A Tutorial for GNUnet 0.9.x (C version)

Christian Grothoff Bart Polot Matthias Wachs

April 25, 2012

1 Getting Started

First of all you have to install a current version of GNUnet. You can obtain the latest development version from subversion by cloning the repository using:

```
$ svn checkout https://gnunet.org/svn/gnunet
```

or you can download a tarball of a more stable version from GNU FTP mirrors:

`ftp://ftp.gnu.org/gnu/gnunet/gnunet-0.9.2.tar.gz`

Detailed installation instructions for various operating systems and a detailed list of all dependencies can found on our website at

`https://gnunet.org/installation`.

Assuming all dependencies are installed, the following commands should work:

```
$ configure --prefix=$HOME
$ make
$ make install
$ export GNUNET_PREFIX=$HOME
$ export PATH=$PATH:$GNUNET_PREFIX/bin
$ make check
$ mkdir ~/.gnunet/
$ touch ~/.gnunet/gnunet.conf
```

The configure prefix defines where to install GNUnet. If you do not specifiy a prefix, GNUnet is installed your home directory. For development purposes you can specify a directory like `/tmp/gnunet`. You should add **export** GNUNET_PREFIX=$HOME to your `.bash_rc` or `.profile` to be sure the environment variable is always set.

1.1 Background: GNUnet Architecture

GNUnet is organized in layers or services. Each service is composed of a main service implementation and a client library for other programs to use, described by an API.

Very often it is other GNUnet services that will use these APIs to build the higher layers of GNUnet on top of the lower ones. Each layer expands or extends the functionality of the service below (for instance, to build a mesh on top of a DHT).

The main service implementation runs as a standalone process in the operating system and the client code runs as part of the client program, so crashes of a client do not affect the service process or other clients. The service and the clients communicate via a message protocol to be defined and implemented by the programmer.

1.2 Baby Steps

First, you may want to just generate the peer's private key:

```
$ gnunet-peerinfo -s
```

GNUnet can then be started with gnunet-arm -s and stopped with gnunet-arm -e. You can specify a configuration for all services using the *-c* switch.

```
$ gnunet-arm -i dht
$ gnunet-dht-monitor -k KEY -T 0 # run in a separate terminal (optional)
$ gnunet-dht-put -k KEY -d VALUE
$ gnunet-dht-get -k KEY
$ gnunet-statistics
$ gnunet-statistics -s dht
```

Using the "monitor" line given above, you can observe the behavior of your own peer's DHT with respect to the specified KEY.

1.3 Starting Two Peers by Hand

For the second peer, you will need to manually create a modified configuration file to avoid conflicts with ports and directories. A peers configuration file is by default located in `~/.gnunet/gnunet.conf`. This file is typically very short as only the differences to the defaults need to be specified. The defaults are located in many files in the `$GNUNET_PREFIX/share/gnunet/config.d` directory.

To configure the second peer, use the files `$GNUNET_PREFIX/share/gnunet/config.d` as a template for your main configuration file:

```
$ cat $GNUNET_PREFIX/share/gnunet/config.d/*.conf > peer2.conf
```

Now you have to edit `peer2.conf` and change:

- SERVICEHOME under PATHS
- Every value for "PORT" (add 10000) in any section
- Every value for "UNIXPATH" in any section

to a fresh, unique value. Make sure that the PORT numbers stay below 65536. From now on, whenever you interact with the second peer, you need to specify `-c peer.conf` as an additional command line argument.

Now, generate the 2nd peer's private key:

```
$ gnunet-peerinfo -s -c peer2.conf
```

This may take a while, generate entropy using your keyboard or mouse as needed. Also, make sure the output is different from the `gnunet-peerinfo` output for the first peer (otherwise you made an error in the configuration).

Then, you can start a second peer using:

```
$ gnunet-arm -c peer2.conf -s
$ gnunet-arm -c peer2.conf -i dht
$ gnunet-dht-put -c peer2.conf -k KEY -d VALUE
$ gnunet-dht-get -c peer2.conf -k KEY
```

If you want the two peers to connect, you can use the peerinfo tool or change the first one to be a hostlist server by:

- Creating an empty configuration file `peer1.conf`
- Adding a line in the "[hostlist]" section with "OPTIONS = -p"

Then change `peer2.conf`, replacing the "SERVERS" line with "http://localhost:8080/". Restart both peers using `gnunet-arm`. Note that if you start your peers without changing these settings, they will use the "global" hostlist servers of the GNUnet P2P network and likely connect to those peers. At that point, debugging might become tricky as you're going to be connected to many more peers and would likely observe traffic and behaviors that are not explicitly controlled by you.

