

Documentation

January 10, 2005

Bowling Green Time Warp  
BGTW<sup>++</sup>

Max Ehammer

[ecmax@bgnet.bgsu.edu](mailto:ecmax@bgnet.bgsu.edu)

Harald Roeck

[hrock@bgnet.bgsu.edu](mailto:hrock@bgnet.bgsu.edu)

Academic Supervisor

Hassan Rajaei, Ph.D

Department of Computer Science  
Bowling Green State University, Ohio

# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	Purpose . . . . .	3
1.2	Objectives . . . . .	3
1.3	Background . . . . .	4
<b>2</b>	<b>Related Work</b>	<b>6</b>
<b>3</b>	<b>Project Description</b>	<b>8</b>
3.1	Project Phases and Efforts . . . . .	8
3.1.1	Research . . . . .	8
3.1.2	Design . . . . .	8
3.1.3	Implementation . . . . .	8
3.1.4	Documentation . . . . .	9
3.2	Design and Methods . . . . .	9
3.2.1	User Interface . . . . .	11
3.3	Implementation and Interfaces . . . . .	11
3.3.1	Process . . . . .	11
3.3.2	Memory Manager . . . . .	13
3.3.3	Msg . . . . .	13
3.3.4	Message Handler . . . . .	14
3.3.5	GVT . . . . .	16
3.3.6	LP and Time Warp . . . . .	16
3.3.7	Random Numbers . . . . .	21

---

3.3.8	Global functions and global variables . . . . .	21
3.4	Problems encountered and Resolving Methods . . . . .	21
3.4.1	Synchronization . . . . .	21
3.4.2	Memory management . . . . .	22
3.4.3	Message Handling and Threads . . . . .	22
<b>4</b>	<b>Simulation Examples, Results and Analysis</b>	<b>24</b>
4.1	Simple Ball Game . . . . .	24
4.2	Analysis . . . . .	26
<b>5</b>	<b>Proposal for Future Work</b>	<b>30</b>

# Chapter 1

## Introduction

### 1.1 Purpose

The purpose of the BGTW project was (and is for future development) to study existing approaches of distributed simulation implementations for distributed memory as well as for shared memory architectures and eventually end up with a own engine (which provides methods for distributed simulation) called BGTW written in C++. This engine is based on a distributed memory architecture and uses the communication library MPI. Gaining a in depth knowledge of the existing approaches should cause enough proficiency to design and implement a engine which provides an appropriate interface for user-applications. The factor of experience and learning was a big issue on this project too!

### 1.2 Objectives

The goal of this project was to develop a small kernel which provides the basic operations needed by most of the existing approaches for distributed simulation. Upon the kernel modules should be used to enhance the functionality of the BGTW engine. As the name of the project already tells, the first phase of this project should support the time warp mechanism developed by Jefferson [10]. Later on it should be easy to add on different methodologies which could be either used by the user-application or by the kernel itself. This would be for instance a different GVT calculation, ways of doing the memory management or a different synchronization approach. Further on the final draft of this document should provide sufficient understanding of the design and implementation of the BGTW engine, hence other students can enhance the existing code and make experiments with their own

approaches and finally compare their results to the well known algorithms in distributed simulation. Since the general tenor of the PADS community is - and of course was - to improve the low level mechanisms of a distributed simulator and hence get more efficiency out of the engine(e.g. [7]), we didn't pay to much attention on this topic. As this engine provides the possibility to add modules with different approaches the usability and functionality gains, the payoff is clearly performance.

## 1.3 Background

Parallel discrete event simulation(PDES), or simply distributed simulation, is a single discrete event simulation running on a parallel computer. This is a simple definition; however, to run a simulation in parallel requires more effort.

"A computer simulation is a computation that models the behavior of some real or imagined system over time"[8]. The reason to run a parallel simulation is to reduce the execution time in comparison to a sequential simulation. Large simulations in engineering, computer science, economics, and military applications require a huge amount of time to finish. Although computers are getting faster and more powerful everyday, so are the problems and systems we are interested to simulate.

The good thing is, a discrete event simulation consists of potential parallelism, but interestingly it is very difficult to parallelize a simulation. Because researchers, especially computer scientists, love difficult problems, the parallel processing community started working on that very early. Although the first algorithms were already published 25 years ago, to develop an efficient distributed simulation still requires expertise and effort. There is no golden rule or a general purpose solution to all simulation problems. All algorithms are based on several restrictions and are good for one simulation but not for another. Additionally, the algorithms have to fulfill at least one important requirement, that both, the sequential and the parallel version, produce the same results.

Independently, Chandy and Misra [4], and Bryant [3] developed some of the first PDES algorithms. These algorithms process only save events. Today this approach is known as *conservative*. The basic idea is that each process or task has to determine that there is no other event with a smaller time stamp before processing an event. Jefferson [10] introduced a new technique named Time Warp. Related algorithms to this approach are called *optimistic*. The reason is that all events are blindly processed without checking their global order. If a process receives an event out of order, it has to roll back the wrong computation. After a roll back it starts to process the new received event and continues to reprocess the rolled back events once again under new conditions.

These are different world views with advantages and disadvantages on each side. In the last decade, new algorithms have been developed which combine these approaches(e.g. [13]) to get a more flexible and general distributed simulation engine.

## Chapter 2

### Related Work

In [7] the Georgia Tech Time Warp (GTW) is presented. This is one of the fastest known Time Warp implementations. This engine is designed to run on a shared memory machine, although an implementation for distributed memory is available. The algorithms used in this engine take advantage of shared memory to increase performance. The GVT computation (for details see [9]) uses global flag variables to calculate the new GVT. Access to this global memory is controlled by a lock. Because of the use of shared memory, no special GVT event or message acknowledgment is needed. Message sending is also done by passing pointers instead of copying memory. This approach implies the algorithm to cancel a message or to send an anti-message: by passing a pointer and connecting the messages through a certain data-structure. Additionally the engine uses a buffer manager which is responsible for memory allocation to create new events. The atomic unit is called buffer. A buffer is a fixed amount of storage and contains an event, checkpointed states and additional pointers to maintain several data-structures. The original Time Warp [10] proposed three distinct data-structures; this implementation uses a single queue. In addition to this single queue, each LP needs three distinct data structures: a message queue, a message cancellation queue, and an event queue.

