Vertical Handover Study Cluster: Control Protocol 2
Introduction and Usage
8.3.2005

Raimo Nikkilä
PM&RG – Product Modelling and Realisation Group
Department of Computer Science and Engineering
Helsinki University of Technology
P.O.Box 5400, FIN-02015 HUT, Finland
Introduction

• Application level text based protocol
• Meant to pass application control information
• Designed to work on top of TCP/IP
• Modular, easily extendable
• Service Independant
• Allows base64 encoded binary to be passed to and from services
• Both server and client implemented
Protocol Specification

• Packet size up to 512 bytes
• All packets end with ”\r\n”
• Client commands are of form:
  – <command> <arguments>
  – service <servicename> <service command> <arguments>

• Server replies are of form:
  – <type><ID> <parameters> :<explanation in text>
  – <type><ID> <servicename> <parameters> :<text>
  – type is one hexadecimal number [0, F]
  – ID is three hexadecimal numbers [000, FFF]

• Minimum server reply is 6 bytes
Implementation

• ISO C99 Server and client
• Supports modern POSIX conforming operating systems.
  – Tested on Linux and FreeBSD 5.X
• Asynchronous (single process) server, makes interclient communication trivial.
• Server can support any number of connections
  – Currently the feasible maximum is ~1000 connections (estimate)
  – If required, can and will be extended to ~10,000 connections
Implementation Features

- Easy to use schedule
- Logging
- Configuration file support
- Callbacks to client and group events
- Parse tree for commands
- Parser for lines, supports quoted strings
- Skeleton service to provide a base for a new service

- Several features have macros to help with common usage
Control Protocol 2 Service Architecture

- Service Module and the Control Protocol 2 server are in the same process
- Service program and module communicate using IPC
- Ideally the service module controls the service program
- Service program and module can be the same
  - Not recommended though
Numeric Codes for Message Types

```c
typedef enum
{
    SM_SYS_MESSAGE     = 0x0,
    SM_CLIENT_MESSAGE  = 0x1,
    SM_CLIENT_REPLY    = 0x2,
    SM_CLIENT_ERROR    = 0x3,
    SM_GROUP_MESSAGE   = 0x4,
    SM_GROUP_REPLY     = 0x5,
    SM_GROUP_ERROR     = 0x6,
    SM_SERVICE_MESSAGE = 0x7,
    SM_SERVICE_REPLY   = 0x8,
    SM_SERVICE_ERROR   = 0x9,
    SM_ADMIN_MESSAGE   = 0xA,
    SM_ADMIN_REPLY     = 0xB,
    SM_ADMIN_ERROR     = 0xC,
    SM_14              = 0xD,
    SM_15              = 0xE,
    SM_DEBUG           = 0xF
} message_type;
```
Service Numerics

- Services have three numeric ranges
  - `SM_SERVICE_MESSAGE`,
  - `SM_SERVICE_REPLY`,
  - `SM_SERVICE_ERROR`

- All services have the same range, services are separated by the service name when using service numerics.

- Service is free to allocate its own numerics within those ranges.
Suspended Client State

- Mobile clients are often on a not-so-stable-a-connection
- Client can quit in ways other than a client initiated QUIT
- Control Protocol 2 server can be set to support suspended clients
- When the connection is lost, the client is transparently suspended
- Within the suspension timeout, the client can reconnect and resume its older connection without any service ever noticing it.
- Connection is resumed by issuing the RESUME command instead of the HELLO command on connect
- Resuming the connection requires the client to remember its former connection ID, which is given on connection.
Security

- Challenges are used as the basic security feature
  - Server sends: `<challenge id> <salt> <command>`
  - Client replies: `<challenge id> MD5(<command> <salt> <password>)`
- Security is optional
  - Turning challenges off for development is recommended
- Service can implement additional security features
- SSH tunnel could be used to protect the entire control connection in high-risk network areas.
Administration of Services

• Client can obtain administrator privileges
  – Command admin
• lsmod
  – lists loaded modules based on pattern
• insmod
  – inserts modules
• rmmod
  – removes modules
• For insmod and rmmod, up to 4 modules can be specified
• Uses the filename of the module
  – usually "/mymodule.so"
Using the Control Protocol 2