If you want to use the peerinfo tool to connect your peers, you should:

- Remove `hostlist` from `DEFAULTSERVICES` (to not connect to the global GNUnet)
- Start both peers running `gnunet-arm -s` and `gnunet-arm -c peer2.conf -s`
- Get HELLO message of the first peer running `gnunet-peerinfo -g`
- Give the output to the second peer by running `gnunet-peerinfo -c peer2.conf -p '<output>'`

Check that they are connected using `gnunet-statistics -s core`.

1.4 Starting Peers Using the Testing library

In case of tests where a big network is needed, the testing library allows to start any number of peers with a single configuration file. On a single not-so-powerful computer, it should be possible to run 100 peers easily.

The only configuration file needed should include a `testing` section with basic performance characteristics; the default settings are:

```
[TESTING]
# How long before failing a connection?
CONNECT_TIMEOUT = 30 s
# How many connect attempts should we make?
CONNECT_ATTEMPTS = 3
# How many connections can happen simultaneously?
MAX_OUTSTANDING_CONNECTIONS = 50
# Should we clean up the files on peer group shutdown?
DELETE_FILES = YES
```

When starting peers using the testing library, the library takes care of modifiying all variable fields (SERVICEHOME, PORT, UNIXPATH), copying the files to different directories (possibly using `scp` for distributed execution), starting and connecting the peers, and many more tasks that need to be done in order to have a running network ready to run experiments on.

```
#include "gnunet_testing_lib.h"

static struct GNUNET_TESTING_PeerGroup *pg;

static void
ready_cb (void *cls, const char *error_msg)
{
    /* Network is running! */
}
```

```
static void
run (void *cls,
     char *const *args,
     const char *cfgfile,
     const struct GNUNET_CONFIGURATION_Handle *cfg)
{
    /* Read testing parameters from config file (number of peers, etc) */
    /* connect_cb and hosts can be NULL */
    pg = GNUNET_TESTING_peergroup_start (cfg, NUMBER_OF_PEERS, TIMEOUT,
                                         NULL /* connect_cb */, &ready_cb,
                                         cls, NULL /* hosts */);
}

int
main (int argc, char *const *argv)
{
    static const struct GNUNET_GETOPT_CommandLineOption options[] = {
      GNUNET_GETOPT_OPTION_END
    };
    return (GNUNET_OK ==
            GNUNET_PROGRAM_run (argc,
                                argv,
                                "binary-name",
                                gettext_noop ("binary description text"),
                                options, &run, NULL)) ? ret : 1;
}
```

Once the `ready_cb` callback is executed the network is up and running. By passing a connect_cb you can receive notifications whenever two peers are connected by testing. With the hosts argument, you can specify which hosts to use for distributed execution. Using `NULL` means to run all peers locally.

You can write some test code in the program itself and call it from there or just interact with the testbed using the command line programs as before. In the latter case, you need to specify the `-c` option with a file in `/tmp/binary-name/NUMBER/gnunet-testing-` to select which of the peers the command line tool should connect to.

Exercise: Find out how many peers you can run on your system.

Exercise: Find out how to create connections from within `run` and create a 2D torus topology. Then use the DHT API to store and retrieve values in the network.

1.5 gnunet-ext

A template build system for writing GNUnet extensions in C can be obtained as follows:

```
$ svn checkout https://gnunet.org/svn/gnunet-ext/
$ cd gnunet-ext/
$ . bootstrap
$ ./configure --prefix=$HOME --with-gnunet=$GNUNET_PREFIX
$ make
```

```
$ make install
$ make check
```

The first step for writing any extension with a new service is to ensure that the `defaults.conf` file contains entries for the UNIXPATH, PORT and BINARYNAME for the service in a section named after the service. The `defaults.conf` is located in the `$GNUNET_PREFIX/share/gnunet/` directory (and will be overwritten if you run `make install` for the main GNUnet distribution afterwards, so be careful about this!).[1]

