NTNU – Trondheim
Norwegian University of
Science and Technology

General Utilisation System for Timed Application and Fast Scheduling Over Network

Knut André Karlsen Vestergren

Master of Science in Engineering Cybernetics
Submission date: June 2012
Supervisor: Sverre Hendseth, ITK

Norwegian University of Science and Technology
Department of Engineering Cybernetics

Problem

Is it possible to utilise the computational power of a multi-computer environment for real-time applications by developing an experimental runtime system and exploring its applications?

Abstract

In this thesis, an experimental runtime system for utilizing the computational power of a multi-computer environment is presented.

Through simple benchmark tests it is shown how some tasks will have a considerate speed-up compared to running on a single computer.

An outline for designing languages and compilers suited for the runtime is also explored and discussed, and it is shown how the system, with some extensions, would be well suited for utilizing the spare computational power in a multi-computer environment. This also holds, with some extra considerations, for a real-time application.

Acknowledgement

I would like to thank my supervisor, Sverre Hendseth, for guidance and interesting digressions.

I would also like to thank my fiancé, Ingrid Hjorth, for her support and patience during the time I have been working on this thesis.

Contents

Code

List of Tables

List of Figures

Chapter 1

Introduction

In this thesis, an experimental runtime system developed for running real-time applications in a multi-computer environment is presented. Structure and implementation of the runtime will be explained and its applications explored.

The thesis also briefly presents a simple benchmark test of the system.

Initially, the GUSTAFSON runtime system was planned to support the design of a multi-computer, real-time language. However, as the planning progressed, the design and implementation of the runtime itself showed to be of considerate size. As such, this project focus on the runtime, with some consideration of language design (in chapter 4).

1.1 The name

The runtime presented in this thesis has been given the name GUSTAFSON, which stands for *General Utilisation System for Timed Application and Fast Scheduling Over Network*. The name is a backronym[1], and the resulting runtime may neither be as general nor fast as the name could imply, and only manual, static scheduling is currently used. It should, however, be a valid description for the ideal the runtime is shaped after.

1.2 Previous work

The author has for his master's specialisation project[1] developed an experimental real-time language with associated compiler and runtime for a single-computer, multi-core environment. Although this thesis should be regarded as an independent project, experiences from the specialisation project will have influenced some decisions made in this project where their scopes overlap. This is especially true for

[1] A backronym or bacronym is a phrase constructed purposely, such that an acronym can be formed to a specific desired word[3].

the task manager (presented in section 2.2), which is designed to avoid problems faced in the specialisation project.

Chapter 2

Structure

2.1 Terminology

This is a short overview of the terminology used in this chapter.

- *System* is used to describe the overall system; that is, the *nodes* connected together.
- *Node* is one specific instance of the runtime, usually running on its own computer.
- *Peer* is sometimes used instead of *node* to distinguish between one node (called *node*) and other nodes (called *peers*) it communicates with.
- *Worker* is a part of the runtime, running in its own thread.
- A *program* is executed on the *system*.
- A *procedure* is a small part of the *program*, designed to run on a single *node*.
- A *task* is an instance of a *procedure*, running on a single *node*, communicating with other *tasks* on the same or other *nodes*.
- Each *node* is assigned its own unique *id*, used to referring to it in the program.

2.2 Task management

2.2.1 Task state

In a concurrent system, a task will often be described to have a state that reflects whether the task is being executed, waiting to be executed or is blocked, waiting for another task to release a resource or finish a computation. An example of such a set of states is given here:

- RUNNING - The task is currently being executed on the CPU.
- READY - The task is ready to run, and is waiting for an available CPU.
- BLOCKED - The task requires a resource currently held by another task, or needs the result from another task's computation.
- FINISHED - The task has completed, and may be deleted.

GUSTAFSON has a set of states that is based on these, but more states are added to distinguish between different reasons for the task to be blocked, and for the task to request actions from the runtime system.

- NEW - The state of the newly created task. The runtime set the state to NEW when the task is created, so that the task will do necessary initialisations when run for the first time.
- READY - The task is ready to run, and is waiting for an available CPU in a FIFO queue. Both the task itself and the runtime may set a task state to READY.
- CHANR - The task requests to read a channel. This may or may not block, depending on whether the channel has data available or not. This state is set by the task, but the runtime may return the task to this state from CHANRNW.
- CHANW - The task requests to write to a channel. This may or may not block, depending on whether the channel has free buffer space or not. This state is set by the task, but the runtime may return the task to this state from CHANWNW.
- NODEWAIT, CHANRNW, CHANWNW - The task is waiting for another node to appear in the system. NODEWAIT is set by the task to tell the runtime to check whether a given node is ready, and to wait for it if it is not. CHANRNW and CHANWNW ("channel read node wait" and "channel write node wait") is set by the runtime if a channel operation could not be performed because the corresponding node is not ready.[1]
- TRANSFER - The task tells the runtime to transfer a procedure to another node.
- SPAWN - The task tells the runtime to execute a task on this or another node. Arguments are given to tell the task what channels to use, and what nodes to communicate with.
- DONE - The task is finished, and the runtime should delete the entry.

[1]The CHANWNW state could have been omitted, since the transmitted data is buffered in both the sending and receiving node until read by the destination task. It has, however, been included to simplify the runtime, and to make read and write operations more similar.

2.2.2 Operation

The task manager contains of a queue of non-blocking tasks, and a number of workers that fetch tasks to run from this queue. The tasks in the queue can both be ready to be executed and ready to perform channel communication. In the first case the task will be in the state *NEW* or *READY*, and in the latter case the state will be *CHANR* or *CHANW*, and has been put in this state by the runtime from *CHANRNW*/*CHANWNW* (see section 2.4).

If the task is ready to run it is executed. Afterwards, the state of the task is checked again, and if the state is *READY* the task is put back in the queue. Any other state will trigger additional actions, such as channel communication or spawning a new task. If this action does not block, the state of the task is then returned to *READY*, and the task is put back in the queue. If the action blocks, the task will be handled by other parts of the runtime (see sections 2.3 and 2.4). If the action is to delete the task (the state is *DONE*), the task is of course neither returned to the queue nor stored in other parts of the runtime.

2.3 Channels

All communications between tasks are made by asynchronous channels. The channels are made asynchronous since they partly communicate over network, and synchronous channels would therefore in many cases cause unacceptable delay. If procedures should need synchronous communication, such channels may easily be built on top of asynchronous channels.

Channels are bi-directional and both the sender and the receiver specify the number of bytes they want to read/write on each operation. It is up to the user[2] of the system to ensure that the transferred data is assembled back into the correct data structure, and that any difference in byte ordering between host is compensated for. Due to this, and other design choices, channels communication is restricted to be one-to-one, although it is not enforced by the runtime and it will again fall to the user to ensure correct use.

The runtime system maintains two buffers for each channel; one for read and one for write. If the communicating tasks are on different nodes both nodes holds a copy of each of the buffers, but the buffers are not duplicated if the tasks reside on the same node. This design is chosen so that the task may return quickly from a channel operation[3] while the potentially slow network communication is carried out by a number of parallel workers.

Each channel holds a queue[4] of blocked tasks. These tasks are blocked due to

[2] *The user* may refere to both machine (compiler) or manual programming.

[3] Given that the operation does not block due to an empty buffer in case of a read operation or a full buffer in case of a write operation.

[4] Due to the channels being one-to-one, the queue size should never exceed one. The design and implementation do, however, not hold this limitation, partly to allow the use of queue design from other parts of the runtime (that allows longer queues of blocked tasks), and partly to allow future implementations of one-to-many/many-to-many channels.

an empty/full buffer. Tasks that are blocked because they are waiting for another node to register in the system are managed by another part of the runtime. (See section 2.4.)

2.3.1 Example

Figures 2.1 through 2.8 shows an example of one task on one node sending the string "Hello, world!" to another task on another node. The top of the figures show the memory in which the string to be sent resides. Below follows the send buffer of the sending node and the receive buffer of the receiving node and finally the memory the receiving task has allocated for the string.

The sending buffer holds three pointers to manage the data transfers; *swPtr*, *ssPtr* and *srPtr* which is the write pointer, the send pointer and the read pointer, respectively. The write pointer points to the next place in the buffer to write to. If it points to the place before the read pointer, the buffer is considered full (meaning that the effective capacity is one byte less than the allocated memory for the buffer.) The send pointer points to the next byte in the buffer to send over the network to the receiving node. The read pointer points to the next byte for the receiving task to read, and reflects the state of the receive buffer on the other node. It is in other words updated when the sending node receives an acknowledgement from the receiving node that the receiving task has read some of the transferred data.

The receiving buffer is similar, but only holds two moving pointers; *rwPtr* and *rrPtr*, the write and read pointers, respectively. Data received over the network is put at the location pointed to by the write pointer, and the receiving task gets data from the location pointed to by the read pointer.

With the assumption of instant network transfers, the two buffers on the two nodes will be identical, and the write and send pointer on the sending side will point to the same location. This assumption does of course not hold, but the buffers will still be identical when the system is in a stable state with no data waiting to be sent over the network, as it is when it is idle (neither of the communicating task wish to send or receive), only the receiver is ready, or only the sender is ready and it has filled the buffers.

The progress of this example is described in the captions of the figures.

2.3.2 Local communication

When both the sender and the receiver resides on the same node the matter is somewhat simplified, but also somewhat complicated. Simplified because all steps involving network transfers and acknowledgements is no longer necessary, but complicated because each buffer is both a read and write buffer, depending on which task that is using it. The system solves this by recording the first task that access the channel, and later it will compare any task accessing the channel to the record, deciding what buffer to use on this basis.

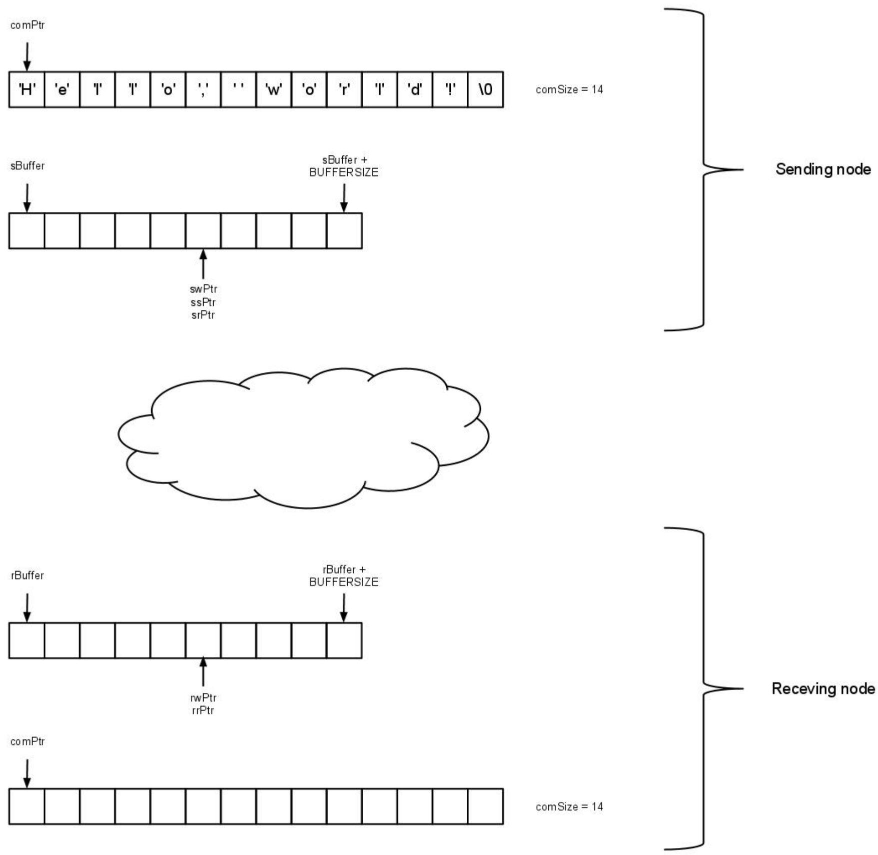

Figure 2.1: Both the sender and the receiver is ready to start the transfer and the buffers are empty. The receiving task is blocked and suspended, and the channel manager will wake it when the buffer holds data for it to read.

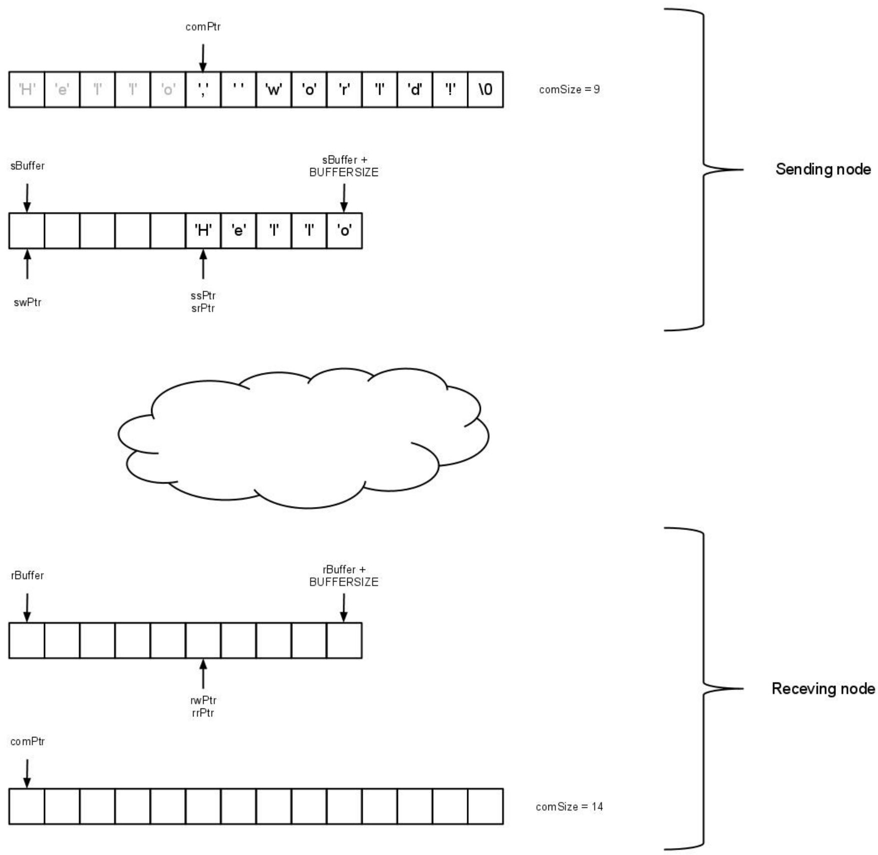

Figure 2.2: The sender has transferred the first part ("Hello") to the send buffer. There is still room for more in the buffer, but the end of the allocated buffer is reached, so the transfer is done in two parts.

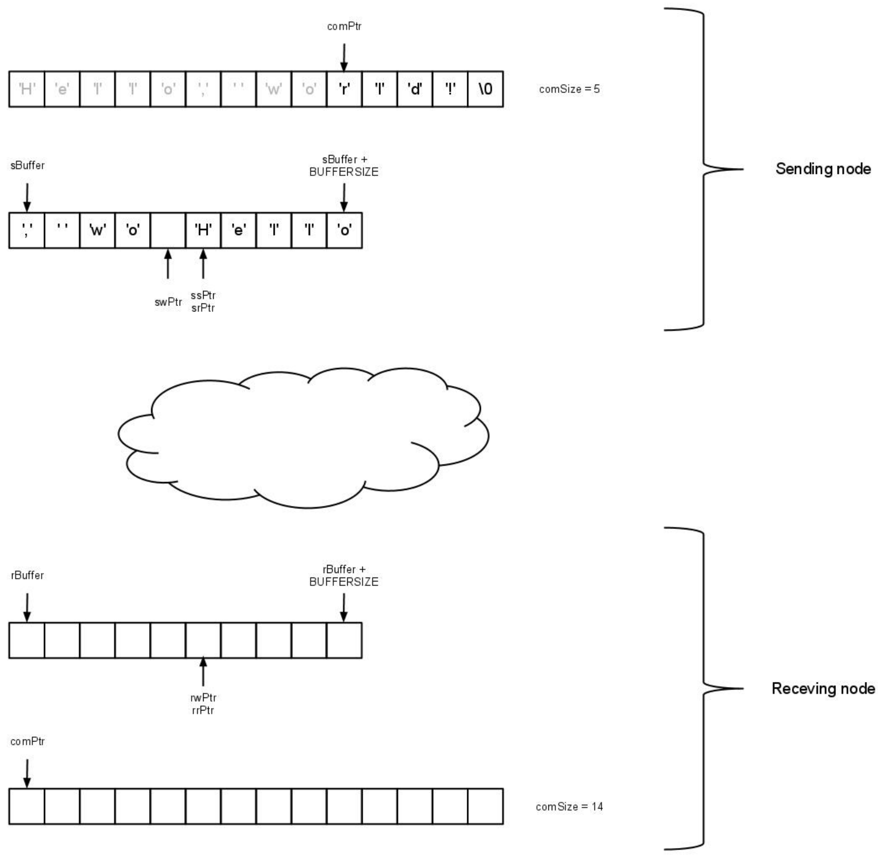

Figure 2.3: The sender has transferred the second part (", wo") to the send buffer. The buffer is now full, as the effective capacity of the buffer is one byte less than the allocated memory. The sending task is now blocked as it still need to send "rld!\0", and it is suspended until the channel manager wakes it up.

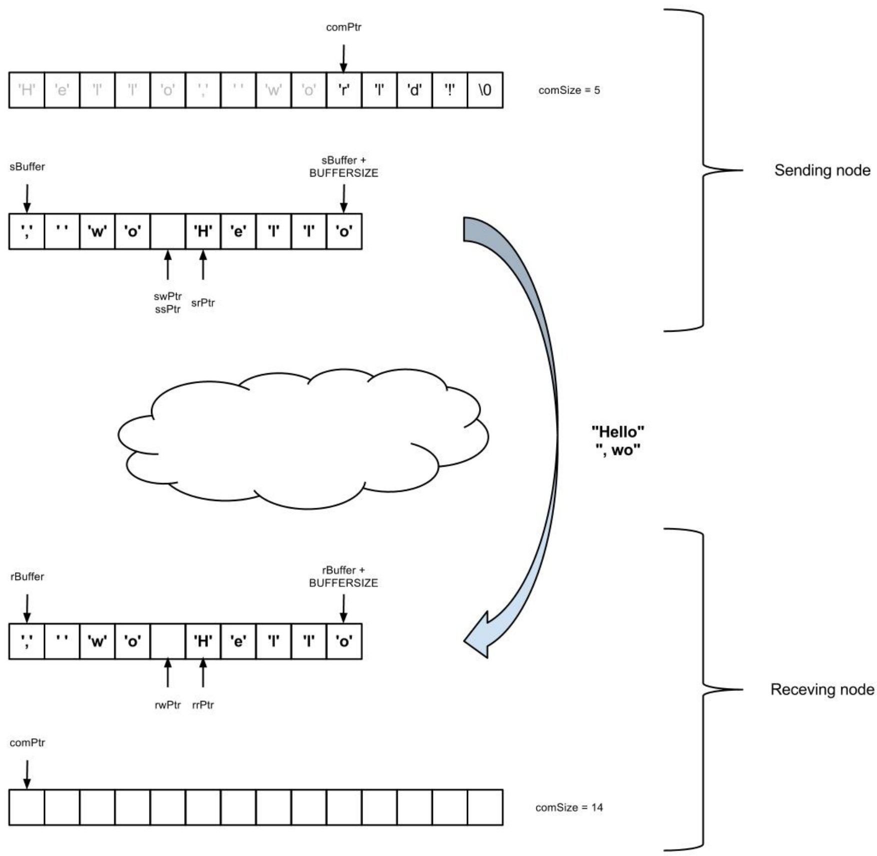

Figure 2.4: The content of the buffer is sent over the network to the receiving node. This is actually done in two parts, similar to the transfer to the buffer, but is shown in one figure to simplify the example. The receiving task is no longer blocked and is waked by the channel manager to resume the write operation.

