# Fundamental Specifications of Tokyo Tyrant Version 1

Last Update: Thu, 05 Aug 2010 15:24:29 +0900

## Table of Contents

## Introduction

Tokyo Tyrant is a package of network interface to the DBM called Tokyo Cabinet. Though the DBM has high performance, you might bother in case that multiple processes share the same database, or remote processes access the database. Thus, Tokyo Tyrant is provided for concurrent and remote connections to Tokyo Cabinet. It is composed of the server process managing a database and its access library for client applications.

The server features high concurrency due to thread-pool modeled implementation and the epoll/kqueue mechanism of the modern Linux/*BSD kernel. The server and its clients communicate with each other by simple binary protocol on TCP/IP. Protocols compatible with memcached and HTTP are also supported so that almost all principal platforms and programming languages can use Tokyo Tyrant. High availability and high integrity are also featured due to such mechanisms as hot backup, update logging, and replication. The server can embed Lua, a lightweight script language so that you can define arbitrary operations of the database.

Because the server uses the abstract API of Tokyo Cabinet, all of the six APIs: the on-memory hash database API, the on-memory tree database API, the hash API, the B+ tree database API, the fixed-length database API, and the table database API, are available from the client with the common interface. Moreover, the table extension is provided to use specific features of the table database.

*As for now, the server works on Linux, FreeBSD, Mac OS X, Solaris only.*

## Installation

Install the latest version of Tokyo Cabinet beforehand and get the package of Tokyo Tyrant.

When an archive file of Tokyo Tyrant is extracted, change the current working directory to the generated directory and perform installation.

Run the configuration script. To enable the Lua extension, add the `--enable-lua' option.

```
./configure
```

Build programs.

```
make
```

Install programs. This operation must be carried out by the root user.

```
make install
```

When a series of work finishes, the following files will be installed.

```
/usr/local/include/ttutil.h
/usr/local/include/tculog.h
/usr/local/include/tcrdb.h
/usr/local/lib/libtokyotyrant.a
/usr/local/lib/libtokyotyrant.so.x.y.z
/usr/local/lib/libtokyotyrant.so.x
/usr/local/lib/libtokyotyrant.so
/usr/local/lib/ttskelmock.so
/usr/local/lib/ttskeldir.so
/usr/local/lib/ttskelproxy.so
/usr/local/lib/ttskelnull.so
/usr/local/lib/pkgconfig/tokyotyrant.pc
```

```
/usr/local/bin/ttserver
/usr/local/bin/ttultest
/usr/local/bin/ttulmgr
/usr/local/bin/tcrtest
/usr/local/bin/tcrmttest
/usr/local/bin/tcrmgr
/usr/local/sbin/ttservctl
/usr/local/share/tokyotyrant/...
/usr/local/man/man1/...
/usr/local/man/man3/...
```

To test the server, perform the following command. To finish it, press Ctrl-C on the terminal.

```
ttserver
```

To test the client connecting to the above running server, perform the following command on another terminal.

```
make check
```

---

## Server Programs

### ttserver

The command `**ttserver**' runs the server managing a database instance. Because the database is treated by the abstract API of Tokyo Cabinet, you can choose the scheme on start-up of the server. Supported schema are on-memory hash database, on-memory tree database, hash database, and B+ tree database. This command is used in the following format. `*dbname*' specifies the database name. If it is omitted, on-memory hash database is specified.

```
ttserver [-host name] [-port num] [-thnum num] [-tout num] [-dmn] [-pid path] [-kl] [-log path] [-ld|-le] [-ulog path] [-ulim num] [-uas] [-sid num] [-mhost name] [-mport num] [-rts path] [-rcc] [-skel name] [-mul num] [-ext path] [-extpc name period] [-mask expr] [-unmask expr] [dbname]
```

Options feature the following.

**`-host`** ***`name`*** : specify the host name or the address of the server. By default, every network address is bound.
**`-port`** ***`num`*** : specify the port number. By default, it is 1978.
**`-thnum`** ***`num`*** : specify the number of worker threads. By default, it is 8.
**`-tout`** ***`num`*** : specify the timeout of each session in seconds. By default, no timeout is specified.
**`-dmn`** : work as a daemon process.
**`-pid`** ***`path`*** : output the process ID into the file.
**`-kl`** : kill the existing process if the process ID file is detected.
**`-log`** ***`path`*** : output log messages into the file.
**`-ld`** : log debug messages also.
**`-le`** : log error messages only.
**`-ulog`** ***`path`*** : specify the update log directory.
**`-ulim`** ***`num`*** : specify the limit size of each update log file.
**`-uas`** : use asynchronous I/O for the update log.
**`-sid`** ***`num`*** : specify the server ID.
**`-mhost`** ***`name`*** : specify the host name of the replication master server.
**`-mport`** ***`num`*** : specify the port number of the replication master server.
**`-rts`** ***`path`*** : specify the replication time stamp file.
**`-rcc`** : check consistency of replication.
**`-skel`** ***`name`*** : specify the name of the skeleton database library.

- `-mul` ***`num`*** : specify the division number of the multiple database mechanism.
- `-ext` ***`path`*** : specify the script language extension file.
- `-extpc` ***`name period`*** : specify the function name and the calling period of a periodic command.
- `-mask` ***`expr`*** : specify the names of forbidden commands.
- `-unmask` ***`expr`*** : specify the names of allowed commands.

To terminate the server normally, send SIGINT or SIGTERM to the process. It is okay to press Ctrl-C on the controlling terminal. To restart the server, send SIGHUP to the process. If the port number is not more than 0, UNIX domain socket is used and the path of the socket file is specified by the host parameter. This command returns 0 on success, another on failure.

The naming convention of the database is specified by the abstract API of Tokyo Cabinet. If the name is "*", the database will be an on-memory hash database. If it is "+", the database will be an on-memory tree database. If its suffix is ".tch", the database will be a hash database. If its suffix is ".tcb", the database will be a B+ tree database. If its suffix is ".tcf", the database will be a fixed-length database. If its suffix is ".tct", the database will be a table database. Otherwise, this function fails. Tuning parameters can trail the name, separated by "#". Each parameter is composed of the name and the value, separated by "=". On-memory hash database supports "bnum", "capnum", and "capsiz". On-memory tree database supports "capnum" and "capsiz". Hash database supports "mode", "bnum", "apow", "fpow", "opts", "rcnum", "xmsiz", and "dfunit". B+ tree database supports "mode", "lmemb", "nmemb", "bnum", "apow", "fpow", "opts", "lcnum", "ncnum", "xmsiz", and "dfunit". Fixed-length database supports "mode", "width", and "limsiz". Table database supports "mode", "bnum", "apow", "fpow", "opts", "rcnum", "lcnum", "ncnum", "xmsiz", "dfunit", and "idx". The tuning parameter "capnum" specifies the capacity number of records. "capsiz" specifies the capacity size of using memory. Records spilled the capacity are removed by the storing order. "mode" can contain "w" of writer, "r" of reader, "c" of creating, "t" of truncating, "e" of no locking, and "f" of non-blocking lock. The default mode is relevant to "wc". "opts" can contains "l" of large option, "d" of Deflate option, "b" of BZIP2 option, and "t" of TCBS option. "idx" specifies the column name of an index and its type separated by ":". For example, "casket.tch#bnum=1000000#opts=ld" means that the name of the database file is "casket.tch", and the bucket number is 1000000, and the options are large and Deflate.

The command mask expression is a list of command names separated by ",". For example, "out,vanish,copy" means a set of "out", "vanish", and "copy". Commands of the memcached compatible protocol and the HTTP compatible protocol are also forbidden or allowed, related by the mask of each original command. Moreover, there are meta expressions. "all" means all commands. "allorg" means all commands of the original binary protocol. "allmc" means all commands of the memcached compatible protocol. "allhttp" means all commands of the HTTP compatible protocol. "allread" is the abbreviation of `get', `mget', `vsiz', `iterinit', `iternext', `fwmkeys', `rnum', `size', and `stat'. "allwrite" is the abbreviation of `put', `putkeep', `putcat', `putshl', `putnr', `out', `addint', `adddouble', `vanish', and `misc'. "allmanage" is the abbreviation of `sync', `optimize', `copy', `restore', and `setmst'. "repl" means replication as master. "slave" means replication as slave.

## ttservctl

The command `ttservctl' is the startup script of the server. It can be called by the RC script of the bootstrap process of the operating system. This command is used in the following format.

**`ttservctl start`**
: Startup the server.

**`ttservctl stop`**
: Stop the server.

**`ttservctl restart`**
: Restart the server.

**`ttservctl hup`**
: Send HUP signal to the server for log rotation.

The database is placed as "/var/ttserver/casket.tch". The log and related files are also placed in "/var/ttserver". This command returns 0 on success, another on failure.

### ttulmgr

The command `ttulmgr' is the utility to export and import the update log. It is useful to filter the update log with such text utilities as `grep' and `sed'. This command is used in the following format. `*upath*' specifies the update log directory.

**`ttulmgr export [-ts num] [-sid num] upath`**
: Export the update log as TSV text data to the standard output.

**`ttulmgr import upath`**
: Import TSV text data from the standard input to the update log.

Options feature the following.

- **`-ts`** ***num*** : specify the beginning time stamp.
- **`-sid`** ***num*** : specify the self server ID.

This command returns 0 on success, another on failure.

---

## Client Programs

### tcrtest

The command `**tcrtest**' is a utility for facility test and performance test. This command is used in the following format. `*host*' specifies the host name of the server. `*rnum*' specifies the number of iterations.

**tcrtest write [-port *num*] [-cnum *num*] [-tout *num*] [-nr] [-rnd] *host* *rnum***
: Store records with keys of 8 bytes. They change as `00000001', `00000002'...

**tcrtest read [-port *num*] [-cnum *num*] [-tout *num*] [-mul *num*] [-rnd] *host***
: Retrieve all records of the database above.

**tcrtest remove [-port *num*] [-cnum *num*] [-tout *num*] [-rnd] *host***
: Remove all records of the database above.

**tcrtest rcat [-port *num*] [-cnum *num*] [-tout *num*] [-shl *num*] [-dai|-dad] [-ext *name*] [-xlr|-xlg] *host* *rnum***
: Store records with partway duplicated keys using concatenate mode.

**tcrtest misc [-port *num*] [-cnum *num*] [-tout *num*] *host* *rnum***
: Perform miscellaneous test of various operations.

**tcrtest wicked [-port *num*] [-cnum *num*] [-tout *num*] *host* *rnum***
: Perform updating operations of list and map selected at random.

**tcrtest table [-port *num*] [-cnum *num*] [-tout *num*] [-exp *num*] *host* *rnum***
: Perform miscellaneous test of the table extension.

Options feature the following.

- **-port *num*** : specify the port number.
- **-cnum *num*** : specify the number of connections.
- **-tout *num*** : specify the timeout of each session in seconds.
- **-nr** : use the function `tcrdbputnr' instead of `tcrdbput'.
- **-rnd** : select keys at random.
- **-mul *num*** : specify the number of records for the mget command.
- **-shl *num*** : use `tcrdbputshl' and specify the width.

**`-dai`** : use `tcrdbaddint' instead of `tcrdbputcat'.
**`-dad`** : use `tcrdbadddouble' instead of `tcrdbputcat'.
**`-ext`** ***`name`*** : call a script language extension function.
**`-xlr`** : perform record locking.
**`-xlg`** : perform global locking.
**`-exp`** ***`num`*** : specify the lifetime of expiration test.

If the port number is not more than 0, UNIX domain socket is used and the path of the socket file is specified by the host parameter. This command returns 0 on success, another on failure.

## tcrmttest

The command `` `**tcrmttest**' `` is a utility for facility test under multi-thread situation. This command is used in the following format. \``*host*' specifies the host name of the server. \``*rnum*' specifies the number of iterations.

```
tcrmttest write [-port num] [-tnum num] [-nr] [-rnd] [-ext name] host rnum
```
Store records with keys of 8 bytes. They change as `00000001', `00000002'...
```
tcrmttest read [-port num] [-tnum num] [-mul num] host
```
Retrieve all records of the database above.
```
tcrmttest remove [-port num] [-tnum num] host
```
Remove all records of the database above.

Options feature the following.