Parsec [2] discusses a different simulation engine developed at UCLA. In comparison to the above introduced engine GTW, which uses only the Time Warp algorithm, this engine is much more flexible. Moreover, it allows the combination of optimistic and conservative sub-models in one PDES. Consequently, parsec is a huge environment with its own programming language as user interface. It translates the user application to a C program which is then compiled to a machine code. Internally, parsec is based on several modules and layers. On top of all them is the programming language which interacts with the virtual time synchronization layer. This layer uses several different modules which implement the synchronization approach. One of these modules is the sequential runtime environment that is used

by a sequential simulation. All others are parallel modules using the Portable multithreaded communication library to perform the communication between the entities. The bottom layer is composed of several system libraries, e.g. MPI, Pthreads or PVM.

Parsec is a general purpose simulator engine. On the other hand Glomosim [14] is a simulator written in parsec for wireless networks. Glomosim provides several different libraries that help the user to implement a simulation of a wireless network. This is a development to satisfy the upcoming need of wireless networks and their testing. With these libraries, it is easy to simulate and test a new wireless network before even purchasing the necessary hardware.

In addition to these popular simulator engines, some smaller environments exist; one of them is Sima [12]. Sima is a experimental environment for a shared memory system and basically provides a set of C library routines. The used algorithms are Conservative Time Windows(CTW) and Three Phase Algorithm(TPA); both are conservative schemes.

In the mid nineties, the Department of Defense realized that there are several existing simulators, each with advantages and disadvantages, that were not able to interact. In response, the High Level Architecture [6, 5] was developed. This is an approach to connect several different existing simulators to get a new, bigger, and more powerful simulation. The layer or middle-ware is called RTI, Runtime Infrastructure. Different simulators and real objects, as well as user interactions, are managed through this layer. An entity connected to the RTI is called federate, and a set of federates together with the RTI is a Federation. To become a federate, an existing simulation has to implement several different well defined interfaces. These interfaces are necessary to enable the communication with other federates. As a result, a HLA simulator is more flexible, and it is possible to reuse existing simulations.

# Chapter 3

## Project Description

### 3.1 Project Phases and Efforts

The project started in the spring semester 2004. After we attended a Distributed Simulation class we had to get familiar with the latest developments and related work. At the end of the semester we got the hardware for the new P4 cluster, hence we focused our work on setting up and configuring the cluster, and we restarted the work on this project by the beginning of the summer term.

#### 3.1.1 Research

The study of existing approaches took a lot of effort. The thought of HLA compliance made the whole story much more complicated. As HLA is a huge standard we soon figured that this might be beyond our current capabilities and our existing time window. Beside many papers we concentrated on the few mentioned above.

#### 3.1.2 Design

The design was done more or less twice. After the first draft and succeeding review many things were changed. The details will be discussed in "Design and Methods".

#### 3.1.3 Implementation

This phase took us about 2 months. After starting programming we figured many flaws in the design. The implementation phase itself was full of surprises, as this

project is using multiple threads and multiple processors with distributed memory, you encounter many problems which you don't think of in advance (if you don't have the proper experience - which we apparently didn't have). A compiled list of encountered problems will be found in "Problems encountered and Resolving Methods". The details of the implementation are discussed in "Implementation and Interfaces".

### 3.1.4 Documentation

The documentation and last phase of this project gets the smallest amount of time. But we hope this document gives you a good understanding of what we did for our project.

## 3.2 Design and Methods

The engines discussed in chapter 2 are either too low-level or too huge to completely understand. GTW is very hard to maintain, and experiments with this engine would be very hard and error prone due to the high performance coding strategies. On the other side, parsec has a nice modular design which allows extensions, but the fact that it uses its own language introduces a new field in computer science: compiler technology.

Our engine should have a small, well defined, and easy understandable user interface. The user of this engine may not be familiar with parallel programming at all. Additionally, extensibility of the engine was requested. It should be extensible by other students to test new synchronization algorithms or other PDES algorithms.

These requirements are to meet, therefore, we decided to work on a modular design. There is a small kernel running on each node, called the `Process`. To ensure that only one process is running on a node we use the Singleton design pattern. The process knows all other processes in the simulation and additionally knows where the user defined LPs are running. Each process is responsible for a distinct subset of the user LPs; the mapping is statically for now. The process uses basically four other modules to delegate the work to: a `GVT`, a `Message Handler`, a `Memory Manager` and a `Queue`. Additionally, we define a `Thread` class to ease pthread handling and we define our own simulation time type, called `Simtime`. A complete class diagram with the names of the classes can be found in figure 3.1. All classes and other code is defined within the namespace `BGTW`.

The `GVT` module is responsible for calculating the lower bound of time stamp. The process doesn't know the algorithm. When a `gvt` message is pushed into the