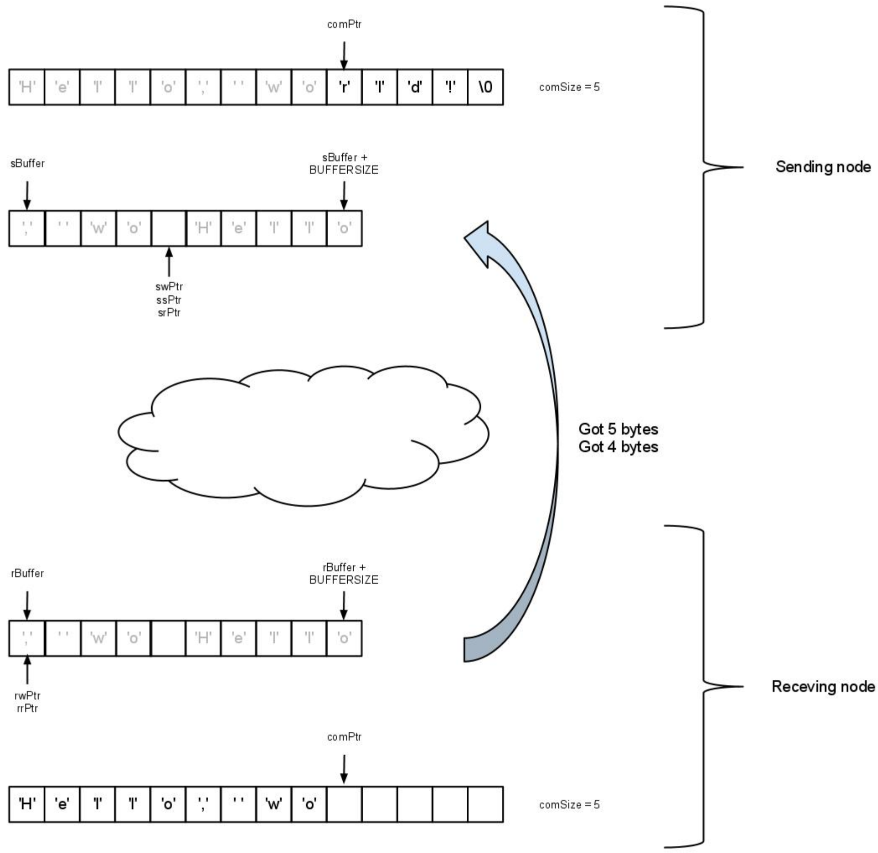

Figure 2.5: The receiver now copies the content of the receive buffer to its allocated memory. Again, this is actually done in two parts, but the example is simplified to show it in one figure. As the receiver reads the data, it sends an acknowledgement to the sender, which in turn moves its read pointer, and frees buffer space. The buffers are now empty again, meaning that the sending task is unblocked and resumes write operation, while the receiving task is once again blocked and suspended.

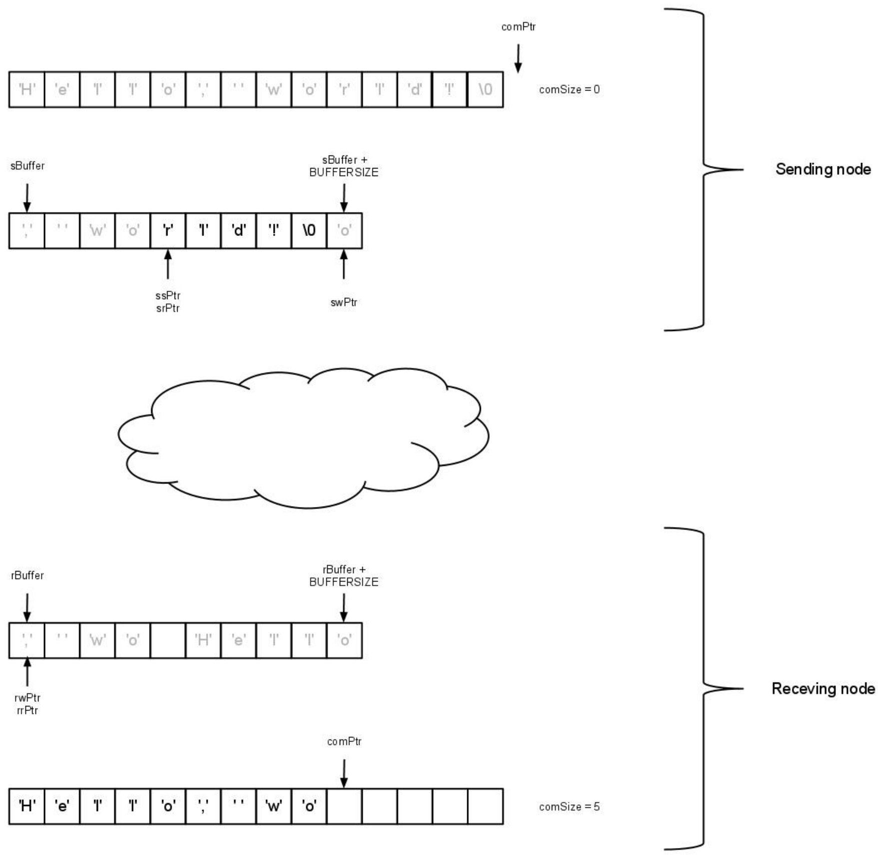

Figure 2.6: The sender now copies the rest of its message ("rld!\0") to the buffer. It has now completed its part of the transfer, and returns to its execution.

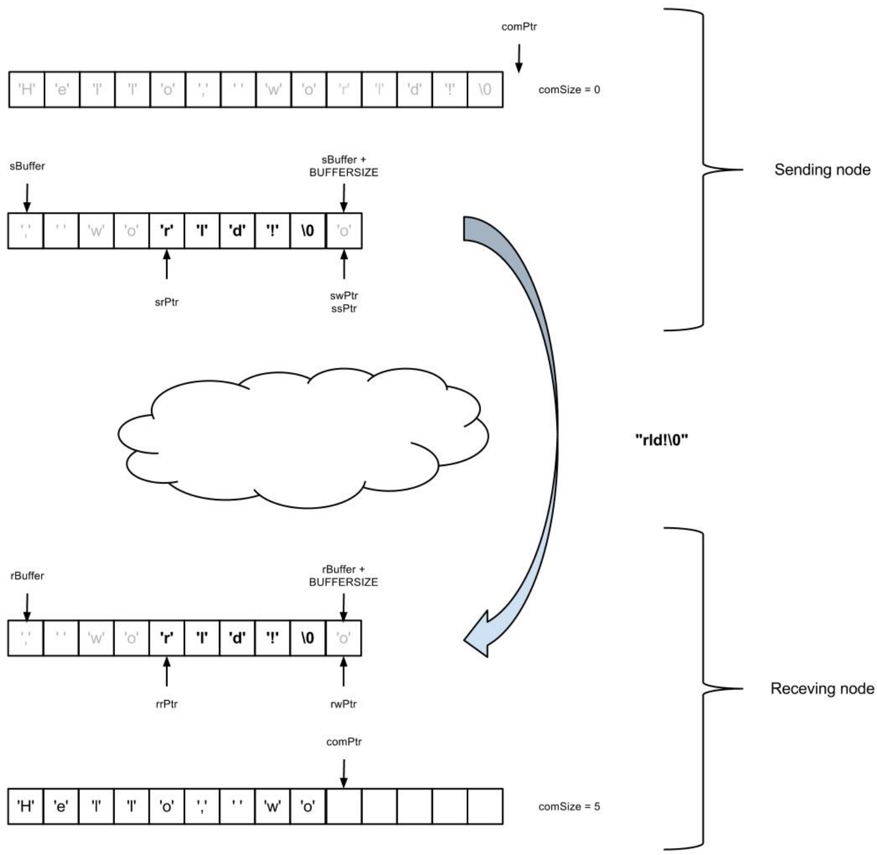

Figure 2.7: The content of the send buffer is again sent over the network to the receiver. The receiving task is unblocked and resumes the read operation.

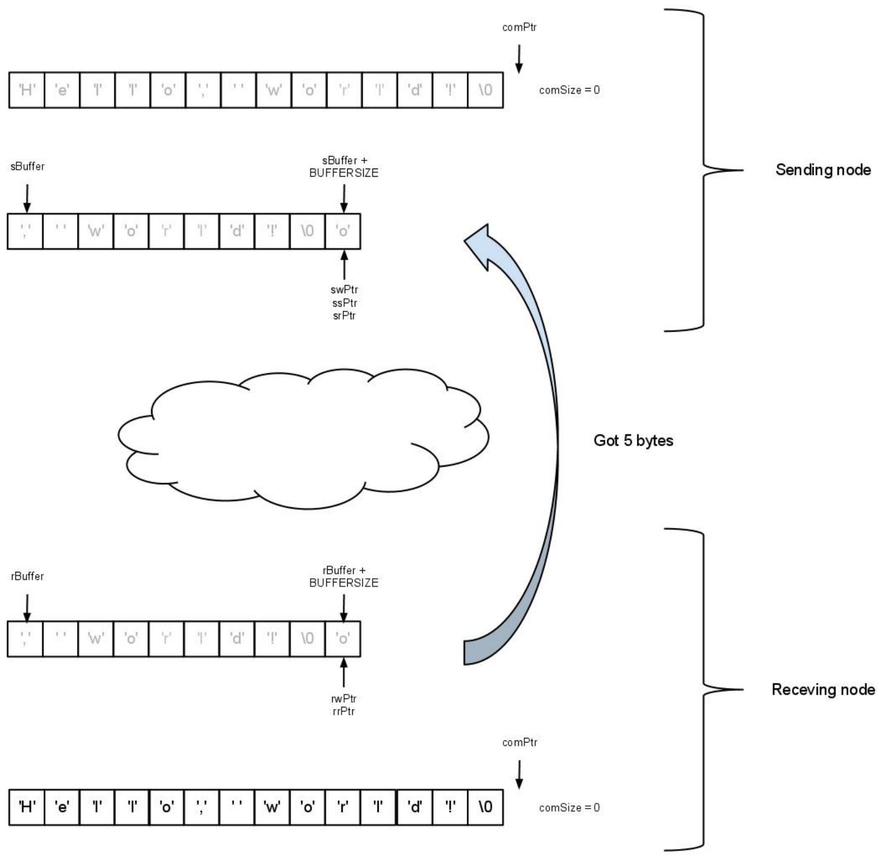

Figure 2.8: The rest of the message is copied from the receive buffer to the allocated memory. An acknowledgement is sent to the sender, and the receiving task has completed the communication and returns to its execution. Both buffers are now empty and ready for a new transfer.

2.4 Node and network managing

2.4.1 Connecting to other nodes

In most aspects, the nodes of the system may be regarded as equals. The system is peer-to-peer and all nodes communicate directly with all other nodes. However, when the system is starting up, one node is designated master and all other nodes are told to connect to this node. When a node connects to the master it is informed of any other node currently connected to the master, and it will in turn connect to these other nodes as well.

Whenever a node connects to the master the master accepts the connection and sends a "handshake" to the node. The handshake contains the id of the master and the IP address and port number of all other nodes already connected to the master. When receiving the handshake, the connecting node stores the id of the master and sends a handshake back to the master. This handshake is on the same form as the one from the master, and contains the id of the node, and the IP address and port number of all connected nodes.[5] When the master receives this handshake, it stores the id of the node, and replies with another handshake. This handshake will be identical to the previous handshake the master sent, unless another node has connected to the master in the meantime and is completely added[6], in which case the IP address and port number of this new node is included in the handshake as well. When this handshake is sent, the master adds the node's IP address and port number to the handshake it will sent to subsequently connecting nodes. Upon receiving this second handshake from the master, the node connects to all the nodes specified in the handshake in the same way it connected to the master, except it will not connect to any nodes received in handshakes from other nodes than the master.

Figure 2.9 illustrates how node 2 connects to the master (node 1). Node 3 is already connected to the master, and node 2 connects to this node after connecting to the master.

Figure 2.10 illustrates how two nodes (node 2 and 3) connects to the master at the same time. Small differences in timing may decide if node 3 should connect to node 2, or vice versa. In this example the master deal with the handshake from node 3 first, and thereby have node 3 completely registered before node 2, and so it will be node 2 which connect to node 3. In the example we can also see that the master sends its first handshake to node 2 first, but this is of little to no consequence.

It is interesting to note that apart from that the master does not try to connect to another node on start-up, it behaves exactly like any other node in the system. In other words, naming any existing node as master to a new node when adding it would work just fine, but with one exception: if two nodes join the network at approximately the same time, connection to two different masters, they may not

[5] This list is empty at this time of the operation, the master is added to the list after the handshake is sent.

[6] "Completely added" means that the master has sent its second handshake to the node.

discover each other. For this reason, all nodes should connect to the same master.

2.4.2 Managing other nodes

All nodes know of all other nodes (its peers). The peers are stored in three different data structures; a linked list, a hash table and a string. In addition to the socket used for communicating with the peer, the id and address informations are stored.

The linked list is used for closing all connections when restarting the system (on critical errors). Peers are added to the list as they connect to the node, or as the node connect to them.

The string is the handshake used when the node connects to peers, or peers to connect to it. If the node is the master, the connecting peer will use this information to connect to all other peers connected to the master. (See section 2.4.1.)

The hash table use the peer id as key and "key modulo number of buckets" as hash function. Peers will be added when the node receives a handshake, but an entry will also be made if a program makes a reference to a peer not yet connected, that is, the program attempts channel communication with a peer that is not connected, or explicitly tells the runtime to wait for a peer to be ready (by setting it's state to *NODEWAIT*, see section 2.2.1).

These three data structures are split between two modules; the network module holds the linked list and the string, while the hash table is held by an individual module called *PeerHash*, after the data structure it holds. While the network module is responsible for the actual communication with, and connection to, other nodes, the PeerHash module offers functionality for quickly retrieving the peer information given the id (used by the channel communication module and the network module) and is responsible for storing and waking tasks that are blocked waiting for a peer to connect.

2.5 Summary of task storage

In summary, when a task is not executed[7] it is stored in one of three possible locations; the queue of ready-to-run tasks (see section 2.2.2), the channel module (tied to a channel blocking the task, see section 2.3), or the PeerHash module (waiting for a peer to connect to the node, see 2.4.2).

2.6 File and procedure managing

A program consists of a number of procedures. When the program is prepared to run on GUSTAFSON, each procedure is compiled as a dynamic linkable library and placed in its own separate file with the same name as the procedure. The entry

[7] Non-blocking channel communication and other non-blocking actions is included in the term *execute* here

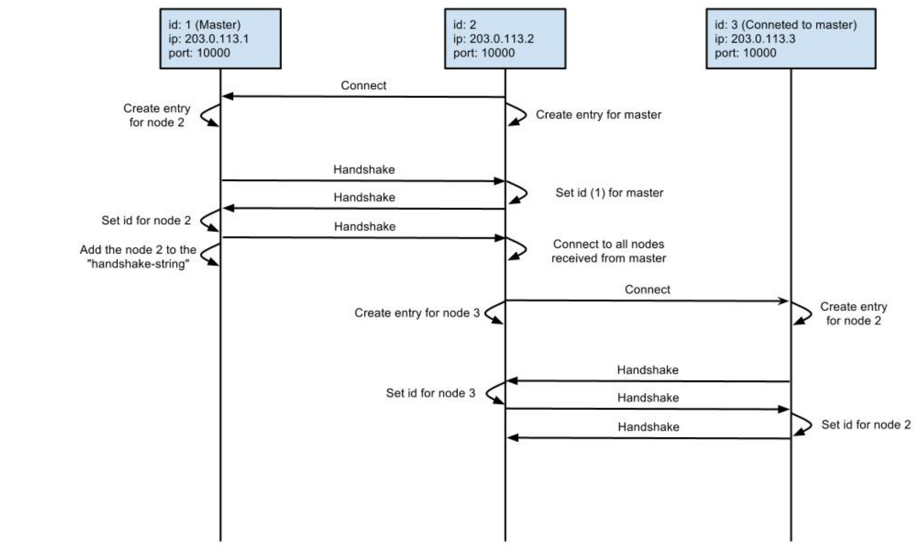

Figure 2.9: Example of simple node connection

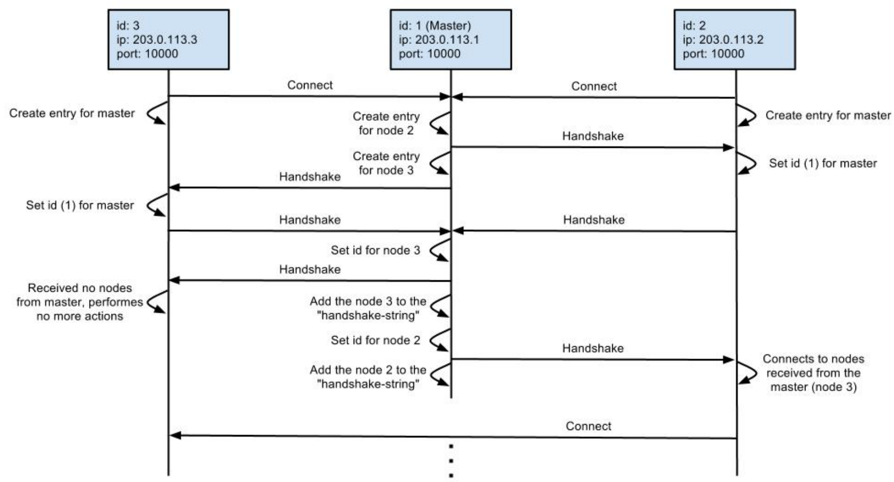

Figure 2.10: Example of interleaved node connection

point (main procedure) of the program to run is specified on one of the nodes when the runtime is started. See section 3.2.2 and 3.3 for more information on this.

It is actually possible to specify several independent, or even dependent, programs to to run simultaneously (one for each node), however, extra care must be taken when designing such programs as the system will not validate that several programs do not use the same channels.

GUSTAFSON holds functionality for transferring the function files to the nodes that needs them. This is invoked manually by the task setting its state to *TRANSFER* (see section 2.2.1). The designer of the program is responsible for ensuring that any node holds the needed files before the program tries to spawn a task from the corresponding function on that node. Since the transferring functions is a relatively slow operation[8], it might be wise for the designer to ensure that all nodes have the needed files before the program is run, and not make the program itself do the transferring of files. The designer should however be aware that trying to spawn a task on a node where the corresponding file does not exist, will cause a critical error, restarting the entire system (see section 2.8).

2.7 Creating new tasks

A task can spawn a task from any procedure on any node (given that the procedure has been transferred to that node, see section 2.6). This is done by the task setting its state to *SPAWN* (see section 2.2.1) after setting the needed arguments for spawning the new task. The arguments are which procedure to create the task from, what node to spawn the task on, what channels to use, and on which nodes the tasks using the other ends of those channels resides.

The runtime will send this information to the given node, which will spawn the new task. If the node to spawn the new task on is the same as the source task is running on, the runtime will of course spawn the new task itself.

2.8 Error handling

To simplify the system, every error is treated as a critical error and restarts the system. If a node encounters an error, it closes down its connections to its peers, and restarts the runtime. The peers will intrepid the closed connection as errors, and will in turn shut down their own connections and restart.