• All services are loaded dynamically as modules using dlfcn.h library
  − Modules need not worry about this

• Service crash will result in a crash of the entire control server
  − *(int*) NULL; is enough, no need for anything fancier

• Following is an introduction to all major features using the example service, demomod
Example: Demomod

• This presentation will cover parts of the demomod service as examples
  – A simple service using most of the servers features

• Examples are interleaved with usage instructions

• Demomod is available with the control server 2
  – /control2/server/demomod

• Should not be used as a basis for new services
  – Skeleton module exists for the purpose
Example: Demomod numerics

```c
#ifndef DEMOMOD NUMERICS_H
#define DEMOMOD NUMERICS_H

typedef enum
{
    DEMOMOD MESSAGE_MSG = 0x100
} demomod_message;

typedef enum
{
    DEMOMOD_REPLY_HELLO = 0x200,
    DEMOMOD_REPLY_BYE = 0x300
} demomod_reply;

typedef enum
{
    DEMOMOD_ERROR_DONT_KNOW = 0x400,
    DEMOMOD_ERROR_KNOWS = 0x500,
    DEMOMOD_ERROR_MISSING_COMMAND = 0x600,
    DEMOMOD_ERROR_UNKNOWN_COMMAND = 0x700
} demomod_error;

#endif
```
Compiling and Running

- Make (traditional) on control2/server
- Targets
  - all (=server)
  - c2s
    - Just the basic server
  - modules
    - All modules (services)
  - server
    - Both c2s and modules
- Copies the module .so files to the same directory with the executable
Usage: Adding a Service

• Compile your code to a shared object ".so” file and place it to the control server directory
  – Makefile provided
  – Only functions required are:
    • void _init(void);
    • void _fini(void);
• Call the add_service() function
  – service add_service(
    – char *service_name,
    – char* service_description,
    – char *version,
    – int flags,
    – receive_binary_handler binary_handler);
• Use either the insmod admin command or the cserver.cfg to automatically load your service to the control server.
Example: Adding a Service

- Return value of add_service stored to a variable of type service

```c
service demomod;

void _init(void)
{
    demomod = add_service("demomod", "demonstration service", "1.00a", 0, NULL);
}```
Usage: Schedule

• Adding to schedule
  – int schedule_event(time_t when, int ntimes, int delay, sch_callback handler);

• Deleting from schedule
  – int unschedule_event(int schedule_id);

• Macros to help using the schedule
  – int schedule_once(time_t when, sch_callback handler);
  – int schedule_now(sch_callback handler);
  – int schedule_repeating(int delay, sch_callback handler);
Example: Using the Schedule

static int demomod_schedule_id;
void _init(void)
{ /* ... */
demomod_schedule_id =
schedule_repeating(60,&demomod_schedule_entry);
}

void _fini (void)
{ /* ... */
unschedule_event(demomod_schedule_id);

void demomod_schedule_entry(int unused)
{
    struct demomod_user_struct *dm_user;
    for (dm_user = demomod_users;dm_user;dm_user = dm_user->next)
        cl_send(dm_user->real_client, demomod, MSG,
                DEMOMOD_MESSAGE_MSG,":%s", STR_SETTING(set_demomod_message));
}
Usage: Logging

• Logging is simple
  – int log_debug(const char* fstr, ...);
  – int log_message(const char* fstr, ...);
  – int log_info(const char* fstr, ...);
  – int log_error(const char* fstr, ...);
  – int log_critical(const char* fstr, ...);
• All logging functions behave like normal printf();
• Logging is configured at the cserver.cfg file.
• Log level setting changes the threshold of messages to log
  – no need to change code
Example: Logging

- Logging is also meant for debug messages during service development
  - No need to printf() yourself
  - All debug messages can then be disabled with one setting

```c
if (load_setting_file("./demomod/demomod.cfg", "demomod"))
    log_error("Unable to open the demomod configuration file");
```
Usage: Configuration Files

• Supports several configuration files
• Entries are of form
  – `<key>`=`<value>`
  – `#` indicates that the line is a comment
• To load a configuration file
  – `int load_settings(const char* fname, const char* owner);`
  – Owner separates the configuration from the configuration of other services
• To remove the configuration loaded
  – `int purge_settings(const char *owner);`
• If names collide (same name, same owner) the old value is overwritten
Example: Configuration Files

- The type of the parameter is guessed
- Spacing is sensitive