**`-port`** ***`num`*** : specify the port number.
**`-tnum`** ***`num`*** : specify the number of running threads.
**`-nr`** : use the function `tcrdbputnr' instead of `tcrdbput'.
**`-rnd`** : select keys at random.
**`-ext`** ***`name`*** : call a script language extension function.

**-mul** ***num*** : specify the number of records for the mget command.

If the port number is not more than 0, UNIX domain socket is used and the path of the socket file is specified by the host parameter. This command returns 0 on success, another on failure.

## tcrmgr

The command \``tcrmgr`' is a utility for test and debugging of the remote database API and its applications. \`*host*' specifies the host name of the server. \`*key*' specifies the key of a record. \`*value*' specifies the value of a record. \`*params*' specifies the tuning parameters. \`*dpath*' specifies the destination file. \`*func* specifies the name of the function. \`*arg*' specifies the arguments of the function. \`*file*' specifies the input file. \`*upath*' specifies the update log directory. \`*mhost*' specifies the host name of the replication master. \`*url*' specifies the target URL.

**tcrmgr inform [-port *num*] [-st] *host***
: Print miscellaneous information to the standard output.

**tcrmgr put [-port *num*] [-sx] [-sep *chr*] [-dk|-dc|-dai|-dad] *host key value***
: Store a record.

**tcrmgr out [-port *num*] [-sx] [-sep *chr*] *host key***
: Remove a record.

**tcrmgr get [-port *num*] [-sx] [-sep *chr*] [-px] [-pz] *host key***
: Print the value of a record.

**tcrmgr mget [-port *num*] [-sx] [-sep *chr*] [-px] *host* [*key*...]**
: Print keys and values of multiple records.

**tcrmgr list [-port *num*] [-sep *chr*] [-m *num*] [-pv] [-px] [-fm *str*] *host***
: Print keys of all records, separated by line feeds.

**tcrmgr ext [-port *num*] [-xlr|-xlg] [-sx] [-sep *chr*] [-px] *host func* [*key* [*value*]]**
: Call a script language extension function.

**`tcrmgr sync [-port`** ***`num`*** **`]`** ***`host`***
: Synchronize updated contents with the database file.

**`tcrmgr optimize [-port`** ***`num`*** **`]`** ***`host`*** **`[`** ***`params`*** **`]`**
: Optimize the database file.

**`tcrmgr vanish [-port`** ***`num`*** **`]`** ***`host`***
: Remove all records.

**`tcrmgr copy [-port`** ***`num`*** **`]`** ***`host dpath`***
: Copy the database file.

**`tcrmgr misc [-port`** ***`num`*** **`] [-mnu] [-sx] [-sep`** ***`chr`*** **`] [-px]`** ***`host func`*** **`[`** ***`arg`*** **`...]`**
: Call a versatile function for miscellaneous operations.

**`tcrmgr importtsv [-port`** ***`num`*** **`] [-nr] [-sc]`** ***`host`*** **`[`** ***`file`*** **`]`**
: Store records of TSV in each line of a file.

**`tcrmgr restore [-port`** ***`num`*** **`] [-ts`** ***`num`*** **`] [-rcc]`** ***`host upath`***
: Restore the database with update log.

**`tcrmgr setmst [-port`** ***`num`*** **`] [-mport`** ***`num`*** **`] [-ts`** ***`num`*** **`] [-rcc]`** ***`host`*** **`[`** ***`mhost`*** **`]`**
: Set the replication master.

**`tcrmgr repl [-port`** ***`num`*** **`] [-ts`** ***`num`*** **`] [-sid`** ***`num`*** **`] [-ph]`** ***`host`***
: Replicate the update log.

**`tcrmgr http [-ah`** ***`name value`*** **`] [-ih]`** ***`url`***
: Fetch the resource of a URL by HTTP.

**`tcrmgr version`**
: Print the version information of Tokyo Tyrant.

Options feature the following.

- **`-port`** ***`num`*** : specify the port number.
- **`-st`** : print miscellaneous status data.
- **`-sx`** : input data is evaluated as a hexadecimal data string.

- **`-sep`** ***`chr`*** : specify the separator of the input data.
- **`-dk`** : use the function `tcrdbputkeep' instead of `tcrdbput'.
- **`-dc`** : use the function `tcrdbputcat' instead of `tcrdbput'.
- **`-dai`** : use the function `tcrdbaddint' instead of `tcrdbput'.
- **`-dad`** : use the function `tcrdbadddouble' instead of `tcrdbput'.
- **`-px`** : output data is converted into a hexadecimal data string.
- **`-pz`** : do not append line feed at the end of the output.
- **`-m`** ***`num`*** : specify the maximum number of the output.
- **`-pv`** : print values of records also.
- **`-fm`** ***`str`*** : specify the prefix of keys.
- **`-xlr`** : perform record locking.
- **`-xlg`** : perform global locking.
- **`-mnu`** : omit the update log.
- **`-nr`** : use the function `tcrdbputnr' instead of `tcrdbput'.
- **`-sc`** : normalize keys as lower cases.
- **`-mport`** ***`num`*** : specify the port number of the replication master.
- **`-ts`** ***`num`*** : specify the beginning time stamp.
- **`-rcc`** : check consistency of replication.
- **`-sid`** ***`num`*** : specify the self server ID.
- **`-ph`** : print human-readable data.
- **`-ah`** ***`name value`*** : add a request header.
- **`-ih`** : output response headers also.

If the port number is not more than 0, UNIX domain socket is used and the path of the socket file is specified by the host parameter. This command returns 0 on success, another on failure.

# Remote Database API

Remote database is a set of interfaces to use an abstract database of Tokyo Cabinet, mediated by a server of Tokyo Tyrant. See `` `tcrdb.h' `` for entire specification.

## Description

To use the remote database API, include `` `tcrdb.h' `` and related standard header files. Usually, write the following description near the front of a source file.

```
#include <tcrdb.h>
#include <stdlib.h>
#include <stdbool.h>
#include <stdint.h>
```

Objects whose type is pointer to `` `TCRDB' `` are used to handle remote databases. A remote database object is created with the function `` `tcrdbnew' `` and is deleted with the function `` `tcrdbdel' ``. To avoid memory leak, it is important to delete every object when it is no longer in use.

Before operations to store or retrieve records, it is necessary to connect the remote database object to the server. The function `` `tcrdbopen' `` is used to open a database connection and the function `` `tcrdbclose' `` is used to close the connection.

## API

The function `tcrdberrmsg' is used in order to get the message string corresponding to an error code.

**`const char *tcrdberrmsg(int ecode);`**

`*ecode*' specifies the error code.
The return value is the message string of the error code.

The function `tcrdbnew' is used in order to create a remote database object.

**`TCRDB *tcrdbnew(void);`**

The return value is the new remote database object.

The function `tcrdbdel' is used in order to delete a remote database object.

**`void tcrdbdel(TCRDB *rdb);`**

`*rdb*' specifies the remote database object.

The function `tcrdbecode' is used in order to get the last happened error code of a remote database object.

**`int tcrdbecode(TCRDB *rdb);`**

`*rdb*' specifies the remote database object.
The return value is the last happened error code.
The following error code is defined: `TTESUCCESS' for success, `TTEINVALID' for invalid operation, `TTENOHOST' for host not found, `TTEREFUSED' for connection refused, `TTESEND' for send error, `TTERECV' for recv error, `TTEKEEP' for existing record, `TTENOREC' for no record found, `TTEMISC' for miscellaneous error.

The function `tcrdbtune' is used in order to set the tuning parameters of a hash database object.

**`bool tcrdbtune(TCRDB *rdb, double timeout, int opts);`**

`*rdb*' specifies the remote database object.
`*timeout*' specifies the timeout of each query in seconds. If it is not more than 0, the timeout is not specified.

`*opts*' specifies options by bitwise-or: `RDBTRECON' specifies that the connection is recovered automatically when it is disconnected.
If successful, the return value is true, else, it is false.
Note that the tuning parameters should be set before the database is opened.

The function `tcrdbopen' is used in order to open a remote database.

**bool tcrdbopen(TCRDB ****rdb***, const char ****host***, int** ***port*****);**

`*rdb*' specifies the remote database object.
`*host*' specifies the name or the address of the server.
`*port*' specifies the port number. If it is not more than 0, UNIX domain socket is used and the path of the socket file is specified by the host parameter.
If successful, the return value is true, else, it is false.

The function `tcrdbopen2' is used in order to open a remote database with a simple server expression.

**bool tcrdbopen2(TCRDB ****rdb***, const char ****expr*****);**

`*rdb*' specifies the remote database object.
`*expr*' specifies the simple server expression. It is composed of two substrings separated by ":". The former field specifies the name or the address of the server. The latter field specifies the port number. If the latter field is omitted, the default port number is specified.
If successful, the return value is true, else, it is false.

The function `tcrdbclose' is used in order to close a remote database object.

**bool tcrdbclose(TCRDB ****rdb*****);**

`*rdb*' specifies the remote database object.
If successful, the return value is true, else, it is false.

The function `tcrdbput' is used in order to store a record into a remote database object.

```
bool tcrdbput(TCRDB *rdb, const void *kbuf, int ksiz, const void *vbuf, int vsiz);
```

`*rdb*' specifies the remote database object.
`*kbuf*' specifies the pointer to the region of the key.
`*ksiz*' specifies the size of the region of the key.
`*vbuf*' specifies the pointer to the region of the value.
`*vsiz*' specifies the size of the region of the value.
If successful, the return value is true, else, it is false.
If a record with the same key exists in the database, it is overwritten.

The function `tcrdbput2' is used in order to store a string record into a remote object.

```
bool tcrdbput2(TCRDB *rdb, const char *kstr, const char *vstr);
```

`*rdb*' specifies the remote database object.
`*kstr*' specifies the string of the key.
`*vstr*' specifies the string of the value.
If successful, the return value is true, else, it is false.
If a record with the same key exists in the database, it is overwritten.

The function `tcrdbputkeep' is used in order to store a new record into a remote database object.

```
bool tcrdbputkeep(TCRDB *rdb, const void *kbuf, int ksiz, const void *vbuf, int vsiz);
```

`*rdb*' specifies the remote database object.
`*kbuf*' specifies the pointer to the region of the key.
`*ksiz*' specifies the size of the region of the key.
`*vbuf*' specifies the pointer to the region of the value.
`*vsiz*' specifies the size of the region of the value.
If successful, the return value is true, else, it is false.
If a record with the same key exists in the database, this function has no effect.

The function `tcrdbputkeep2' is used in order to store a new string record into a remote database object.

**bool tcrdbputkeep2(TCRDB ****rdb***, const char ****kstr***, const char ****vstr***);**

`*rdb*' specifies the remote database object.
`*kstr*' specifies the string of the key.
`*vstr*' specifies the string of the value.
If successful, the return value is true, else, it is false.
If a record with the same key exists in the database, this function has no effect.

The function `tcrdbputcat' is used in order to concatenate a value at the end of the existing record in a remote database object.

**bool tcrdbputcat(TCRDB ****rdb***, const void ****kbuf***, int** ***ksiz*****, const void ****vbuf***, int** ***vsiz*****);**

`*rdb*' specifies the remote database object.
`*kbuf*' specifies the pointer to the region of the key.
`*ksiz*' specifies the size of the region of the key.
`*vbuf*' specifies the pointer to the region of the value.
`*vsiz*' specifies the size of the region of the value.
If successful, the return value is true, else, it is false.
If there is no corresponding record, a new record is created.

The function `tcrdbputcat2' is used in order to concatenate a string value at the end of the existing record in a remote database object.

**bool tcrdbputcat2(TCRDB ****rdb***, const char ****kstr***, const char ****vstr***);**

`*rdb*' specifies the remote database object.
`*kstr*' specifies the string of the key.
`*vstr*' specifies the string of the value.
If successful, the return value is true, else, it is false.
If there is no corresponding record, a new record is created.

The function `tcrdbputshl' is used in order to concatenate a value at the end of the existing record and shift it to the left.

