Periodic HWDB Client Architecture

The initial implementation of the Homework stream database is in a daemon process, accessed using the SRPC simple RPC system. This document describes the general architecture and displays sample code for clients of the database that need to obtain tuples from the database at regular, periodic time intervals.

# Background

The Homework database (HWDB) records tuples in time order in different tables as a result of “insert” SQL statements issued by other processes. The SQL supported by HWDB enables one to select tuples from a table that have arrived in a specified time period. Applications that wish to monitor the recent activity, whether for display purposes or to drive policy-based management, need to periodically obtain the latest tuples to process.

Initially, we thought that we would use query subscription on tables to achieve this style of working. The original subscription approach requires subscribed queries to be re-evaluated every time a tuple was added or removed from a table (removals occur when the oldest tuple is overwritten in the circular buffer). This approach simply does not work for tables that change frequently (i.e. for nearly all of the tables we would be populating), so one should not use it, even though it is still supported in the code. We anticipate providing syntax and support for subscription of a query to a table that the database server implements as a periodic task according to a user-defined time interval; if we do determine that this approach is appropriate, then one can achieve the functionality provided by the architecture defined in this document by such a subscription. In the meantime, this is the only game in town.

# The architecture

This architecture permits a singly-threaded process to periodically query the database at reasonably-precise time intervals of order seconds. General pseudocode for main() is as follows:

process arguments

initialize RPC system

connect to “HWDB”

generate appropriate select query to issue at TIME\_DELTA intervals

note current time of day (seconds, microseconds) and assign to expected

forever

expected time += TIME\_DELTA

get current time

time-delay = expected time – current time

nanosleep(time\_delay)

invoke query, receiving response

process results

The minimum granularity for TIME\_DELTA is seconds, as an RPC takes of order milliseconds (if on the same host) or of order 10-20 ms (if the client is on a different host). Of course, you must be able to complete your processing within the remaining time before you then sleep before the next call.

# An example

The remainder of the document shows the main() from the flowmonitor application that has been provided with the database. The source code in the distribution can be viewed if you want more detail.

#define USAGE "./flowmonitor [-h host] [-p port] [-m ports] [-m hosts] [-m proto]"

#define TIME\_DELTA 5 /\* in seconds \*/

static struct timespec time\_delay = {TIME\_DELTA, 0};

static int must\_exit = 0;

static int exit\_status = 0;

static int sig\_received;

static int mapPorts; /\* true if mapping ports to appl protocols \*/

static int mapHosts; /\* true if mapping ip addresses to hostnames \*/

static int mapProto; /\* true if mapping protocol numbers to protocol names \*/

static tstamp\_t processresults(char \*buf, unsigned int len);

static void signal\_handler(int signum) {

must\_exit = 1;

sig\_received = signum;

}

int main(int argc, char \*argv[]) {

RpcConnection rpc;

char query[SOCK\_RECV\_BUF\_LEN];

char resp[SOCK\_RECV\_BUF\_LEN];

int qlen;

unsigned len;

char \*host;

unsigned short port;

int i, j;

struct timeval expected, current;

tstamp\_t last = 0LL;

host = HWDB\_SERVER\_ADDR;

port = HWDB\_SERVER\_PORT;

mapPorts = 0;

mapHosts = 0;

mapProto = 0;

for (i = 1; i < argc; ) {

if ((j = i + 1) == argc) {

fprintf(stderr, "usage: %s\n", USAGE);

exit(1);

}

if (strcmp(argv[i], "-h") == 0)

host = argv[j];

else if (strcmp(argv[i], "-p") == 0)

port = atoi(argv[j]);

else if (strcmp(argv[i], "-m") == 0) {

if (strcmp(argv[j], "ports") == 0)

mapPorts = 1;

else if (strcmp(argv[j], "hosts") == 0)

mapHosts = 1;

else if (strcmp(argv[j], "proto") == 0)

mapProto = 1;

else

fprintf(stderr, "Don't know how to map %s\n", argv[j]);

} else {

fprintf(stderr, "Unknown flag: %s %s\n", argv[i], argv[j]);

}

i = j + 1;

}

/\* first attempt to connect to the database server \*/

if (!rpc\_init(0)) {

fprintf(stderr, "Failure to initialize rpc system\n");

exit(-1);

}

if (!(rpc = rpc\_connect(host, port, "HWDB", 1l))) {

fprintf(stderr, "Failure to connect to HWDB at %s:%05u\n", host, port);

exit(-1);

}

if (mapPorts) {

portmap\_init(TCP, TCP\_FILE);

portmap\_init(UDP, UDP\_FILE);

}

if (mapHosts) {

hostmap\_init();

}

if (mapProto) {

protomap\_init(PROTO\_FILE);

}

/\* now establish signal handlers to gracefully exit from loop \*/

if (signal(SIGTERM, signal\_handler) == SIG\_IGN)

signal(SIGTERM, SIG\_IGN);

if (signal(SIGINT, signal\_handler) == SIG\_IGN)

signal(SIGINT, SIG\_IGN);

if (signal(SIGHUP, signal\_handler) == SIG\_IGN)

signal(SIGHUP, SIG\_IGN);

gettimeofday(&expected, NULL);

expected.tv\_usec = 0;

while (! must\_exit) {

tstamp\_t last\_seen;

expected.tv\_sec += TIME\_DELTA;

if (last) {

char \*s = timestamp\_to\_string(last);

sprintf(query,

"SQL:select \* from Flows [ range %d seconds] where timestamp > %s\n",

TIME\_DELTA+1, s);

free(s);

} else

sprintf(query, "SQL:select \* from Flows [ range %d seconds]\n",

TIME\_DELTA);

qlen = strlen(query) + 1;

gettimeofday(&current, NULL);

if (current.tv\_usec > 0) {

time\_delay.tv\_nsec = 1000 \* (1000000 - current.tv\_usec);

time\_delay.tv\_sec = expected.tv\_sec - current.tv\_sec - 1;

} else {

time\_delay.tv\_nsec = 0;

time\_delay.tv\_sec = expected.tv\_sec - current.tv\_sec;

}

nanosleep(&time\_delay, NULL);

if (! rpc\_call(rpc, query, qlen, resp, sizeof(resp), &len)) {

fprintf(stderr, "rpc\_call() failed\n");

break;

}

resp[len] = '\0';

if ((last\_seen = processresults(resp, len)))

last = last\_seen;

}

rpc\_disconnect(rpc);

if (mapPorts) {

portmap\_free(TCP);

portmap\_free(UDP);

}

if (mapHosts) {

hostmap\_free();

}

if (mapProto) {

protomap\_free();

}

exit(exit\_status);

}