2 Writing a Client Application

When writing any client application (for example, a command-line tool), the basic structure is to start with the `GNUNET_PROGRAM_run` function. This function will parse command-line options, setup the scheduler and then invokes the `run` function (with the remaining non-option arguments) and a handle to the parsed configuration (and the configuration file name that was used, which is typically not needed):

```
#include <gnunet/gnunet_util_lib.h>

static void
run (void *cls,
     char *const *args,
     const char *cfgfile,
     const struct GNUNET_CONFIGURATION_Handle *cfg)
{
  /* main code here */
}

int
main (int argc, char *const *argv)
{
  static const struct GNUNET_GETOPT_CommandLineOption options[] = {
    GNUNET_GETOPT_OPTION_END
  };
  return (GNUNET_OK ==
          GNUNET_PROGRAM_run (argc,
                              argv,
                              "binary-name",
                              gettext_noop ("binary description text"),
                              options, &run, NULL)) ? ret : 1;
}
```

2.1 Handling command-line options

Options can then be added easily by adding global variables and expanding the `options` array. For example, the following would add a string-option and a binary flag (defaulting to `NULL` and `GNUNET_NO` respectively):

[1] A more elegant solution for extending `defaults.conf` will be provided in the future.

```
static char *string_option;
static int a_flag;

// ...
  static const struct GNUNET_GETOPT_CommandLineOption options[] = {
  {'s', "name", "SOMESTRING",
     gettext_noop ("text describing the string option NAME"), 1,
     &GNUNET_GETOPT_set_string, &string_option},
    {'f', "flag", NULL,
     gettext_noop ("text describing the flag option"), 0,
     &GNUNET_GETOPT_set_one, &a_flag},
    GNUNET_GETOPT_OPTION_END
  };
// ...
```

Issues such as displaying some helpful text describing options using the `--help` argument and error handling are taken care of when using this approach. Other `GNUNET_GETOPT_`-functions can be used to obtain integer value options, increment counters, etc. You can even write custom option parsers for special circumstances not covered by the available handlers.

Inside the `run` method, the program would perform the application-specific logic, which typically involves initializing and using some client library to interact with the service. The client library is supposed to implement the IPC whereas the service provides more persistent P2P functions.

Exercise: Add a few command-line options and print them inside of `run`. What happens if the user gives invalid arguments?

2.2 Writing a Client Library

The first and most important step in writing a client library is to decide on an API for the library. Typical API calls include connecting to the service, performing application-specific requests and cleaning up. Many examples for such service APIs can be found in the `gnunet/src/include/gnunet_*_service.h` files.

Then, a client-service protocol needs to be designed. This typically involves defining various message formats in a header that will be included by both the service and the client library (but is otherwise not shared and hence located within the service's directory and not installed by `make install`). Each message must start with a `struct GNUNET_MessageHeader` and must be shorter than 64k. By convention, all fields in IPC (and P2P) messages must be in big-endian format (and thus should be read using `ntohl` and similar functions and written using `htonl` and similar functions). Unique message types must be defined for each message struct in the `gnunet_protocols.h` header (or an extension-specific include file).

2.2.1 Connecting to the Service

Before a client library can implement the application-specific protocol with the service, a connection must be created:

```
client = GNUNET_CLIENT_connect ("service-name", cfg);
```

As a result a `GNUNET_CLIENT_Connection` handle is returned which has to used in later API calls related to this service. The complete client API can be found in `gnunet_client_lib.h`

2.2.2 GNUnet Messages

In GNUnet, messages are always sent beginning with a `struct GNUNET_MessageHeader` in big endian format. This header defines the size and the type of the message, the payload follows after this header.

```
struct GNUNET_MessageHeader
{

  /**
   * The length of the struct (in bytes, including the length field itself),
   * in big-endian format.
   */
  uint16_t size GNUNET_PACKED;

  /**
   * The type of the message (GNUNET_MESSAGE_TYPE_XXXX), in big-endian format.
   */
  uint16_t type GNUNET_PACKED;

};
```

Existing message types are defined in `gnunet_protocols.h`
A common way to create a message is:

```
struct GNUNET_MessageHeader *msg =
  GNUNET_malloc(payload_size + sizeof(struct GNUNET_MessageHeader));
msg->size = htons(payload_size + sizeof(struct GNUNET_MessageHeader));
msg->type = htons(GNUNET_MY_MESSAGE_TYPE);
memcpy(&msg[1], &payload, payload_size);
// use 'msg'
```

Exercise: Define a message struct that includes a 32-bit unsigned integer in addition to the standard GNUnet MessageHeader. Add a C struct and define a fresh protocol number for your message.