**bool tcrdbputshl(TCRDB **rdb*, const void **kbuf*, int *ksiz*, const void **vbuf*, int *vsiz*, int *width*);**

`*rdb*' specifies the remote database object.
`*kbuf*' specifies the pointer to the region of the key.
`*ksiz*' specifies the size of the region of the key.
`*vbuf*' specifies the pointer to the region of the value.
`*vsiz*' specifies the size of the region of the value.
`*width*' specifies the width of the record.
If successful, the return value is true, else, it is false.
If there is no corresponding record, a new record is created.

The function `tcrdbputshl2' is used in order to concatenate a string value at the end of the existing record and shift it to the left.

**bool tcrdbputshl2(TCRDB **rdb*, const char **kstr*, const char **vstr*, int *width*);**

`*rdb*' specifies the remote database object.
`*kstr*' specifies the string of the key.
`*vstr*' specifies the string of the value.
`*width*' specifies the width of the record.
If successful, the return value is true, else, it is false.
If there is no corresponding record, a new record is created.

The function `tcrdbputnr' is used in order to store a record into a remote database object without response from the server.

**bool tcrdbputnr(TCRDB **rdb*, const void **kbuf*, int *ksiz*, const void **vbuf*, int *vsiz*);**

`*rdb*' specifies the remote database object.
`*kbuf*' specifies the pointer to the region of the key.

`*ksiz*' specifies the size of the region of the key.
`*vbuf*' specifies the pointer to the region of the value.
`*vsiz*' specifies the size of the region of the value.
If successful, the return value is true, else, it is false.
If a record with the same key exists in the database, it is overwritten.

The function `tcrdbputnr2' is used in order to store a string record into a remote object without response from the server.

**bool tcrdbputnr2(TCRDB ****rdb***, const char ****kstr***, const char ****vstr***);**

`*rdb*' specifies the remote database object.
`*kstr*' specifies the string of the key.
`*vstr*' specifies the string of the value.
If successful, the return value is true, else, it is false.
If a record with the same key exists in the database, it is overwritten.

The function `tcrdbout' is used in order to remove a record of a remote database object.

**bool tcrdbout(TCRDB ****rdb***, const void ****kbuf***, int** ***ksiz*****);**

`*rdb*' specifies the remote database object.
`*kbuf*' specifies the pointer to the region of the key.
`*ksiz*' specifies the size of the region of the key.
If successful, the return value is true, else, it is false.

The function `tcrdbout2' is used in order to remove a string record of a remote database object.

**bool tcrdbout2(TCRDB ****rdb***, const char ****kstr***);**

`*rdb*' specifies the remote database object.
`*kstr*' specifies the string of the key.
If successful, the return value is true, else, it is false.

The function `tcrdbget' is used in order to retrieve a record in a remote database object.

**void *tcrdbget(TCRDB ****rdb***, const void ****kbuf***, int** ***ksiz***, int ****sp***);**

`*rdb*' specifies the remote database object.
`*kbuf*' specifies the pointer to the region of the key.
`*ksiz*' specifies the size of the region of the key.
`*sp*' specifies the pointer to the variable into which the size of the region of the return value is assigned.
If successful, the return value is the pointer to the region of the value of the corresponding record. `NULL' is returned if no record corresponds.
Because an additional zero code is appended at the end of the region of the return value, the return value can be treated as a character string. Because the region of the return value is allocated with the `malloc' call, it should be released with the `free' call when it is no longer in use.

The function `tcrdbget2' is used in order to retrieve a string record in a remote database object.

**char *tcrdbget2(TCRDB ****rdb***, const char ****kstr***);**

`*rdb*' specifies the remote database object.
`*kstr*' specifies the string of the key.
If successful, the return value is the string of the value of the corresponding record. `NULL' is returned if no record corresponds.
Because the region of the return value is allocated with the `malloc' call, it should be released with the `free' call when it is no longer in use.

The function `tcrdbget3' is used in order to retrieve records in a remote database object.

**bool tcrdbget3(TCRDB ****rdb***, TCMAP ****recs***);**

`*rdb*' specifies the remote database object.
`*recs*' specifies a map object containing the retrieval keys. As a result of this function, keys existing in the database have the corresponding values and keys not existing in the database are removed.
If successful, the return value is true, else, it is false.

The function `tcrdbvsiz' is used in order to get the size of the value of a record in a remote database object.

```
int tcrdbvsiz(TCRDB *rdb, const void *kbuf, int ksiz);
```

`*rdb*' specifies the remote database object.
`*kbuf*' specifies the pointer to the region of the key.
`*ksiz*' specifies the size of the region of the key.
If successful, the return value is the size of the value of the corresponding record, else, it is -1.

The function `tcrdbvsiz2' is used in order to get the size of the value of a string record in a remote database object.

```
int tcrdbvsiz2(TCRDB *rdb, const char *kstr);
```

`*rdb*' specifies the remote database object.
`*kstr*' specifies the string of the key.
If successful, the return value is the size of the value of the corresponding record, else, it is -1.

The function `tcrdbiterinit' is used in order to initialize the iterator of a remote database object.

```
bool tcrdbiterinit(TCRDB *rdb);
```

`*rdb*' specifies the remote database object.
If successful, the return value is true, else, it is false.
The iterator is used in order to access the key of every record stored in a database.

The function `tcrdbiternext' is used in order to get the next key of the iterator of a remote database object.

```
void *tcrdbiternext(TCRDB *rdb, int *sp);
```

`*rdb*' specifies the remote database object.
`*sp*' specifies the pointer to the variable into which the size of the region of the return value is assigned.
If successful, the return value is the pointer to the region of the next key, else, it is `NULL'. `NULL' is returned when no record is to be get out of the iterator.
Because an additional zero code is appended at the end of the region of the return value, the return value can

be treated as a character string. Because the region of the return value is allocated with the `malloc' call, it should be released with the `free' call when it is no longer in use. The iterator can be updated by multiple connections and then it is not assured that every record is traversed.

The function `tcrdbiternext2' is used in order to get the next key string of the iterator of a remote database object.

**char *tcrdbiternext2(TCRDB ****rdb***);**

`*rdb*' specifies the remote database object.

If successful, the return value is the string of the next key, else, it is `NULL'. `NULL' is returned when no record is to be get out of the iterator.

Because the region of the return value is allocated with the `malloc' call, it should be released with the `free' call when it is no longer in use. The iterator can be updated by multiple connections and then it is not assured that every record is traversed.

The function `tcrdbfwmkeys' is used in order to get forward matching keys in a remote database object.

**TCLIST *tcrdbfwmkeys(TCRDB ****rdb***, const void ****pbuf***, int** ***psiz***, int** ***max***);**

`*rdb*' specifies the remote database object.

`*pbuf*' specifies the pointer to the region of the prefix.

`*psiz*' specifies the size of the region of the prefix.

`*max*' specifies the maximum number of keys to be fetched. If it is negative, no limit is specified.

The return value is a list object of the corresponding keys. This function does never fail. It returns an empty list even if no key corresponds.

Because the object of the return value is created with the function `tclistnew', it should be deleted with the function `tclistdel' when it is no longer in use.

The function `tcrdbfwmkeys2' is used in order to get forward matching string keys in a remote database object.

**TCLIST *tcrdbfwmkeys2(TCRDB *_rdb_, const char *_pstr_, int _max_);**

`_rdb_' specifies the remote database object.
`_pstr_' specifies the string of the prefix.
`_max_' specifies the maximum number of keys to be fetched. If it is negative, no limit is specified.
The return value is a list object of the corresponding keys. This function does never fail. It returns an empty list even if no key corresponds.
Because the object of the return value is created with the function `tclistnew', it should be deleted with the function `tclistdel' when it is no longer in use.

The function `tcrdbaddint' is used in order to add an integer to a record in a remote database object.

**int tcrdbaddint(TCRDB *_rdb_, const void *_kbuf_, int _ksiz_, int _num_);**

`_rdb_' specifies the remote database object.
`_kbuf_' specifies the pointer to the region of the key.
`_ksiz_' specifies the size of the region of the key.
`_num_' specifies the additional value.
If successful, the return value is the summation value, else, it is `INT_MIN'.
If the corresponding record exists, the value is treated as an integer and is added to. If no record corresponds, a new record of the additional value is stored.

The function `tcrdbadddouble' is used in order to add a real number to a record in a remote database object.

**double tcrdbadddouble(TCRDB *_rdb_, const void *_kbuf_, int _ksiz_, double _num_);**

`_rdb_' specifies the remote database object.
`_kbuf_' specifies the pointer to the region of the key.
`_ksiz_' specifies the size of the region of the key.
`_num_' specifies the additional value.
If successful, the return value is the summation value, else, it is Not-a-Number.
If the corresponding record exists, the value is treated as a real number and is added to. If no record

corresponds, a new record of the additional value is stored.

The function `tcrdbext' is used in order to call a function of the script language extension.

**void *tcrdbext(TCRDB ****rdb***, const char ****name***, int** ***opts***, const void ****kbuf***, int** ***ksiz***, const void ****vbuf***, int** ***vsiz***, int ****sp***);**

`*rdb*' specifies the remote database object.
`*name*' specifies the function name.
`*opts*' specifies options by bitwise-or: `RDBXOLCKREC' for record locking, `RDBXOLCKGLB' for global locking.
`*kbuf*' specifies the pointer to the region of the key.
`*ksiz*' specifies the size of the region of the key.
`*vbuf*' specifies the pointer to the region of the value.
`*vsiz*' specifies the size of the region of the value.
`*sp*' specifies the pointer to the variable into which the size of the region of the return value is assigned.
If successful, the return value is the pointer to the region of the value of the response. `NULL' is returned on failure.
Because an additional zero code is appended at the end of the region of the return value, the return value can be treated as a character string. Because the region of the return value is allocated with the `malloc' call, it should be released with the `free' call when it is no longer in use.

The function `tcrdbext2' is used in order to call a function of the script language extension with string parameters.

**char *tcrdbext2(TCRDB ****rdb***, const char ****name***, int** ***opts***, const char ****kstr***, const char ****vstr***);**

`*rdb*' specifies the remote database object.
`*name*' specifies the function name.
`*opts*' specifies options by bitwise-or: `RDBXOLCKREC' for record locking, `RDBXOLCKGLB' for global locking.

`*kstr*' specifies the string of the key.
`*vstr*' specifies the string of the value.
If successful, the return value is the string of the value of the response. `NULL' is returned on failure.
Because the region of the return value is allocated with the `malloc' call, it should be released with the `free' call when it is no longer in use.

The function `tcrdbsync' is used in order to synchronize updated contents of a remote database object with the file and the device.

**bool tcrdbsync(TCRDB ****rdb***);**

`*rdb*' specifies the remote database object.
If successful, the return value is true, else, it is false.

The function `tcrdboptimize' is used in order to optimize the storage of a remove database object.

**bool tcrdboptimize(TCRDB ****rdb***, const char ****params***);**

`*rdb*' specifies the remote database object.
`*params*' specifies the string of the tuning parameters. If it is `NULL', it is not used.
If successful, the return value is true, else, it is false.

The function `tcrdbvanish' is used in order to remove all records of a remote database object.

**bool tcrdbvanish(TCRDB ****rdb***);**

`*rdb*' specifies the remote database object.
If successful, the return value is true, else, it is false.

The function `tcrdbcopy' is used in order to copy the database file of a remote database object.

**bool tcrdbcopy(TCRDB ****rdb***, const char ****path***);**

`*rdb*' specifies the remote database object.
`*path*' specifies the path of the destination file. If it begins with `@', the trailing substring is executed as a

command line.
If successful, the return value is true, else, it is false. False is returned if the executed command returns non-zero code.
The database file is assured to be kept synchronized and not modified while the copying or executing operation is in progress. So, this function is useful to create a backup file of the database file.

The function `tcrdbrestore' is used in order to restore the database file of a remote database object from the update log.

**bool tcrdbrestore(TCRDB ****rdb***, const char ****path***, uint64_t** ***ts***, int** ***opts***);**

`*rdb*' specifies the remote database object.
`*path*' specifies the path of the update log directory.
`*opts*' specifies options by bitwise-or: `RDBROCHKCON' for consistency checking.
`*ts*' specifies the beginning time stamp in microseconds.
If successful, the return value is true, else, it is false.

The function `tcrdbsetmst' is used in order to set the replication master of a remote database object.