```plaintext
#demomod example configuration
demomod_message=Hello I'm demomod and I know you
```
Usage: Configuration

• The four supported setting types are boolean, integer, float and string
• First, read the configuration file
• Declare a variable of type setting
  – common usage: setting set_my_setting; as global variable
• Load the setting using the get_setting(); function
  – int get_setting(char *name, setting &set, setting_type type, ...);
  – Example: get_setting(“number”, &set_my_setting, SET_INT, 3);
• Following macros can be used to easily access the settings
  – BOOL_SETTING(setting);
  – INT_SETTING(setting);
  – FLOAT_SETTING(setting);
  – STR_SETTING(setting);
Example: Using Configuration

- Hard coded default values insure operation even if the configuration file is missing or invalid

```c
setting set_demomod_message;

void _init (void)
{ /* ... */
get_setting("demomod_message", &set_demomod_message, SET_STR, "I'm demomod
and I know you");

void demomod_schedule_entry(int unused)
{ /* ... */
c1_send(dm_user->real_client, demomod, MSG, DEMOMOD_MESSAGE_MSG,":%s",
STR_SETTING(set_demomod_message));

void _fini (void)
{ /* ... */
purge_settings("demomod");
```
Usage: Client Callbacks

• A service can register client callbacks for certain client events
  – Several callbacks can be registered
• Callbacks are registered using the function `add_client_callbacks();`
  – `int add_client_callbacks(const char *owner, void (*connection)(client), void (*disconnection)(client), void (*data_in)(client, const char *line));`
• Callbacks are unregistered using the function `del_client_callbacks();`
  – `int del_client_callbacks(const char *owner);`
• If registering for only certain events, `NULL` can be used as the callback for events of no interest
Example: Client Callbacks

- No need to handle uninteresting events
- All callbacks must be removed at module _fini

```c
void _init(void)
{ /* ... */
  add_client_callbacks("demomod", NULL, demomod_client_disconnection_callback, NULL);
}

void demomod_client_disconnection_callback(client clt)
{
    struct demomod_user_struct *tmp;
    if ((tmp = demomod_get_user_by_client(clt)) != NULL)
        demomod_del_user(tmp);
}

void _fini (void)
{ /* ... */
  del_client_callbacks("demomod");
}  ```
Usage: Group Callbacks

• Similar to the client callbacks
  – Several callbacks can also be registered

• Registered using `add_group_callbacks()`
  – `int add_group_callbacks(char *owner, void (*)(group, client), void (*)(group, client), void (*)(group, const char *))`;

• Unregistered using the `del_group_callbacks()`
  – `int del_group_callbacks(char *)`;

• Like with client callbacks, use NULL for non-interesting events
Usage: Parse Tree

- Commands are handled using a parse tree
  - Average complexity $O(\log_2 n)$
- Important functions for parse trees
  - `parse_tree get_parse_tree(void);`
  - `void free_parse_tree(parse_tree);`
  - `int parse_tree_add(parse_tree, const char *cname, const parse_function pfunc);`
- The parse function is of type
  - `void (*parse_function)(client, char* ostr, int argc, char *argv[]);`
- Almost all services require a parse tree for their commands.
Example: Parse Tree 1/2

```c
parse_tree pt_demomod;
void _init(void)
{
    /* ... */
    /* register demomod’s command handler */
    parse_tree_add(&pt_services, "demomod", demomod_parser_entry);
    /* register commands */
    parse_tree_add(&pt_demomod, "hello", demomod_cmd_hello);
    parse_tree_add(&pt_demomod, "bye", demomod_cmd_bye);
}

void _fini(void)
{
    /* ... */
    free_parse_tree(pt_demomod);
    parse_tree_del(&pt_services, "demomod");
}
```
Example: Parse Tree 2/2