2.2.3 Sending Requests to the Service

Any client-service protocol must start with the client sending the first message to the service, since services are only notified about (new) clients upon receiving a the first message.

Clients can transmit messages to the service using the `GNUNET_CLIENT_notify_transmit_ready` API:

```
static size_t
transmit_cb (void *cls, size_t size, void *buf)
{
  // ...
  if (NULL == buf) { handle_error(); return 0; }
  GNUNET_assert (size >= msg_size);
  memcpy (buf, my_msg, msg_size);
  // ...
  return msg_size;
}

// ...
th = GNUNET_CLIENT_notify_transmit_ready (client,
```

```
                                msg_size,
                        timeout,
                                GNUNET_YES,
                                &transmit_cb, cls);
// ...
```

The client-service protocoll calls `GNUNET_CLIENT_notify_transmit_ready` to be notified when the client is ready to send data to the service. Besides other arguments, you have to pass the client returned from the `connect` call, the message size and the callback function to call when the client is ready to send.

Only a single transmission request can be queued per client at the same time using this API. The handle `th` can be used to cancel the request if necessary (for example, during shutdown).

When `transmit_cb` is called the message is copied in the buffer provided and the number of bytes copied into the buffer is returned. `transmit_cb` could also return 0 if for some reason no message could be constructed; this is not an error and the connection to the service will persist in this case.

Exercise: Define a helper function to transmit a 32-bit unsigned integer (as payload) to a service using some given client handle.

2.2.4 Receiving Replies from the Service

Clients can receive messages from the service using the `GNUNET_CLIENT_receive` API:

```
/**
 * Function called with messages from stats service.
 *
 * @param cls closure
 * @param msg message received, NULL on timeout or fatal error
 */
static void
receive_message (void *cls, const struct GNUNET_MessageHeader *msg)
{
  struct MyArg *arg = cls;

  // process 'msg'
}

// ...
  GNUNET_CLIENT_receive (client,
                         &receive_message,
                         arg,
                         timeout);
// ...
```

It should be noted that this receive call only receives a single message. To receive additional messages, `GNUNET_CLIENT_receive` must be called again.

Exercise: Expand your helper function to receive a response message (for example, containing just the GNUnet MessageHeader without any payload). Upon receiving the service's response, you should call a callback provided to your helper function's API. You'll need to define a new 'struct' to hold your local context ("closure").

2.3 Writing a user interface

Given a client library, all it takes to access a service now is to combine calls to the client library with parsing command-line options.

Exercise: Call your client API from your `run` method in your client application to send a request to the service. For example, send a 32-bit integer value based on a number given at the command-line to the service.

3 Writing a Service

Before you can test the client you've written so far, you'll need to also implement the corresponding service.

3.1 Code Placement

New services are placed in their own subdirectory under `gnunet/src`. This subdirectory should contain the API implementation file `SERVICE_api.c`, the description of the client-service protocol `SERVICE.h` and P2P protocol `SERVICE_protocol.h`, the implementation of the service itself `gnunet-service-SERVICE.h` and several files for tests, including test code and configuration files.

3.2 Starting a Service

The key API definitions for starting services are"

```
typedef void (*GNUNET_SERVICE_Main) (void *cls,
                                     struct GNUNET_SERVER_Handle *server,
                                     const struct GNUNET_CONFIGURATION_Handle *cfg);
int GNUNET_SERVICE_run (int argc,
                        char *const *argv,
                        const char *serviceName,
                        enum GNUNET_SERVICE_Options opt,
                        GNUNET_SERVICE_Main task,
                        void *task_cls);
```

Here is a starting point for your main function for your service:

```
static void my_main (void *cls,
                     struct GNUNET_SERVER_Handle *server,
                     const struct GNUNET_CONFIGURATION_Handle *cfg)
{
  /* do work */
}

int main (int argc, char *const*argv)
{
  if (GNUNET_OK !=
      GNUNET_SERVICE_run (argc, argv, "my",
                          GNUNET_SERVICE_OPTION_NONE,
                          &my_main, NULL));
    return 1;
```

```
  return 0;
}
```

Exercise: Write a stub service that processes no messages at all in your code. Create a default configuration for it, integrate it with the build system and start the service from `gnunet-service-arm` using `gnunet-arm -i NAME`.