There is a delay on the restart to allow all connections to be closed, and all peers to go in error mode before the restart. This delay is shorter for the master node than the slaves. This makes it probable that the master is ready to accept incoming connections before the slaves restart.

[8] In this implementation, the operations ties up a worker for the entire duration of the transfer (in contrast to channel communication, which has its own dedicated set workers), and thereby slows down the program additionally. This is done to simplify the system, as the transferring of files is somewhat less interesting than the remaining scope of this project.

Chapter 3

Implementation

3.1 Runtime

The source code for the runtime is given in appendix C, and the electronic attachment (see appendix A). The description of its structure and behaviour is given in chapter 2. This section gives a short description of the modules/files the runtime is divided into.

3.1.1 Channel manager

The source code is given in appendix C.2 and C.3. This module is responsible for the channel communication.

It uses the network and PeerHash modules (see sections 2.3 and 2.4). It is used by the Task manager.

3.1.2 Function manager

The source code is given in appendix C.4 and C.5. This module is responsible for loading procedures from files and instantiating them to tasks (together with the task manager). It also stores files/procedures received from other nodes, and reads files to send to other nodes.

The module uses the task manager, and is used by the task manager and the network module.

3.1.3 Network and PeerHash

The source code is given in appendix C.8, C.9, C.10 and C.11. The network module is the largest module in the system (in terms of code lines). It is responsible for connecting to, and communicating with other nodes. The PeerHash module stores tasks blocked due to missing (not yet connected) peers, and retrieves peer information given the peers id/node id (see section 2.4.2).

These modules use, and/or are used by, all the other modules.

3.1.4 Task manager

The source code is given in appendix C.12 and C.13. The task manager manages the queue of the tasks that are ready to run, and holds a number of workers that execute the tasks from this queue (see section 2.2).

The module uses all the other modules, and is used by many of them.

3.1.5 Other

A main-function, given in C.1, reads the needed arguments for the system, and starts it. It also restarts the program in case of critical errors.

The system has one global variable (given by “Global.h” and “Global.c”, see section C.6 and C.7) used to coordinate the restart of the system in case of errors.

3.2 Prepared programs

3.2.1 Syntax and structure

This section describes the form the procedures to be run on GUSTAFSON must have, before being compiled/linked[1]. Each procedure must reside in its own file, with the general form shown in code 3.1. The argument to the procedure (*instanceStruct*, see code 3.2) contains fields needed to communicate with the runtime, remembering what part of the (re-entrant) procedure that currently is executed, and pointers to the memory used internally in the procedure and memory used by channel communication[2].

Code 3.1 General structure for the ready-to-compile procedure

```
void procedure_name(struct InstanceStruct *instance){
  switch(instance->step){
    /*...*/
  }
}
```

As shown in code 3.1, all code in the procedure is placed in a *switch*. When the procedure is given CPU-time, it runs the *case* given by argument *instance->step*. Before relinquishing the CPU, the procedure updates *instance->step* to the next case to run, typically incrementing it for sequentially code, or setting it to lower

[1]That is, the form when written in C. In a practical application, the procedures would probable not be translated to C, but rather an intermediate/assembly language. It is, however, more practical to present the form in C, and this form will of course also tell the seasoned compiler designer much about the form of the intermediate/assembly language.

[2]These memory areas may, or may not, overlap.

Code 3.2 The struct holding the needed data for each task. The task itself uses all fields except the first (*funStruct*) and the last two (*next/prev*).

```
struct InstanceStruct{
  struct FunStruct *funStruct;
  void *memPtr;
  int *chanTrans;
  int step;
  enum InstanceState state;
  void *comPtr;
  int comSize;
  int localCh;
  int nodeWait;

  struct InstanceStruct *next, *prev;
};
```

or higher values to implement branches and loops. A simple case is given by code 3.3.

It is also shown in code 3.3 how the procedure updates its state before it returns. The states are explained in section 2.2.1, but it is in this section shown the practical use. Apart from updating the *instance->step*, no additional information is needed for the procedure to pass to the runtime for the state READY. Most other states, however, needs additional information to be saved in the *instance* struct before returning.

The state NODEWAIT is illustrated in code 3.4. What node to wait for is given to the runtime (in *instance->nodeWait*).

Code 3.3 The typical *case* of the ready-to-compile procedure

```
case N:
  /*Do work*/
  instance->step = N + 1;
  instance->state = READY;
  return;
```

Code 3.4 Code for blocking the task until node 7 is connected

```
case N:
  instance->nodeWait = 7;
  instance->step = N + 1;
  instance->state = NODEWAIT;
  return;
```

The states TRANSFER and SPAWN has some similarities. In both cases a struct *SpawnStruct* (see 3.7) must be filled with needed information. The name of the procedure to be transferred or spawned must be supplied in both cases. SPAWN also needs information of the channels the procedure will use (see 2.3). Examples of cases for TRANSFER and SPAWN are given in codes 3.5 and 3.6.

Code 3.5 Code for transferring a procedure to another node. The allocation of memory may have been done already, in the initialisation, or a previous transfer. The *SpawnStruct* (see code 3.7) is filled with the data needed to transfer the procedure; the name of the procedure (line 5) and the node to transfer it to (node 9, line 6).

```
1  case N:
2    instance->comSize = sizeof(struct SpawnStruct);
3    instance->comPtr = malloc(instance->comSize);
4    ((struct SpawnStruct*)instance->comPtr)->name = malloc(strlen(
         "proc_name") + 1);
5    strcpy(((struct SpawnStruct*)instance->comPtr)->name,"
         proc_name");
6    ((struct SpawnStruct*)instance->comPtr)->peerId = 9;
7
8    instance->step = N + 1;
9    instance->state = TRANSFER;
10   return;
```

Cases for sending and receiving data over channels are given by code 3.8 and 3.9. In addition to pointers to the memory area to read from/send to, the procedure needs to supply the number of bytes to be sent/received.

Finally, a case for cleaning up after the procedure is given in code 3.10. All allocated memory is freed (it may be more than given in code 3.10) and the state is set to DONE.

3.2.2 Compilation

To prepare a procedure formatted as shown in section 3.2 to be run on the runtime GUSTAFSON, it should be compiled as a dynamic linkable library. Each procedure needs to reside in its own file, and the file must have the same name as the procedure. Code 3.11 shows the compilation in gcc.

Code 3.6 Code for spawning a new task on node 11. The allocation of memory may have been done already, in the initialisation or a previous transfer. The *SpawnStruct* (see code 3.7) is filled with the data needed to spawn a new task; the name of the procedure ("proc_name", line 5), the node to run it on (node 11, line 6), the number of channels (N_CHANNELS, line 7) and the data for each channel (CHAN_ID/PEER_ID, line 9-10).

```
1  case N:
2    instance->comSize = sizeof(struct SpawnStruct);
3    instance->comPtr = malloc(instance->comSize);
4    ((struct SpawnStruct*)instance->comPtr)->name = malloc(strlen(
         "proc_name") + 1);
5    strcpy(((struct SpawnStruct*)instance->comPtr)->name,"
         proc_name");
6    ((struct SpawnStruct*)instance->comPtr)->peerId = 11;
7    ((struct SpawnStruct*)instance->comPtr)->ctSize = 2 *
         N_CHANNELS * sizeof(int);
8    ((struct SpawnStruct*)instance->comPtr)->chanTrans = malloc(
         N_CHANNELS * sizeof(int));
9    ((struct SpawnStruct*)instance->comPtr)->chanTrans[0] =
         CHAN_ID;
10   ((struct SpawnStruct*)instance->comPtr)->chanTrans[1] =
         PEER_ID;
11   /*And so on for other channels*/
12
13   instance->step = N + 1;
14   instance->state = SPAWN;
15   return;
```

Code 3.7 Struct holding data for spawning tasks and transferring procedures

```
struct SpawnStruct{
  char *name;
  int *chanTrans;
  int ctSize;
  int peerId;
};
```

Code 3.8 Code for sending "Hello, world!" to another task. The id of the node the receiving task is running on and a global channel identifier in the array *chanTrans* (see code 3.2) based on the number CHAN_ID. "comPtr" will often be set to point to an existing memory area (an offset of instance->memPtr), rather than allocating a new memory area and copying data to it (line 3-4).

```
1 case N:
2   instance->comSize = 14;
3   instance->comPtr = malloc(14);
4   strcpy(intance->comPtr, "Hello, world!");
5   instance->localCh = CHAN_ID;
6
7   instance->step = N + 1;
8   instance->state = CHANW;
9   return;
```

Code 3.9 Code for receiving 14 bytes from another task. The received data is stored at the existing memory area pointed to by "instance->memPtr + 42" (line 3).

```
1 case N:
2   instance->comSize = 14;
3   instance->comPtr = instance->memPtr + 42;
4   instance->localCh = CHAN_ID;
5
6   instance->step = N + 1;
7   instance->state = CHANR;
8   return;
```

Code 3.10 Code for cleaning up when task is complete. Freeing of other memmory areas may be needed, depending on the procedure. The instance state is set to *DONE*, so that the runtime will delete the task.

```
1 case N:
2   free(instance->memPtr);
3   instance->step = 0;
4   instance->state = DONE;
5   return;
```

Code 3.11 Compilation of GUSTAFSON procedures

```
1 gcc -shared -nostartfiles -o procedure_name procedure_name.c -g
```

3.3 Running GUSTAFSON

This section briefly describes how to run GUSTAFSON in a multi-computer environment.

When starting an instance of GUSTAFSON on a node the instance must be set up as either a master or a slave (see section 2.4.1), and it may or may not be given a procedure to run. This totals to 4 different modes to run GUSTAFSON in.

Common for all modes is that the two first arguments should specify the unique id of the node and the local (tcp) port to listen for new connections on. If no other arguments are given, the instance starts as a master, with no procedure running on it initially. See code 3.12 for an example.

Code 3.12 Starting an instance of GUSTAFSON as a master with no procedure initially running. The node id is set to 2, and the instance listens to tcp port 1045 for incoming connections.

```
./runtime 2 1045
```

To start the instance as a slave the flag *-c* is given, followed by the ip address and tcp port of the master. See code 3.13 for an example.

Code 3.13 Starting an instance of GUSTAFSON as a slave with no procedure initially running. The node id is set to 3, and the instance listens to tcp port 1045 for incoming connections. The slave will connect to the node with ip address 10.0.0.1 on tcp port 1045.

```
./runtime 3 1045 -c 10.0.0.1 1045
```

To run a procedure on the instance the flag *-p* is given, followed by the procedure name. This extension can be added both to master and slave instances. See code 3.14 for an example.

Code 3.14 Starting an instance of GUSTAFSON as a slave with the procedure "myProcedure" initially running. In all other aspects, the instance is equal to the one given in code 3.13.

```
./runtime 3 1045 -c 10.0.0.1 1045 -p myProcedure
```

Chapter 4

Applications

The runtime presented so far in this report would of course have little practical use without a language to use with it. Although the full design and implementations of such languages and associated compilers fall outside the scope of this thesis, this chapter will present a rough outline of such languages.

In this chapter, two levels of abstraction that can be used when programming for GUSTAFSON is shown. The lower level of abstraction is to apply a simple language where it is still the programmer's responsibility to specify what part of the program that should be split in separate tasks and on what nodes to execute each task.

On the higher level of abstraction, a more standard type of language is applied. In this case the compiler will split the program in tasks and assign the tasks to different nodes, based on simple or complex analysis.

4.1 Low level abstraction

4.1.1 Example

Code 4.1 shows an example of a simple producer/consumer pair, written in a language suited to be converted to a program intended for GUSTAFSON. This language contains, in addition to the usual *if*, *while*, *procedures* and so on, syntax for:

- waiting for other peers to connect/be connected to - *WAITFOR <node id>*
- transferring files/functions to other nodes - *TRANSFER <procedure name> <node id>*
- reading from and writing to channels - *CHAN(<chan number>) ! var* and *CHAN(<chan number>) ? var*
- spawning tasks on other peers (or the same node) - *SPAWN <procedure name> <node id> <channel information>*

Code 4.1 An example of a simple producer/consumer pair

```
PROCEDURE f1
  FOR a = 1 TO 42
    CHAN(1) ! a
  END
  CHAN(2) ! a
END

PROCEDURE f2
  b = 0
  WHILE b != 42
    CHAN(1) ? b
    PRINT b
  END
  CHAN(2) ! b
END

PROCEDURE main
  WAITFOR 2
  WAITFOR 3
  TRANSFER f1 2
  TRANSFER f2 3
  SPAWN f1 2 (1:3, 2:1)
  SPAWN f2 3 (1:2, 3:1)

  CHAN(1) ? a
  CHAN(2) ? a

  PRINT "DONE!"
END
```

The syntax *WAITFOR* and *TRANSFER* should be relatively simple to understand; see sections 2.4.2 and 2.6 for descriptions of their functions.

The syntax for reading from and writing to channels are partly inspired by occam[2]; *!* and *?* is used to indicate writing and reading, respectively. *CHAN(...)* is used to indicate the channel to use. *Chan number* is an identification local to the current procedure. The runtime will translate this to a globally valid channel id.

SPAWN starts a new task. The first two arguments are the same as for *TRANSFER*; they indicate the procedure to create a task from and the node to run it on. The last argument is a translation from the local channel identification used in procedures, to globally valid channel id and peer id. The argument is on the form (<chan id>:<peer id> [,<chan id>:<peer id>]*) and contains a chan id/peer id pair for each channel used in the procedure.

In the example (code 4.1), lines 1 through 6 gives the producer (called *f1*). The producer produces the numbers from 1 through 42 and writes them to channel 1 (local id). Afterwards it writes 42 to channel 2.

The consumer (called *f2*) on lines 8 through 15 is similar; the consumer reads numbers from channel 1 and prints them. When the consumer reads the number 42, it exits, after writing 42 to channel 2.

Lines 17 through 29 gives the *main*. The main, designed to run on node 1, spawns a producer on node 2 and a consumer on node 3. Lines 18 and 19 instructs the main to wait to nodes 2 and 3 are connected. Then the procedure files for the producer and consumer are transferred to nodes 2 and 3 (lines 20-21). Lines 22 and 23 spawns the producer and consumer on the remote nodes, and sets up the channels. Channel 1 on the producer is tied to channel 1 on the consumer, and channel 2 on both the producer and consumer is tied to the main, to channels 1 and 2, respectively. Finally, the main listens to channels 1 and 2, to tell when the producer and consumer are finished, and prints "DONE" when they are.

The translated versions of main, f1 and f2 are given in the electronic attachment (main.c, f1.c and f2.c), and in the appendix, section B.2.

4.1.2 Application

Manually programming in this low level abstraction does not seem feasible, since the programmer is charged with the responsibility of managing and assigning tasks to nodes, and the set up and use of channels is somewhat complex.

However, consider the same example given in code 4.2 with a slightly higher lever of abstraction. In this example it is not the concern of the programmer to decide what tasks should run on what nodes, nor to manually check which nodes are ready; this responsibility is left to the compiler[1], or even the runtime. The programmer also uses variables for the channels, both for the actual communica-

[1] The compiler would of cause not check which nodes are ready, since this obviously must be done at runtime, but rather insert the code for checking if nodes are ready at the appropriate place.

Code 4.2 An example of a simple producer/consumer pair - modified

```
PROCEDURE f1(ch1, ch2)
  FOR a = 1 TO 42
    ch1 ! a
  END
  ch2 ! a
END

PROCEDURE f2(ch1, ch2)
  b = 0
  WHILE b != 42
    ch1 ? b
    PRINT b
  END
  ch2 ! b
END

PROCEDURE main
  CHAN cha, chb, chc
  SPAWN f1(cha, chb)
  SPAWN f2(cha, chc)

  chb ? a
  chc ? a

  PRINT "DONE!"
END
```

tion, and when setting up the tasks. This level of abstraction may be suited for actual use.

Without these modifications, however, this low level abstraction is still suited for an intermediate language.

4.2 High level abstraction

4.2.1 Example

As specifying the code for each task as an individual procedure is both time consuming and potentially greatly increase the number of code lines, a higher abstraction is desired.

Consider we want to encrypt a string with the hypothetical function *encrypt()*. Assume the unspecified method of encryption lets us split the string in several parts, encrypt them separately, and reassemble the encrypted strings, forming the same encrypted message as if we where to have encrypted the whole string in one piece. The task is in other words well suited for parallelisation.

Code 4.3 shows a simple program to encrypt two strings in parallel. The keyword *PAR* (loosely inspired by occam[2]) indicate that every statement between it and the associated *END* should be run in parallel. The compiler is left responsible for splitting the code into procedures and setting up the needed channels.

The same program is transposed to a lower level abstraction in code 4.4. The number of lines are approximately doubled (not counting blank lines), even when the *TRANSFER* and *WAITFOR* commands used in code 4.1 are omitted. The readability is also reduced, even in this simple example.

4.2.2 Application

A high level abstraction language like this would be suited for many applications. However, the ability to manually specify tasks is still in many cases useful, so the functionality of a high level abstraction like in code 4.3 should come in addition to the functionality shown in section 4.1.2.

Code 4.3 Simple example of distributing work to two nodes

```
FUNCTION main
  PAR
    a = encrypt("This string should be encrypted")
    b = encrypt("And so should this");
  END
  PRINT a + b
END
```

Code 4.4 Code 4.3 rewritten to a lower level abstraction

```
FUNCTION f1
  CHAN(1) ! encrypt("This string should be encrypted")
END

FUNCTION f2
  CHAN(1) ! encrypt("And so should this")
END


FUNCTION main
  SPAWN f1 2 (1:1)
  SPAWN f2 3 (2:1)

  CHAN(1) ? a
  CHAN(2) ? b
  PRINT a + b
END
```

Chapter 5

Benchmark

5.1 The benchmark program

This chapter presents a simple benchmark test. By using a simple (and inefficient) algorithm for factorising a number into its prime components, it is shown how a near ideal[1] task for parallelisation is executed on nine[2] nodes. The program, written in the "Low level GUSTAFSON language" given in section 4.1, is given in code 5.1. Three different implementations are used (given in appendix B.2), differing on how the *while-* and *for-loops* (lines 4 and 5 in code 5.1) are implemented.

The implementation referred to as "-O0" returns to the runtime for each iteration of the loops, clearly resulting in massive overhead as the inner loop (the *for-loop* on line 5) totally iterates approximately equal to the sum of the factors of number being factorised, which may be in the millions, and even billions for some numbers.

The "-O1" implementation does not return to the runtime for each *for-loop* iteration, but rather uses the *for-loop* directly. It still returns to the runtime for each iteration of the outer loop (the *while-loop* on line 4). This reduces the overhead from the "-O0" implementation.

Finally the "-O2" implementation returns to the runtime at neither loop. As the *while-loop* typically has few iterations, this should not have a large impact on performance compared to the "-O1" implementation.

A single code version of the program is given in 5.2 and is used as a reference.

[1]Ideal in the sense that it has one independent component for each node, and the components are of the same size.

[2]Eight nodes are doing the computations, while one node acts as a controller. Having a separate node as a controller is not necessary, but simplify the example.