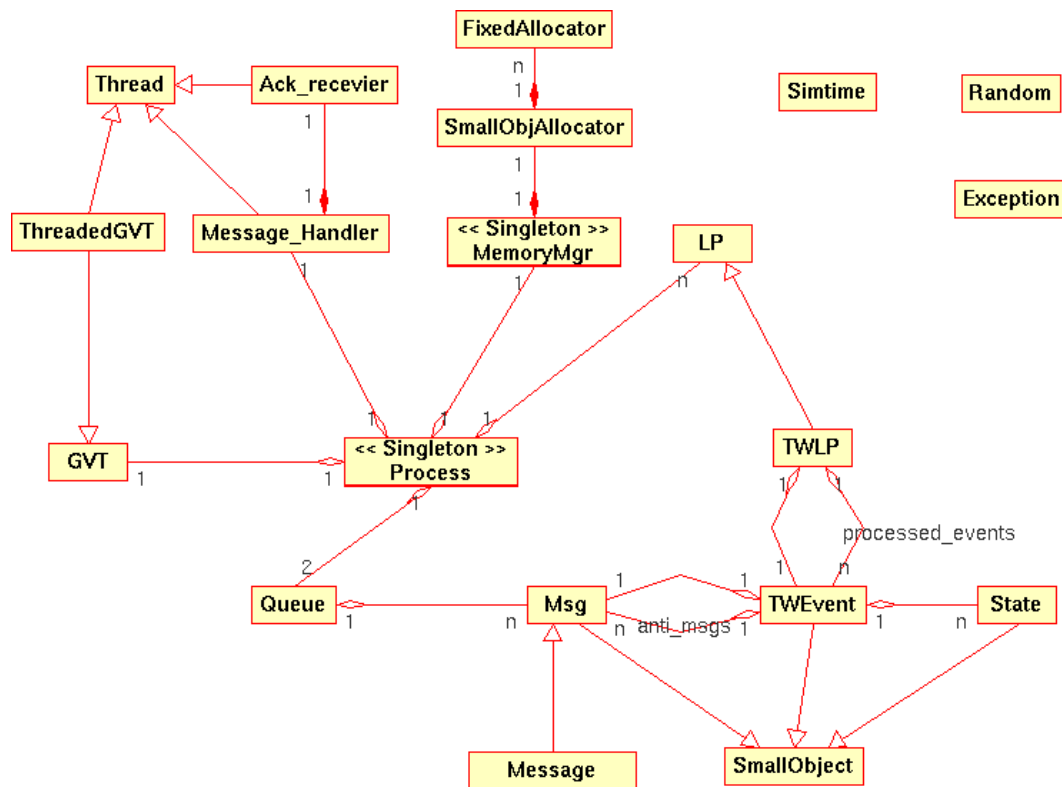


Figure 3.1: Complete class diagram

queue the process takes the message and sends it to the GVT module. When the gvt module returns the new gvt time all LPs in this process are informed to advance to that time and the process continues.

The Message Handler provides an interface to send messages, invoke barriers and to initialize the underlying communication library. We are focusing on MPI, hence it is basically an abstraction of the necessary MPI calls.

The Memory Manager is an allocator for small objects as described in [1]. All objects derived from `SmallObject` are allocated and deleted by the memory manager. Additionally, the memory manager provides the functions `alloc()` and `free()` to allocate or delete memory blocks.

The Queue is derived from the `priority queue` of the standard template library, but it stores a pointer to the object instead of the object itself.

### 3.2.1 User Interface

The messages are represented by the `Msg` class. This is the base class for the user defined message types. The template `Message` derives from `Msg` and the user defines its Messages by using this template with his/her payload class.

To define an LP the user derives their class from an abstract LP class, e.g. `TWLP`, and implements the virtual functions `run()`, `init()`, and `finish()`. The first is called when a message for that LP is processed and the latter two are to initialize and finalize the LP.

The user has to write the `main` method where the process is initialized and the LPs are registered to the engine. The simulation starts by calling `simulate()` of the process. Finally, `finalize()` must be called before exiting the program. The output is written to files, one for each process. Optionally a debug output is produced to trace the internals of the simulation engine.

## 3.3 Implementation and Interfaces

In this section we describe the details of the design and the implementation.

### 3.3.1 Process

The process class (figure 3.2) has several public methods, some are called by the user (`init()`, `finalize()`, `register_LP()`, and `simulate()`) whereas the rest is for internal use by different modules. This class is a Singleton, therefore the constructor is protected and a static function `instance()` is used to get a pointer to this class.

The process contains all unprocessed events for this node. We use two queues: an event queue for user messages and an event queue for system messages (e.g. `gvt msg`). We distinguish between messages with type numbers. The user is free to use positive numbers smaller than 127, and the engine uses negative numbers for system messages.

Every process has a vector of `lp` pointers, which holds pointers to all registered LP objects in the simulation. Additionally, we use two maps, `lpmap` and `mapped_lps`. The first one takes an LP id and returns the id of the responsible process, and the second one takes an LP id of an LP in this process and returns a link to that LP.

The simulation is started by calling `simulate()`. It starts off by mapping the LPs to the available processes (`mapping()`), starting the Message Handler thread, initializing the LPs mapped to this process (`init_mapped_lps()`), invoking the sched-

Process	Simtime
- gvt : ThreadedGVT *	- simtime : float
- lp_count : int	+ !=(rhs : const Simtime &) : bool
- m_event_queue : Queue	+ !=(rhs : const double &) : bool
- m_instance : Process *	+ +(_s : Simtime) : Simtime
- m_rank : int	+ +(_s : double) : Simtime
- m_simtime : Simtime	+ ++() : Simtime
- m_size : int	+ ++(x : int) : Simtime
- m_sys_event_queue : Queue	+ -( _s : Simtime) : Simtime
- msg_handler : Message_Handler *	+ -( _s : double) : Simtime
- all_lps : std::vector<LP*>	+ --() : Simtime
- mapped_lps : std::map<int,LP*>	+ --(x : int) : Simtime
- lpsmap : std::map<int,int>	+ <(rhs : const Simtime &) : bool
# =( : const Process &) : Process &	+ <(rhs : const double &) : bool
# Process()	+ <=(rhs : const Simtime &) : bool
+ finalize() : void	+ <=(rhs : const double &) : bool
# finish_mapped_lps() : void	+ =( _s : Simtime) : Simtime
+ init(argc : int *, argv : char * * *) : void	+ =( _s : double) : Simtime
# init_mapped_lps() : void	+ ==(rhs : const Simtime &) : bool
+ instance() : Process *	+ ==(rhs : const double &) : bool
# mapping() : void	+ >(rhs : const Simtime &) : bool
# min_time() : Simtime	+ >(rhs : const double &) : bool
+ num_lps() : int	+ >=(rhs : const Simtime &) : bool
+ pop() : Msg *	+ >=(rhs : const double &) : bool
+ push(msg : Msg *) : void	+ Simtime()
+ rank() : int	+ Simtime(_s : double)
+ register_LP(lp : LP *) : void	+ value() : double
# scheduler(simtime : Simtime) : void	
+ send(msg : Msg *) : void	
+ simtime() : Simtime	
+ simulate(simtime : Simtime) : void	
+ size() : int	

Figure 3.2: Details of the Process and the Simtime class

uler (scheduler()), and eventually calling the finish function of the mapped LPs (finish\_mapped\_lps()). The scheduler runs in a loop until the gvt exceeds the specified value. It takes the next event from the queue by calling pop(). If the message is a gvt message it will be forwarded to the gvt module which computes the new gvt value, and all LPs are informed about the new gvt value. Otherwise it is a regular user message, hence process\_msg() of the correct LP is called.

The procedures push() and pop() are used to insert or to abstract messages into or from the queues. Both use mutex locks to access the queues, because they could be called by different running threads. pop() will block if no messages are in the queues. The send() function is used by an LP to send new messages to another LP.

The Simtime class is basically a wrapper around a float value. We overload nearly all operators to our preferences.