**bool tcrdbsetmst(TCRDB ****rdb***, const char ****host***, int** ***port***, uint64_t** ***ts***, int** ***opts***);**

`*rdb*' specifies the remote database object.
`*host*' specifies the name or the address of the server. If it is `NULL', replication of the database is disabled.
`*port*' specifies the port number.
`*ts*' specifies the beginning timestamp in microseconds.
`*opts*' specifies options by bitwise-or: `RDBROCHKCON' for consistency checking.
If successful, the return value is true, else, it is false.

The function `tcrdbsetmst2' is used in order to set the replication master of a remote database object with a simple server expression.

**bool tcrdbsetmst2(TCRDB ****rdb***, const char ****expr***, uint64_t** ***ts***, int** ***opts***);**

`*rdb*' specifies the remote database object.
`*expr*' specifies the simple server expression. It is composed of two substrings separated by ":". The former field specifies the name or the address of the server. The latter field specifies the port number. If the latter field is omitted, the default port number is specified.
`*ts*' specifies the beginning timestamp in microseconds.
`*opts*' specifies options by bitwise-or: `RDBROCHKCON' for consistency checking.
If successful, the return value is true, else, it is false.

The function `tcrdbrnum' is used in order to get the number of records of a remote database object.

**uint64_t tcrdbrnum(TCRDB ****rdb***);**

`*rdb*' specifies the remote database object.
The return value is the number of records or 0 if the object does not connect to any database server.

The function `tcrdbsize' is used in order to get the size of the database of a remote database object.

**uint64_t tcrdbsize(TCRDB ****rdb***);**

`*rdb*' specifies the remote database object.
The return value is the size of the database or 0 if the object does not connect to any database server.

The function `tcrdbstat' is used in order to get the status string of the database of a remote database object.

**char *tcrdbstat(TCRDB ****rdb***);**

`*rdb*' specifies the remote database object.
The return value is the status message of the database or `NULL' if the object does not connect to any database server. The message format is TSV. The first field of each line means the parameter name and the second field means the value.
Because the region of the return value is allocated with the `malloc' call, it should be released with the `free' call when it is no longer in use.

The function `tcrdbmisc' is used in order to call a versatile function for miscellaneous operations of a remote database object.

**TCLIST *tcrdbmisc(TCRDB ****rdb***, const char ****name***, int** ***opts***, const TCLIST ****args***);**

`*rdb*' specifies the remote database object.
`*name*' specifies the name of the function. All databases support "put", "out", "get", "putlist", "outlist", and "getlist". "put" is to store a record. It receives a key and a value, and returns an empty list. "out" is to remove a record. It receives a key, and returns an empty list. "get" is to retrieve a record. It receives a key, and returns a list of the values. "putlist" is to store records. It receives keys and values one after the other, and returns an empty list. "outlist" is to remove records. It receives keys, and returns an empty list. "getlist" is to retrieve records. It receives keys, and returns keys and values of corresponding records one after the other.
`*opts*' specifies options by bitwise-or: `RDBMONOULOG' for omission of the update log.
`*args*' specifies a list object containing arguments.
If successful, the return value is a list object of the result. `NULL' is returned on failure.
Because the object of the return value is created with the function `tclistnew', it should be deleted with the function `tclistdel' when it is no longer in use.

## API of the Table Extension

The function `tcrdbtblput' is used in order to store a record into a remote database object.

**bool tcrdbtblput(TCRDB ****rdb***, const void ****pkbuf***, int** ***pksiz***, TCMAP ****cols***);**

`*rdb*' specifies the remote database object.
`*pkbuf*' specifies the pointer to the region of the primary key.
`*pksiz*' specifies the size of the region of the primary key.
`*cols*' specifies a map object containing columns.
If successful, the return value is true, else, it is false.
If a record with the same key exists in the database, it is overwritten.

The function `tcrdbtblputkeep' is used in order to store a new record into a remote database object.

**bool tcrdbtblputkeep(TCRDB ****rdb***, const void ****pkbuf***, int** ***pksiz***, TCMAP ****cols***);**

`*rdb*' specifies the remote database object.
`*pkbuf*' specifies the pointer to the region of the primary key.
`*pksiz*' specifies the size of the region of the primary key.
`*cols*' specifies a map object containing columns.
If successful, the return value is true, else, it is false.
If a record with the same key exists in the database, this function has no effect.

The function `tcrdbtblputcat' is used in order to concatenate columns of the existing record in a remote database object.

```
bool tcrdbtblputcat(TCRDB *rdb, const void *pkbuf, int pksiz, TCMAP *cols);
```

`*rdb*' specifies the remote database object.
`*pkbuf*' specifies the pointer to the region of the primary key.
`*pksiz*' specifies the size of the region of the primary key.
`*cols*' specifies a map object containing columns.
If successful, the return value is true, else, it is false.
If there is no corresponding record, a new record is created.

The function `tcrdbtblout' is used in order to remove a record of a remote database object.

```
bool tcrdbtblout(TCRDB *rdb, const void *pkbuf, int pksiz);
```

`*rdb*' specifies the remote database object.
`*pkbuf*' specifies the pointer to the region of the primary key.
`*pksiz*' specifies the size of the region of the primary key.
If successful, the return value is true, else, it is false.

The function `tcrdbtblget' is used in order to retrieve a record in a remote database object.

```
TCMAP *tcrdbtblget(TCRDB *rdb, const void *pkbuf, int pksiz);
```

`*rdb*' specifies the remote database object.
`*pkbuf*' specifies the pointer to the region of the primary key.

`*pksiz*' specifies the size of the region of the primary key.
If successful, the return value is a map object of the columns of the corresponding record. `NULL' is returned if no record corresponds.
Because the object of the return value is created with the function `tcmapnew', it should be deleted with the function `tcmapdel' when it is no longer in use.

The function `tcrdbtblsetindex' is used in order to set a column index to a remote database object.

**bool tcrdbtblsetindex(TCRDB ****rdb***, const char ****name***, int** ***type*****);**

`*rdb*' specifies the remote database object.
`*name*' specifies the name of a column. If the name of an existing index is specified, the index is rebuilt. An empty string means the primary key.
`*type*' specifies the index type: `RDBITLEXICAL' for lexical string, `RDBITDECIMAL' for decimal string, `RDBITTOKEN' for token inverted index, `RDBITQGRAM' for q-gram inverted index. If it is `RDBITOPT', the index is optimized. If it is `RDBITVOID', the index is removed. If `RDBITKEEP' is added by bitwise-or and the index exists, this function merely returns failure.
If successful, the return value is true, else, it is false.

The function `tcrdbtblgenuid' is used in order to generate a unique ID number of a remote database object.

**int64_t tcrdbtblgenuid(TCRDB ****rdb*****);**

`*rdb*' specifies the remote database object.
The return value is the new unique ID number or -1 on failure.

The function `tcrdbqrynew' is used in order to create a query object.

**RDBQRY *tcrdbqrynew(TCRDB ****rdb*****);**

`*rdb*' specifies the remote database object.
The return value is the new query object.

The function `tcrdbqrydel' is used in order to delete a query object.

**void tcrdbqrydel(RDBQRY ****qry***);**

`*qry*' specifies the query object.

The function `tcrdbqryaddcond' is used in order to add a narrowing condition to a query object.

**void tcrdbqryaddcond(RDBQRY ****qry***, const char ****name***, int** ***op*****, const char ****expr***);**

`*qry*' specifies the query object.
`*name*' specifies the name of a column. An empty string means the primary key.
`*op*' specifies an operation type: `RDBQCSTREQ' for string which is equal to the expression, `RDBQCSTRINC' for string which is included in the expression, `RDBQCSTRBW' for string which begins with the expression, `RDBQCSTREW' for string which ends with the expression, `RDBQCSTRAND' for string which includes all tokens in the expression, `RDBQCSTROR' for string which includes at least one token in the expression, `RDBQCSTROREQ' for string which is equal to at least one token in the expression, `RDBQCSTRRX' for string which matches regular expressions of the expression, `RDBQCNUMEQ' for number which is equal to the expression, `RDBQCNUMGT' for number which is greater than the expression, `RDBQCNUMGE' for number which is greater than or equal to the expression, `RDBQCNUMLT' for number which is less than the expression, `RDBQCNUMLE' for number which is less than or equal to the expression, `RDBQCNUMBT' for number which is between two tokens of the expression, `RDBQCNUMOREQ' for number which is equal to at least one token in the expression, `RDBQCFTSPH' for full-text search with the phrase of the expression, `RDBQCFTSAND' for full-text search with all tokens in the expression, `RDBQCFTSOR' for full-text search with at least one token in the expression, `RDBQCFTSEX' for full-text search with the compound expression. All operations can be flagged by bitwise-or: `RDBQCNEGATE' for negation, `RDBQCNOIDX' for using no index.
`*expr*' specifies an operand exression.

The function `tcrdbqrysetorder' is used in order to set the order of a query object.

**void tcrdbqrysetorder(RDBQRY ****qry***, const char ****name***, int** ***type*****);**

`*qry*' specifies the query object.
`*name*' specifies the name of a column. An empty string means the primary key.

`*type*' specifies the order type: `RDBQOSTRASC' for string ascending, `RDBQOSTRDESC' for string descending, `RDBQONUMASC' for number ascending, `RDBQONUMDESC' for number descending.

The function `tcrdbqrysetlimit' is used in order to set the limit number of records of the result of a query object.

**void tcrdbqrysetlimit(RDBQRY ****qry***, int** ***max***, int** ***skip***);**

`*qry*' specifies the query object.
`*max*' specifies the maximum number of records of the result. If it is negative, no limit is specified.
`*skip*' specifies the number of skipped records of the result. If it is not more than 0, no record is skipped.

The function `tcrdbqrysearch' is used in order to execute the search of a query object.

**TCLIST *tcrdbqrysearch(RDBQRY ****qry***);**

`*qry*' specifies the query object.
The return value is a list object of the primary keys of the corresponding records. This function does never fail. It returns an empty list even if no record corresponds.
Because the object of the return value is created with the function `tclistnew', it should be deleted with the function `tclistdel' when it is no longer in use.

The function `tcrdbqrysearchout' is used in order to remove each record corresponding to a query object.

**bool tcrdbqrysearchout(RDBQRY ****qry***);**

`*qry*' specifies the query object of the database.
If successful, the return value is true, else, it is false.

The function `tcrdbqrysearchget' is used in order to get records corresponding to the search of a query object.

**TCLIST *tcrdbqrysearchget(RDBQRY ****qry***);**

`*qry*' specifies the query object.

The return value is a list object of zero separated columns of the corresponding records.
This function does never fail. It returns an empty list even if no record corresponds. Each element of the list can be treated with the function `tcrdbqryrescols'. Because the object of the return value is created with the function `tclistnew', it should be deleted with the function `tclistdel' when it is no longer in use.

The function `tcrdbqryrescols' is used in order to get columns of a record in a search result.

**`TCMAP *tcrdbqryrescols(TCLIST *res, int index);`**

`*res*' specifies a list of zero separated columns of the search result.
`*index*' the index of a element of the search result.
The return value is a map object containing columns.
Because the object of the return value is created with the function `tcmapnew', it should be deleted with the function `tcmapdel' when it is no longer in use.

The function `tcrdbqrysearchcount' is used in order to get the count of corresponding records of a query object.

**`int tcrdbqrysearchcount(RDBQRY *qry);`**

`*qry*' specifies the query object.
The return value is the count of corresponding records or 0 on failure.

The function `tcrdbqryhint' is used in order to get the hint string of a query object.

**`const char *tcrdbqryhint(RDBQRY *qry);`**

`*qry*' specifies the query object.
The return value is the hint string.
This function should be called after the query execution by `tcrdbqrysearch' and so on. The region of the return value is overwritten when this function is called again.

The function `tcrdbmetasearch' is used in order to retrieve records with multiple query objects and get the set of the result.

**TCLIST *tcrdbmetasearch(RDBQRY *****qrys***, int** ***num***, int** ***type***);**

`*qrys*' specifies an array of the query objects.
`*num*' specifies the number of elements of the array.
`*type*' specifies a set operation type: `RDBMSUNION' for the union set, `RDBMSISECT' for the intersection set, `RDBMSDIFF' for the difference set.
The return value is a list object of the primary keys of the corresponding records. This function does never fail. It returns an empty list even if no record corresponds.
If the first query object has the order setting, the result array is sorted by the order. Because the object of the return value is created with the function `tclistnew', it should be deleted with the function `tclistdel' when it is no longer in use.

