill, N.J., September 27, 1981.

[3] Gifford, D.K., Lucassen, J.M., and Berlin, S.T., *The Application of Digital Broadcast Communication to Large Scale Information Systems*, IEEE Trans. on Selected Areas in Comm. (May 1985).

[4] Gifford, D.K., and Donahue, J.D., *Coordinating Independent Atomic Actions*, Proc. of IEEE Spring CompCon 85, Feb. 25-28, 1985, San Francisco, CA, pp. 92-95.

[5] Rothnie, J.B *et al.*, *Introduction to a System for Distributed Databases (SDD-1)*, ACM Trans. on Database Systems 5, 1 (March 1980).

[6] Stonebreaker, M., and Neuhold, E., *A Distributed Data Base Version of INGRES*, Proc. 2nd Berkeley Workshop on Distributed Data Management and Computer Networks, Berkeley, CA, May 1977, pp. 19-36.

[7] Tanton, N.E., *Teletex - Evaluation and Potential*, IEEE Trans. on Consumer Electronics CE-25, 3 (July 1979), pp. 246-250.

[8] Williams, R. *et al.*, *R*: An Overview of the Architecture*, Rep. RJ3325, IBM Research, San Jose, CA, December 2, 1981.