3.3 Receiving Requests from Clients

Inside of the `my_main` method, a service typically registers for the various message types from clients that it supports by providing a handler function, the message type itself and possibly a fixed message size (or 0 for variable-size messages):

```
static void
handle_get (void *cls,
            struct GNUNET_SERVER_Client *client,
            const struct GNUNET_MessageHeader *message)
{
  GNUNET_SERVER_receive_done (client, GNUNET_OK);
}
static void
handle_get (void *cls,
            struct GNUNET_SERVER_Client *client,
            const struct GNUNET_MessageHeader *message)
{
  GNUNET_SERVER_receive_done (client, GNUNET_OK);
}

static void my_main (void *cls,
                     struct GNUNET_SERVER_Handle *server,
                     const struct GNUNET_CONFIGURATION_Handle *cfg)
{
  static const struct GNUNET_SERVER_MessageHandler handlers[] = {
    {&handle_set, NULL, GNUNET_MESSAGE_TYPE_MYNAME_SET, 0},
    {&handle_get, NULL, GNUNET_MESSAGE_TYPE_MYNAME_GET, 0},
    {NULL, NULL, 0, 0}
  };
  GNUNET_SERVER_add_handlers (server, handlers);
   /* do more setup work */
}
```

Each handler function **must** eventually (possibly in some asynchronous continuation) call `GNUNET_SERVER_receive_done`. Only after this call additional messages from the same client may be processed. This way, the service can throttle processing messages from the same client. By passing `GNUNET_SYSERR`, the service can close the connection to the client, indicating an error.

Services must check that client requests are well-formed and must not crash on protocol violations by the clients. Similarly, client libraries must check replies from servers and should gracefully report errors via their API.

Exercise: Change the service to "handle" the message from your client (for now, by printing a message). What happens if you forget to call `GNUNET_SERVER_receive_done`?

3.4 Responding to Clients

Servers can send messages to clients using the `GNUNET_SERVER_notify_transmit_ready` API:

```
static size_t
transmit_cb (void *cls, size_t size, void *buf)
{
  // ...
  if (NULL == buf) { handle_error(); return 0; }
  GNUNET_assert (size >= msg_size);
  memcpy (buf, my_msg, msg_size);
  // ...
  return msg_size;
}

// ...
th = GNUNET_SERVER_notify_transmit_ready (client,
                                          msg_size,
                                          timeout,
                                          &transmit_cb, cls);
// ...
```

Only a single transmission request can be queued per client at the same time using this API. Additional APIs for sending messages to clients can be found in the `gnunet_server_lib.h` header.

Exercise: Change the service respond to the request from your client. Make sure you handle malformed messages in both directions.

4 Interacting directly with other Peers using the CORE Service

One of the first things any service that extends the P2P protocol typically does is connect to the CORE service:

```
struct GNUNET_CORE_Handle *
GNUNET_CORE_connect (struct GNUNET_SCHEDULER_Handle *sched,
                     const struct GNUNET_CONFIGURATION_Handle *cfg,
                     struct GNUNET_TIME_Relative timeout,
                     void *cls,
                     GNUNET_CORE_StartupCallback init,
                     GNUNET_CORE_ConnectEventHandler connects,
                     GNUNET_CORE_DisconnectEventHandler disconnects,
                     GNUNET_CORE_MessageCallback inbound_notify,
                     int inbound_hdr_only,
                     GNUNET_CORE_MessageCallback outbound_notify,
                     int outbound_hdr_only,
                     const struct GNUNET_CORE_MessageHandler *handlers);
```

4.1 New P2P connections

Before any traffic with a different peer can be exchanged, the peer must be known to the service. This is notified by the CORE `connects` callback, which communicates the identity of the new peer to the service:

```
void
connects (void *cls,
          const struct GNUNET_PeerIdentity * peer,
          const struct GNUNET_ATS_Information * atsi,
          unsigned int atsi_count)
{
    /* Save identity for later use */
    /* Optional: start sending messages to peer */
}
```

Exercise: Create a service that connects to the CORE. Then start (and connect) two peers and print a message once your connect callback is invoked.