The function `tcrdbparasearch' is used in order to search records for multiple servers in parallel.

**TCLIST *tcrdbparasearch(RDBQRY *****qrys***, int** ***num***);**

`*qrys*' specifies an array of the query objects.
`*num*' specifies the number of elements of the array.
The return value is a list object of zero separated columns of the corresponding records.
This function does never fail. It returns an empty list even if no record corresponds. Each element of the list can be treated with the function `tcrdbqryrescols'. Because the object of the return value is created with the function `tclistnew', it should be deleted with the function `tclistdel' when it is no longer in use.

## Example Code

The following code is an example to use a remote database.

```
#include <tcrdb.h>
#include <stdlib.h>
#include <stdbool.h>
#include <stdint.h>
```

```
int main(int argc, char **argv){

  TCRDB *rdb;
  int ecode;
  char *value;

  /* create the object */
  rdb = tcrdbnew();

  /* connect to the server */
  if(!tcrdbopen(rdb, "localhost", 1978)){
    ecode = tcrdbecode(rdb);
    fprintf(stderr, "open error: %s\n", tcrdberrmsg(ecode));
  }

  /* store records */
  if(!tcrdbput2(rdb, "foo", "hop") ||
     !tcrdbput2(rdb, "bar", "step") ||
     !tcrdbput2(rdb, "baz", "jump")){
    ecode = tcrdbecode(rdb);
    fprintf(stderr, "put error: %s\n", tcrdberrmsg(ecode));
  }

  /* retrieve records */
  value = tcrdbget2(rdb, "foo");
  if(value){
    printf("%s\n", value);
    free(value);
  } else {
    ecode = tcrdbecode(rdb);
    fprintf(stderr, "get error: %s\n", tcrdberrmsg(ecode));
  }

  /* close the connection */
  if(!tcrdbclose(rdb)){
    ecode = tcrdbecode(rdb);
    fprintf(stderr, "close error: %s\n", tcrdberrmsg(ecode));
  }

  /* delete the object */
  tcrdbdel(rdb);

  return 0;
}
```

The following code is an example to use a remote database with the table extension.

```
#include <tcrdb.h>
#include <stdlib.h>
#include <stdbool.h>
#include <stdint.h>

int main(int argc, char **argv){

  TCRDB *rdb;
  int ecode, pksiz, i, rsiz;
  char pkbuf[256];
  const char *rbuf, *name;
  TCMAP *cols;
  RDBQRY *qry;
  TCLIST *res;

  /* create the object */
  rdb = tcrdbnew();

  /* connect to the server */
  if(!tcrdbopen(rdb, "localhost", 1978)){
    ecode = tcrdbecode(rdb);
    fprintf(stderr, "open error: %s\n", tcrdberrmsg(ecode));
  }

  /* store a record */
  pksiz = sprintf(pkbuf, "%ld", (long)tcrdbtblgenuid(rdb));
  cols = tcmapnew3("name", "mikio", "age", "30", "lang", "ja,en,c", NULL);
  if(!tcrdbtblput(rdb, pkbuf, pksiz, cols)){
    ecode = tcrdbecode(rdb);
    fprintf(stderr, "put error: %s\n", tcrdberrmsg(ecode));
  }
  tcmapdel(cols);
```

```
/* store a record in a naive way */
pksiz = sprintf(pkbuf, "12345");
cols = tcmapnew();
tcmapput2(cols, "name", "falcon");
tcmapput2(cols, "age", "31");
tcmapput2(cols, "lang", "ja");
if(!tcrdbtblput(rdb, pkbuf, pksiz, cols)){
  ecode = tcrdbecode(rdb);
  fprintf(stderr, "put error: %s\n", tcrdberrmsg(ecode));
}
tcmapdel(cols);

/* search for records */
qry = tcrdbqrynew(rdb);
tcrdbqryaddcond(qry, "age", RDBQCNUMGE, "20");
tcrdbqryaddcond(qry, "lang", RDBQCSTROR, "ja,en");
tcrdbqrysetorder(qry, "name", RDBQOSTRASC);
tcrdbqrysetlimit(qry, 10, 0);
res = tcrdbqrysearch(qry);
for(i = 0; i < tclistnum(res); i++){
  rbuf = tclistval(res, i, &rsiz);
  cols = tcrdbtblget(rdb, rbuf, rsiz);
  if(cols){
    printf("%s", rbuf);
    tcmapiterinit(cols);
    while((name = tcmapiternext2(cols)) != NULL){
      printf("\t%s\t%s", name, tcmapget2(cols, name));
    }
    printf("\n");
    tcmapdel(cols);
  }
}
tclistdel(res);
tcrdbqrydel(qry);

/* close the connection */
if(!tcrdbclose(rdb)){
  ecode = tcrdbecode(rdb);
  fprintf(stderr, "close error: %s\n", tcrdberrmsg(ecode));
}
```

```
  /* delete the object */
  tcrdbdel(rdb);

  return 0;
}
```

### How to Use the Library

The API of C is available by programs conforming to the C89 (ANSI C) standard or the C99 standard. As the header files of Tokyo Tyrant are provided as `` `tcrdb.h' ``, applications should include it to use the API. As the library is provided as `` `libtokyotyrant.a' `` and `` `libtokyotyrant.so' `` and they depend on `` `libtokyocabinet.so' ``, `` `libz.so' ``, `` `libbz2.so' ``, `` `libresolv.so' ``, `` `libnsl.so' ``, `` `libdl.so' ``, `` `librt.so' ``, `` `libpthread.so' ``, `` `libm.so' ``, and `` `libc.so' ``, linker options corresponding to them are required by the build command. The typical build command is the following.

```
gcc -I/usr/local/include tt_example.c -o tt_example \
  -L/usr/local/lib -ltokyotyrant -ltokyocabinet -lz -lbz2 \
  -lresolv -lnsl -ldl -lrt -lpthread -lm -lc
```

You can also use Tokyo Tyrant in programs written in C++. Because each header is wrapped in C linkage (`` `extern "C"' `` block), you can simply include them into your C++ programs.

---

## Lua Extension

The database server can starts reading a Lua script file by the `` `-ext' `` option. Clients can call functions defined in the script file by the \``tcrdbext`' function of the remote database API.

## User-defined Functions

You can define some arbitrary functions in the script file. Each function receives two string parameters of the key and the value. The return value is sent back to the client. If the function returns \`nil', the server send the error code to the client.

Two kinds of locking options are provided to \``tcrdbext`'. One is global locking which means that only one thread can operate the function at the same time. The other is record locking which means that only one thread can operate the record of the specified key at the same time.

Note that instances of Lua interpreter are handled separately by each native thread. Because global variables of Lua are not useful to share some data among native threads or sessions, shared data should be handled in the database or by the stash functions.

## Built-in Functions

The following build-in functions for database operations are available in user defined functions. The type of \`key' and \`value' parameters should be string or number. If number is given, it is converted as decimal string.

**`_eval(`*`chunk`*`)`**

Evaluate a Lua chunk in each native thread.
\`*chunk*' specifies the Lua chunk string.
If successful, the return value is true, else, it is false.

**`_log(`*`message, level`*`)`**

Print a message into the server log.
\`*message*' specifies the message string.
\`*level*' specifies the log level; 0 for debug, 1 for information, 2 for error, 3 for system. It can be omitted

and the default value is 1.

**_put(*key*, *value*)**

Store a record.
`*key*' specifies the key.
`*value*' specifies the value.
If successful, the return value is true, else, it is false.

**_putkeep(*key*, *value*)**

Store a new record.
`*key*' specifies the key.
`*value*' specifies the value.
If successful, the return value is true, else, it is false.

**_putcat(*key*, *value*)**

Concatenate a value at the end of the existing record.
`*key*' specifies the key.
`*value*' specifies the value.
If successful, the return value is true, else, it is false.

**_out(*key*)**

Remove a record.
`*key*' specifies the key.
If successful, the return value is true, else, it is false.

**_get(*key*)**

Retrieve a record.
`*key*' specifies the key.
If successful, the return value is the value of the corresponding record. `nil' is returned if no record corresponds.

**_vsiz(*key*)**

Get the size of the value of a record.

`*key*' specifies the key.
If successful, the return value is the size of the value of the corresponding record, else, it is -1.

**`_iterinit()`**

Initialize the iterator.
If successful, the return value is true, else, it is false.

**`_iternext()`**

Get the next key of the iterator.
If successful, the return value is the next key, else, it is `nil'. `nil' is returned when no record is to be get out of the iterator.

**`_fwmkeys(`*`prefix, max`*`)`**

Get forward matching keys.
The return value an array of the keys of the corresponding records. This function does never fail. It returns an empty array even if no record corresponds.

**`_addint(`*`key, value`*`)`**

Add an integer to a record.
`*key*' specifies the key.
`*value*' specifies the value.
If successful, the return value is the summation value, else, it is `nil'.

**`_adddouble(`*`key, value`*`)`**

Add a real number to a record.
`*key*' specifies the key.
`*value*' specifies the value.
If successful, the return value is the summation value, else, it is `nil'.

**`_vanish()`**

Remove all records.
If successful, the return value is true, else, it is false.

**`_rnum()`**

Get the number of records.

The return value is the number of records.

**`_size()`**

Get the size of the database file.

The return value is the size of the database file.

**`_misc(`*`name`*`, `*`args`*`, ...)`**

Call a versatile function for miscellaneous operations.

\`*name*' specifies the name of the function. If it begins with "$", the update log is omitted.

\`*args*' specifies an array of arguments.

If successful, the return value is an array of the result. \`nil' is returned on failure.

**`_foreach(`*`func`*`)`**

Process each record atomically.

\`*func*' the iterator function called for each record. It receives two parameters of the key and the value, and returns true to continue iteration or false to stop iteration.

If successful, the return value is true, else, it is false.

**`_mapreduce(`*`mapper`*`, `*`reducer`*`, `*`keys`*`)`**

Perform operations based on MapReduce.

\`*mapper*' specifies the mapper function. It is called for each target record and receives the key, the value, and the function to emit the mapped records. The emitter function receives a key and a value. The mapper function should return true normally or false on failure.

\`*reducer*' specifies the reducer function. It is called for each record generated by sorting emitted records by keys, and receives the key and an array of values. The reducer function should return true normally or false on failure.

\`*keys*' specifies the keys of target records. If it is not defined, every record in the database is processed.

If successful, the return value is true, else, it is false.

**`_stashput(`*`key`*`, `*`value`*`)`**

Store a record into the stash.

`*key*' specifies the key.
`*value*' specifies the value.
The return value is always true.

**`_stashout(`*`key`*`)`**

Remove a record from the stash.
`*key*' specifies the key.
If successful, the return value is true, else, it is false.

**`_stashget(`*`key`*`)`**

Retrieve a record in the stash.
`*key*' specifies the key.
If successful, the return value is the value of the corresponding record. `nil' is returned if no record corresponds.

**`_stashvanish()`**

Remove all records of the stash.

**`_stashforeach(`*`func`*`)`**

Process each record atomically of the stash.
`*func*' the iterator function called for each record. It receives two parameters of the key and the value, and returns true to continue iteration or false to stop iteration.
If successful, the return value is true, else, it is false.

**`_lock(`*`key`*`)`**

Lock an arbitrary key.
`*key*' specifies the key. The locked key should be unlocked in the same operation.
If successful, the return value is true, else, it is false.

**`_unlock(`*`key`*`)`**

Unock an arbitrary key.
`*key*' specifies the key.
If successful, the return value is true, else, it is false.

**_pack(*format*, *ary*, *...*)**

Serialize an array of numbers into a string.

`*format*' specifies the format string. It should be composed of conversion characters; `c' for int8_t, `C' for uint8_t, `s' for int16_t, `S' for uint16_t, `i' for int32_t, `I' for uint32_t, `l' for int64_t, `L' for uint64_t, `f' for float, `d' for double, `n' for uint16_t in network byte order, `N' for uint32_t in network byte order, `M' for uint64_t in network byte order, and `w' for BER encoding. They can be trailed by a numeric expression standing for the iteration count or by `*' for the rest all iteration.