Code 5.1 Inefficient factorisation program for benchmark tests

```
PROCEDURE work
  CHAN(1) ? number

  WHILE number != 1
    FOR factor = 2 TO number
      IF number MOD factor == 0
        PRINT factor
        number = number / factor
        BREAK
      END
    END
  END
  CHAN(1) ! number
END

PROCEDURE main
  WAITFOR 2
   ...
  WAITFOR 9

  SPAWN work 2 (1:1)
  SPAWN work 3 (2:1)
  SPAWN work 4 (3:1)
  SPAWN work 5 (4:1)
  SPAWN work 6 (5:1)
  SPAWN work 7 (6:1)
  SPAWN work 8 (7:1)
  SPAWN work 9 (8:1)

  //Example values, the actual values used differs
  CHAN(1) ! 70312316987348207
  CHAN(2) ! 85600508411905222549
   ...
  CHAN(8) ! 9223372036854775783

  CHAN(1) ? a
  CHAN(2) ? a
   ...
  CHAN(8) ? a

  PRINT "DONE"
END
```

Code 5.2 A single core C reference program for the benchmark tests

```c
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

int main(int argc, char **argv){
  if(argc < 2)
    return -1;
  long long unsigned int n = strtoull(argv[1], NULL, 10);
  if(n == ULLONG_MAX || n == 0)
    return -1;

  long long unsigned int f;
  while(n != 1){
    for(f = 2; f <= n; ++f){
      if(n % f == 0){
        printf("%llu\n", f);
        n = n / f;
        break;
      }
    }
  }

  return 0;
}
```

5.2 The benchmark setup

Nine identical machines were used in the benchmark tests. The reference program ran on one, while the GUSTAFSON program used one computer as a controller and an other eight to do the computations.

The GUSTAFSON program does eight times the work of the reference program (it factorises the same number eight times, once on each work node), meaning the computation time for it is directly comparable with the computation time of the reference program, provided we ignore the controller node (which is a reasonable assumption in this example, as the factorisation demands much more computational power than the controlling node).

The scripts used to execute the program (on several computers by ssh) is given in appendix B.2.

5.3 The benchmark results

Two different numbers is used in the bench mark, 15310972286449713778 with factors 2, 401, 991, 4801, 22159 and 181081, and 15310972286449713776 with factors 2, 2, 2, 2, 103, 1468189 and 903994019. The results are given in tables 5.1 and 5.2.

From the first example (table 5.1) it is shown that there is, as expected, a large improvement from the "-O0" to "-O1" and "-O2" implementations (the latter running about 2.75 times faster), but it is also shown that even the fast GUSTAFSON implementations are much slower than the reference program (which runs about 40 times faster). For a small computation this should be expected, as a number of (relatively slow) messages needs to be sent over the network.

From the second, much more computationally heavy, example (table 5.2) it is again shown a large improvement the "-O0" to "-O1" and "-O2" implementations (this time "-O1" is almost 50 times faster than "-O0"). More surprisingly is is shown a significant speed-up from "-O1" to "-O2". The reason for this eludes the author, but it is of little consequence for the conclusions of the benchmark. Finally, it is shown that the difference between the reference and the GUSTAFSON program is much smaller here, with the reference program running only twice as fast as the "-O1" implementation, meaning the the use of the GUSTAFSON program (running 8 times the calculations of the reference) has a considerate speed-up in this case. It should be noted that there is still a massive room for improvement, but tweaking the runtime for maximal efficiency is not considered within the scope of this thesis.

5.4 Summary

While this chapter has briefly demonstrated the plausibility and potential of the runtime GUSTAFSON, it should be noted that this benchmark on no expense pretends to cover all aspects of GUSTAFSON and its efficiency, nor gives a complete picture on when it may be beneficial to use GUSTAFSON.

Table 5.1: Results of benchmark tests factorising 15310972286449713778

Test no	Reference [ms]	-O0 [ms]	-O1 [ms]	-O2 [ms]
1	2	221	81	80
2	2	212	80	81
3	2	241	80	84
4	2	222	80	80
5	2	215	81	80
6	2	214	80	81
7	2	221	81	80
8	2	204	80	81

Table 5.2: Results of benchmark tests factorising 15310972286449713776

Test no	Reference [ms]	-O0 [ms]	-O1 [ms]	-O2 [ms]
1	7207	735499	15011	11430
2	7208	N/A	15269	11439
3	7207	N/A	15019	11431

Chapter 6

Discussion

6.1 Considerations for real-time applications

It is possible to argue that the most important aspect of a real-time application is *predictability*. So, is it possible to claim that GUSTAFSON is predictable?

As the system is experimental, in many aspects unfinished, and also largely untested, it will in all probability contain several bugs and faults, making it unpredictable. However, those errors lay outside the scope of this thesis, and may be ignored in this discussion.

The nodes of the system communicates with its peers over network, and the system may hence suffer from some of the inherit unpredictability of the the network. As the communication happens over TCP, it may be assumed that the received data is correct[1], and that lost packages is retransmitted. This leaves two issues; total loss of network and unpredictable transmission time.

Measures may be made to reduce the chance of loss of network, but it will be impossible to guarantee against failure. In this implementation, loss of network is considered a critical, unrecoverable error, much in the same way as loss of a node. If needed, the system could be extended to provide support for both redundant nodes and networks.

Not being able to reliable predict transmission times may pose a problem in critical real-time applications. Again, measures may be made to increase the quality of the network, and hence increase the predictability, but some level of uncertainty may be unavoidable.

The rest of the system should in most aspects be predictable, assuming it is possible to predict how the OS grants the systems resources, but it may be needed to tweak the number and priorities of workers.

[1] Assuming the system is not deliberately attacked. The security measures to prevent this is considered outside the scope of this thesis.

6.2 Efficiency

Although some high level considerations (like the choice of asynchronous channels over synchronous channels, see section 2.3) have been made, many aspects of making the runtime as efficient as possible have not been touched. It is likely that both minor adjustments to the code and larger redesigns could lead to a considerate better efficiency.

As it is briefly shown in chapter 5 it is necessary with some improvement to efficiency for GUSTAFSON to have a practical value. However, the same chapter show that considerate speed-up is already achievable for some tasks.

6.3 Further work

The previous sections of this chapter have already suggested several aspects of the system that would benefit from further development. This section will briefly touch a few more aspects.

6.3.1 Applications

As discussed in chapter 4, a language with a corresponding compiler is needed for GUSTAFSON to have any practical value.

The development of one or more such languages and compilers would form an interesting thesis on its own. A similar task (but somewhat simpler as it only consider one (multi-core) computer) is examined in [1].

6.3.2 Ease of use

Ease of use has not been within the scope of this project. There are several additions to the system that would improve usability, most notably:

- *GUI:* A simple, intuitive user graphical user interface could greatly improve the usability. Currently information is printed on the command line once, with no way of query for it.

- *Remote control:* Currently, remote control is only available in the sense of remote controlling the target computer (remote desktop, remote shell or similar). By integrating some remote control features into GUSTAFSON, combined with the previous point of a GUI, usability could be greatly increased. This is especially true when the nodes of the system is not placed in the same location.

6.3.3 Multi-platform usage

The implementation presented in this thesis is build on Linux/POSIX. By extending the system to work on multiple platforms can get the following advantages:

- *Increased computational power:* By including platforms now unavailable, it is possible to increase the computational power.
- *Increased availability:* Allowing the system to work on hand held devices, e.g. a smartphone, will greatly increase the users access to heavy computational power.
- *Utilize specialised platforms:* Some platforms may be specialised in solving particular tasks, e.g. doing matrix operation or digital signal processing. By dividing the program in tasks suited for running on different specialised platforms it is possible finish calculations faster, but also minimise the amount of resources uses, freeing computational power for other tasks.

6.3.4 Other Extensions

In any future work with applications and languages for GUSTAFSON, it would probably surface the need for additional support from the runtime. For instance, it might be of use for an application to receive information about the workload from the different nodes, for dynamically decide what node to run a task on.

Alternatively, built-in support in the runtime for scheduling, and even rescheduling, of task to nodes may be of interest.

Information of network bandwidth and round-trip delay between nodes may be of interest for both static and dynamic scheduling, as would the computational power, and number of cores, of the different nodes.

The ability to shut down one node without resetting all the other nodes would greatly improve the system. In addition, the system should be able to handle if a node goes down due to an error. The system would need to redistribute the work of the affected node to other nodes. Alternatively, redundant nodes doing the same work could be utilised, as briefly touched in section 6.1, but the system would still need to appoint new nodes to act as new backup nodes. As the number of nodes in the system grows, this extension would grown more important, as the chance of an error would grow as well.

An other issue when the size of the system grows, is the way the nodes are connected. Currently, all the nodes communicates with all the nodes, making the total number of connections grow exponentiation with the number of nodes, generating a lot of unnecessary traffic on the network. By letting a few larger nodes acts as relays, it is possible to greatly reduce the number of connections. This would of course increase the transmission time between some of the nodes, but care could be taken in the design of the network and scheduling of tasks to minimize the problems arising from this.

Chapter 7

Conclusion

> Is it possible to utilise the computational power of a multi-computer environment for real-time applications by developing an experimental runtime system and exploring its applications?

The findings presented in this thesis have shown how it is possible to solve the problem presented above. In chapter 6 several alterations and extensions that are needed before the system has a practical use are discussed, but as an experimental system, GUSTAFSON is suited to prove the plausibility of solving the proposed problem.

In chapter 5 it is shown how it is still room to make the system more efficient, but also that GUSTAFSON can lead to considerate speed-up of suitable tasks. It should, however, be noted that the benchmark test performed covered too few aspects to fully conclude anything about the efficiency of the concept. More benchmark test, covering different patterns of task-to-task communications, must be performed to fully explore the potential of the system.

A brief outline for suitable languages for the system has been proposed, and it would seem a plausible task to further develop a language bases on one or more of these and write a compiler for it to use with GUSTAFSON.

Bibliography

[1] Knut André Karlsen Vestergren. Cb - en utvidelse av c for enkel parallellisering og samtidighet i et flerkjernemiljø. Specialisation thesis, NTNU, 2011.

[2] Wikipedia. occam. http://en.wikipedia.org/wiki/Occam_(programming_language), 2001-2011. [Read 23.01.12].

[3] Wikipedia. Backronym. http://en.wikipedia.org/wiki/Backronym, 2011-2012. [Read 23.05.12].

Appendix A

Electronic attachment

The electronic attachment should contain the following folders and files:

- *Runtime* - Folder containing the source code for the runtime.
- *Examples* - Folder containing the example code from various examples in this report.
- *Bin* - Destination folder for the compiled code (both runtime and examples). This folder is initially empty, except for the folder *dlibs*, which is the target folder for the compiled examples.
- *Tmp* - Folder for temporary files, initially empty.
- *Makefile* - Makefile for compiling both the runtime and the (executable) example files.

Appendix B

Translated programs

This chapter holds the source code of the example programs from chapter 4, translated to C-code ready to be compiled and run on GUSTAFSON.

Font size is reduced. See the electronic appendix for a more detailed study.

B.1 Simple low level abstraction

B.1.1 simple.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "TaskManager.h"

void simple(struct InstanceStruct *instance){
  switch(instance->step){
    case 0:
      instance->memPtr = malloc(4);
      instance->chanTrans = malloc(4 * sizeof(int));
      instance->chanTrans[0] = 2;
      instance->chanTrans[1] = 2;
      instance->chanTrans[2] = 3;
      instance->chanTrans[3] = 3;
      instance->step = 1;
      instance->state = READY;
      return;
    case 1:
      instance->nodeWait = 2;
      instance->step = 2;
      instance->state = NODEWAIT;
      return;
    case 2:
      instance->nodeWait = 3;
      instance->step = 3;
      instance->state = NODEWAIT;
      return;
    case 3:
      instance->comSize = sizeof(struct SpawnStruct);
      instance->comPtr = malloc(instance->comSize);
      ((struct SpawnStruct*)instance->comPtr)->name = malloc(3);
      strcpy(((struct SpawnStruct*)instance->comPtr)->name,"f1");
      ((struct SpawnStruct*)instance->comPtr)->peerId = 2;
      instance->step = 4;
      instance->state = TRANSFER;
      return;
    case 4:
      strcpy(((struct SpawnStruct*)instance->comPtr)->name,"f2");
      ((struct SpawnStruct*)instance->comPtr)->peerId = 3;
      instance->step = 5;
      instance->state = TRANSFER;
      return;
    case 5:
      strcpy(((struct SpawnStruct*)instance->comPtr)->name,"f1");
      ((struct SpawnStruct*)instance->comPtr)->peerId = 2;
      ((struct SpawnStruct*)instance->comPtr)->ctSize = 4 * sizeof(int);
      ((struct SpawnStruct*)instance->comPtr)->chanTrans = malloc(4 * sizeof(int));
      ((struct SpawnStruct*)instance->comPtr)->chanTrans[0] = 1;
      ((struct SpawnStruct*)instance->comPtr)->chanTrans[1] = 3;
      ((struct SpawnStruct*)instance->comPtr)->chanTrans[2] = 2;
      ((struct SpawnStruct*)instance->comPtr)->chanTrans[3] = 1;
      instance->step = 6;
      instance->state = SPAWN;
      return;
    case 6:
      strcpy(((struct SpawnStruct*)instance->comPtr)->name,"f2");
      ((struct SpawnStruct*)instance->comPtr)->peerId = 3;
      ((struct SpawnStruct*)instance->comPtr)->chanTrans[0] = 1;
      ((struct SpawnStruct*)instance->comPtr)->chanTrans[1] = 2;
      ((struct SpawnStruct*)instance->comPtr)->chanTrans[2] = 3;
      ((struct SpawnStruct*)instance->comPtr)->chanTrans[3] = 1;
      instance->step = 7;
      instance->state = SPAWN;
      return;
    case 7:
      free(((struct SpawnStruct*)instance->comPtr)->name);
      free(((struct SpawnStruct*)instance->comPtr)->chanTrans);
      free(instance->comPtr);
      instance->localCh = 0;
      instance->comPtr = instance->memPtr;
      instance->comSize = 4;
      instance->step = 8;
      instance->state = CHANR;
      return;
    case 8:
```

```
        instance->localCh = 2;
        instance->comPtr = instance->memPtr;
        instance->comSize = 4;
        instance->step = 9;
        instance->state = CHANR;
        return;
      case 9:
        printf("DONE!\n");
        instance->state = READY;
        instance->step = 10;
        return;
      case 10:
        free(instance->memPtr);
        instance->memPtr = NULL;
        instance->state = DONE;
        instance->step = 0;
        return;
    }
  }
```

B.1.2 f1.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "TaskManager.h"

void f1(struct InstanceStruct *instance){
  switch(instance->step){
    case 0:
      instance->memPtr = malloc(4);
      *(int*)instance->memPtr = 0;
      instance->step = 1;
      instance->state = READY;
      return;
    case 1:
      instance->localCh = 0;
      instance->comPtr = instance->memPtr;
      instance->comSize = 4;
      (*(int*)instance->memPtr)++;
      instance->step = 2;
      instance->state = CHANW;
      return;
    case 2:
      if(*(int*)instance->memPtr == 42)
        instance->step = 3;
      else
        instance->step = 1;
      instance->state = READY;
      return;
    case 3:
      instance->localCh = 2;
      instance->comPtr = instance->memPtr;
      instance->comSize = 4;
      instance->step = 4;
      instance->state = CHANW;
      return;
    case 4:
      free(instance->memPtr);
      instance->memPtr = NULL;
      instance->state = DONE;
      instance->step = 0;
      return;
  }
}
```

B.1.3 f2.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "TaskManager.h"

void f2(struct InstanceStruct *instance){
  switch(instance->step){
    case 0:
      instance->memPtr = malloc(4);
      instance->step = 1;
      instance->state = READY;
      return;
    case 1:
      instance->localCh = 0;
      instance->comPtr = instance->memPtr;
      instance->comSize = 4;
      instance->step = 2;
      instance->state = CHANR;
      return;
    case 2:
      printf("%d\n", *(int*)instance->memPtr);
      if(*(int*)instance->memPtr == 42)
        instance->step = 3;
      else
        instance->step = 1;
      instance->state = READY;
      return;
    case 3:
      instance->localCh = 2;
      instance->comPtr = instance->memPtr;
      instance->comSize = 4;
      instance->step = 4;
      instance->state = CHANW;
      return;
    case 4:
      free(instance->memPtr);
      instance->memPtr = NULL;
      instance->state = DONE;
      instance->step = 0;
      return;
  }
}
```

B.2 Factorisation

B.2.1 factorisation.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

#include "TaskManager.h"

#define PREFIX "FACTORISATION: "

void factorisation(struct InstanceStruct *instance){
  switch(instance->step){
    case 0:
      instance->memPtr = malloc(12);

      int tmp = open("number.txt", O_RDONLY);
      char *tmpb = malloc(24);
      read(tmp, tmpb, 24);
      close(tmp);
      tmpb[23] = 0;
      *(unsigned long long *)(instance->memPtr + 4) = strtoull(tmpb, NULL, 10);
      free(tmpb);

      *(int *)(instance->memPtr) = 1;
      instance->chanTrans = malloc(16 * sizeof(int));
      instance->chanTrans[0] = 2;
      instance->chanTrans[1] = 2;
      instance->chanTrans[2] = 3;
      instance->chanTrans[3] = 3;
      instance->chanTrans[4] = 4;
      instance->chanTrans[5] = 4;
      instance->chanTrans[6] = 5;
      instance->chanTrans[7] = 5;
      instance->chanTrans[8] = 6;
      instance->chanTrans[9] = 6;
      instance->chanTrans[10] = 7;
      instance->chanTrans[11] = 7;
      instance->chanTrans[12] = 8;
      instance->chanTrans[13] = 8;
      instance->chanTrans[14] = 9;
      instance->chanTrans[15] = 9;
      instance->step = 1;
      instance->state = READY;
      return;
    case 1:
      *(int *)(instance->memPtr) += 1;
      if(*(int *)(instance->memPtr) < 10){
        instance->nodeWait = *(int *)(instance->memPtr);
        instance->step = 1;
        instance->state = NODEWAIT;
        printf(PREFIX"NODEWAIT %d\n", instance->nodeWait);
      }
      else{
        instance->step = 2;
        instance->state = READY;
      }
      return;
    case 2:
      instance->comSize = sizeof(struct SpawnStruct);
      instance->comPtr = malloc(instance->comSize);
      ((struct SpawnStruct*)instance->comPtr)->name = malloc(5);
      strcpy(((struct SpawnStruct*)instance->comPtr)->name,"work");
      ((struct SpawnStruct*)instance->comPtr)->peerId = 1;
      ((struct SpawnStruct*)instance->comPtr)->ctSize = 2 * sizeof(int);
      ((struct SpawnStruct*)instance->comPtr)->chanTrans = malloc(2 * sizeof(int));
      ((struct SpawnStruct*)instance->comPtr)->chanTrans[0] = 1;
      ((struct SpawnStruct*)instance->comPtr)->chanTrans[1] = 1;
      instance->step = 3;
      instance->state = READY;
      return;
    case 3:
      ((struct SpawnStruct*)instance->comPtr)->peerId += 1;
      if(((struct SpawnStruct*)instance->comPtr)->peerId < 10){
        ((struct SpawnStruct*)instance->comPtr)->chanTrans[0] = ((struct SpawnStruct*)
            instance->comPtr)->peerId;
```

```
        instance->step = 3;
        instance->state = SPAWN;
        printf(PREFIX"SPAWN %d\n", ((struct SpawnStruct*)instance->comPtr)->peerId);
      }
      else{
        free(((struct SpawnStruct*)instance->comPtr)->name);
        free(((struct SpawnStruct*)instance->comPtr)->chanTrans);
        free(instance->comPtr);
        instance->localCh = -2;
        instance->comSize = 8;
        instance->step = 4;
        instance->state = READY;
      }
      return;
    case 4:
      instance->localCh += 2;
      if(instance->localCh < 15){
        instance->comSize = 8;
        instance->comPtr = instance->memPtr + 4;
        instance->step = 4;
        instance->state = CHANW;
        printf(PREFIX"SEND %llu TO %d:%d\n", *(unsigned long long *)(instance->memPtr + 4), instance->chanTrans[instance->localCh], instance->chanTrans[instance->localCh + 1]);
      }
      else{
        instance->localCh = -2;
        instance->step = 5;
        instance->state = READY;
      }
      return;
    case 5:
      instance->localCh += 2;
      if(instance->localCh < 15){
        instance->comPtr = instance->memPtr + 4;
        instance->comSize = 8;
        instance->step = 5;
        instance->state = CHANR;
      }
      else{
        instance->step = 6;
        instance->state = READY;
      }
      return;
    case 6:
      printf(PREFIX"DONE\n");
      exit(0);
      free(instance->memPtr);
      instance->step = 0;
      instance->state = DONE;
      return;
  }
}
```