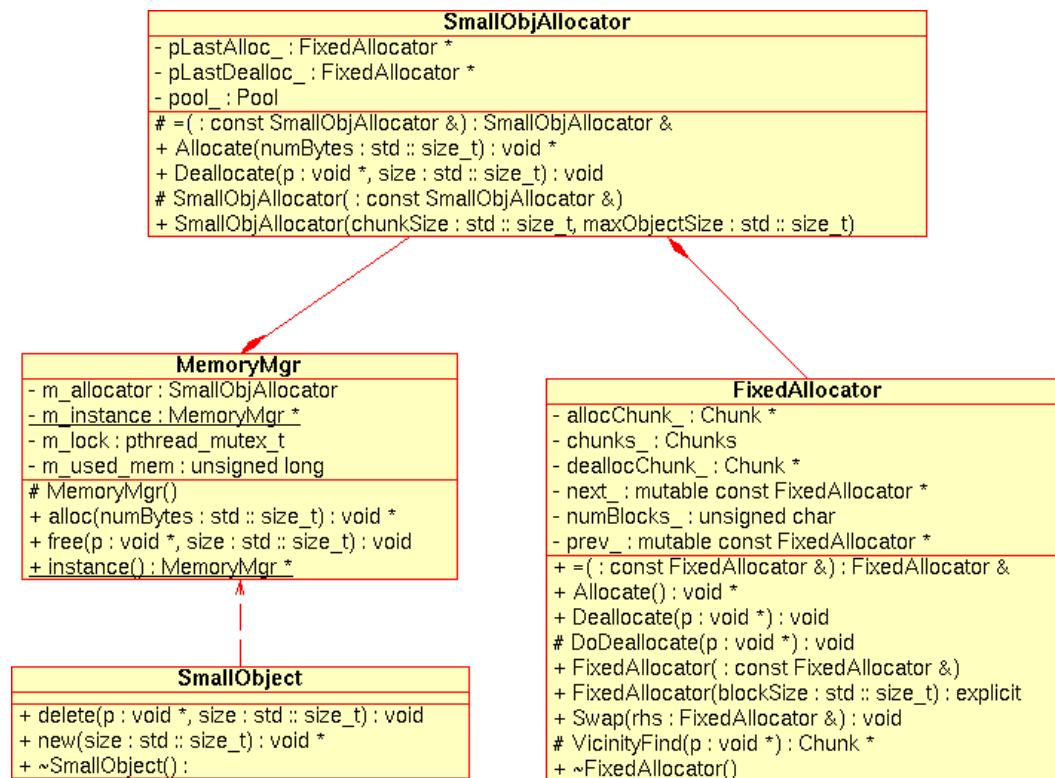


Figure 3.3: Details of the Memory Manager

### 3.3.2 Memory Manager

The design of the Memory Manager (`MemoryMgr` figure 3.3) is extracted from [1]. For details about the `SmallObjectAllocator` and `FixedObjectAllocator` we refer to the book. The memory manager is a Singleton to ensure that only one object is present at each node. It provides only two functions `alloc()` to allocate memory and `free()` to free a memory block. Both of these functions use a mutex to guarantee thread safety. The `SmallObject` class is a base class with overloaded operators `new()` and `delete()`. All objects derived from this class are automatically generated and destroyed by the memory manager. Classes derived from `SmallObject`: `Msg` and hence all user messages, `TWEvent`, `State`.

### 3.3.3 Msg

The `Msg` class (figure 3.4) represents the base class and the header for all messages in the engine. All user messages and system messages are derived from `Msg`. The template `Message` saves a user type `_T` as data. The `==` and the `<` operator are over-

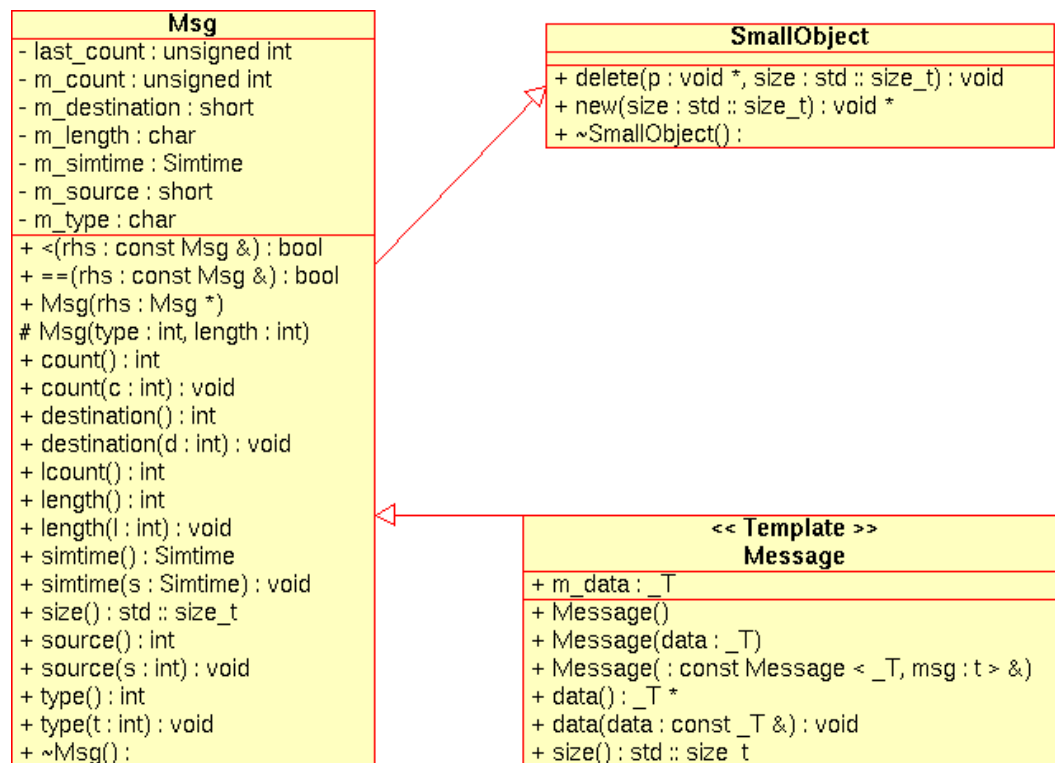


Figure 3.4: Details of the Msg class

loaded to implement priorities between different messages. System messages have the highest priority with a type value smaller than zero, then there are anti messages at the next level and finally normal user messages. Within a level the simtime is used to compare two messages. Two messages are equal if they have the same type, count, destination and source. All messages created at the same process have a unique id, called count. This is necessary to distinguish messages which are resent after a rollback. All the code for Msg is in the header file `msg.h`.