`*ary*' specifies the array of numbers.

The return value is the serialized string.

**_unpack(*format*, *str*)**

Deserialize a binary string into an array of numbers.

`*format*' specifies the format string. It should be composed of conversion characters as with `_pack'.

`*str*' specifies the binary string.

The return value is the deserialized array.

**_split(*str*, *delims*)**

Split a string into substrings.

`*str*' specifies the string.

`*delims*' specifies a string including separator characters. If it is not defined, the zero code is specified.

The return value is an array of substrings.

**_codec(*mode*, *str*)**

Encode or decode a string.

`*mode*' specifies the encoding method; "url" for URL encoding, "~url" for URL decoding, "base" for Base64 encoding, "~base" for Base64 decoding, "hex" for hexadecimal encoding, "~hex" for hexadecimal decoding, "pack" for PackBits encoding, "~pack" for PackBits decoding, "tcbs" for TCBS encoding, "~tcbs" for TCBS decoding, "deflate" for Deflate encoding, "~deflate" for Deflate decoding, "gzip" for GZIP encoding, "~gzip" for GZIP decoding, "bzip" for BZIP2 encoding, "~bzip" for BZIP2 decoding, "xml" for XML escaping, "~xml" for XML unescaping.

`*str*' specifies the string.

The return value is the encoded or decoded string.

**`_hash(`*`mode`*`, `*`str`*`)`**

Get the hash value of a string.

\`*mode*' specifies the hash method; "md5" for MD5 in hexadecimal format, "md5raw" for MD5 in raw format, "crc32" for CRC32 checksum number.

\`*str*' specifies the string.

The return value is the hash value.

**`_bit(`*`mode`*`, `*`num`*`, `*`aux`*`)`**

Perform bit operation of an integer.

\`*mode*' specifies the operator; "and" for bitwise-and operation, "or" for bitwise-or operation, "xor" for bitwise-xor operation, "not" for bitwise-not operation, "left" for left shift operation, "right" for right shift operation.

\`*num*' specifies the integer, which is treated as a 32-bit unsigned integer.

\`*aux*' specifies the auxiliary operand for some operators.

The return value is the result value.

**`_strstr(`*`str`*`, `*`pattern`*`, `*`alt`*`)`**

Perform substring matching or replacement without evaluating any meta character.

\`*str*' specifies the source string.

\`*pattern*' specifies the matching pattern.

\`*alt*' specifies the alternative string corresponding for the pattern. If it is not defined, matching check is performed.

If the alternative string is specified, the converted string is returned. If the alternative string is not specified, the index of the substring matching the given pattern or 0 is returned.

**`_regex(`*`str`*`, `*`pattern`*`, `*`alt`*`)`**

Perform pattern matching or replacement with regular expressions.

\`*str*' specifies the source string.

\`*pattern*' specifies the pattern of regular expressions.

\`*alt*' specifies the alternative string corresponding for the pattern. If it is not defined, matching check is performed.

If the alternative string is specified, the converted string is returned. If the alternative string is not specified, the boolean value of whether the source string matches the pattern is returned.

**`_ucs(`*`data`*`)`**

Convert a UTF-8 string into a UCS-2 array or its inverse.

\`*data*' specifies the target data. If it is a string, convert it into a UCS-array. If it is an array, convert it into a UTF-8 string.

The return value is the result data.

**`_dist(`*`astr`*`, `*`bstr`*`, `*`isutf`*`)`**

Calculate the edit distance of two UTF-8 strings.

\`*astr*' specifies a string.

\`*bstr*' specifies the other string.

\`*isutf*' specifies whether to calculate cost by Unicode character. If it is not defined, false is specified and calculate cost by ASCII character.

The return value is the edit distance which is known as the Levenshtein distance.

**`_isect(ary, ...)`**

Calculate the intersection set of arrays.

\`*ary*' specifies the arrays. Arbitrary number of arrays can be specified as the parameter list.

The return value is the array of the intersection set.

**`_union(ary, ...)`**

Calculate the union set of arrays.

\`*ary*' specifies the arrays. Arbitrary number of arrays can be specified as the parameter list.

The return value is the array of the union set.

**`_time()`**

Get the time of day in seconds.

The return value is the time of day in seconds. The accuracy is in microseconds.

**`_sleep(`*`sec`*`)`**

Suspend execution for the specified interval.
`*sec*' specifies the interval in seconds.
If successful, the return value is true, else, it is false.

**`_stat(`*`path`*`)`**

Get the status of a file.
`*path*' specifies the path of the file.
If successful, the return value is a table containing status, else, it is `nil'. There are keys of status name; "dev", "ino", "mode", "nlink", "uid", "gid", "rdev", "size", "blksize", "blocks", "atime", "mtime", "ctime", which have same meanings of the POSIX "stat" call. Additionally, "_regular" for whether the file is a regular file, "_directory" for whether the file is a directory, "_readable" for whether the file is readable by the process, "_writable" for whether the file is writable by the process, "_executable" for whether the file is executable by the process, "_realpath" for the real path of the file, are supported.

**`_glob(`*`pattern`*`)`**

Find pathnames matching a pattern.
`*pattern*' specifies the matching pattern. "?" and "*" are meta characters.
The return value is an array of matched paths. If no path is matched, an empty array is returned.

**`_remove(`*`path`*`)`**

Remove a file or a directory and its sub ones recursively.
`*path*' specifies the path of the link.
If successful, it is true, else, it is false.

**`_mkdir(`*`path`*`)`**

Create a directory.
`*path*' specifies the path of the directory.
If successful, it is true, else, it is false.

Built-in functions, whose names start with "_", cannot be called directly by clients. When the server starts, the function `**`_begin`**' is called implicitly if it has been defined. When the server starts, the function `**`_end`**' is called implicitly if it has been defined.

The global variable `**_version**' contains the version information of the server. The global variable `**_pid**' contains the process ID. The global variable `**_sid**' contains the server ID. The global variable `**_thnum**' contains the number of native threads. The global variable `**_thid**' contains the ID number of each native thread.

## Example Code

The following code is an example to increment the record value and store as a decimal number string. This function should be called with the record locking option to ensure atomicity.

```
function incr(key, value)
   value = tonumber(value)
   if not value then
      return nil
   end
   local old = tonumber(_get(key))
   if old then
      value = value + old
   end
   if not _put(key, value) then
      return nil
   end
   return value
end
```

---

# Protocol

The protocol between the server and clients stands on TCP/IP. By default, the service port is bound to every address of the local host and the port number is 1978. Each session of the server and a client is composed of a request and a response. The server speaks three protocols at the same port.

## Original Binary Protocol

In the original binary protocol, requests are classified into the following commands. Structure of request and response is determined by the command. The byte order of integer in request and response is big endian.

**`put`**: for the function `tcrdbput'

Request: **`[magic:2][ksiz:4][vsiz:4][kbuf:*][vbuf:*]`**

Two bytes of the command ID: 0xC8 and 0x10
A 32-bit integer standing for the length of the key
A 32-bit integer standing for the length of the value
Arbitrary data of the key
Arbitrary data of the value

Response: **`[code:1]`**

An 8-bit integer whose value is 0 on success or another on failure

**`putkeep`**: for the function `tcrdbputkeep'

Request: **`[magic:2][ksiz:4][vsiz:4][kbuf:*][vbuf:*]`**

Two bytes of the command ID: 0xC8 and 0x11
A 32-bit integer standing for the length of the key
A 32-bit integer standing for the length of the value
Arbitrary data of the key
Arbitrary data of the value

Response: **`[code:1]`**

An 8-bit integer whose value is 0 on success or another on failure

**`putcat`**: for the function `tcrdbputcat'

Request: **`[magic:2][ksiz:4][vsiz:4][kbuf:*][vbuf:*]`**

Two bytes of the command ID: 0xC8 and 0x12
A 32-bit integer standing for the length of the key
A 32-bit integer standing for the length of the value
Arbitrary data of the key
Arbitrary data of the value

Response: **`[code:1]`**

An 8-bit integer whose value is 0 on success or another on failure

**`putshl`**: for the function `tcrdbputshl'

Request: **`[magic:2][ksiz:4][vsiz:4][width:4][kbuf:*][vbuf:*]`**

Two bytes of the command ID: 0xC8 and 0x13
A 32-bit integer standing for the length of the key
A 32-bit integer standing for the length of the value
A 32-bit integer standing for the width
Arbitrary data of the key
Arbitrary data of the value

Response: **`[code:1]`**

An 8-bit integer whose value is 0 on success or another on failure

**`putnr`**: for the function `tcrdbputnr'

Request: **`[magic:2][ksiz:4][vsiz:4][kbuf:*][vbuf:*]`**

Two bytes of the command ID: 0xC8 and 0x18
A 32-bit integer standing for the length of the key
A 32-bit integer standing for the length of the value
Arbitrary data of the key
Arbitrary data of the value

Response: (none)

**`out`**: for the function `tcrdbout'

Request: **`[magic:2][ksiz:4][kbuf:*]`**

Two bytes of the command ID: 0xC8 and 0x20
A 32-bit integer standing for the length of the key
Arbitrary data of the key

Response: **`[code:1]`**

An 8-bit integer whose value is 0 on success or another on failure

**`get`**: for the function `tcrdbget'

Request: **`[magic:2][ksiz:4][kbuf:*]`**

Two bytes of the command ID: 0xC8 and 0x30
A 32-bit integer standing for the length of the key
Arbitrary data of the key

Response: **`[code:1]([vsiz:4][vbuf:*])`**

An 8-bit integer whose value is 0 on success or another on failure
on success: A 32-bit integer standing for the length of the value
on success: Arbitrary data of the value

**`mget`**: for the function `tcrdbget3'

Request: **`[magic:2][rnum:4][{[ksiz:4][kbuf:*]}:*]`**

Two bytes of the command ID: 0xC8 and 0x31
A 32-bit integer standing for the number of keys
iteration: A 32-bit integer standing for the length of the key
iteration: Arbitrary data of the key

Response: **`[code:1][rnum:4][{[ksiz:4][vsiz:4][kbuf:*][vbuf:*]}:*]`**

An 8-bit integer whose value is 0 on success or another on failure

A 32-bit integer standing for the number of records
iteration: A 32-bit integer standing for the length of the key
iteration: A 32-bit integer standing for the length of the value
iteration: Arbitrary data of the key
iteration: Arbitrary data of the value

**`vsiz`**: for the function `tcrdbvsiz'

Request: **`[magic:2][ksiz:4][kbuf:*]`**

Two bytes of the command ID: 0xC8 and 0x38
A 32-bit integer standing for the length of the key
Arbitrary data of the key

Response: **`[code:1]([vsiz:4])`**

An 8-bit integer whose value is 0 on success or another on failure
on success: A 32-bit integer standing for the length of the value

**`iterinit`**: for the function `tcrdbiterinit'

Request: **`[magic:2]`**

Two bytes of the command ID: 0xC8 and 0x50

Response: **`[code:1]`**

An 8-bit integer whose value is 0 on success or another on failure

**`iternext`**: for the function `tcrdbiternext'

Request: **`[magic:2]`**

Two bytes of the command ID: 0xC8 and 0x51

Response: **`[code:1]([ksiz:4][kbuf:*])`**

An 8-bit integer whose value is 0 on success or another on failure
on success: A 32-bit integer standing for the length of the key
on success: Arbitrary data of the key

`fwmkeys`: for the function `tcrdbfwmkeys'

Request: `[magic:2][psiz:4][max:4][pbuf:*]`

Two bytes of the command ID: 0xC8 and 0x58
A 32-bit integer standing for the length of the prefix
A 32-bit integer standing for the maximum number of keys to be fetched
Arbitrary data of the prefix

Response: `[code:1][knum:4][{[ksiz:4][kbuf:*]}:*]`

An 8-bit integer whose value is 0 on success or another on failure
A 32-bit integer standing for the number of keys
iteration: A 32-bit integer standing for the length of the key
iteration: Arbitrary data of the key

`addint`: for the function `tcrdbaddint'

