Ubik -- A Library for Managing Ubiquitous Data

> *"Yes," Runciter's dark voice resumed, "by making use of the most advanced techniques of present-day science, the reversion of matter to earlier forms can be reversed, and at a price any conapt owner can afford. Ubik is sold by leading home-art stores throughout Earth. Do not take internally. Keep away from open flame. Do not deviate from printed procedural approaches as expressed on label. So look for it, Joe. Don't just sit there; go out and buy a can of Ubik and spray it all around you night and day."*
>
> *from "Ubik"*
> *by Philip K. Dick*

Introduction

Ubik is a library for managing a single database, replicated across some number of servers. It provides the abstraction of a file that is read or written to in the context of a read or write transaction.

To read data from the replicated file, one starts a transaction (using BeginTrans), sets the appropriate locks (using SetLock), reads the data from the file and terminates the transaction (using EndTrans, there's no reason to abort a read transaction).

At other times, you may want to write data to the file. Again, you start a transaction with BeginTrans (you must declare at this time that you are performing a write transaction). You then set the the appropriate locks on the data you'll be reading or writing. Next, you then do a set of ubik_Read and ubik_Write operations to read and update the data. Finally, if everything looks fine, you can commit the transaction with ubik_EndTrans. If you instead reach a point where you decide you can not commit the transaction, you can abort the whole thing by calling ubik_AbortTrans.

Note that you can mix reading, writing and setting locks in any order. However, you, as the programmer, are responsible for avoiding deadlock when setting locks. The traditional method of avoiding deadlock is to set locks according to a partial order so that lock dependency cycles can not be formed.

Before performing any transactions, you must initialize the ubik library. This is done by calling ubik_ServerInit with a number of parameters. You must pass it your own internet address in network byte order (the parameter localHost), a network order short port address (thus there must be one port used by all the replicates in the system for this service), an array of internet server addresses for the remaining servers (aside from yourself), terminated by a 0. In addition, you must pass it a pathname prefix such that the database file names can be derived by appending the appropriate strings to the end of the pathname parameter. For instance, if the pathname is "/user/x/foo", then the log file will be written to "/usr/x/foo.DBSYS1" and the data file will be written to "/usr/x/foo.DB0". Finally, there is an output parameter, in which a pointer to the newly-created database structure is returned, if the call succeeds and returns a 0 return code. This value is passed into the ubik_BeginTrans procedure as its first parameter.

The remainder of this document describes both a client and a server procedural interface. Ubik is designed to provide a basic replicated database facility for writing servers that are accessed via RPC, in particular via our "R" (later "R_X") remote procedure call packages. The model you should have in mind is this: there is a collection of N servers, all sharing one replicated database (replicated by ubik). Other clients make RPC calls to these servers, which then perform read and or write transactions for the clients, and return the appropriate results to the clients. When a client contacts a server to perform a request, it may fail due to the server's being down, or certain requests may fail because they may have to be performed at the synchronization site, and you have contacted a non-synchronization site. The ubik client interface provides routines that hide these difficulties, trying multiple servers when operations fail at one server, and tracking down the synchronization site if it fails to contact the synchronization site on its first attempt.

Server Procedural Interface

The ubik procedural interface is quite simple. The data space is two-dimensional, with one dimension being a file name (a single 32-bit integer file identifier), and the other being a byte position within the that file.

int **ubik_ServerInit**(localHost, localPort, serverList, pathName, dbase)
IN sockaddr_in localHost;
IN short localPort;
IN long serverList[];
char *pathName;
OUT struct ubik_dbase *dbase;

This routine initializes the ubik database for a set of servers. It returns 0 for success, or an error code on failure. The set of servers is specified by *serverList*; this list should not include the network address of this server, which instead is provided by the *localHost* parameter. The parameter *dbase* is the returned structure representing this instance of a ubik; it is passed to various calls below. The variable *pathName* provides an initial prefix used for naming storage files used by this database. The database itself consists of several files (two at present) whose file names begin with the *pathName* prefix. One of these files is the log file, used for crash recovery; the other is the database file itself, and stores, after a fixed-sized short header, the user's actual data bytes.