### 3.3.4 Message Handler

The Message\_Handler (figure 3.5) is basically used for sending and receiving messages from and accordingly to other processors. The Message\_Handler uses two threads and therefore derives from the class Thread which provides a virtual `void run()` function. This function must be implemented by the user class. The first thread is used for receiving messages therefore this thread is constantly listening - polling - if messages sent over the network belong to it. The `run()` method is implemented by the Message\_Handler class. The second thread is used for acknowl-

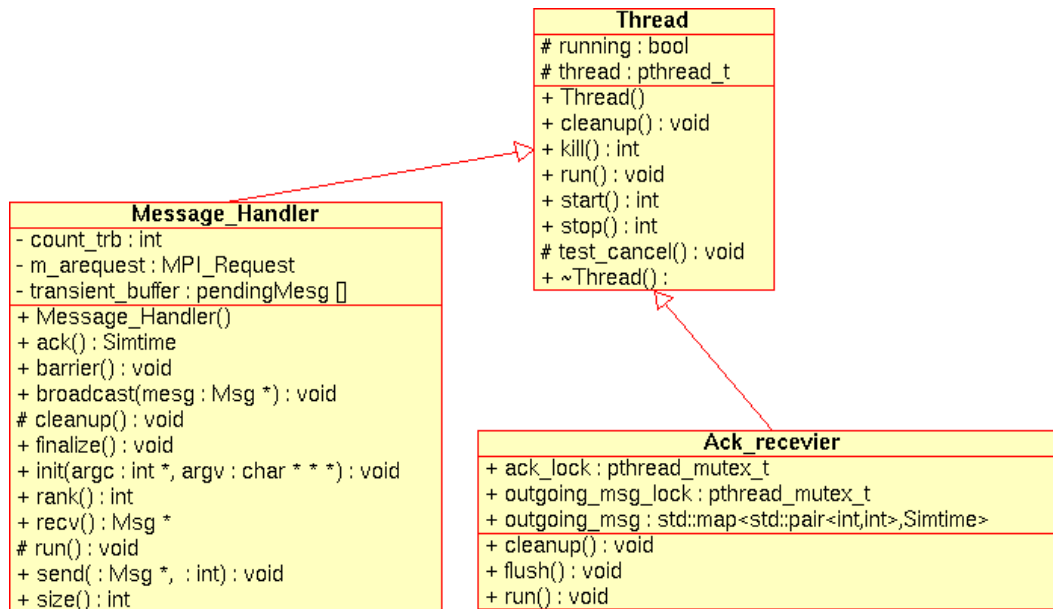


Figure 3.5: Details of the Message\_Handler class

edging sent messages. The `run()` method is implemented by the `Ack_recevier` class which is an inner class of the `Message_Handler` class. The `init(int* argc, char*** argv)` command initializes the MPI environment and sets up the transient messages buffer. The `size()` and `rank()` commands are just wrappers around the original MPI calls. The `ack()` returns the lowest time stamp of unacknowledged messages therefore the entries in the `std::map` `outgoing_msg` are checked. This map is maintained by the second thread of the `Message_Handler` class which is responsible for acknowledging messages.

One of the main routines of the `Message_Handler` class is the `send(Msg* mesg, int destination)` function. The `send` method is handled asynchronously, this means there is no blocking and we have explicitly to ask if the message has been delivered yet or not. The message which should be sent is first pushed into the transient message buffer which is capable of holding `TRANSIENT` (actually set to 20) pointers to the corresponding messages. If the buffer is full the process is blocked until the oldest message (in a manner of first in last out) has been delivered successfully and therefore takes the place of the old message. After the message has been pushed into the buffer it will be sent over the network. If the message type is greater than zero (then it's a user message) the message will be registered with its `count`, `length` and `simtime` to be acknowledged after received by the destination.

The `barrier()` and `broadcast(Msg* mesg)` calls are again just wrappers, they belong to the corresponding MPI calls.

The `void Message_Handler::run()` method is as already said responsible for receiving messages. The receive method is as the send method handled asynchronous. After receiving a message the size of the message is determined and adequate memory is allocated. The memory allocation is done through the `MemoryMgr` class which provides sufficient small junks for storage. Through `memcpy((void*)p, (void*)recv_buffer, recv_size)` the data is copied to the main memory of this machine and a pointer is pushed into the receive queue of the `Process` class. After the message is pushed into the proper queue an acknowledgment is sent to the sender if the message type was a user message.

The second thread of the `Message_Handler` class is the `void Message_Handler :: Ack_receiver :: run()` method which basically just receives the acknowledgments for sent user messages and erases the corresponding requests for acknowledgment out of the `outgoing_mesg` map.

The `finalize()` method ends the running threads properly and calls the `mpi` routine `MPI_Finalize()`.

Note! As this is multithreaded each `mpi` call has to be locked and especially within the `run()` method - actually the thread itself - the possibility of canceling the running thread during a `mpi` routine has to be disabled, otherwise processes might have troubles cleaning up the `mpi` environment through `MPI_Finalize()` as they wait for the thread to finalize which has already been canceled.

### 3.3.5 GVT

GVT is an abstract class and defines only the interface to calculate the new gvt value. `ThreadedGVT` is derived from `GVT` and `Thread` (figure 3.6). It implements a simple gvt algorithm: it blocks the process, calculates the minimum of the process by calling `min_time()` and invokes an allreduce to distribute the new gvt. Additionally, a thread on node zero is pushing a gvt message into the process every 100ms.

Calculating the minimum of a process, involves several steps. First all anti messages are processed, then the time stamp of the next message to process is calculated. The message handler is asked for the timestamp of the last unacknowledged message. The smaller value of these two is the lower bound of timestamp in this process.