Request: `[magic:2][ksiz:4][num:4][kbuf:*]`

Two bytes of the command ID: 0xC8 and 0x60
A 32-bit integer standing for the length of the key
A 32-bit integer standing for the additional number
Arbitrary data of the key

Response: `[code:1]([sum:4])`

An 8-bit integer whose value is 0 on success or another on failure
on success: A 32-bit integer standing for the summation value

`adddouble`: for the function `tcrdbadddouble'

Request: `[magic:2][ksiz:4][integ:8][fract:8][kbuf:*]`

Two bytes of the command ID: 0xC8 and 0x61
A 32-bit integer standing for the length of the key
A 64-bit integer standing for the integral of the additional number

A 64-bit integer standing for the trillionfold fractional of the additional number
Arbitrary data of the key

Response: **`[code:1]([integ:8][fract:8])`**

An 8-bit integer whose value is 0 on success or another on failure
on success: A 64-bit integer standing for the integral of the summation value
on success: A 64-bit integer standing for the trillionfold fractional of the summation value

**`ext`**: for the function `tcrdbext'

Request: **`[magic:2][nsiz:4][opts:4][ksiz:4][vsiz:4][nbuf:*][kbuf:*][vbuf:*]`**

Two bytes of the command ID: 0xC8 and 0x68
A 32-bit integer standing for the length of the function name
A 32-bit integer standing for the options
A 32-bit integer standing for the length of the key
A 32-bit integer standing for the length of the value
Arbitrary data of the function name
Arbitrary data of the key
Arbitrary data of the value

Response: **`[code:1]([rsiz:4][rbuf:*])`**

An 8-bit integer whose value is 0 on success or another on failure
on success: A 32-bit integer standing for the length of the result
on success: Arbitrary data of the result

**`sync`**: for the function `tcrdbsync'

Request: **`[magic:2]`**

Two bytes of the command ID: 0xC8 and 0x70

Response: **`[code:1]`**

An 8-bit integer whose value is 0 on success or another on failure

**`optimize`**: for the function `tcrdboptimize'

Request: **`[magic:2][psiz:4][pbuf:*]`**

Two bytes of the command ID: 0xC8 and 0x71
A 32-bit integer standing for the length of the parameter string
Arbitrary data of the parameter string

Response: **`[code:1]`**

An 8-bit integer whose value is 0 on success or another on failure

**`vanish`**: for the function `tcrdbvanish'

Request: **`[magic:2]`**

Two bytes of the command ID: 0xC8 and 0x72

Response: **`[code:1]`**

An 8-bit integer whose value is 0 on success or another on failure

**`copy`**: for the function `tcrdbcopy'

Request: **`[magic:2][psiz:4][pbuf:*]`**

Two bytes of the command ID: 0xC8 and 0x73
A 32-bit integer standing for the length of the path
Arbitrary data of the path

Response: **`[code:1]`**

An 8-bit integer whose value is 0 on success or another on failure

**`restore`**: for the function `tcrdbrestore'

Request: **`[magic:2][psiz:4][ts:8][opts:4][pbuf:*]`**

Two bytes of the command ID: 0xC8 and 0x74
A 32-bit integer standing for the length of the path
A 64-bit integer standing for the beginning time stamp in microseconds
A 32-bit integer standing for the options

Arbitrary data of the path

Response: **`[code:1]`**

An 8-bit integer whose value is 0 on success or another on failure

### **`setmst`**: for the function `tcrdbsetmst'

Request: **`[magic:2][hsiz:4][port:4][ts:8][opts:4][host:*]`**

Two bytes of the command ID: 0xC8 and 0x78
A 32-bit integer standing for the length of the host name
A 32-bit integer standing for the port number
A 64-bit integer standing for the beginning time stamp in microseconds
A 32-bit integer standing for the options
Arbitrary data of the host name

Response: **`[code:1]`**

An 8-bit integer whose value is 0 on success or another on failure

### **`rnum`**: for the function `tcrdbrnum'

Request: **`[magic:2]`**

Two bytes of the command ID: 0xC8 and 0x80

Response: **`[code:1][rnum:8]`**

An 8-bit integer whose value is always 0
A 64-bit integer standing for the number of records

### **`size`**: for the function `tcrdbsize'

Request: **`[magic:2]`**

Two bytes of the command ID: 0xC8 and 0x81

Response: **`[code:1][rnum:8]`**

An 8-bit integer whose value is always 0
A 64-bit integer standing for the size of the database

**`stat`**: for the function `tcrdbstat'

Request: **`[magic:2]`**

Two bytes of the command ID: 0xC8 and 0x88

Response: **`[code:1][ssiz:4][sbuf:*]`**

An 8-bit integer whose value is always 0
A 32-bit integer standing for the length of the status message
Arbitrary data of the result

**`misc`**: for the function `tcrdbmisc'

Request: **`[magic:2][nsiz:4][opts:4][rnum:4][nbuf:*][{[asiz:4][abuf:*]}:*]`**

Two bytes of the command ID: 0xC8 and 0x90
A 32-bit integer standing for the length of the function name
A 32-bit integer standing for the options
A 32-bit integer standing for the number of arguments
Arbitrary data of the function name
iteration: A 32-bit integer standing for the length of the argument
iteration: Arbitrary data of the argument

Response: **`[code:1][rnum:4][{[esiz:4][ebuf:*]}:*]`**

An 8-bit integer whose value is 0 on success or another on failure
A 32-bit integer standing for the number of result elements
iteration: A 32-bit integer standing for the length of the element
iteration: Arbitrary data of the element

To finish the session, the client can shutdown and close the socket at any time. If not closed, the connection can be reused for the next session. If protocol violation or some fatal error occurs, the server immediately breaks the session and closes the connection.

## Memcached Compatible Protocol

As for the memcached (ASCII) compatible protocol, the server implements the following commands; "set", "add", "replace", "get", "delete", "incr", "decr", "stats", "flush_all", "version", and "quit". "noreply" options of update commands are also supported. However, "flags", "exptime", and "cas unique" parameters are ignored.

## HTTP Compatible Protocol

As for the HTTP (1.1) compatible protocol, the server implements the following commands; "GET" (relevant to `tcrdbget'), "HEAD" (relevant to `tcrdbvsiz'), "PUT" (relevant to `tcrdbput'), "POST" (relevant to `tcrdbext'), "DELETE" (relevant to `tcrdbout'), and "OPTIONS" (relevant to `tcrdbstat'). The URI of each request is treated as the key encoded by the URL encoding. And the entity body is treated as the value. However, headers except for "Connection" and "Content-Length" are ignored. "PUT" can have the header "X-TT-PDMODE" whose value is either of 1 (relevant to `tcrdbputkeep'), 2 (relevant to `tcrdbputcat'), or else (relevant to `tcrdbput').

"POST" should have one of the header "X-TT-XNAME" or the header "X-TT-MNAME". "X-TT-XNAME" is relevant to `tcrdbext' and specifies the function name. The header "X-TT-XOPTS" stands for bitwise-or options of 1 (record locking) and 2 (global locking). The URI of each request is treated as the key encoded by the URL encoding. And the entity body is treated as the value. The result is expressed as the entity body of the response. "X-TT-MNAME" is relevant to `tcrdbmisc' and specifies the function name. The header "X-TT-MOPTS" stands for bitwise-or options of 1 (omission of the update log). The request parameters are expressed as the entity body in the "application/x-www-form-urlencoded" format. The names are ignored and the values are treated as a list of the parameters. The result is expressed as the entity body of the response in the "application/x-www-form-urlencoded" format.

# Tutorial

## Basic Use

After installation of Tokyo Tyrant, you can start the server immediately by executing the command `**`ttserver**' in the terminal. By default, the server listens to the port 1978 and serves as the accessor of an on-memory hash database, which is useful to store cache data.

```
[terminal-1]$ ttserver
```

To test storing operations, execute the following commands in another terminal. `**`tcrmgr put**' calls the function `**`tcrdbput**'.

```
[terminal-2]$ tcrmgr put localhost one first
[terminal-2]$ tcrmgr put localhost two second
[terminal-2]$ tcrmgr put localhost three third
```

To test retrieving operations, execute the following commands in another terminal. `**`tcrmgr get**' calls the function `**`tcrdbget**'.

```
[terminal-2]$ tcrmgr get localhost one
[terminal-2]$ tcrmgr get localhost two
[terminal-2]$ tcrmgr get localhost three
```

To retrieve multiple records at once, execute the following command. `**`tcrmgr mget**' calls the function `**`tcrdbget3**'.

```
[terminal-2]$ tcrmgr mget localhost one two three
```

To terminate the server, input Ctrl-C in the terminal of the server.

Next, let's run the server that handles a hash database, by specifying the file name whose suffix is `.tch'.

```
[terminal-1]$ ttserver casket.tch
```

Store some records.

```
[terminal-2]$ tcrmgr put localhost one first
[terminal-2]$ tcrmgr put localhost two second
[terminal-2]$ tcrmgr put localhost three third
```

Terminate the server by Ctrl-C, and then restart the server.

```
[terminal-1]$ ttserver casket.tch
```

Check consistency of stored records.

```
[terminal-2]$ tcrmgr mget localhost one two three
```

Terminate the server by Ctrl-C and remove the database, for the successive tutorial.

```
[terminal-1]$ rm casket.tch
```

## Daemon

To run the server as a daemon process, specify the option `-dmn'. Moreover, the option `-pid' should be specified to record the process ID into a file. Note that the current directory of the daemon process is changed to the root directory. So, the file path parameter should be expressed as the absolute path.

```
[terminal-1]$ ttserver -dmn -pid /tmp/ttserver.pid /tmp/casket.tch
```

To terminate the daemonized server, check the process ID from the file specified by `**-pid**' and send the SIGTERM signal to the process.

```
[terminal-1]$ kill -TERM `cat /tmp/ttserver.pid`
```

To run the server by the RC script of the operating system, use `**ttservctl**'. As for most Linux distribution, append the following line to `**/etc/rc.local**'.

```
/usr/local/sbin/ttservctl start
```

By default, the database file and the related files are placed under `**/var/ttserver**'. Because `**ttservctl**' is a tiny shell script, copy and edit it for your purpose. Also, it is suitable to install the modified script into `**/etc/init.d**' and set symbolic links from `**/etc/rc3.d/S98ttserver**' and `**/etc/rc5.d/S98ttserver**'.

## Backup and Recovery

Let's run the server again to continue this tutorial.

```
[terminal-1]$ ttserver casket.tch
```

Store some records.

```
[terminal-2]$ tcrmgr put localhost one first
[terminal-2]$ tcrmgr put localhost two second
[terminal-2]$ tcrmgr put localhost three third
```

To back up the database file, indicate the destination path to the server by the command `**tcrmgr copy**'. Note that the backup file is created on the local file system of the server (not on the client side).

```
[terminal-2]$ tcrmgr copy localhost backup.tch
```

Terminate the server by Ctrl-C and remove the database.

```
[terminal-1]$ rm casket.tch
```

Recover the database from the backup file and restart the server.

```
[terminal-1]$ cp backup.tch casket.tch
[terminal-1]$ ttserver casket.tch
```

Check consistency of stored records.

```
[terminal-2]$ tcrmgr mget localhost one two three
```

Terminate the server by Ctrl-C and remove the databases, for the successive tutorial.

```
[terminal-1]$ rm casket.tch backup.tch
```

## Update Log

Let's run the server with update logging enabled. The option `**-ulog**' specifies the directory to contain the update log files.

```
[terminal-1]$ mkdir ulog
[terminal-1]$ ttserver -ulog ulog casket.tch
```

Store some records.

```
[terminal-2]$ tcrmgr put localhost one first
[terminal-2]$ tcrmgr put localhost two second
[terminal-2]$ tcrmgr put localhost three third
```

Terminate the server by Ctrl-C and remove the database.

```
[terminal-1]$ rm casket.tch
```

Escape the update log directoty and restart the server.

```
[terminal-1]$ mv ulog ulog-back
[terminal-1]$ mkdir ulog
[terminal-1]$ ttserver -ulog ulog casket.tch
```

Restore the database from the escaped update log, by the command `**tcrmgr restore**' on the client side.

```
[terminal-2]$ tcrmgr restore localhost ulog-back
```

Check consistency of stored records.

```
[terminal-2]$ tcrmgr mget localhost one two three
```

Terminate the server by Ctrl-C and remove the database, for the successive tutorial.

```
[terminal-1]$ rm -rf casket.tch ulog ulog-back
```

## Replication

Replication is a mechanism to synchronize two or more database servers for high availability and high integrity. The replication source server is called "master" and each destination server is called "slave". Replication requires the following preconditions.