You can only call this function once per Unix process.

int **ubik_BeginTrans**(dbase, transMode, transPtr)
IN struct ubik_dbase *dbase;
IN long transMode;
OUT struct ubik_trans *transPtr;

This routine begins a read or write transaction on the transaction identified by *transPtr*, in the database named by *dbase*. An open mode of UBIK_READTRANS identifies this as a read transaction, while a mode of UBIK_WRITETRANS identifies this as a write transaction. Finally, *transPtr* is set to the returned transaction control block.

int **ubik_EndTrans**(transPtr)
IN struct ubik_trans *transPtr;

This routine ends a read or write transaction on the open transaction identified by *transPtr* by committing the transaction. It returns an error code, or 0 if the operation succeeds. All transaction locks are released.

int **ubik_AbortTrans**(transPtr)
IN struct ubik_trans *transPtr;

This routine ends a read or write transaction on the open transaction identified by *transPtr* by aborting the transaction. All transaction locks are released, and all changes made by the transaction are undone.

int **ubik_Read**(transPtr, buffer, length)
IN struct ubik_trans *transPtr;
IN char *buffer;
IN long length;

This routine reads *length* bytes into *buffer* from the current position in the database. The file pointer is updated appropriately (by adding the number of bytes actually transferred), and the length actually transferred is stored in the long integer pointed to by *length*. This call returns either -1 if an I/O error occurs, or the number of characters actually read.

Note that ubik is page oriented, and a read may read to the end of a page instead of stopping as soon as the file's size is exceeded. This means that server programmers can not assume that they'll get an EOF precisely at the point at which they last wrote data (although they won't get an EOF that point); instead, ubik may return a little extra data past the actual end of file.

int **ubik_Write**(transPtr, buffer, length)
IN struct ubik_trans *transPtr;
IN char *buffer;
IN long length;

This routine writes *length* bytes into *buffer* at the current position in the database. The byte pointer is updated appropriately (by adding the number of bytes actually written). Unlike ubik_Read, a short write does return a non-zero error code.

int **ubik_Seek**(transPtr, fileid, position)
IN struct ubik_trans *transPtr;
IN long fileid;
IN long position;

This sets the file pointer associated with the current transaction to the appropriate file and byte position. Unlike Unix files, a transaction is labeled by both a file number (*fileid*) and a byte position relative to the specified file (*position*).

int **ubik_Tell**(transPtr, fileid, position)
IN struct ubik *transPtr;
OUT long *fileid;
OUT long *position;

This call returns the file pointer associated with the specified transaction in *fileid* and *position*.

int **ubik_Truncate**(transPtr, length)
IN struct ubik_trans *transPtr;
IN long length;

This sets the file size for the currently-selected file to length bytes, if *length* is less than the file's current size.

int **ubik_SetLock**(transPtr, pos, length, type)
IN struct ubik_trans *transPtr;
IN long pos, length;
IN int type;

This function sets a lock at position *pos* for *length* bytes, in lock mode *type*. There is no hard connection between lock positions and file positions; this mechanism simply provides a way of locking an integer name space. In the current implementation, the length of the lock stream must be 1, or the call will fail. Finally, one can set either shared or exclusive locks, by setting *type* to be either LOCKREAD or LOCKWRITE, respectively.

Since ubik expects people to do two-phase locking, there is no unlock primitive provided. Locks are released at the end of the transaction, when the transaction is either aborted or committed.

Finally a routine is provided to make it easier to parse command lines and generate the host lists required for ubik_ServerInit:

int **ubik_GetVersion**(transPtr, versionP)
IN struct ubik_trans *transPtr;
OUT struct ubik_version *versionP;

This function can be called to return the version number of the ubik database. This can be compared for equality using the vcmp macro declared in ubik.h.

int **ubik_CacheUpdate**(transPtr)
IN struct ubik_trans *transPtr;

This proviced a facility to simplify database caching. It returns zero if last transaction was done on the local server and was successful. If the return value is non-zero and the caller is a server caching part of the ubik database, he should invalidate that cache. A return value of -1 means bad (NULL) argument.

ubik_ParseServerList(argc, argv, ahost, aothers)
IN int argc;
IN char **argv;
OUT long *ahost;
OUT long aothers[];

This routine parses the command line argument list (argc and argv) for a series of tokens consisting of "-servers *host-1 host-2 ... host-n*", and sets both *ahost* and *aothers* to the appropriate values. The server list should include the host you are running on, so as to allow the same command line to be used to start all of the servers in a group. The aothers array should be declared

```
long others[MAXSERVERS]
```

where MAXSERVERS is a #define from ubik.h to avoid memory trashing; the space for others is not allocated dynamically by ubik_ParseServerList.