B.2.2 factorisation_o1.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

#include "TaskManager.h"

#define PREFIX "FACTORISATION: "

void factorisation_o1(struct InstanceStruct *instance){
  switch(instance->step){
    case 0:
      instance->memPtr = malloc(12);

      int tmp = open("number.txt", O_RDONLY);
      char *tmpb = malloc(24);
      read(tmp, tmpb, 24);
      close(tmp);
      tmpb[23] = 0;
      *(unsigned long long *)(instance->memPtr + 4) = strtoull(tmpb, NULL, 10);
      free(tmpb);

      *(int *)(instance->memPtr) = 1;
      instance->chanTrans = malloc(16 * sizeof(int));
      instance->chanTrans[0] = 2;
      instance->chanTrans[1] = 2;
      instance->chanTrans[2] = 3;
      instance->chanTrans[3] = 3;
      instance->chanTrans[4] = 4;
      instance->chanTrans[5] = 4;
      instance->chanTrans[6] = 5;
      instance->chanTrans[7] = 5;
      instance->chanTrans[8] = 6;
      instance->chanTrans[9] = 6;
      instance->chanTrans[10] = 7;
      instance->chanTrans[11] = 7;
      instance->chanTrans[12] = 8;
      instance->chanTrans[13] = 8;
      instance->chanTrans[14] = 9;
      instance->chanTrans[15] = 9;
      instance->step = 1;
      instance->state = READY;
      return;
    case 1:
      *(int *)(instance->memPtr) += 1;
      if(*(int *)(instance->memPtr) < 10){
        instance->nodeWait = *(int *)(instance->memPtr);
        instance->step = 1;
        instance->state = NODEWAIT;
        printf(PREFIX"NODEWAIT %d\n", instance->nodeWait);
      }
      else{
        instance->step = 2;
        instance->state = READY;
      }
      return;
    case 2:
      instance->comSize = sizeof(struct SpawnStruct);
      instance->comPtr = malloc(instance->comSize);
      ((struct SpawnStruct*)instance->comPtr)->name = malloc(5);
      strcpy(((struct SpawnStruct*)instance->comPtr)->name,"work_o1");
      ((struct SpawnStruct*)instance->comPtr)->peerId = 1;
      ((struct SpawnStruct*)instance->comPtr)->ctSize = 2 * sizeof(int);
      ((struct SpawnStruct*)instance->comPtr)->chanTrans = malloc(2 * sizeof(int));
      ((struct SpawnStruct*)instance->comPtr)->chanTrans[0] = 1;
      ((struct SpawnStruct*)instance->comPtr)->chanTrans[1] = 1;
      instance->step = 3;
      instance->state = READY;
      return;
    case 3:
      ((struct SpawnStruct*)instance->comPtr)->peerId += 1;
      if(((struct SpawnStruct*)instance->comPtr)->peerId < 10){
        ((struct SpawnStruct*)instance->comPtr)->chanTrans[0] = ((struct SpawnStruct*)
            instance->comPtr)->peerId;
        instance->step = 3;
        instance->state = SPAWN;
        printf(PREFIX"SPAWN %d\n", ((struct SpawnStruct*)instance->comPtr)->peerId);
      }
```

```
      else{
        free(((struct SpawnStruct*)instance->comPtr)->name);
        free(((struct SpawnStruct*)instance->comPtr)->chanTrans);
        free(instance->comPtr);
        instance->localCh = -2;
        instance->comSize = 8;
        instance->step = 4;
        instance->state = READY;
      }
      return;
    case 4:
      instance->localCh += 2;
      if(instance->localCh < 15){
        instance->comSize = 8;
        instance->comPtr = instance->memPtr + 4;
        instance->step = 4;
        instance->state = CHANW;
        printf(PREFIX"SEND %llu TO %d:%d\n", *(unsigned long long *)(instance->memPtr + 4), instance->chanTrans[instance->localCh], instance->chanTrans[instance->localCh + 1]);
      }
      else{
        instance->localCh = -2;
        instance->step = 5;
        instance->state = READY;
      }
      return;
    case 5:
      instance->localCh += 2;
      if(instance->localCh < 15){
        instance->comPtr = instance->memPtr + 4;
        instance->comSize = 8;
        instance->step = 5;
        instance->state = CHANR;
      }
      else{
        instance->step = 6;
        instance->state = READY;
      }
      return;
    case 6:
      printf(PREFIX"DONE\n");
      exit(0);
      free(instance->memPtr);
      instance->step = 0;
      instance->state = DONE;
      return;
  }
}
```

B.2.3 factorisation_o2.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

#include "TaskManager.h"

#define PREFIX "FACTORISATION: "

void factorisation_o2(struct InstanceStruct *instance){
  switch(instance->step){
    case 0:
      instance->memPtr = malloc(12);

      int tmp = open("number.txt", O_RDONLY);
      char *tmpb = malloc(24);
      read(tmp, tmpb, 24);
      close(tmp);
      tmpb[23] = 0;
      *(unsigned long long *)(instance->memPtr + 4) = strtoull(tmpb, NULL, 10);
      free(tmpb);

      *(int *)(instance->memPtr) = 1;
      instance->chanTrans = malloc(16 * sizeof(int));
      instance->chanTrans[0] = 2;
      instance->chanTrans[1] = 2;
      instance->chanTrans[2] = 3;
      instance->chanTrans[3] = 3;
      instance->chanTrans[4] = 4;
      instance->chanTrans[5] = 4;
      instance->chanTrans[6] = 5;
      instance->chanTrans[7] = 5;
      instance->chanTrans[8] = 6;
      instance->chanTrans[9] = 6;
      instance->chanTrans[10] = 7;
      instance->chanTrans[11] = 7;
      instance->chanTrans[12] = 8;
      instance->chanTrans[13] = 8;
      instance->chanTrans[14] = 9;
      instance->chanTrans[15] = 9;
      instance->step = 1;
      instance->state = READY;
      return;
    case 1:
      *(int *)(instance->memPtr) += 1;
      if(*(int *)(instance->memPtr) < 10){
        instance->nodeWait = *(int *)(instance->memPtr);
        instance->step = 1;
        instance->state = NODEWAIT;
        printf(PREFIX"NODEWAIT %d\n", instance->nodeWait);
      }
      else{
        instance->step = 2;
        instance->state = READY;
      }
      return;
    case 2:
      instance->comSize = sizeof(struct SpawnStruct);
      instance->comPtr = malloc(instance->comSize);
      ((struct SpawnStruct*)instance->comPtr)->name = malloc(5);
      strcpy(((struct SpawnStruct*)instance->comPtr)->name,"work_o2");
      ((struct SpawnStruct*)instance->comPtr)->peerId = 1;
      ((struct SpawnStruct*)instance->comPtr)->ctSize = 2 * sizeof(int);
      ((struct SpawnStruct*)instance->comPtr)->chanTrans = malloc(2 * sizeof(int));
      ((struct SpawnStruct*)instance->comPtr)->chanTrans[0] = 1;
      ((struct SpawnStruct*)instance->comPtr)->chanTrans[1] = 1;
      instance->step = 3;
      instance->state = READY;
      return;
    case 3:
      ((struct SpawnStruct*)instance->comPtr)->peerId += 1;
      if(((struct SpawnStruct*)instance->comPtr)->peerId < 10){
        ((struct SpawnStruct*)instance->comPtr)->chanTrans[0] = ((struct SpawnStruct*)
            instance->comPtr)->peerId;
        instance->step = 3;
        instance->state = SPAWN;
        printf(PREFIX"SPAWN %d\n", ((struct SpawnStruct*)instance->comPtr)->peerId);
      }
```

```
      else{
        free(((struct SpawnStruct*)instance->comPtr)->name);
        free(((struct SpawnStruct*)instance->comPtr)->chanTrans);
        free(instance->comPtr);
        instance->localCh = -2;
        instance->comSize = 8;
        instance->step = 4;
        instance->state = READY;
      }
      return;
    case 4:
      instance->localCh += 2;
      if(instance->localCh < 15){
        instance->comSize = 8;
        instance->comPtr = instance->memPtr + 4;
        instance->step = 4;
        instance->state = CHANW;
        printf(PREFIX"SEND %llu TO %d:%d\n", *(unsigned long long *)(instance->memPtr + 4), instance->chanTrans[instance->localCh], instance->chanTrans[instance->localCh + 1]);
      }
      else{
        instance->localCh = -2;
        instance->step = 5;
        instance->state = READY;
      }
      return;
    case 5:
      instance->localCh += 2;
      if(instance->localCh < 15){
        instance->comPtr = instance->memPtr + 4;
        instance->comSize = 8;
        instance->step = 5;
        instance->state = CHANR;
      }
      else{
        instance->step = 6;
        instance->state = READY;
      }
      return;
    case 6:
      printf(PREFIX"DONE\n");
      exit(0);
      free(instance->memPtr);
      instance->step = 0;
      instance->state = DONE;
      return;
  }
}
```

B.2.4 work.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "TaskManager.h"

#define PREFIX "WORK: "

void work(struct InstanceStruct *instance){
  switch(instance->step){
    case 0:
      instance->memPtr = malloc(16);
      instance->step = 1;
      instance->state = READY;
      return;
    case 1:
      instance->localCh = 0;
      instance->comPtr = instance->memPtr;
      instance->comSize = 8;
      instance->step = 2;
      instance->state = CHANR;
      return;
    case 2:
      *(unsigned long long*)(instance->memPtr + 8) = 2;
      if(*(unsigned long long*)(instance->memPtr) == 1)
        instance->step = 4;
      else
        instance->step = 3;
      instance->state = READY;
      return;
    case 3:
      if(*(unsigned long long*)(instance->memPtr) % *(unsigned long long*)(instance->
          memPtr + 8) == 0){
        *(unsigned long long*)(instance->memPtr) /= *(unsigned long long*)(instance->
            memPtr + 8);
        printf(PREFIX"Factor: %llu (%llu left)\n", *(unsigned long long*)(instance->
            memPtr + 8), *(unsigned long long*)(instance->memPtr));
        instance->step = 2;
      }
      else{
        (*(unsigned long long*)(instance->memPtr + 8))++;
        instance->step = 3;
      }
      instance->state = READY;
      return;
    case 4:
      instance->localCh = 0;
      instance->comPtr = instance->memPtr;
      instance->comSize = 8;
      instance->step = 5;
      instance->state = CHANW;
      return;
    case 5:
      free(instance->memPtr);
      instance->step = 0;
      instance->state = DONE;
      break;
  }
}
```

B.2.5 work_o1.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "TaskManager.h"

#define PREFIX "WORK: "

void work_o1(struct InstanceStruct *instance){
  switch(instance->step){
    case 0:
      instance->memPtr = malloc(24);
      instance->step = 1;
      instance->state = READY;
      return;
    case 1:
      instance->localCh = 0;
      instance->comPtr = instance->memPtr;
      instance->comSize = 8;
      instance->step = 2;
      instance->state = CHANR;
      return;
    case 2:
      *(unsigned long long*)(instance->memPtr + 8) = 2;
      if(*(unsigned long long*)(instance->memPtr) == 1)
        instance->step = 4;
      else
        instance->step = 3;
      instance->state = READY;
      return;
    case 3:
      instance->step = 3;
      for(*(unsigned long long*)(instance->memPtr + 16) = *(unsigned long long*)(
            instance->memPtr + 8);
        *(unsigned long long*)(instance->memPtr + 16) + 1000 > *(unsigned long long*)(
              instance->memPtr + 8);
        (*(unsigned long long*)(instance->memPtr + 8))++){
          if(*(unsigned long long*)(instance->memPtr) % *(unsigned long long*)(instance
                ->memPtr + 8) == 0){
            *(unsigned long long*)(instance->memPtr) /= *(unsigned long long*)(instance
                  ->memPtr + 8);
            printf(PREFIX"Factor: %llu (%llu left)\n", *(unsigned long long*)(instance
                  ->memPtr + 8), *(unsigned long long*)(instance->memPtr));
            instance->step = 2;
            break;
          }
      }
      instance->state = READY;
      return;
    case 4:
      instance->localCh = 0;
      instance->comPtr = instance->memPtr;
      instance->comSize = 8;
      instance->step = 5;
      instance->state = CHANW;
      return;
    case 5:
      free(instance->memPtr);
      instance->step = 0;
      instance->state = DONE;
      break;
  }
}
```

B.2.6 work_o2.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "TaskManager.h"

#define PREFIX "WORK: "

void work_o2(struct InstanceStruct *instance){
  switch(instance->step){
    case 0:
      instance->memPtr = malloc(16);
      instance->step = 1;
      instance->state = READY;
      return;
    case 1:
      instance->localCh = 0;
      instance->comPtr = instance->memPtr;
      instance->comSize = 8;
      instance->step = 2;
      instance->state = CHANR;
      return;
    case 2:
      *(unsigned long long*)(instance->memPtr + 8) = 2;
      if(*(unsigned long long*)(instance->memPtr) == 1)
        instance->step = 4;
      else
        instance->step = 3;
      instance->state = READY;
      return;
    case 3:
      while(*(unsigned long long*)(instance->memPtr) % *(unsigned long long*)(instance
          ->memPtr + 8) != 0)
        (*(unsigned long long*)(instance->memPtr + 8))++;
      *(unsigned long long*)(instance->memPtr) /= *(unsigned long long*)(instance->
          memPtr + 8);
      printf(PREFIX"Factor: %llu (%llu left)\n", *(unsigned long long*)(instance->
          memPtr + 8), *(unsigned long long*)(instance->memPtr));
      instance->step = 2;
      instance->state = READY;
      return;
    case 4:
      instance->localCh = 0;
      instance->comPtr = instance->memPtr;
      instance->comSize = 8;
      instance->step = 5;
      instance->state = CHANW;
      return;
    case 5:
      free(instance->memPtr);
      instance->step = 0;
      instance->state = DONE;
      break;
  }
}
```

B.2.7 Benchmark script - copy

```bash
#!/bin/bash

IP[2]="129.241.187.142"
IP[3]="129.241.187.144"
IP[4]="129.241.187.145"
IP[5]="129.241.187.148"
IP[6]="129.241.187.151"
IP[7]="129.241.187.152"
IP[8]="129.241.187.155"
IP[9]="129.241.187.157"


for ID in 2 3 4 5 6 7 8 9
do
  ssh ${IP[ID]} "rm ~/GUSTAFSON/ -rf; mkdir ~/GUSTAFSON"
  scp node.zip ${IP[ID]}:~/GUSTAFSON
  ssh ${IP[ID]} "cd GUSTAFSON; unzip node.zip"
done
```

B.2.8 Benchmark script - generate

```bash
#!/bin/bash

IP[2]="129.241.187.142"
IP[3]="129.241.187.144"
IP[4]="129.241.187.145"
IP[5]="129.241.187.148"
IP[6]="129.241.187.151"
IP[7]="129.241.187.152"
IP[8]="129.241.187.155"
IP[9]="129.241.187.157"

RUNSTR="gnome-terminal --tab -t h2 --command=\"ssh -t ${IP[2]} 'cd ~/GUSTAFSON; ./runtime 2 10002'\""
for ID in 3 4 5 6 7 8 9
do
  RUNSTR="$RUNSTR --tab -t h$ID --command=\"ssh -t ${IP[ID]} 'sleep ${ID};cd ~/GUSTAFSON; ./runtime ${ID} 1000${ID} -c ${IP[2]} 10002'\""
done
echo "#!/bin/bash" > ex.sh
echo $RUNSTR >> ex.sh
```

B.2.9 Benchmark script - execution 1

```bash
#!/bin/bash
gnome-terminal --tab -t h2 --command="ssh -t 129.241.187.142 'cd ~/GUSTAFSON; ./runtime 2 10002'" --tab -t h3 --command="ssh -t 129.241.187.144 'sleep 3;cd ~/GUSTAFSON; ./runtime 3 10003 -c 129.241.187.142 10002'" --tab -t h4 --command="ssh -t 129.241.187.145 'sleep 4;cd ~/GUSTAFSON; ./runtime 4 10004 -c 129.241.187.142 10002'" --tab -t h5 --command="ssh -t 129.241.187.148 'sleep 5;cd ~/GUSTAFSON; ./runtime 5 10005 -c 129.241.187.142 10002'" --tab -t h6 --command="ssh -t 129.241.187.151 'sleep 6;cd ~/GUSTAFSON; ./runtime 6 10006 -c 129.241.187.142 10002'" --tab -t h7 --command="ssh -t 129.241.187.152 'sleep 7;cd ~/GUSTAFSON; ./runtime 7 10007 -c 129.241.187.142 10002'" --tab -t h8 --command="ssh -t 129.241.187.155 'sleep 8;cd ~/GUSTAFSON; ./runtime 8 10008 -c 129.241.187.142 10002'" --tab -t h9 --command="ssh -t 129.241.187.157 'sleep 9;cd ~/GUSTAFSON; ./runtime 9 10009 -c 129.241.187.142 10002'"
```

B.2.10 Benchmark script - execution 2

```bash
#!/bin/bash

time ./runtime 1 10001 -c 129.241.187.142 10002 -r factorisation # or factorisations_o1 / factorisation_o2
```

Appendix C

Runtime Source Code

This chapter holds the source code of the GUSTAFSON runtime for quick reference.

Font size is reduced. See the electronic appendix for a more detailed study.

C.1 main.c

```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>

#include "Global.h"
#include "Network.h"
#include "FunctionManager.h"
#include "TaskManager.h"
#include "PeerHash.h"
#include "ChanManager.h"


int main(int argc, char **argv){
  if(argc < 3){
    fprintf(stderr,ERRFIX"Not enough arguments!\n Usage: %s <id> <port> [-c <ip> <port>] [-r <function>]\n", argv[0]);
    return -1;
  }
  char *tmp1, *tmp2;
  int id = strtol(argv[1], &tmp1, 10);
  int port = strtol(argv[2], &tmp2, 10);
  if(*tmp1 != 0 || *tmp2 != 0){
    fprintf(stderr,ERRFIX"Id and port must be a numbers!\n Usage: %s <id> <port> [-c <ip> <port>] [-r <function>]\n", argv[0]);
    return -1;
  }
  if(port < 1025 || port > 65535){
    fprintf(stderr,ERRFIX"Port out of range! Valid range is 1025-65535\n Usage: %s <id> <port> [-c <ip> <port>] [-r <function>]\n", argv[0]);
    return -1;
  }
  if(id < 0 || id > 9999){
    fprintf(stderr,ERRFIX"Id out of range! Valid range is 0-9999\n Usage: %s <id> <port> [-c <ip> <port>] [-r <function>]\n", argv[0]);
    return -1;
  }


  if(tm_init(4)) //TODO dynamic set number of workers?
    goto errorLbl;
  if(fm_init())
    goto errorLbl;
  if(ph_init())
    goto errorLbl;
  if(ch_init(4)) //TODO dynamic set number of workers?
    goto errorLbl;


  char *mip = NULL;
  char *mpt = NULL;
  if(argc > 5 && strcmp(argv[3], "-c") == 0){
    mip = argv[4];
    mpt = argv[5];
  }

  int e = nw_init(id, port, mip, mpt);
  if(e == -2)
    goto panicLbl;
  if(e)
    goto errorLbl;

  char *prog = NULL;
  if(argc > 4 && strcmp(argv[3], "-r") == 0)
    prog = argv[4];
  else if(argc > 7 && strcmp(argv[6], "-r") == 0)
    prog = argv[7];

  if(prog != NULL){
    if(fm_loadFunction(prog) || fm_createInstance(prog, NULL))
      fprintf(stderr,ERRFIX"Could not run program!\n");
    else
      printf(PREFIX"Running program: %s\n", prog);
  }

  printf(PREFIX"Running\n");

  while(masterSwitch)
    sleep(1);
```

```
panicLbl:
  nw_panic();
errorLbl:
  sleep((argc > 4 ? 5 : 3)); //Sleep shorter if "master"
  printf(PREFIX"Restarting\n");
  char **argvv = malloc(sizeof(char**) * (argc + 1));
  int i;
  for(i = 0; i < argc; ++i)
    argvv[i] = argv[i];
  argvv[argc] = NULL;

  execve(argv[0], argvv, NULL);
  perror("Execve");

  return -1;
}
```

C.2 ChanManager.h

```
#ifndef _CHAN_MANAGER_H
#define _CHAN_MANAGER_H

#include <pthread.h>

#define NCHANBUCKETS 256
#define BUFFERSIZE 32768

struct PeerList;
struct InstanceStruct;

struct ChanStruct{
  pthread_mutex_t lock;
  int chid;
  struct PeerList *peer;
  volatile void *volatile rbuffer, *volatile rrPtr, *volatile rwPtr;
  volatile void *volatile sbuffer, *volatile srPtr, *volatile swPtr, *volatile ssPtr;
  struct InstanceStruct *waitingInstance;
  struct InstanceStruct *orgWriter;

  struct ChanStruct *next;
};


int ch_init(int workers);

int ch_receive();

int ch_action(struct InstanceStruct *instance);