4.2 Receiving P2P Messages

To receive messages from CORE, services register a set of handlers (parameter `*handlers` in the CORE_connect call) that are called by CORE when a suitable message arrives.

```
static int
callback_function_for_type_one(void *cls,
                               const struct GNUNET_PeerIdentity *peer,
                               const struct GNUNET_MessageHeader *message,
                               const struct GNUNET_TRANSPORT_ATS_Information
                               *atsi)
{
    /* Do stuff */
    return GNUNET_OK; /* or GNUNET_SYSERR to close the connection */
}

/**
 * Functions to handle messages from core
 */
static struct GNUNET_CORE_MessageHandler core_handlers[] = {
  {&callback_function_for_type_one, GNUNET_MESSAGE_TYPE_MYSERVICE_TYPE_ONE, 0},
  /* more handlers*/
  {NULL, 0, 0}
};
```

Exercise: Start one peer with a new service that has a message handler and start a second peer that only has your "old" service without message handlers. Which "connect" handlers are invoked when the two peers are connected? Why?

4.3 Sending P2P Messages

In response to events (connect, disconnect, inbound messages, timing, etc.) services can then use this API to transmit messages:

```
typedef size_t
(*GNUNET_CONNECTION_TransmitReadyNotify) (void *cls,
                                          size_t size,
                                          void *buf)
{
    /* Fill "*buf" with up to "size" bytes, must start with GNUNET_MessageHeader */
    return n; /* Total size of the message put in "*buf" */
}

struct GNUNET_CORE_TransmitHandle *
GNUNET_CORE_notify_transmit_ready (struct GNUNET_CORE_Handle *handle,
                                   int cork, uint32_t priority,
                                   struct GNUNET_TIME_Relative maxdelay,
                                   const struct GNUNET_PeerIdentity *target,
                                   size_t notify_size,
                                   GNUNET_CONNECTION_TransmitReadyNotify notify,
                                   void *notify_cls);
```

Exercise: Write a service that upon connect sends messages as fast as possible to the other peer (the other peer should run a service that "processes" those messages). How fast is the transmission? Count using the STATISTICS service on both ends. Are messages lost? How can you transmit messages faster? What happens if you stop the peer that is receiving your messages?

4.4 End of P2P connections

If a message handler returns `GNUNET_SYSERR`, the remote peer shuts down or there is an unrecoverable network disconnection, CORE notifies the service that the peer disconnected. After this notification no more messages will be received from the peer and the service is no longer allowed to send messages to the peer. The disconnect callback looks like the following:

```
void
disconnects (void *cls,
             const struct GNUNET_PeerIdentity * peer)
{
    /* Remove peer's identity from known peers */
    /* Make sure no messages are sent to peer from now on */
}
```

Exercise: Fix your service to handle peer disconnects.

5 Using the DHT

The DHT allows to store data so other peers in the P2P network can access it and retrieve data stored by any peers in the network. This section will explain how to use the DHT. Of course, the first thing to do is to connect to the DHT service:

```
dht_handle = GNUNET_DHT_connect (cfg, parallel_requests);
```

The second parameter indicates how many requests in parallel to expect. It is not a hard limit, but a good approximation will make the DHT more efficiently.

5.1 Storing data in the DHT

Since the DHT is a dynamic environment (peers join a leave frequently) the data that we put in the DHT does not stay there indefinitely. It is important to "refresh" the data periodically by simply storing it again, in order to make sure other peers can access it.

The put API call offers a callback to signal that the PUT request has been sent. This does not guarantee that the data is accessible to others peers, or even that is has been stored, only that the service has requested to a neighboring peer the retransmission of the PUT request towards its final destination. Currently there is no feedback about whether or not the data has been sucessfully stored or where it has been stored. In order to improve the availablilty of the data and to compensate for possible errors, peers leaving and other unfavorable events, just make several PUT requests!

```
void
message_sent_cont (void *cls, const struct GNUNET_SCHEDULER_TaskContext *tc)
{
    /* Request has left local node */
}

GNUNET_DHT_put (dht_handle,
                &key,
                replication,
                GNUNET_DHT_RO_NONE, /* Route options, see next call */
                block_type,
                data_size,
                data,
                expiration, /* When does the data expire? */
                timeout, /* How long to try to send the request */
                &message_sent_cont,
                cls);
```

Exercise: Store a value in the DHT periodically to make sure it is available over time. You might consider using the function GNUNET_SCHEDULER_add_delayed and call GNUNET_DHT_put from inside a helper function.

5.2 Obtaining data from the DHT

As we saw in the previous example, the DHT works in an asynchronous mode. Each request to the DHT is executed "in the background" and the API calls return immediately. In order to receive results from the DHT, the API provides a callback. Once started, the request runs in the service, the service will try to get as many results as possible (filtering out duplicates) until the timeout expires or we explicitly stop the request. It is possible to give a "forever" timeout with `GNUNET_TIME_UNIT_FOREVER_REL`.