### 3.3.6 LP and Time Warp

The `LP` and `TWLP` are abstract classes (figure 3.7). The user derives their class from `TWLP` to implement a time warp simulation. The user is responsible to implement the three abstract functions `run()`, `init()`, and `finish()`. An `LP` must have the

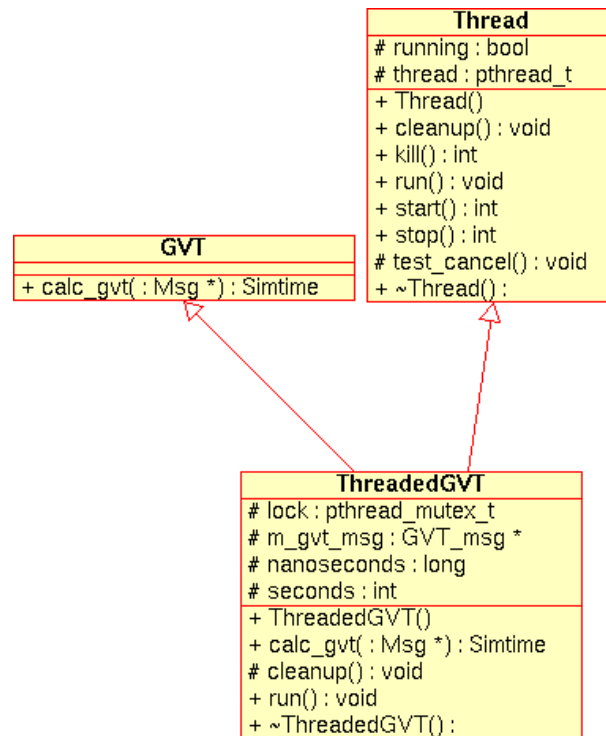


Figure 3.6: Details of the GVT and ThreadedGVT class

following functions which are called by the process:

**void initialize()** to initialize the lp; calls the user function `init()`.

**void finalize()** to finalize the lp; calls the user function `finish()`

**Simtime advance(Simtime)** to inform the lp about the new minimal timestamp or gvt value.

**void process\_msg(Msg\*)** sends the lp a new message to process; calls the user function `run()`.

Additionally, there are the following functions which are called by the user:

**Msg\* recv()** returns the message which started this event

**void send(Msg\*, int id, Simtime)** sends a message to the LP `id` at the specified `simtime`

**Random\* randgen(int)** returns a new random number generator

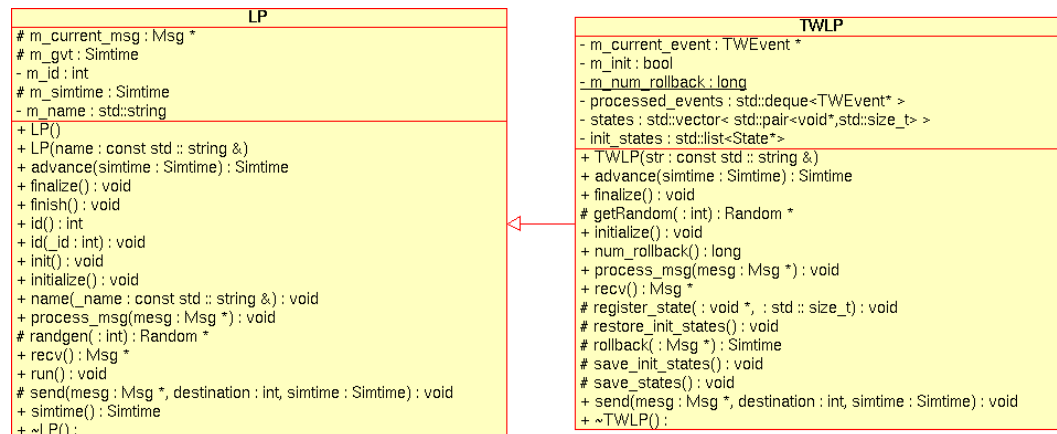


Figure 3.7: Details of the LP and TWLP class

An implementation of an LP is TWLP. It implements the functions mentioned above and an additional function called by the user: `void register_state(void*, std::size_t)`. It is necessary to register the state variables to the TWLP, because we have to be able to rollback wrong computations. This function takes a pointer to a memory location and the size of that memory block which will be saved to restore later on.

Internally the TWLP has several private functions:

**Simtime rollback(Msg\*)** is called when the LP receives a message with timestamp smaller or equal to the last processed message. It takes this message as argument.

**void save\_states()** makes a copy of all registered state variables and connects them to the current event. It is called after the user function `run()`.

**void save\_init\_states()** saves the initial state variables

**void restore\_init\_states()** restores the initial state variables

The necessary data structures for these functions are:

**std::deque<TWEvent\*> processed\_events** holds the processed events

**std::vector<std::pair<void\*,std::size\_t>> states** stores the registered state variables addresses and lengths

**std::list<State\*> init\_states** the initial states of the TWLP

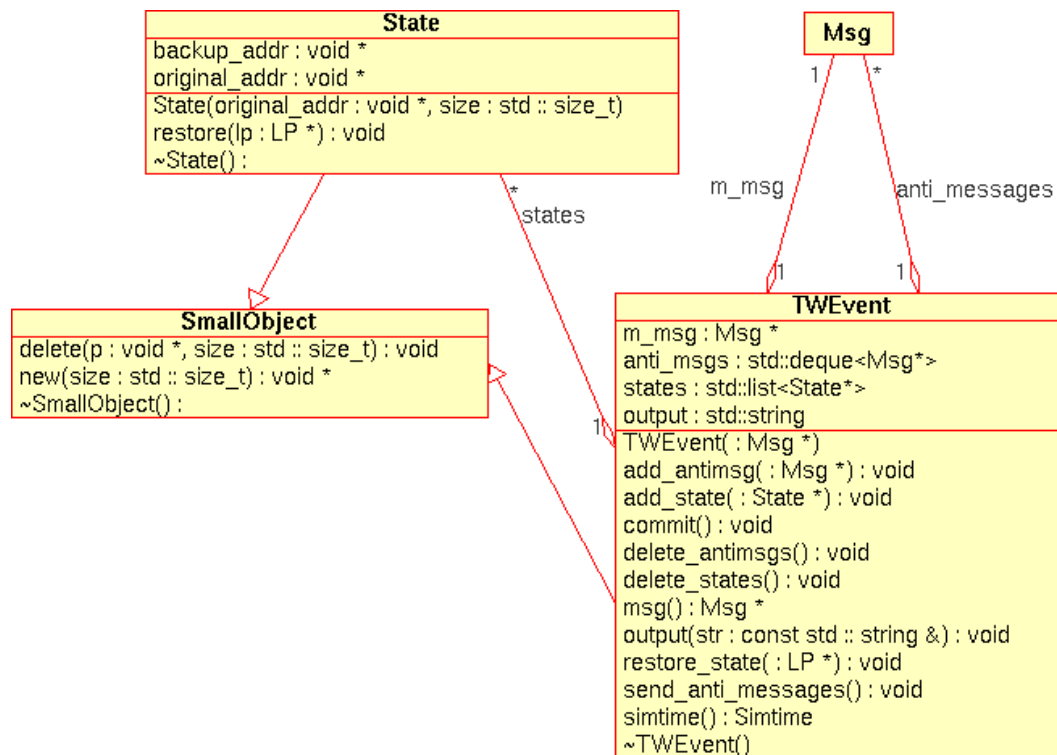


Figure 3.8: Details of the TWEvent class

Closely connected to the TWLP is the class TWEvent and the class State (figure 3.8). TWEvent is a wrapper around a received message with additional data structures necessary to save information about that message, to safely rollback the user computation. State stores the address of a memory block, the size of the memory block and the address of the backup copy of that block. It provides one function which copies the memory from the backup address to the original address.

The TWEvent has to save a copy of all messages (anti messages) which are sent during processing this event and it has to save the status of the TWLP after processing the event. The status of a TWLP is the set of all registered state variables and the output string. TWEvent provides several functions, which are all called by the TWLP.

**Msg\* msg()** returns the message which initiated this event

**void commit()** commits this event, hence writes the output and deletes the message for this event

**void send\_anti\_messages()** sends all anti messages; called during a rollback

**void restore\_state(LP\*)** restores all state variables of an LP

**void add\_antimsg(Msg\*)** connects an anti message to the event

**void add\_state(State\*)** inserts a state to the event

The anti messages and states are deleted when an object is destroyed. The message which initiated this event is deleted during the commit function.

### Time Warp

There are two points which must be considered in a time warp system. Firstly, all changes between processing two events must be saved, and secondly, we have to be able to rollback these changes. We are accomplishing that with the `TWEvent` class. When the `process_msg()` function from the `TWLP` is called we create a new `TWEvent` object with the received message and call the user function `run()`. When the user sends messages we create anti messages, copies of the original messages, and connect them to the current event. When the user pushes something in the `cout` stream we save that in a string variable. After the user is finished with their computation we take a copy of all registered variables and the output string and attach them to the current event too. Finally we push the event into the `processed_events` list.

If the timestamp of a new message is smaller or equal as the timestamp of the last processed message we rollback until it is safe to process the new message. The `rollback()` function takes all messages with timestamps greater than the new message, calls `send_antimessages()` for each of them, pushes the messages of the events back into the process, and deletes the `TWEvent` objects. Then it takes a closer look to all messages with the same timestamp as the new one, because one of them could be a corresponding anti message. If a corresponding message is found all events to this message are rolled back too, the corresponding message and the new message are deleted and the internal variables are set to `NULL`. Finally the status of the `TWLP` is restored by calling `restore_state()` of the last event in the processed events list.

Events are committed when the `TWLP` gets informed about the new `gvt` value. The process calls `advance()` of all `LPs` to push the computation forward. `advance()` in a `TWLP` takes all processed events with timestamps smaller than the new value, calls `commit()` for each of them, and deletes the objects, hence deletes all anti messages and saved states connected to these objects. More precisely it doesn't commit all events, it leaves the one event which was processed just before the new `gvt` time in the list, so that we are able to rollback to `gvt` time. This is necessary because the state of the `LP` at simulation time `gvt` is connected to that event.

### 3.3.7 Random Numbers

The random number generator used in our engine is implemented by Agner Fog. It is open source and available free of charge at <http://www.agner.org/random/>.

### 3.3.8 Global functions and global variables

In the namespace `BGTW` are some additional global variables and functions defined. The global declarations are in the file `defines.h` and the implementation is in `defines.cpp`.

The variables are:

`std::ofstream cout` to overload the standard cout stream

`bool debugFlag` a boolean variable to enable debug output

`std::ofstream debug` the stream for the debug output

`pthread_mutex_t mpi_lock` a lock for the mpi calls

The functions are:

`void setDebug(const std::string&)` sets the `debugFlag` to true and the name of the file where debug output is stored.

`void setOutput(const std::string&)` sets the name of the output file where the standard output is saved.

**overloaded stream operator <<** the stream operator is overloaded for the `Simtime` class and for pointers to messages and for pointers to LPs.

## 3.4 Problems encountered and Resolving Methods

### 3.4.1 Synchronization

As this software is using multiple threads and is running on multiple CPUs you run into several synchronization issues you should be aware of. Each object which is accessed by any thread has to be locked through a `pthread_mutex_lock` before using this specific object afterwards it has to be unlocked through `pthread_mutex_unlock` hence other threads can use this object. This has more or less the same value for

each MPI call you make as the suspension of a running thread while executing a MPI routine may mess up the whole system. Therefore you have to use again a `pthread_mutex_lock` before executing any MPI call and afterwards release the resource through `pthread_mutex_unlock`. These locks belong to different objects and hence are named differently. One important issue which belongs to this topic is the cancelation of threads. At the end of each mpi program the call `MPI_Finalize()` cleans up. A thread which was canceled during a mpi routine messes up the whole system. As now every other process is waiting for the canceled thread to finalize, but never will. To prevent this each thread has to disable the cancel option before executing a mpi routine, afterwards the cancelation option has to be turned on again.

### 3.4.2 Memory management

It is important to be aware when and where memory is allocated and deleted in the engine. Internally we pass only pointers between the different modules. Especially the message handling is very difficult, because a message can be created by the message handler which is fine because no other LP in this process knows about that message. The problem is with messages created in the same process. These messages are generated by an LP in the same process and these LPs could keep a pointer to that message, and hence could end up with a dangling pointer, because the receiver LP is finally responsible for deleting the processed messages. To understand the policy that the receiver LP deletes the messages is very important. When the user is processing an event it is possible to send the message which initiated this event, but this will probably crash the engine. The user is responsible for making a copy of the message and send the copy if he/she wants to send the same message again. Messages sent through the Message Handler are deleted by the message handler.