#endif//_CHAN_MANAGER_H
```

C.3 ChanManager.c

```c
#include <pthread.h>
#include <string.h>
#include <stdlib.h>
#include <semaphore.h>

#include "TaskManager.h"
#include "ChanManager.h"
#include "PeerHash.h"
#include "Network.h"
#include "Global.h"

static pthread_mutex_t chanHashLock = PTHREAD_MUTEX_INITIALIZER;
static struct ChanStruct **chanHash;

static int sendQueue[256];
static unsigned char sqrPtr = 0;
static unsigned char sqwPtr = 0;
static sem_t sqrSem, sqwSem;
static pthread_mutex_t sqLock = PTHREAD_MUTEX_INITIALIZER;


static struct ChanStruct *allocateNew(int chid, int peerid){
  struct ChanStruct *ret = malloc(sizeof(*ret));
  ret->chid = chid;
  if(peerid == -1 || peerid == nw_getNodeId())
    ret->peer = NULL;
  else
    ret->peer = ph_getPeer(peerid);
  ret->waitingInstance = NULL;
  ret->orgWriter = NULL;
  ret->next = NULL;
  pthread_mutex_init(&(ret->lock), NULL);

  ret->rrPtr = ret->rwPtr = ret->rbuffer = malloc(BUFFERSIZE);
  ret->ssPtr = ret->srPtr = ret->swPtr = ret->sbuffer = malloc(BUFFERSIZE);

  return ret;
}

static int min3(int i1, int i2, int i3){
  i1 = (i1 < i2 ? i1 : i2);
  return (i1 < i3 ? i1 : i3);
}

static void volatile_memcpy(volatile void *dest, volatile void *src, int n){
  int i;
  for(i = 0; i < n; ++i)
    *((unsigned char*)dest + i) = *((unsigned char*)src + i);
}

static struct ChanStruct *getChan(int chid, int peerid){
  pthread_mutex_lock(&chanHashLock);
  struct ChanStruct *ptr = chanHash[chid % NCHANBUCKETS];
  struct ChanStruct *last = NULL;
  while(ptr != NULL && ptr->chid != chid){
    last = ptr;
    ptr = ptr->next;
  }
  if(ptr == NULL){
    ptr = allocateNew(chid, peerid);
    if(last == NULL)
      chanHash[chid % NCHANBUCKETS] = ptr;
    else
      last->next = ptr;
  }
  if(ptr != NULL && ptr->peer == NULL && peerid != -1 && peerid != nw_getNodeId())
    ptr->peer = ph_getPeer(peerid);

  pthread_mutex_unlock(&chanHashLock);
  return ptr;
}

static void *worker(void *data){
  while(masterSwitch){
    sem_wait(&sqrSem);
    pthread_mutex_lock(&sqLock);
    int chid = sendQueue[sqrPtr++];
    pthread_mutex_unlock(&sqLock);
    sem_post(&sqwSem);
```

```c
    struct ChanStruct *ptr = getChan(chid, -1);
    if(ptr == NULL){
      fprintf(stderr, ERRFIX"Error in send worker - channel not found (%s:%d)\n", __FILE__,__LINE__);
      SHUTDOWN;
    }

    pthread_mutex_lock(&(ptr->lock));
    int maxTransfer = (int)ptr->swPtr - (int)ptr->ssPtr;
    if(maxTransfer < 0)
      maxTransfer += BUFFERSIZE;
    if(maxTransfer > 0){
      int splitPoint = BUFFERSIZE - (int)ptr->ssPtr + (int)ptr->sbuffer;
      if(splitPoint <= maxTransfer){
        if(nw_chsend(ptr->peer, ptr->chid, ptr->ssPtr, splitPoint)){
          fprintf(stderr, ERRFIX"Error on chansend (%s:%d)\n",__FILE__,__LINE__);
          SHUTDOWN;
        }
        ptr->ssPtr = ptr->sbuffer;
        if(splitPoint < maxTransfer){
          if(nw_chsend(ptr->peer, ptr->chid, ptr->ssPtr, maxTransfer - splitPoint)){
            fprintf(stderr, ERRFIX"Error on chansend (%s:%d)\n",__FILE__,__LINE__);
            SHUTDOWN;
          }
          ptr->ssPtr += maxTransfer - splitPoint;
        }
      }
      else{
        if(nw_chsend(ptr->peer, ptr->chid, ptr->ssPtr, maxTransfer)){
          fprintf(stderr, ERRFIX"Error on chansend (%s:%d)\n",__FILE__,__LINE__);
          SHUTDOWN;
        }
        ptr->ssPtr += maxTransfer;
      }
    }
    pthread_mutex_unlock(&(ptr->lock));
  }
  return NULL;
}



int ch_init(int workers){
  chanHash = malloc(NCHANBUCKETS * sizeof(struct ChanStruct*));
  memset(chanHash, 0, NCHANBUCKETS * sizeof(struct ChanHash*));

  sem_init(&sqrSem, 0, 0);
  sem_init(&sqwSem, 0, 256);

  int i;
  for(i = 1; i <= workers; ++i){
    pthread_t t;
    if(pthread_create(&t, NULL, worker, NULL)){
      fprintf(stderr, ERRFIX"Could not create required number of threads. (Failed on %d of %d) (%s:%d)\n", i, workers, __FILE__, __LINE__);
      return -1;
    }
  }
  return 0;
}




int ch_action(struct InstanceStruct *instance){
  int chid = instance->chanTrans[instance->localCh];
  int peerid = instance->chanTrans[instance->localCh + 1];

  struct ChanStruct *ptr = getChan(chid, peerid);
  if(ptr == NULL){
    fprintf(stderr, ERRFIX"Could not find chan (%s:%d)\n",__FILE__,__LINE__);
    return -1;
  }
  if(ptr->peer == NULL && nw_getNodeId() != peerid){
    instance->nodeWait = peerid;
    if(instance->state == CHANR)
      instance->state = CHANRNW;
    else if(instance->state == CHANW)
      instance->state = CHANWNW;
    else{
      fprintf(stderr, ERRFIX"Erroneously state %d (%s:%d)\n", instance->state, __FILE__, __LINE__);
      return -1;
    }
    tm_requeue(instance);
    return 0;
  }

```

```c

    pthread_mutex_lock(&(ptr->lock));

    unsigned char trade = 0;
    void volatile *volatile rbuffer;
    void volatile *volatile rwPtr;
    void volatile *volatile rrPtr;
    void volatile *volatile sbuffer;
    void volatile *volatile swPtr;
    void volatile *volatile srPtr;

    if(ptr->peer == NULL){ //Local
      if(ptr->orgWriter == NULL && instance->state == CHANW){
        ptr->orgWriter = instance;
      }
      if(ptr->orgWriter == instance){
        rbuffer = ptr->sbuffer;
        rwPtr = ptr->swPtr;
        rrPtr = ptr->srPtr;
        sbuffer = ptr->rbuffer;
        swPtr = ptr->rwPtr;
        srPtr = ptr->rrPtr;
        trade = 1;
      }
    }
    if(trade == 0){
      rbuffer = ptr->rbuffer;
      rwPtr = ptr->rwPtr;
      rrPtr = ptr->rrPtr;
      sbuffer = ptr->sbuffer;
      swPtr = ptr->swPtr;
      srPtr = ptr->srPtr;
    }

    if(instance->state == CHANR){
      int trans = 0;
      int maxTransfer = (int)rwPtr - (int)rrPtr;
      if(maxTransfer < 0)
        maxTransfer += BUFFERSIZE;
      while(instance->comSize > 0 && maxTransfer > 0){
        int splitPoint = BUFFERSIZE - (int)rrPtr + (int)rbuffer;
        int toTransfer = min3(instance->comSize, splitPoint, maxTransfer);
        volatile_memcpy(instance->comPtr, rrPtr, toTransfer);
        instance->comPtr += toTransfer;
        instance->comSize -= toTransfer;
        maxTransfer -= toTransfer;
        rrPtr += toTransfer;
        if((int)rrPtr == (int)rbuffer + BUFFERSIZE)
          rrPtr = rbuffer;
        trans += toTransfer;
      }
      if(instance->comSize == 0){
        instance->state = READY;
        tm_requeue(instance);
      }
      else{
        instance->prev = NULL;
        instance->next = ptr->waitingInstance;
        if(ptr->waitingInstance != NULL){
          ptr->waitingInstance->prev = instance;
        }
        ptr->waitingInstance = instance;
      }
      if(trans){
        if(ptr->peer == NULL){//same peer
          while(ptr->waitingInstance != NULL){
            if(ptr->waitingInstance != instance){
              struct InstanceStruct *tmp = ptr->waitingInstance;
              ptr->waitingInstance = ptr->waitingInstance->next;
              tm_requeue(tmp);
            }
            else{;
              ptr->waitingInstance->prev = NULL;
              ptr->waitingInstance = ptr->waitingInstance->next;
            }
          }
        }
        else{
          nw_chsend(ptr->peer, ptr->chid, NULL, trans);
        }
      }
    }
    else if(instance->state == CHANW){
      int trans = 0;
      int maxTransfer = (int)srPtr - (int)swPtr - 1;
      if(maxTransfer < 0)
        maxTransfer += BUFFERSIZE;

```

```
    while(instance->comSize > 0 && maxTransfer > 0){
      int splitPoint = BUFFERSIZE - (int)swPtr + (int)sbuffer;
      int toTransfer = min3(instance->comSize, splitPoint, maxTransfer);
      volatile_memcpy(swPtr, instance->comPtr, toTransfer);
      instance->comPtr += toTransfer;
      instance->comSize -= toTransfer;
      maxTransfer -= toTransfer;
      swPtr += toTransfer;
      trans += toTransfer;
      if((int)swPtr == (int)sbuffer + BUFFERSIZE)
        swPtr -= BUFFERSIZE;
    }
    if(instance->comSize == 0){
      instance->state = READY;
      tm_requeue(instance);
    }
    else{
      instance->prev = NULL;
      instance->next = ptr->waitingInstance;
      if(ptr->waitingInstance != NULL){
        ptr->waitingInstance->prev = instance;
      }
      ptr->waitingInstance = instance;
    }
    if(trans){
      if(ptr->peer == NULL){//same peer
        while(ptr->waitingInstance != NULL){
          if(ptr->waitingInstance != instance){
            struct InstanceStruct *tmp = ptr->waitingInstance;
            ptr->waitingInstance = ptr->waitingInstance->next;
            tm_requeue(tmp);
          }
          else{
            ptr->waitingInstance->prev = NULL;
            ptr->waitingInstance = ptr->waitingInstance->next;
          }
        }
      }
      else{
        sem_wait(&sqwSem);
        pthread_mutex_lock(&sqLock);
        sendQueue[sqwPtr++] = chid;
        pthread_mutex_unlock(&sqLock);
        sem_post(&sqrSem);
      }
    }
  }
  else{
    pthread_mutex_unlock(&(ptr->lock));
    fprintf(stderr, ERRFIX"Erroneously state %d (%s:%d)\n", instance->state, __FILE__, __LINE__);
    return -1;
  }
  if(trade){
    ptr->rbuffer = sbuffer;
    ptr->rwPtr = swPtr;
    ptr->rrPtr = srPtr;
    ptr->sbuffer = rbuffer;
    ptr->swPtr = rwPtr;
    ptr->srPtr = rrPtr;
  }
  else{
    ptr->rbuffer = rbuffer;
    ptr->rwPtr = rwPtr;
    ptr->rrPtr = rrPtr;
    ptr->sbuffer = sbuffer;
    ptr->swPtr = swPtr;
    ptr->srPtr = srPtr;
  }

  pthread_mutex_unlock(&(ptr->lock));
  return 0;
}


int ch_receive(int chid, void *data, int size){
  struct ChanStruct *ptr = getChan(chid, -1);

  if(ptr == NULL){
    fprintf(stderr, ERRFIX"Error on receive - channel not found (%s:%d)\n",__FILE__, __LINE__);
    return -1;
  }

  pthread_mutex_lock(&(ptr->lock));

  if(data == NULL){ //Ack
    int maxTransfer = (int)ptr->ssPtr - (int)ptr->srPtr;
```

```
      if(maxTransfer < 0)
        maxTransfer += BUFFERSIZE;
      if(size > maxTransfer){
        fprintf(stderr, ERRFIX"Buffer overflow %d/%d (%s:%d)\n",size,maxTransfer,__FILE__
            ,__LINE__);
        pthread_mutex_unlock(&(ptr->lock));
        return -1;
      }
      ptr->srPtr += size;
      if(ptr->srPtr > ptr->sbuffer + BUFFERSIZE)
        ptr->srPtr -= BUFFERSIZE;
      while(ptr->waitingInstance != NULL){
          struct InstanceStruct *tmp = ptr->waitingInstance;
          ptr->waitingInstance = ptr->waitingInstance->next;
          tm_requeue(tmp);
      }
    }
    else{ //Receive
      int maxTransfer = (int)ptr->rrPtr - (int)ptr->rwPtr - 1;
      if(maxTransfer < 0)
        maxTransfer += BUFFERSIZE;

      if(maxTransfer < size){
        fprintf(stderr, ERRFIX"Buffer overflow %d/%d (%s:%d)\n",size,maxTransfer,__FILE__
            ,__LINE__);
        pthread_mutex_unlock(&(ptr->lock));
        return -1;
      }
      int splitPoint = BUFFERSIZE - (int)ptr->rwPtr + (int)ptr->rbuffer;
      if(splitPoint <= size){
        volatile_memcpy(ptr->rwPtr, data, splitPoint);
        ptr->rwPtr = ptr->rbuffer;
        if(splitPoint < size){
          volatile_memcpy(ptr->rwPtr, data + splitPoint, size - splitPoint);
          ptr->rwPtr += size - splitPoint;
        }
      }
      else{
        volatile_memcpy(ptr->rwPtr, data, size);
        ptr->rwPtr += size;
      }
      while(ptr->waitingInstance != NULL){
        struct InstanceStruct *tmp = ptr->waitingInstance;
        ptr->waitingInstance = ptr->waitingInstance->next;
        tm_requeue(tmp);
      }
    }
    pthread_mutex_unlock(&(ptr->lock));
    return 0;
  }
```

C.4 FunctionManager.h

```
#ifndef _FUNCTIONMANAGER_H
#define _FUNCTIONMANAGER_H

#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>
#include <stdlib.h>

struct InstanceStruct;

struct FunStruct{
  char *name;
  FILE *file;
  void *handle;
  void (*fun)(struct InstanceStruct *);
};


int fm_init();
int fm_writeToFile(char *name, char *data, int len, int remainder);
int fm_createInstance(char *name, int *chanTrans);
int fm_loadFunction(char *name);
int fm_readFunction(char *name, void **ptr, unsigned long *fileLen);

#endif //_FUNCTIONMANAGER_H
```

C.5 FunctionManager.c

```c

#define _GNU_SOURCE
#include <search.h>

#include <stdio.h>
#include <stdlib.h>
#include <dlfcn.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <pthread.h>
#include <errno.h>

#include "FunctionManager.h"
#include "TaskManager.h"
#include "Global.h"

#define PATH "./dlibs/"
#define PERM 0777

static pthread_mutex_t tabLock = PTHREAD_MUTEX_INITIALIZER;
static struct hsearch_data *tab;

int fm_init(){
  umask(~PERM);
  tab = malloc(sizeof(*tab));
  bzero(tab, sizeof(*tab));

  if(hcreate_r(50, tab) == 0){
    fprintf(stderr,ERRFIX"Error creating hash table in function manager. (%s:%d)\n", __FILE__, __LINE__);
    return -1;
  }
  return 0;
}

ENTRY *fm_createFile(char *name){
  struct FunStruct *fun = malloc(sizeof(struct FunStruct));
  fun->name = malloc(strlen(name) + 1);
  strcpy(fun->name, name);
  fun->handle = NULL;
  fun->fun = NULL;

  char *tmp = malloc(strlen(fun->name) + strlen(PATH) + 1);
  sprintf(tmp, PATH"%s", fun->name);

  if((fun->file = fopen(tmp, "wb")) == NULL){
    free(tmp);
    fprintf(stderr,ERRFIX"Error creating file! (%s:%d)\n", __FILE__, __LINE__);
    return NULL;
  }
  free(tmp);

  ENTRY entry, *res;
  entry.key = name;
  entry.data = fun;
  int success;

  pthread_mutex_lock(&tabLock);
  success = hsearch_r(entry, ENTER, &res, tab);
  pthread_mutex_unlock(&tabLock);

  if(success == 0){
    fprintf(stderr,ERRFIX"Error creating hash entry! (%s:%d)\n", __FILE__, __LINE__);
    return NULL;
  }
  if(res->data != entry.data)
    fprintf(stderr,ERRFIX"Warning: Hash table overwrite! (%s:%d)\n", __FILE__, __LINE__); //TODO send warning to source?

  return res;
}

int fm_completeFile(struct FunStruct *fun){
  if(fclose(fun->file)){
    fprintf(stderr,ERRFIX"Error closing file! (%s:%d)\n", __FILE__, __LINE__);
    return -1;
  }
  fun->file = NULL;
```