If we give a route option `GNUNET_DHT_RO_RECORD_ROUTE` the callback will get a list of all the peers the data has travelled, both on the PUT path and on the GET path.

```
static void
get_result_iterator (void *cls, struct GNUNET_TIME_Absolute expiration,
                     const GNUNET_HashCode * key,
                     const struct GNUNET_PeerIdentity *get_path,
                     unsigned int get_path_length,
                     const struct GNUNET_PeerIdentity *put_path,
                     unsigned int put_path_length,
```

```
                    enum GNUNET_BLOCK_Type type, size_t size, const void *data)
{
    /* Do stuff with the data and/or route */
    /* Optionally: */
    GNUNET_DHT_get_stop (get_handle);
}

get_handle =
      GNUNET_DHT_get_start (dht_handle,
                            timeout,
                            block_type,
                            &key,
                            replication,
                            GNUNET_DHT_RO_NONE, /* Route options */
                            NULL, /* xquery: not used here */
                            0, /* xquery size */
                            &get_result_iterator,
                            cls)
```

Exercise: Store a value in the DHT and after a while retrieve it. Show the IDs of all the peers the requests have gone through. In order to convert a peer ID to a string, use the function GNUNET_i2s. Pay attention to the route option parameters in both calls!

5.3 Implementing a block plugin

In order to store data in the DHT, it is necessary to provide a block plugin. The DHT uses the block plugin to ensure that only well-formed requests and replies are transmitted over the network.

The block plugin should be put in a file `plugin_block_SERVICE.c` in the service's respective directory. The mandatory functions that need to be implemented for a block plugin are described in the following sections.

5.3.1 Validating requests and replies

The evaluate function should validate a reply or a request. It returns a `GNUNET_BLOCK_EvaluationResult`, which is an enumeration. All possible answers are in `gnunet_block_lib.h`. The function will be called with a `reply_block` argument of `NULL` for requests. Note that depending on how `evaluate` is called, only some of the possible return values are valid. The specific meaning of the `xquery` argument is application-specific. Applications that do not use an extended query should check that the `xquery_size` is zero. The Bloom filter is typically used to filter duplicate replies.

```
static enum GNUNET_BLOCK_EvaluationResult
block_plugin_SERVICE_evaluate (void *cls,
                               enum GNUNET_BLOCK_Type type,
                               const GNUNET_HashCode * query,
                               struct GNUNET_CONTAINER_BloomFilter **bf,
                               int32_t bf_mutator,
                               const void *xquery,
                               size_t xquery_size,
                               const void *reply_block,
                               size_t reply_block_size)
```

```
{
  /* Verify type, block and bloomfilter */
}
```

5.3.2 Deriving a key from a reply

The DHT can operate more efficiently if it is possible to derive a key from the value of the corresponding block. The `get_key` function is used to obtain the key of a block — for example, by means of hashing. If deriving the key is not possible, the function should simply return `GNUNET_SYSERR` (the DHT will still work just fine with such blocks).

```
static int
block_plugin_SERVICE_get_key (void *cls, enum GNUNET_BLOCK_Type type,
                              const void *block, size_t block_size,
                              GNUNET_HashCode * key)
{
    /* Store the key in the key argument, return GNUNET_OK on success. */
}
```

5.3.3 Initialization of the plugin

The plugin is realized as a shared C library. The library must export an initialization function which should initialize the plugin. The initialization function specifies what block types the plugin cares about and returns a struct with the functions that are to be used for validation and obtaining keys (the ones just defined above).

```
void *
libgnunet_plugin_block_SERVICE_init (void *cls)
{
  static enum GNUNET_BLOCK_Type types[] =
  {
    GNUNET_BLOCK_TYPE_SERVICE_BLOCKYPE, /* list of blocks we care about, from gnunet_block_lib.h
    GNUNET_BLOCK_TYPE_ANY       /* end of list */
  };
  struct GNUNET_BLOCK_PluginFunctions *api;

  api = GNUNET_malloc (sizeof (struct GNUNET_BLOCK_PluginFunctions));
  api->evaluate = &block_plugin_SERICE_evaluate;
  api->get_key = &block_plugin_SERVICE_get_key;
  api->types = types;
  return api;
}
```