- The master must record the update log.
- The master must specify the unique server ID.
- Each slave must record the update log because it may become the master when fail over.
- Each slave must specify the unique server ID because it may become the master when fail over.
- Each slave must specify the address and the port number of the master server.
- Each slave must specify the replication time stamp file.

This section describes how to set up one master (at port 1978) and one slave (at port 1979) replication. First, let's run the master server.

```
[terminal-1]$ mkdir ulog-1
[terminal-1]$ ttserver -port 1978 -ulog ulog-1 -sid 1 casket-1.tch
```

Next, let's run the slave server in another terminal.

```
[terminal-2]$ mkdir ulog-2
[terminal-2]$ ttserver -port 1979 -ulog ulog-2 -sid 2 \
                -mhost localhost -mport 1978 -rts 2.rts casket-2.tch
```

Store some records into the master.

```
[terminal-3]$ tcrmgr put -port 1978 localhost one first
[terminal-3]$ tcrmgr put -port 1978 localhost two second
[terminal-3]$ tcrmgr put -port 1978 localhost three third
```

Check consistency of stored records in the master and the slave.

```
[terminal-2]$ tcrmgr mget -port 1978 localhost one two three
[terminal-2]$ tcrmgr mget -port 1979 localhost one two three
```

Let's simulate the case that the master is crashed. Terminate the master by Ctrl-C and remove the database file.

```
[terminal-1]$ rm casket-1.tch
```

Terminate the slave by Ctrl-C and restart it as the new master.

```
[terminal-2]$ ttserver -port 1979 -ulog ulog-2 -sid 2 casket-2.tch
```

Add the new slave (at port 1980).

```
[terminal-1]$ mkdir ulog-3
[terminal-1]$ ttserver -port 1980 -ulog ulog-3 -sid 3 \
                 -mhost localhost -mport 1979 -rts 3.rts casket-3.tch
```

Check consistency of stored records in the new master and the new slave.

```
[terminal-2]$ tcrmgr mget -port 1979 localhost one two three
[terminal-2]$ tcrmgr mget -port 1980 localhost one two three
```

Terminate the two servers by Ctrl-C and remove the database and related files.

```
[terminal-1]$ rm -rf casket-1.tch ulog-1 1.rts
[terminal-2]$ rm -rf casket-2.tch ulog-2 2.rts
[terminal-1]$ rm -rf casket-3.tch ulog-3 3.rts
```

Tokyo Tyrant supports "dual master" replication which realizes higher availability. To do it, run two servers which replicate each other. Note that updating both of the masters at the same time may cause inconsistency of their databases. By default, the servers do not complain even if inconsistency is detected. The option \``-rcc`' make them check the consistency and stop replication when inconsistency is detected.

## Setting Replication on Demand

You can set replication of the running database service without any downtime. First, prepare the following script for backup operation and save it as "ttbackup.sh" with executable permission (0755).

```
#! /bin/sh
srcpath="$1"
destpath="$1.$2"
rm -f "$destpath"
cp -f "$srcpath" "$destpath"
```

Next, let's run the master with update log enabled.

```
[terminal-1]$ mkdir ulog-1
```

```
[terminal-1]$ ttserver -port 1978 -ulog ulog-1 -sid 1 casket-1.tch
```

Store a volume of records into the master.

```
[terminal-2]$ tcrtest write -port 1978 localhost 10000
```

Check consistency of stored records.

```
[terminal-2]$ tcrmgr list -port 1978 -pv localhost
```

Backup the database.

```
[terminal-2]$ tcrmgr copy -port 1978 localhost '@./ttbackup.sh'
```

Confirm that the backup file was saved as "casket-1.tch.*xxxxx*" ("*xxxxx*" stands for the time stamp of the backup file). Then, run the slave with the backup file.

```
[terminal-2]$ ls
[terminal-2]$ cp casket-1.tch.xxxxx casket-2.tch
[terminal-2]$ echo xxxxx > 2.rts
[terminal-2]$ mkdir ulog-2
[terminal-2]$ ttserver -port 1979 -ulog ulog-2 -sid 2 -rts 2.rts casket-2.tch
```

Note that the above operation did not specify the master server to the slave. For tutorial, let's simulate that some records are stored into the master by users while you are setting replication.

```
[terminal-3]$ tcrmgr put -port 1978 localhost one first
[terminal-3]$ tcrmgr put -port 1978 localhost two second
[terminal-3]$ tcrmgr put -port 1978 localhost three third
```

Check the difference between the master and the slave.

```
[terminal-3]$ tcrmgr inform -port 1978 localhost
[terminal-3]$ tcrmgr inform -port 1979 localhost
```

Specify the master to the slave so that replication will start and the difference will be resolved.

```
[terminal-3]$ tcrmgr setmst -port 1979 -mport 1978 localhost localhost
```

Confirm that the slave knows the master and the difference has been resolved.

```
[terminal-3]$ tcrmgr inform -port 1979 -st localhost
```

Terminate the two servers by Ctrl-C and remove the database and related files.

```
[terminal-1]$ rm -rf casket-1.tch casket-1.tch.* ulog-1 1.rts ttbackup.sh
[terminal-2]$ rm -rf casket-2.tch ulog-2 2.rts
```

## Tuning

If you use a hash database, set the tuning parameter "#bnum=*xxx*" to improve performance. It specifies the bucket number and should be more than the number of record to be stored.

If you use a B+ tree database, set the tuning parameters "#lcnum=*xxx*#bnum=*yyy*" to improve performance. The former specifies the maximum number of leaf nodes to be cached. It should be larger as long as the capacity of RAM on the system allows. The latter specifies the bucket number and should be more than 1/128 of the number of records to be stored.

If huge number of clients access the server, make sure the limit number of file descriptors per process is cleared. By default on most systems, it is set as 1024. If so, use `**ulimit**' to clear it.

In order to deal with rushing queries at the peak time of your service, replication combining the on-memory hash/tree database and the file hash/tree database is useful. The master server handles the on-memory database and it can come through rushing queries at the peak time. Though the on-memory database can not assure the data persistence, the slave of replication compensates the shortage by storing records in the file database.

## Lua Extension

If you want more complex database operations than existing ones, use the Lua extension. For example, prepare the following script and save it as "test.lua". There is a function "fibonacci" which returns the Fibonacci number of a number of the key.

```
function fibonacci(key, value)
   local num = tonumber(key)
   local large = math.pow((1 + math.sqrt(5)) / 2, num)
   local small = math.pow((1 - math.sqrt(5)) / 2, num)
   return (large - small) / math.sqrt(5)
end
```

Let's start the server by making it read the script file.

```
[terminal-1]$ ttserver -ext test.lua
```

Call the function from the client command.

```
[terminal-2]$ tcrmgr ext localhost fibonacci 1
[terminal-2]$ tcrmgr ext localhost fibonacci 2
[terminal-2]$ tcrmgr ext localhost fibonacci 3
[terminal-2]$ tcrmgr ext localhost fibonacci 4
[terminal-2]$ tcrmgr ext localhost fibonacci 5
[terminal-2]$ tcrmgr ext localhost fibonacci 6
```

Fibonacci numbers can be generated by another algorithm, which is naive and stateful. Add the following script to "test.lua". There is a function "fibnext" which returns the next Fibonacci number from the database. The state information are stored in the database.

```
function fibnext(key, value)
   local cur = tonumber(_get("fibcur"))
   if not cur then
      _put("fibold", 0)
      _put("fibcur", 1)
      return 1
   end
   local old = tonumber(_get("fibold"))
   _put("fibold", cur)
   cur = old + cur
   _put("fibcur", cur)
   return cur
end
```

Then, restart the server and test the new algorithm.

```
[terminal-2]$ tcrmgr ext localhost fibnext
[terminal-2]$ tcrmgr ext localhost fibnext
[terminal-2]$ tcrmgr ext localhost fibnext
[terminal-2]$ tcrmgr ext localhost fibnext
[terminal-2]$ tcrmgr ext localhost fibnext
[terminal-2]$ tcrmgr ext localhost fibnext
```

As you see, the called function receives two string parameters of the key and the value. The return value is sent back to the client. You can use such built-in functions for database operations as "_put", "_out", "_get", and so on. There is a sample file `**ext/senatus.lua**'.

## Using memcached Client

This section describes how to use a memcached client library of Perl (Cache::Memcached) with Tokyo Tyrant. Run the server of Tokyo Tyrant as usual. And, the following script is a typical example.

```
use Cache::Memcached;

my $memd = Cache::Memcached->new();
$memd->set_servers(['localhost:1978']);

$memd->set('one', 'first');
```

```
$memd->set('two', 'second');
$memd->set('three', 'third');

my $val = $memd->get('one');
printf("one: %s\n", $val);

$val = $memd->get_multi('one', 'two', 'three');
printf("one: %s\n", $val->{one});
printf("two: %s\n", $val->{two});
printf("three: %s\n", $val->{three});

$memd->delete('one');
```

## Using HTTP Client

This section describes how to use an HTTP client library of Perl (LWP::UserAgent) with Tokyo Tyrant. Run the server of Tokyo Tyrant as usual. And, the following script is a typical example.

```
use LWP::UserAgent;

my $ua = LWP::UserAgent->new(keep_alive => 1);
my $baseurl = 'http://localhost:1978/';

my $req;
$req = HTTP::Request->new(PUT => $baseurl . 'one', [], 'first');
$ua->request($req);
$req = HTTP::Request->new(PUT => $baseurl . 'two', ["X-TT-PDMODE" => 1], 'second');
$ua->request($req);
$req = HTTP::Request->new(PUT => $baseurl . 'three', ["X-TT-PDMODE" => 2], 'third');
$ua->request($req);

$req = HTTP::Request->new(GET => $baseurl . 'one');
my $res = $ua->request($req);
if($res->is_success()){
    printf("%s\n", $res->content());
}

$req = HTTP::Request->new(DELETE => $baseurl . 'one');
```

```
$res = $ua->request($req);

$req = HTTP::Request->new(POST => $baseurl . 'foo',
  ["X-TT-XNAME" => "echo", "X-TT-XOPTS" => 1], 'bar');
$res = $ua->request($req);
if($res->is_success()){
    printf("%s\n", $res->content());
}
```

## Persistent but Expirable Cache

If you want to cache data like session information for your Web application but want to avoid data loss because of the server crash, using Tokyo Tyrant can be the solution, that is to say, "persistent" but expirable cache. It requires the following preconditions.

- The server must open a table database.
- Clients should store each record with an expiration date column.
- The database should have an index for the expiration date column.
- The database should enable auto defragmentation.
- The server must periodically call the user defined function provided through the Lua extension.

First, prepare the following script for expiration and save it as "ttexpire.lua". It will expire records where the value of the "x" column exceeds the current date.

```
function expire()
   local args = {}
   local cdate = string.format("%d", _time())
   table.insert(args, "addcond\0x\0NUMLE\0" .. cdate)
   table.insert(args, "out")
   local res = _misc("search", args)
   if not res then
      _log("expiration was failed", 2)
   end
end
```

Start the server by opening a table database which has the index for the "x" column, and by scheduling it to call the expiration function per second.

```
[terminal-1]$ ttserver -ext ttexpire.lua -extpc expire 1.0 "casket.tct#idx=x:dec#dfunit=8"
```

Store test records from another terminal.

```
[terminal-2]$ now=`date +%s`
for((i=1;i<=60;i++)); do
  tcrmgr put -sep '|' localhost "$i" "x|$((now+i))"
done
```

You can confirm that the records are being removed by expiration.

```
[terminal-2]$ tcrmgr list -pv -sep '|' localhost
```

---

## License

Tokyo Tyrant is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation; either version 2.1 of the License or any later version.

Tokyo Tyrant is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with Tokyo Tyrant (See

Tokyo Tyrant was written by FAL Labs. You can contact the author by e-mail to `**`info@fallabs.com`**'.

---