This function returns 0 if it found and parsed all of the servers, otherwise it returns a ubik error code.

Client Procedural Interface

The client library consists of a few procedures to make it easier to ubik-based servers. The ubik_ClientInit function initializes the client-side ubik system, and returns a client structure used in the ubik_Call procedure. The ubik_ParseClientList is a utility provided to ease parsing of command line options; it is useful for bridging the gap between command line options and ubik_ClientInit. Finally, ubik_Call is used to actually invoke, via RPC, a procedure exported by a ubik server. It handles such common problems as ignoring dead hosts and tracking down synchronization sites in a collection of servers.

ubik_ClientInit(aservers, aport, alocalPort, aclient)
IN long aservers[];
IN short aport;
IN short alocalPort;
OUT struct ubik_client **aclient;

The first parameter to this function is a set of server addresses (network byte order internet host addresses), terminated by a zero entry, and representing the set of hosts to try to contact when making a call to the ubik servers backing the service that the client wishes to invoke. The second parameter is the port (a network byte order short integer) that represents the port at which this service can be contacted. The next parameter is the socket which the local end of the newly-created UDP connection should be assigned; if you are not also a server (how many levels can there be!) then you should just specify this parameter as 0, and the system will assign a local port number for you. Finally, aclient should be a pointer to a struct ubik_client * variable that will be set to a structure used by the ubik_Call procedure, described below.

ubik_ParseClientList(argc, argv, aservers)
IN argc;
IN char **argv;
OUT long *aservers;

This function parses the same command line syntax as ubik_ParseServerList (that is, "-server *server-1 ... server-N*", but returns a single array of network byte order long integers representing the addresses of the servers named in the -server command line option. This array is suitable for passing to ubik_ClientInit. The array itself is not allocated, the caller should declare it as

```
long others[MAXSERVERS]
```

to avoid memory corruption.

ubik_Call(aproc, aclient, aflags, p1, p2, ..., p8)

IN int (*aproc)();
IN struct ubik_client *aclient;
IN aflags;
IN long p1, p2, p3, ..., p8

This function invokes the R stub routine named by aproc to each of the servers represented by the client structure, until it finds one that executes the routine correctly. It handles UNOTSYNC return codes by locating the sychronization site retrying the operation there, and handles network communication problems by trying the next host in order.

The stub is invoked as follows:

```
    (*aproc)(<appropriate RPC connection>, p1, p2, ...,
    p8)
```

It returns the results of the first call that did not encounter one of the above recoverable errors.

The *flags* parameter is reserved for future use, and must be 0.

Server Example

A working example can be found in the file userver.c. Here are some additional comments on using ubik for writing a server.

Server programmers start of writing a server by calling ubik_ServerInit; this returns a pointer to a struct ubik_dbase, representing the ubik database you'll be processing. At present, you can only initialize one ubik database per process.

From that point on, your ubik server can perform read or write transactions by creating a new transaction (read or write) using ubik_BeginTrans, reading data, setting locks and writing data, and finally committing (with ubik_EndTrans) or aborting (with ubik_AbortTrans) the transaction.

Note that you must set appropriate locks when reading or writing data (even if you just decide to lock the whole database), otherwise transactions may read uncommitted data.

There is one relatively minor change in the programmer's interface to R when using ubik: instead of calling r_GetRequest, one calls ubik_GetRequest. These two functions take the same parameters and return the same values, but ubik_GetRequest also dispatches those functions that ubik sends from server to server, and does so in a safe manner guaranteed to avoid deadlocks over lack of light-weight processes.

Client Example

A working client example can be found in the file uclient.c. Here are some additional comments on using ubik for writing a client.

A client generally starts off parsing the command line to determine the hosts it will communicate with; generally using ubik_ParseClientList. It then calls ubik_ClientInit with the results of the parse, and its own local port number, if any, and obtains a ubik_client structure with which it can invoke procedures replicated on multiple servers. Finally, a client uses ubik_Call to invoke these server procedures, passing in the stub to invoke, a ubik_client structure representing the set of potential hosts to use, and a set of parameters to this procedure. It returns the first interesting return code it obtains.