```c
  char *tmp = malloc(strlen(fun->name) + strlen(PATH) + 1);
  sprintf(tmp, PATH"%s", fun->name);
  fun->handle = dlopen(tmp, RTLD_LAZY);
  if (!fun->handle){
    free(tmp);
    fprintf (stderr, "Error in %s:%d - %s\n", __FILE__, __LINE__, dlerror());
    return -1;
  }
  fun->fun = dlsym(fun->handle, fun->name);
  if (!fun->fun){
    free(tmp);
    fprintf(stderr, "Error in %s:%d - %s\n", __FILE__, __LINE__, dlerror());
    return -1;
  }
  free(tmp);
  return 0;
}

int fm_writeToFile(char *name, char *data, int len, int remainder){
  ENTRY entry, *res = NULL;
  entry.key = name;

  pthread_mutex_lock(&tabLock);
  hsearch_r(entry, FIND, &res, tab);
  pthread_mutex_unlock(&tabLock);

  if(res == NULL)
    if((res = fm_createFile(name)) == NULL)
      return -1;

  struct FunStruct *fun = (struct FunStruct*)res->data;
  if(fun->file == NULL){
    fprintf(stderr,ERRFIX"Error in %s:%d - possible overwrite\n", __FILE__, __LINE__);
    return -1;
  }

  if(fwrite(data, 1, len, fun->file) < len){
    fprintf(stderr,ERRFIX"Error writing to file! (%s:%d)\n", __FILE__, __LINE__);
    return -1;
  }

  if(remainder == 0)
    if(fm_completeFile(fun))
      return -1;

  return 0;
}

int fm_loadFunction(char *name){
  struct FunStruct *fun = malloc(sizeof(struct FunStruct));
  fun->name = malloc(strlen(name) + 1);
  strcpy(fun->name, name);
  fun->handle = NULL;
  fun->fun = NULL;
  fun->file = NULL;


  char *tmp = malloc(strlen(fun->name) + strlen(PATH) + 1);
  sprintf(tmp, PATH"%s", fun->name);

  fun->handle = dlopen(tmp, RTLD_LAZY);
  if (!fun->handle){
    free(tmp);
    fprintf (stderr, "Error in %s:%d - %s\n", __FILE__, __LINE__, dlerror());
    return -1;
  }
  fun->fun = dlsym(fun->handle, fun->name);
  if (fun->fun == NULL){
    free(tmp);
    fprintf(stderr, "Error in %s:%d - %s\n", __FILE__, __LINE__, dlerror());
    return -1;
  }
  free(tmp);

  ENTRY entry, *res;
  entry.key = fun->name;
  entry.data = fun;
  int success;

  pthread_mutex_lock(&tabLock);
  success = hsearch_r(entry, ENTER, &res, tab);
  pthread_mutex_unlock(&tabLock);

  if(success == 0){
    fprintf(stderr,ERRFIX"Error creating hash entry! (%s:%d)\n", __FILE__, __LINE__);
    return -1;
  }
  if(res->data != entry.data)
```

```c
      fprintf(stderr,ERRFIX"Warning: Hash table overwrite! (%s:%d)\n", __FILE__, __LINE__); //TODO send warning to source?

  return 0;
}

int fm_readFunction(char *name, void **ptr, unsigned long *fileLen){

  char *tmp = malloc(strlen(name) + strlen(PATH) + 1);
  sprintf(tmp, PATH"%s", name);

  FILE *file;
  file = fopen(tmp, "rb");
  if (!file){
    fprintf(stderr, "Unable to open file %s - %s - (%s:%d)\n", name, strerror(errno), __FILE__, __LINE__);
    return -1;
  }
  fseek(file, 0, SEEK_END);
  *fileLen = ftell(file);
  fseek(file, 0, SEEK_SET);
  *ptr = malloc(*fileLen);
  if(ptr == NULL){
    fprintf(stderr,ERRFIX"Unable to allocate %lu bytes of memory (%s:%d)\n", *fileLen, __FILE__, __LINE__);
    return -1;
  }
  if(fread(*ptr, 1, *fileLen, file) != *fileLen){
    fprintf(stderr,ERRFIX"Error while reading from file %s (%s:%d)\n", name, __FILE__, __LINE__);
    return -1;
  }
  fclose(file);
  free(tmp);
  return 0;
}

int fm_createInstance(char *name, int *chanTrans){
  ENTRY entry, *res = NULL;
  entry.key = name;

  pthread_mutex_lock(&tabLock);
  int success = hsearch_r(entry, FIND, &res, tab);
  pthread_mutex_unlock(&tabLock);

  if(res == NULL || success == 0){
      if(fm_loadFunction(name)){
        fprintf(stderr,ERRFIX"File not found! %s:%d\n", __FILE__, __LINE__);
        return -1;
      }
      pthread_mutex_lock(&tabLock);
      success = hsearch_r(entry, FIND, &res, tab);
      pthread_mutex_unlock(&tabLock);
  }
  if(res == NULL || success == 0){//Should not happen
    fprintf(stderr,ERRFIX"Entry not found! %s:%d\n", __FILE__, __LINE__);
    return -1;
  }

  if(tm_createNew((struct FunStruct*)res->data, chanTrans)){
    fprintf(stderr,ERRFIX"Error on creating new instance! %s:%d\n", __FILE__, __LINE__);
    return -1;
  }

  return 0;

}
```

C.6 Global.h

```
#ifndef _GLOBAL_H
#define _GLOBAL_H

extern char masterSwitch;
#define SHUTDOWN {fprintf(stderr,"!!! SHUTDOWN (%s:%d)\n",__FILE__,__LINE__);
      masterSwitch = 0; return NULL;}
#define PREFIX ">>> "
#define ERRFIX "!!! "


#endif //_GLOBAL_H
```

C.7 Global.c

```

char masterSwitch = 1;
```

C.8 Network.h

```c
#ifndef _NETWORK_H
#define _NETWORK_H

#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>

struct PeerList{
  int socket;
  int id;
  char cascade;
  char *rcvBuffer;
  char rPtr, wPtr;
  sem_t rSem, wSem;
  pthread_mutex_t sendLock;
  struct sockaddr_storage *saddr;

  struct PeerList *prev;
  struct PeerList *next;
};


int nw_init(int id, int port, char *mip, char *mpt);
int nw_getNodeId();
int nw_sendFile(int id, char *name);
int nw_chsend(struct PeerList *peer, int chid, volatile void * volatile data, int size);
int nw_spawn(int id, char *name, int *chanTrans, int ctSize);
int nw_close();
int nw_panic();


#endif //_NETWORK_H
```

C.9 Network.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <pthread.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <semaphore.h>

#include "Network.h"
#include "FunctionManager.h"
#include "Global.h"
#include "PeerHash.h"
#include "ChanManager.h"

#define BLOCKS      256
#define BLOCKSIZE   1024
#define TBUFFERSIZE 2048

struct ThreadList{
  pthread_t *t;
  int listenSocket;
  struct ThreadList *next;
};


enum Mode{IDLE, PARTLY, BUFFER, TRANSFER, EXECUTE, WRITE, READ, HANDSHAKE, ERROR}; //
     BUFFER is unused
struct ParseRet{
  enum Mode mode;
  char *name;
  int id; //Chan id, peer id or file socket/ptr, subject to change
  int aux; //Aux data (port, meassage id)
  int n;  //number of bytes or args
};
//List of threads to cancel in panic-mode
//Must always be consistent, and be read without allocating lock.
//(Aquire lock for writing)
static pthread_mutex_t threadsLock = PTHREAD_MUTEX_INITIALIZER;
static struct ThreadList *threadsFirst = NULL;
static struct ThreadList *threadsLast = NULL;

//List of peers
static pthread_mutex_t peersLock = PTHREAD_MUTEX_INITIALIZER;
static struct PeerList *peers = NULL;
struct PeerString{
  int allocSize;
  int usedSize;
  char *string;
}peerString;


//Node info
static int nodeId = -1;
static int nodePort;

//Internal prototypes
void *receive_thread(void *);
void *recvWrk_thread(void *);


//List aux functions
static void addThread(pthread_t *t, int listenSocket){
  struct ThreadList *tmp = malloc(sizeof(*tmp));
  tmp->t = t;
  tmp->listenSocket = listenSocket;
  tmp->next = NULL;
  pthread_mutex_lock(&threadsLock);
  if(threadsFirst == NULL)
    threadsFirst = tmp;
  if(threadsLast != NULL)
    threadsLast->next = tmp;
  threadsLast = tmp;
  pthread_mutex_unlock(&threadsLock);
```

```
}

struct PeerList *addPeer(int nodeSocket, struct sockaddr_storage *client, char cascade, char isAccept){
  struct PeerList *peer = malloc(sizeof(*peer));
  peer->prev = NULL;
  peer->socket = nodeSocket;
  peer->saddr = client;
  peer->id = -1;
  peer->cascade = cascade;
  peer->rcvBuffer = malloc(BLOCKS * BLOCKSIZE);
  peer->rPtr = peer->wPtr = 0;
  sem_init(&peer->rSem, 0, 0);
  sem_init(&peer->wSem, 0, BLOCKS);
  pthread_mutex_init(&peer->sendLock, NULL);

  pthread_mutex_lock(&peersLock);

  if(isAccept){
    pthread_mutex_lock(&peer->sendLock);
    char *tmp = peerString.string;
    int left = peerString.usedSize;
    while(left){
      int n = send(peer->socket, (void*)tmp, left, 0);
      if(n < 1){
        fprintf(stderr,ERRFIX"Error on send! (%s:%d)\n", __FILE__, __LINE__);
        pthread_mutex_unlock(&peer->sendLock);
        pthread_mutex_unlock(&peersLock);
        return NULL;
      }
      left -= n;
      tmp += n;
    }
  pthread_mutex_unlock(&peer->sendLock);
  }

  peer->next = peers;
  if(peers != NULL)
    peers->prev = peer;
  peers = peer;

  char host[256];
  char port[10];
  if(getnameinfo((struct sockaddr*)client, sizeof(*client), host, 255, port, 10, NI_NUMERICHOST | NI_NUMERICSERV) == 0){
    if(isAccept)
      printf(PREFIX"[%s]:%s connected\n", host, port);
    else
      printf(PREFIX"Connected to [%s]:%s\n", host, port);
  }
  else{
    if(isAccept)
      printf(PREFIX"Node connected, IP not found (ERROR) - (%s:%d)\n", __FILE__, __LINE__);
    else
      printf(PREFIX"Connected to node, IP not found (ERROR) - (%s:%d)\n", __FILE__, __LINE__);
    return NULL;
  }
  pthread_mutex_unlock(&peersLock);

  pthread_t *t1 = malloc(sizeof (*t1));
  pthread_t *t2 = malloc(sizeof (*t2));
  pthread_create(t1, NULL, receive_thread, peer);
  pthread_create(t2, NULL, recvWrk_thread, peer);
  return peer;
}

int connectPeer(char *peerInfo){
  char *savePtr = NULL;
  char *ip = strtok_r(peerInfo, "|", &savePtr);
  char *port = strtok_r(NULL, "|", &savePtr);

  struct addrinfo hints;
  memset(&hints, 0, sizeof hints);
  hints.ai_family = AF_UNSPEC;
  hints.ai_socktype = SOCK_STREAM;
  hints.ai_flags = AI_PASSIVE;

  struct addrinfo *list;
  if(getaddrinfo(ip, port, &hints, &list)){
    fprintf(stderr,ERRFIX"Error on getaddrinfo (%s:%d)\n", __FILE__, __LINE__);
    return -1;
  };
  struct addrinfo *ptr;
  struct PeerList *peer;
  for(ptr = list; ptr != NULL; ptr = ptr->ai_next){
    int serverSocket = socket(ptr->ai_family, ptr->ai_socktype, ptr->ai_protocol);
```

```c
    if(serverSocket < 0) continue;
    if(connect(serverSocket, ptr->ai_addr, ptr->ai_addrlen) < 0) continue;

    struct sockaddr *addr = malloc(sizeof *addr);
    socklen_t len = sizeof *addr;
    if(getsockname(serverSocket, addr, &len)) continue;

    if((peer = addPeer(serverSocket, (struct sockaddr_storage*)addr, 0, 0)) == NULL) continue;
    break;
  }
  freeaddrinfo(list);
  if(peer == NULL){
    perror("Connect");
    fprintf(stderr,ERRFIX"Unable to connect (%s:%d)\n", __FILE__, __LINE__);
    return -1;
  }
  return 0;
}

int cascadePeer(char *buffer){
  char cmpl = 0;
  if(buffer[strlen(buffer) - 1] == ' ')
    cmpl = 1;

  char *savePtr = NULL;
  char *tok = strtok_r(buffer," ",&savePtr);
  char *last = NULL;
  while(tok != NULL){
    if(last != NULL)
      if(connectPeer(last))
        return -1;
    last = tok;
    tok = strtok_r(NULL," ",&savePtr);

  }
  if(cmpl){
    if(last != NULL)
      if(connectPeer(last))
        return -1;
    buffer[0] = 0;
  }
  else{
    memmove(buffer, last, strlen(last) + 1);
  }
  return 0;
}

struct ParseRet parse(char *msg){
  struct ParseRet ret = {IDLE, NULL, -1, 0};
  char *ptr;
  switch(*msg){
    case 'H': //H<id> <port> <size of list of nodes>
      ret.mode = HANDSHAKE;
      ret.id = strtol(msg + 1, &ptr, 10);
      ret.aux = strtol(ptr + 1, &ptr, 10);
      ret.n = strtol(ptr + 1, NULL, 10);
      break;
    case 'T': //T<name> <size of file>
      ret.mode = TRANSFER;
      ret.name = strndup(msg + 1, (int)strchr(msg + 1, ' ') - (int)msg - 1);
      ret.n = strtol(msg + strlen(ret.name) + 1, NULL, 10);
      break;
    case 'E': //E<name> <size of arguments>
      ret.mode = EXECUTE;
      ret.name = strndup(msg + 1, (int)strchr(msg + 1, ' ') - (int)msg - 1);
      ret.n = strtol(msg + strlen(ret.name) + 1, NULL, 10);
      break;
    case 'W': //W<chan> <size>
      ret.mode = WRITE;
      ret.id = strtol(msg + 1, &ptr, 10);
      ret.n = strtol(ptr + 1, NULL, 10);
      break;
    case 'R': //R<chan> <size>
      ret.mode = READ;
      ret.id = strtol(msg + 1, &ptr, 10);
      ret.n = strtol(ptr + 1, NULL, 10);
      break;
    default:
      ret.mode = ERROR;
  }
  return ret;
}

//Thread functions
void *receive_thread(void *data){
  struct PeerList *peer = (struct PeerList*)data;
  while(masterSwitch){
```

```
      int n = 0;
      sem_wait(&peer->wSem);
      if((n = recv(peer->socket, &peer->rcvBuffer[peer->wPtr * BLOCKSIZE + 4], BLOCKSIZE - 4, 0)) < 1)
        SHUTDOWN;
      *(short*)&peer->rcvBuffer[peer->wPtr * BLOCKSIZE] = (short)n;
      *(short*)&peer->rcvBuffer[peer->wPtr * BLOCKSIZE + 2] = 4;
      ++peer->wPtr;
      sem_post(&peer->rSem);
    }
    return NULL;
}

void *recvWrk_thread(void *data){
    struct PeerList *peer = (struct PeerList*)data;

    char *buffer = malloc(TBUFFERSIZE);
    buffer[0] = 0;
    void *chanTrans;
    int ctPtr = 0;
    struct ParseRet state = {IDLE, NULL, -1, 0};

    //Wait for first item
    sem_wait(&peer->rSem);

    char advance = 0;
    while(masterSwitch){

      short len = *(short*)&peer->rcvBuffer[peer->rPtr * BLOCKSIZE];
      short off = *(short*)&peer->rcvBuffer[peer->rPtr * BLOCKSIZE + 2];
      char *msg = &peer->rcvBuffer[peer->rPtr * BLOCKSIZE + off];
      int partLen = strnlen(msg, len);
      char end = 0;
      if(partLen < len){
        ++partLen;
        end = 1;
      }

      switch(state.mode){
        case IDLE:
          if(!end){
            strncpy(buffer, msg, len);
            state.mode = PARTLY;
          }
          else
            state = parse(msg);
          if(partLen == len)
            advance = 1;
          else{
            advance = 0;
            *(short*)&peer->rcvBuffer[peer->rPtr * BLOCKSIZE]     -= partLen; //new len
            *(short*)&peer->rcvBuffer[peer->rPtr * BLOCKSIZE + 2] += partLen; //new offset
          }
          break;
        case PARTLY:
          strncat(buffer, msg, partLen);
          if(end){
            state = parse(buffer);
            buffer[0] = 0;
          }
          if(partLen == len)
            advance = 1;
          else{
            advance = 0;
            *(short*)&peer->rcvBuffer[peer->rPtr * BLOCKSIZE]     -= partLen; //new len
            *(short*)&peer->rcvBuffer[peer->rPtr * BLOCKSIZE + 2] += partLen; //new offset
          }
          break;
        case HANDSHAKE:
          partLen = (len < state.n ? len : state.n);
          state.n -= partLen;

          if(peer->id != -1 && peer->cascade){ //Cascade on 2nd handshake
            strncat(buffer, msg, partLen);
            if(cascadePeer(buffer)){
              fprintf(stderr,ERRFIX"Could not cascade! (%s:%d)\n",__FILE__, __LINE__);
              SHUTDOWN;
            }
          }

          if(state.n == 0){
            if(peer->id == -1){
              peer->id = state.id;
              ph_add(peer);
              pthread_mutex_lock(&peersLock);
              pthread_mutex_lock(&peer->sendLock);
```

```c
          char *tmp = peerString.string;
          int left = peerString.usedSize;
          while(left){
            int n = send(peer->socket, (void*)tmp, left, 0);
            if(n < 1){
              fprintf(stderr,ERRFIX"Error on send! (%s:%d)\n", __FILE__, __LINE__);
              pthread_mutex_unlock(&peer->sendLock);
              pthread_mutex_unlock(&peersLock);
              SHUTDOWN;
            }
            left -= n;
            tmp += n;
          }

          char host[256];
          char port[10];
          if(getnameinfo((struct sockaddr*)peer->saddr, sizeof(*peer->saddr), host, 255, port, 10, NI_NUMERICHOST | NI_NUMERICSERV) == 0){
            printf(PREFIX"Handshake from [%s]:%s - id:%d\n", host, port, state.id);
            if(peerString.allocSize < peerString.usedSize + 22){
              peerString.allocSize *= 2;
              char *nstring = malloc(peerString.allocSize);
              strcpy(nstring, peerString.string);
              free(peerString.string);
              peerString.string = nstring;
            }
            sprintf(peerString.string + peerString.usedSize - 1, "%s|%d ", host, state.aux);
            peerString.usedSize = strlen(peerString.string + 17) + 18;
            sprintf(peerString.string, "H%04d %05d %04d", nodeId, nodePort, peerString.usedSize - 17);
          }
          else{
            printf(PREFIX"Client connected (ip unavailible)\n");
            return NULL;
          }
          pthread_mutex_unlock(&peer->sendLock);
          pthread_mutex_unlock(&peersLock);
        }
        state.mode = IDLE;
      }
      if(partLen == len)
        advance = 1;
      else{
        advance = 0;
        *(short*)&peer->rcvBuffer[peer->rPtr * BLOCKSIZE]     -= partLen; //new len
        *(short*)&peer->rcvBuffer[peer->rPtr * BLOCKSIZE + 2] += partLen; //new offset
      }
      break;
    case TRANSFER:
      partLen = (len < state.n ? len : state.n);
      state.n -= partLen;

      if(fm_writeToFile(state.name, msg, partLen, state.n)){
        fprintf(stderr,ERRFIX"Error writing to file! (%s:%d)\n", __FILE__, __LINE__);
        SHUTDOWN;
      }
      if(state.n == 0){
        free(state.name);
        state.mode = IDLE;
      }

      if(partLen == len)
        advance = 1;
      else{
        advance = 0;
        *(short*)&peer->rcvBuffer[peer->rPtr * BLOCKSIZE]     -= partLen; //new len
        *(short*)&peer->rcvBuffer[peer->rPtr * BLOCKSIZE + 2] += partLen; //new offset
      }
      break;
    case EXECUTE:
      partLen = (len < state.n ? len : state.n);
      state.n -= partLen;

      if(ctPtr == 0)
        chanTrans = malloc(state.n + partLen);
      memcpy(chanTrans + ctPtr, msg, partLen);
      ctPtr += partLen;

      if(state.n == 0)
        state.mode = IDLE;

      if(partLen == len)
        advance = 1;
      else{
        advance = 0;
```

```c
          *(short*)&peer->rcvBuffer[peer->rPtr * BLOCKSIZE]     -= partLen; //new len
          *(short*)&peer->rcvBuffer[peer->rPtr * BLOCKSIZE + 2] += partLen; //new offset
        }


        if(state.mode == IDLE){
          if(fm_createInstance(state.name, chanTrans)){
            fprintf(stderr,ERRFIX"Error on execute! (%s:%d)\n", __FILE__, __LINE__);
            SHUTDOWN;
          }
          ctPtr = 0;
          free(state.name);
          state.name = NULL;
        }
        break;
      case WRITE:
        partLen = (len < state.n ? len : state.n);
        state.n -= partLen;
        if(ch_receive(state.id, msg, partLen)){
          fprintf(stderr,ERRFIX"Error on write (%s:%d)\n",__FILE__,__LINE__);
          SHUTDOWN;
        }
        if(state.n == 0)
          state.mode = IDLE;
        if(partLen == len)
          advance = 1;
        else{
          advance = 0;
          *(short*)&peer->rcvBuffer[peer->rPtr * BLOCKSIZE]     -= partLen; //new len
          *(short*)&peer->rcvBuffer[peer->rPtr * BLOCKSIZE + 2] += partLen; //new offset
        }
        break;
      case READ:
        if(ch_receive(state.id, NULL, state.n)){
          fprintf(stderr,ERRFIX"Error on read (%s:%d)\n",__FILE__,__LINE__);
          SHUTDOWN;
        }
        state.mode = IDLE;
        break;
      case ERROR:
      default:
        fprintf(stderr,ERRFIX"Error (%s:%d)\n",__FILE__,__LINE__);
        SHUTDOWN;
        break;
    }
    if(advance && state.mode != READ){
      //Advance to next queue item
      ++peer->rPtr;
      sem_post(&peer->wSem);
      sem_wait(&peer->rSem);
      advance = 0;
    }
  }
  return NULL;
}

void *accept_thread(void *data){
  while(masterSwitch){
    struct sockaddr_storage *client;
    size_t size = sizeof(*client);
    client = malloc(size);
    int clientSocket = accept(*(int*)data, (struct sockaddr *)client, &size); //Cancel point
    if(clientSocket < 0)
      fprintf(stderr,ERRFIX"Error on accept\n");
    else if(addPeer(clientSocket, client, 0, 1) == NULL)
      SHUTDOWN;

  }
  return NULL;
}

//Public functions
int nw_init(int id, int port, char *mip, char *mpt){
  nodeId = id;

  peerString.string = malloc(1024);
  peerString.allocSize = 1024;
  peerString.usedSize = 19;
  sprintf(peerString.string, "H%04d %05d 0002", id, port);
  sprintf(peerString.string + 17, " ");

  struct addrinfo hints;
  memset(&hints, 0, sizeof hints);
  hints.ai_family = AF_INET; //CHANGE TO AF_NET6 for IPv6
  hints.ai_socktype = SOCK_STREAM;
```

```c
  hints.ai_flags = AI_PASSIVE;

  struct addrinfo *list;

  char portStr[6];
  sprintf(portStr, "%d", port);

  if(getaddrinfo(NULL, portStr, &hints, &list)){
    fprintf(stderr,ERRFIX"Error on getaddrinfo (%s:%d)\n", __FILE__, __LINE__);
    return -1;
  }

  int errorListen = -1;
  struct addrinfo *ptr;
  for(ptr = list; ptr != NULL; ptr = ptr->ai_next){
    int yes = 1;
    int serverSocket = socket(ptr->ai_family, ptr->ai_socktype, ptr->ai_protocol);
    if(serverSocket < 0) continue;
    if(setsockopt(serverSocket, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(int)) < 0) continue;
    if(bind(serverSocket, ptr->ai_addr, ptr->ai_addrlen) < 0) continue;
    if(listen(serverSocket, 10) < 0) continue;

    pthread_t *t = malloc(sizeof(*t));
    int *socket = malloc(sizeof(int));
    *socket = serverSocket;

    if(pthread_create(t, NULL, accept_thread, socket)){
      free(t);
      free(socket);
      continue;
    }
    addThread(t, serverSocket);
    errorListen = 0;
  }
  freeaddrinfo(list);
  if(errorListen){
    perror("Listen");
    fprintf(stderr,ERRFIX"Unable to socket/bind/listen (%s:%d)\n", __FILE__, __LINE__);
    return -1;
  }

  if(mip != NULL && mpt != NULL){
    if(getaddrinfo(mip, mpt, &hints, &list)){
      fprintf(stderr,ERRFIX"Error on getaddrinfo (%s:%d)\n", __FILE__, __LINE__);
      return -1;
    };
    struct addrinfo *ptr;
    struct PeerList *peer;
    for(ptr = list; ptr != NULL; ptr = ptr->ai_next){
      int serverSocket = socket(ptr->ai_family, ptr->ai_socktype, ptr->ai_protocol);
      if(serverSocket < 0) continue;
      if(connect(serverSocket, ptr->ai_addr, ptr->ai_addrlen) < 0) continue;

      struct sockaddr *addr = malloc(sizeof *addr);
      socklen_t len = sizeof *addr;
      if(getsockname(serverSocket, addr, &len)) continue;

      if((peer = addPeer(serverSocket, (struct sockaddr_storage*)addr, 1, 0)) == NULL) continue;
      break;
    }
    freeaddrinfo(list);
    if(peer == NULL){
      fprintf(stderr,ERRFIX"Unable to connect (%s:%d)\n", __FILE__, __LINE__);
      return -2;
    }
  }

  return 0;

}

int nw_getNodeId(){
  return nodeId;
}

int nw_sendFile(int id, char *name){
  struct PeerList *peer = ph_getPeer(id);
  if(peer == NULL){
    fprintf(stderr,ERRFIX"Trying to transfer to non-excisting peer - id=%d name=%s (%s:%d)\n", id, name, __FILE__, __LINE__);
    return -1;
  }
  void *ptr = NULL;
  unsigned long size;
  if(fm_readFunction(name, &ptr, &size)){
    fprintf(stderr,ERRFIX"Error! (%s:%d)\n", __FILE__, __LINE__);
```

```c
    return -1;
  }
  char *header = malloc(strlen(name) + 15);
  sprintf(header, "T%s %lu", name, size);
  pthread_mutex_lock(&peer->sendLock);
  int left = strlen(header) + 1;
  while(left){
    int n = send(peer->socket, (void*)header, left, 0);
    if(n < 1){
      fprintf(stderr,ERRFIX"Error on send! (%s:%d)\n", __FILE__, __LINE__);
      pthread_mutex_unlock(&peer->sendLock);
      return -1;
    }
    left -= n;
    header += n;
  }
  left = size;
  while(left){
    int n = send(peer->socket, (void*)ptr, left, 0);
    if(n < 1){
      fprintf(stderr,ERRFIX"Error on send! (%s:%d)\n", __FILE__, __LINE__);
      pthread_mutex_unlock(&peer->sendLock);
      return -1;
    }
    left -= n;
    ptr += n;
  }
  pthread_mutex_unlock(&peer->sendLock);
  return 0;
}

int nw_spawn(int id, char *name, int *chanTrans, int ctSize){
  struct PeerList *peer = ph_getPeer(id);
  if(peer == NULL){
    fprintf(stderr,ERRFIX"Trying to execute on non-excisting peer (%s:%d)\n",__FILE__,
        __LINE__);
    return -1;
  }
  char *buffer = malloc(strlen(name) + 12 + ctSize);
  sprintf(buffer, "E%s %d", name, ctSize);
  memcpy(buffer + strlen(buffer) + 1, chanTrans, ctSize);
  pthread_mutex_lock(&peer->sendLock);
  int left = strlen(buffer) + ctSize + 1;
  while(left){
    int n = send(peer->socket, (void*)buffer, left, 0);
    if(n < 1){
      fprintf(stderr,ERRFIX"Error on send! (%s:%d)\n", __FILE__, __LINE__);
      pthread_mutex_unlock(&peer->sendLock);
      return -1;
    }
    left -= n;
    buffer += n;
  }
  pthread_mutex_unlock(&peer->sendLock);
  return 0;
}

int nw_chsend(struct PeerList *peer, int chid, volatile void * volatile data, int size)
    {
  char *buffer = malloc(32);
  char *tmp = buffer;
  if(data == NULL)//R<id> <size>\0
    sprintf(buffer, "R%d %d", chid, size);
  else//W<id> <size>\0<data>
    sprintf(buffer, "W%d %d", chid, size);
  pthread_mutex_lock(&peer->sendLock);
  int left = strlen(buffer) + 1;
  while(left){
    int n = send(peer->socket, (void*)buffer, left, 0);
    if(n < 1){
      fprintf(stderr,ERRFIX"Error on send! (%s:%d)\n", __FILE__, __LINE__);
      pthread_mutex_unlock(&peer->sendLock);
      return -1;
    }
    left -= n;
    buffer += n;
  }
  free(tmp);
  if(data){
    left = size;
    while(left){
      int n = send(peer->socket, (void*)data, left, 0);
      if(n < 1){
        fprintf(stderr,ERRFIX"Error on send! (%s:%d)\n", __FILE__, __LINE__);
        pthread_mutex_unlock(&peer->sendLock);
        return -1;
      }
      left -= n;
```

```
            data += n;
        }
    }
    pthread_mutex_unlock(&peer->sendLock);
    return 0;
}

int nw_panic(){
    printf(PREFIX"Panic!\n");
    pthread_mutex_lock(&peersLock);
    struct ThreadList *pt;
    for(pt = threadsFirst; pt != NULL; pt = pt->next){
        shutdown(pt->listenSocket, 2);
        pthread_cancel(*pt->t);
    }
    struct PeerList *pp;
    for(pp = peers; pp != NULL; pp = pp->next)
        shutdown(pp->socket, SHUT_RDWR);
    pthread_mutex_unlock(&peersLock);
    return 0;
}
```