5.3.4 Shutdown of the plugin

Following GNUnet's general plugin API concept, the plugin must export a second function for cleaning up. It usually does very little.

```
void *
libgnunet_plugin_block_SERVICE_done (void *cls)
{
  struct GNUNET_TRANSPORT_PluginFunctions *api = cls;

  GNUNET_free (api);
  return NULL;
}
```

5.3.5 Integration of the plugin with the build system

In order to compile the plugin, the `Makefile.am` file for the service should contain a rule similar to this:

```
plugin_LTLIBRARIES = \
  libgnunet_plugin_block_SERVICE.la
libgnunet_plugin_block_SERVICE_la_SOURCES = \
  plugin_block_SERVICE.c
libgnunet_plugin_block_SERVICE_la_LIBADD = \
  $(top_builddir)/src/hello/libgnunethello.la \
  $(top_builddir)/src/block/libgnunetblock.la \
  $(top_builddir)/src/util/libgnunetutil.la
libgnunet_plugin_block_SERVICE_la_LDFLAGS = \
 $(GN_PLUGIN_LDFLAGS)
libgnunet_plugin_block_SERVICE_la_DEPENDENCIES = \
  $(top_builddir)/src/block/libgnunetblock.la
```

Exercise: Write a block plugin that accepts all queries and all replies but prints information about queries and replies when the respective validation hooks are called.

5.4 Monitoring the DHT

It is possible to monitor the functioning of the local DHT service. When monitoring the DHT, the service will alert the monitoring program of any events, both started locally or received for routing from another peer. The are three different types of events possible: a GET request, a PUT request or a response (a reply to a GET).

Since the different events have different associated data, the API gets 3 different callbacks (one for each message type) and optional type and key parameters, to allow for filtering of messages. When an event happens, the appropiate callback is called with all the information about the event.

```
void
get_callback (void *cls,
              enum GNUNET_DHT_RouteOption options,
              enum GNUNET_BLOCK_Type type,
              uint32_t hop_count,
              uint32_t desired_replication_level,
              unsigned int path_length,
              const struct GNUNET_PeerIdentity *path,
              const GNUNET_HashCode * key)
```

```
{
}

void
get_resp_callback (void *cls,
                   enum GNUNET_BLOCK_Type type,
                   const struct GNUNET_PeerIdentity *get_path,
                   unsigned int get_path_length,
                   const struct GNUNET_PeerIdentity *put_path,
                   unsigned int put_path_length,
                   struct GNUNET_TIME_Absolute exp,
                   const GNUNET_HashCode * key,
                   const void *data,
                   size_t size)
{
}

void
put_callback (void *cls,
              enum GNUNET_DHT_RouteOption options,
              enum GNUNET_BLOCK_Type type,
              uint32_t hop_count,
              uint32_t desired_replication_level,
              unsigned int path_length,
              const struct GNUNET_PeerIdentity *path,
              struct GNUNET_TIME_Absolute exp,
              const GNUNET_HashCode * key,
              const void *data,
              size_t size)
{
}

monitor_handle = GNUNET_DHT_monitor_start (dht_handle,
                                           block_type, /* GNUNET_BLOCK_TYPE_ANY for all */
                                           key, /* NULL for all */
                                           &get_callback,
                                           &get_resp_callback,
                                           &put_callback,
                                           cls);
```

6 Debugging with `gnunet-arm`

Even if services are managed by `gnunet-arm`, you can start them with `gdb` or `valgrind`. For example, you could add the following lines to your configuration file to start the DHT service in a `gdb` session in a fresh `xterm`:

```
[dht]
```

```
PREFIX=xterm -e gdb --args
```

Alternatively, you can stop a service that was started via ARM and run it manually:

```
$ gnunet-arm -k dht
$ gdb --args gnunet-service-dht -L DEBUG
$ valgrind gnunet-service-dht -L DEBUG
```

Assuming other services are well-written, they will automatically re-integrate the restarted service with the peer.

Finally, set the option "DEBUG=YES" to start services with logging of DEBUG messages (if you are using `GNUNET_log` for printf-style debugging).

You should also probably enable the creation of core files, by setting `ulimit`, and echo'ing 1 into `/proc/sys/kernel/core_uses_pid`. Then you can investigate the core dumps with `gdb`, which is often the fastest method to find simple errors.

Exercise: Add a memory leak to your service and obtain a trace pointing to the leak using `valgrind` while running the service from `gnunet-service-arm`.