• Following is required of all non-trivial services

```c
void demomod_parser_entry(client clt, char* ostring, int argc, char** argv)
{
    PARSER_FORWARD;
    if (argc == 0)
    {
        service_send_client(clt, demomod, ERR, DEMOMOD_ERROR_MISSING_COMMAND, ":You forgot the specify the command");
        return;
    }

    if (parse_tree_call(pt_demomod, clt, ostring, argc, argv))
        cl_send(clt, demomod, ERR, DEMOMOD_ERROR_UNKNOWN_COMMAND, ":Sorry, I don't know command \%s\", argv[0]);
}
```
Usage: Parser

- Server has an inbuilt string parser with quoted string support.
  - `int parse_line(struct parser_control_struct*,
    const char* line);`
  - returns the number of parameters successfully parsed

- For most services, using the parser should not be necessary as all incoming lines are parsed.
  - Initialising the parser_control_struct is non-trivial.
  - Only 8 first parameters are parsed

- Additional white space is removed by the parser

- There is a macro to be used with parsed lines
  - `PARSER_FORWARD;`
  - Requires the calling function to have a few specially named variables
Usage: Communicating With Clients

• All client messages should go through these four functions:
  – int send_client(client clt, message_type mtype, int mid, const char *fstr, ...);
  – int client_send_binary(client, const char* sid, int nid, int data_len, void *data);
  – int service_send_client(client clt, message_type mtype, int mid, const char *fstr, ...);
  – int send_group(group grp, message_type mtype, int mid, const char *fstr, ...);

• For services a shorter function is provided
  – cl_send == service_send_client

• Also message types have the following abbreviations for services
  – MSG SM_SERVICE_MESSAGE
  – RPL SM_SERVICE_REPLY
  – ERR SM_SERVICE_ERROR
Example: Communicating with Clients

- Several parameters need to be used to send something to a client

```c
void demomod_cmd_hello(client clt, char* ostring, int argc, char** argv)
{
    if (demomod_get_user_by_client(clt) != NULL)
        cl_send(clt, demomod, ERR, DEMOMOD_ERROR_KNOWS, "I already know you");
    else
    {
        demomod_add_user(clt);
        cl_send(clt, demomod, RPL, DEMOMOD_REPLY_HELLO, "Nice to meet you");
    }
}
```
Usage: Using Groups

- Groups are handled using only a few functions
  - `group add_group(char *name, char *secret);`
  - `group del_group(group);`
  - `group get_group(char *name);`

- Group client interaction
  - `group join_group(group, client);`
  - `group part_group(group, client);`

- Several other functions are also provided
  - Some can return a strange pointer, but it’s enough to check if it is NULL
  - File `groups.h` lists all group operations

- Something missing?
  - Groups can be used for many things
  - It does not take long to add new functionality
Usage: Removing a Service

• A service can either remove itself or be removed using the rmmod administrator command
• Service must free everything it has reserved, especially itself, callbacks and schedule entries.
Example: Removing a Service

- Nothing is removed automatically
- Forgetting to remove something will usually lead to a segmentation fault

```c
void _fini (void)
{
    fini_demomod_users();
    unschedule_event(demomod_schedule_id);
    purge_settings("demomod");
    del_client_callbacks("demomod");
    free_parse_tree(pt_demomod);
    parse_tree_del(&pt_services, "demomod");
}
```
Usage: Skeleton Service

• Can be found in the directory skeleton/ under the control server.
  – Similar to the demomod
  – Has no functionality, only structure
• Presents most features of the server implementation
• Meant to be a usable base for almost any service.
• Comes with a simple Makefile that should be used
Control Protocol 2 Client

- Client for the Control Protocol 2 server.
  - Mostly reused server code
  - Shares some headers with the server
    - Compatibility

- Symmetric
  - Will block, makes writing clients easier

- Non-modular

- Not meant to contain large quantities of service logic
Future Work

• Callbacks might be replaced with a more general solution.
  – Compatibility to the old will be kept if needed

• Some basis for the IPC code
  – Not trivial with C/UNIX
The End

• Questions, comments and feedback are appreciated
• Control Protocol 2 is work in progress
  – Server internals can change without notice...
  – but the function calls will almost always remain the same

• Avoid making changes to the server code yourself.