C.10 PeerHash.h

```
#ifndef _PEER_HASH_H
#define _PEER_HASH_H

#define NPEERBUCKETS 64

struct PeerHash{
  int id;
  struct PeerList *peer;
  struct InstanceStruct *waitingInstance;
  struct PeerHash *next;
};


int ph_init();
int ph_add(struct PeerList *peer);
int ph_waitForNode(struct InstanceStruct *instance, int id);
struct PeerList *ph_getPeer(int id);

#endif//_PEER_HASH_H
```

C.11 PeerHash.c

```
#include <pthread.h>
#include <string.h>
#include <stdlib.h>

#include "TaskManager.h"
#include "PeerHash.h"
#include "Network.h"

static pthread_mutex_t peerHashLock = PTHREAD_MUTEX_INITIALIZER;
struct PeerHash **peerHash;

int ph_add(struct PeerList *peer){
  if(peer->id < 0)
    return -1;
  pthread_mutex_lock(&peerHashLock);
  struct PeerHash *ptr = peerHash[peer->id % NPEERBUCKETS];
  struct PeerHash *last = NULL;

  while(ptr != NULL && ptr->id != peer->id){
    last = ptr;
    ptr = ptr->next;
  }

  if(ptr == NULL){
    ptr = malloc(sizeof *ptr);
    ptr->id = peer->id;
    ptr->peer = peer;
    ptr->waitingInstance = NULL;
    ptr->next = NULL;
    if(last == NULL)
      peerHash[peer->id % NPEERBUCKETS] = ptr;
    else
      last->next = ptr;
  }
  else{
    ptr->peer = peer;
    while(ptr->waitingInstance != NULL){
      struct InstanceStruct *instance = ptr->waitingInstance;
      ptr->waitingInstance = ptr->waitingInstance->next;
      if(instance->state == CHANRNW)
        instance->state = CHANR;
      else if(instance->state == CHANWNW)
        instance->state = CHANW;
      else
        instance->state = READY;
      tm_requeue(instance);
    }
  }
  pthread_mutex_unlock(&peerHashLock);
  return 0;
}

int ph_waitForNode(struct InstanceStruct *instance, int id){
  if(id < 0)
    return -1;
  pthread_mutex_lock(&peerHashLock);
  struct PeerHash *ptr = peerHash[id % NPEERBUCKETS];
  struct PeerHash *last = NULL;

  while(ptr != NULL && ptr->id != id){
    last = ptr;
    ptr = ptr->next;
  }

  if(ptr == NULL){
    ptr = malloc(sizeof *ptr);
    ptr->id = id;
    ptr->peer = NULL;
    ptr->waitingInstance = instance;
    ptr->next = NULL;
    if(last == NULL)
      peerHash[id % NPEERBUCKETS] = ptr;
    else
      last->next = ptr;
  }
  else if(ptr->peer != NULL){
    if(instance->state == CHANRNW)
      instance->state = CHANR;
    else if(instance->state == CHANWNW)
```

```
        instance->state = CHANW;
      else
        instance->state = READY;
      tm_requeue(instance);
    }
    else{
      instance->prev = NULL;
      instance->next = ptr->waitingInstance;
      if(ptr->waitingInstance != NULL){
        ptr->waitingInstance->prev = instance;
      }
      ptr->waitingInstance = instance;
    }
    pthread_mutex_unlock(&peerHashLock);
    return 0;
}

struct PeerList *ph_getPeer(int id){
    if(id < 0)
      return NULL;
    pthread_mutex_lock(&peerHashLock);
    struct PeerHash *ptr = peerHash[id % NPEERBUCKETS];
    while(ptr != NULL && ptr->id != id)
      ptr = ptr->next;

    pthread_mutex_unlock(&peerHashLock);
    if(ptr == NULL)
      return NULL;
    return ptr->peer;
}

int ph_init(){
    peerHash = malloc(NPEERBUCKETS * sizeof(struct PeerHash*));
    memset(peerHash, 0, NPEERBUCKETS * sizeof(struct PeerHash*));
    return 0;
}
```

C.12 TaskManager.h

```
#ifndef _TASK_MANAGER_H
#define _TASK_MANAGER_H

struct FunStruct; //Forward from FunctionManager.h

enum InstanceState{NEW = 1, READY = 2, CHANR = 4, CHANW = 8, CHANRNW = 16, CHANWNW =
      32, NODEWAIT = 64, TRANSFER = 128, SPAWN = 256, DONE = 512};

struct InstanceStruct{
  struct FunStruct *funStruct;
  void *memPtr;
  int *chanTrans;
  int step;
  enum InstanceState state;
  void *comPtr;
  int comSize;
  int localCh;
  int nodeWait;

  struct InstanceStruct *next, *prev;
};

struct SpawnStruct{
  char *name;
  int *chanTrans;
  int ctSize;
  int peerId;
};


int tm_init(int workers);
int tm_createNew(struct FunStruct *funStruct, int *chanTrans);
int tm_requeue(struct InstanceStruct *instance);


#endif//_TASK_MANAGER_H
```

C.13 TaskManager.c

```c
#include <pthread.h>
#include <stdlib.h>
#include <string.h>

#include "FunctionManager.h"
#include "TaskManager.h"
#include "PeerHash.h"
#include "Global.h"
#include "Network.h"
#include "ChanManager.h"

//Add to last, take from first;
static pthread_mutex_t queueLock = PTHREAD_MUTEX_INITIALIZER;
static pthread_cond_t queueCond = PTHREAD_COND_INITIALIZER;
static struct InstanceStruct *first, *last;

static void addToQueue(struct InstanceStruct *instance){
  pthread_mutex_lock(&queueLock);
  instance->next = last;
  instance->prev = NULL;
  if(last != NULL)
    last->prev = instance;
  last = instance;
  if(first == NULL)
    first = instance;
  else if(first->prev == NULL)
    first->prev = instance;
  pthread_cond_broadcast(&queueCond);
  pthread_mutex_unlock(&queueLock);
}

//Blocks if queue is empty
static struct InstanceStruct *getFirstItem(){
  struct InstanceStruct *ret;
  pthread_mutex_lock(&queueLock);
  while(first == NULL)
    pthread_cond_wait(&queueCond, &queueLock);
  ret = first;
  if(first != NULL){
    if(first->prev){
      first->prev->next = NULL;
      first = first->prev;
    }
    else{
      first = NULL;
      last = NULL;
    }
  }
  pthread_mutex_unlock(&queueLock);
  return ret;
}

static void *worker(void *data){
  while(masterSwitch){
    struct InstanceStruct *item = getFirstItem();
    if(item->state & (READY | NEW))
      item->funStruct->fun(item);
    switch(item->state){
      case NEW:
        fprintf(stderr,ERRFIX"Task has invalid state (%s:%d)\n",__FILE__,__LINE__);
        SHUTDOWN;
        break;
      case READY:
        addToQueue(item);
        break;
      case CHANR:
      case CHANW:
        if(ch_action(item)){
          fprintf(stderr,ERRFIX"Error on ch_action (%s:%d)\n", __FILE__, __LINE__);
          SHUTDOWN;
        }
        break;
      case CHANRNW:
      case CHANWNW:
      case NODEWAIT:
        ph_waitForNode(item, item->nodeWait);
        break;
      case TRANSFER:
        if(nw_sendFile(((struct SpawnStruct*)item->comPtr)->peerId,
```

```
            ((struct SpawnStruct*)item->comPtr)->name)){
              fprintf(stderr,ERRFIX"Error on transfer (%s:%d)\n",__FILE__,__LINE__);
              SHUTDOWN;
            }
            item->state = READY;
            addToQueue(item);
            break;
          case SPAWN:
            if(((struct SpawnStruct*)item->comPtr)->peerId == nw_getNodeId()){
              int *chanTrans = malloc(((struct SpawnStruct*)item->comPtr)->ctSize * sizeof(
                  int));
              memcpy(chanTrans, ((struct SpawnStruct*)item->comPtr)->chanTrans, ((struct
                  SpawnStruct*)item->comPtr)->ctSize * sizeof(int));
              if(fm_createInstance(((struct SpawnStruct*)item->comPtr)->name, chanTrans)){
                    fprintf(stderr,ERRFIX"Error on spawn (%s:%d)\n",__FILE__,__LINE__);
                    SHUTDOWN;
                    }
            }
            else if(nw_spawn(((struct SpawnStruct*)item->comPtr)->peerId,
              ((struct SpawnStruct*)item->comPtr)->name,
              ((struct SpawnStruct*)item->comPtr)->chanTrans,
              ((struct SpawnStruct*)item->comPtr)->ctSize)){
                fprintf(stderr,ERRFIX"Error on spawn (%s:%d)\n",__FILE__,__LINE__);
                SHUTDOWN;
            }
            item->state = READY;
            addToQueue(item);;
            break;
          case DONE:
            free(item->chanTrans);
            free(item);
            break;
      }
    }
    return NULL;
}


int tm_init(int workers){
    first = NULL;
    last = NULL;
    int i;
    for(i = 1; i <= workers; ++i){
      pthread_t t;
      if(pthread_create(&t, NULL, worker, NULL)){
        fprintf(stderr,ERRFIX"Could not create required number of threads. (Failed on %d
            of %d) (%s:%d)\n", i, workers, __FILE__, __LINE__);
        return -1;
      }
    }
    return 0;
}

int tm_createNew(struct FunStruct *funStruct, int *chanTrans){
    struct InstanceStruct *instance = malloc(sizeof *instance);
    instance->funStruct = funStruct;
    instance->memPtr = NULL;
    instance->chanTrans = chanTrans;
    instance->step = 0;
    instance->state = NEW;
    addToQueue(instance);
    return 0;
}

int tm_requeue(struct InstanceStruct *instance){
    addToQueue(instance);
    return 0;
}
```