The problem of when and where deleting saved states and anti messages is taken care by the `TWEvent` class. When a `TWEvent` object is deleted all states and anti messages connected to that object are deleted.

### 3.4.3 Message Handling and Threads

Message acknowledgment is another point we didn't consider in the first design. We were trusting the MPI layer, but this is not enough. The receiver has to acknowledge all messages to the sender when the message is pushed into the process. The MPI layer considers messages which are at the receiver node as received, but this does not mean that the messages are in the engine, or more precisely in the process, they could be in the MPI buffer at the receiver node. Hence, we have to acknowledge

each message explicitly. It's enough to send the `id(count field)` and the length of the received message back to the sender. Within the message handler is a small thread which polls these messages from the network and deletes the entries in the sending buffer. This is important to compute a correct `gvt` value.

Another point is that an MPI implementation may not be thread safe, but we use several threads internally. Therefore, a global mutex variable called `mpi_lock` is necessary to control all MPI calls. To prevent deadlocks no blocking calls are used throughout the engine.

## Chapter 4

# Simulation Examples, Results and Analysis

### 4.1 Simple Ball Game

The following small simulation sends pairs of messages in a circle; one clockwise and the other counter clockwise.

The first step is to define our message. In this example the message is defined at lines 005 – 007. Followed by the LP declaration at lines 010 – 021. The mandatory implementation of the three functions `init()`, `run()`, and `finish()` are at lines 023–078. The user has to write the main method. This example demonstrates a typical main function. First the process is generated, then the output file and the debug file are created. The next statement at line 088 initiates the simulation. This must be done before any other statement. The following lines show different ways to register LPs to the simulation kernel. When all LPs are registered, the simulation is started by executing the `simulate()` function. Finally, `finalize()` is called to clean up the simulation and to close all open files. The simulation won't print any output to the console, because the output is redirected to files, one for each process. The `setDebug()` function is used to trace the simulation run or to debug the kernel; it creates one file per process.

```

001 #include "BGTW/bgtw.h"
002 using namespace BGTW;
003
004 // define my msg -- ball
005 struct mydata{ int a,b; };
006 #define MYDATA 13
007 typedef Message<mydata,MYDATA> my_msg;
008 // -----
009 // my lp declaration
010 class my_lp : public TWLP
011 {
012 public:
013   my_lp(const std::string& str) : TWLP(str)
014   {
015     virtual void run();
016     virtual void init();
017     virtual void finish();
018 private:
019     int count;
020     Random * rand;
021 };
022 // -----
023 void my_lp::run()
024 {
025   my_msg* mmsg = static_cast<my_msg*> (recv());
026
027   // number of balls received
028   ++count;
029
030   // this will be written to a file
031   cout << " got ball " << mmsg
032   << " " << (mmsg->data()->a)
033   << " count: "
034   << count
035   << " rounds: "
036   << (mmsg->data()->b)
037   << endl;
038
039   if(id()==0)// count the number of rounds
040     mmsg->data()->b++;
041
042   // one ball clock wise
043   if(mmsg->data()->a == 0)
044     send(new my_msg(static_cast<my_msg*>(*mmsg)),
045          (id()+1)%NUM_LPS(), simtime() +0.2);
046   // the other counter clock wise
047   else if(mmsg->data()->a == 1)
048     send(new my_msg(static_cast<my_msg*>(*mmsg)),
049          (id()+NUM_LPS()-1)%NUM_LPS(), simtime() +0.2);
050 }
051 // -----
052 void my_lp::init()
053 {
054   cout << " number of lps: " << NUM_LPS() << endl;
055   if(id()%100 == 0)
056   { // play the first balls
057     my_msg* mmsg = new my_msg();
058     mmsg->data()->a = 0;
059     mmsg->data()->b = 0;
060     send(mmsg, id(), Simtime(0.1));
061     cout << " scheduled first ball " << endl;
062     mmsg = new my_msg();
063     mmsg->data()->a = 1;
064     mmsg->data()->b = 0;
065     send(mmsg, id(), Simtime(0.2));
066     cout << " scheduled second ball " << endl;
067   }
068   // count is state variable
069   REGISTER_STATE(count);
070   // this returns a random number generator
071   rand = getRandom(123*id());
072 }
073 //
074 // -----
075 void my_lp::finish()
076 {
077   cout << " game finished " << endl;
078   cout << " got the ball " << count << " times " << endl;
079 }
080 // -----
081 int main(int argc, char** argv)
082 {
083   Process* process = Process::instance();
084   // init the simulation
085   process->init(&argc, &argv);
086   // open a file for debug output
087   setDebug("debug");
088   // open a file for stdout
089   setOutput("output");
090   // different ways to register lps
091   process->register_LP(new my_lp("first"));
092   process->register_LP(new my_lp("second"));
093   process->register_LP(new my_lp("third"));
094   process->register_LP(new my_lp("fourth"));
095   REGISTER_LP(my_lp);
096   // same as process->register_LP(new
097   my_lp("my_lp"))
098   REGISTER_LPS(my_lp,10);
099   // same as
100   // for(int i=0;i<15;++i)
101   // process->register_LP(new my_lp("my_lp"));
102   // start the simulation
103   process->simulate(Simtime(100.0));
104
105   // end of simulation
106   process->finalize();
107 }
108
109

```

Figure 4.1 shows a job file to execute the small simulation on our Pentium 4 cluster. A BGTW application is compiled with the `-lbgtw` flag in order to link it with the library. The same simulation is executed with 4, 3, and 2 CPUs, and the `time` command is used to time the execution time of each run.

## 4.2 Analysis

Figures 4.2, 4.3, 4.4 and 4.5 are showing the timings of the above simulation with different numbers of LPs on 4, 3 and 2 CPUs. The last run with a problem size of 5000 LPs shows a speedup with more CPUs, but if the problem size is too small the overhead of the parallel execution is too high.

```
#PBS -l nodes=4
#PBS -N appl
#PBS -l walltime=10:00
#PBS -j oe

cd $PBS_O_WORKDIR
mpiCC -lbgtw appl.cpp

cd $TMPDIR

echo 4
time mpiexec $PBS_O_WORKDIR/a.out
echo =====
echo 3
time mpiexec -n 3 $PBS_O_WORKDIR/a.out
echo =====
echo 2
time mpiexec -n 2 $PBS_O_WORKDIR/a.out
echo =====

mpiexec -comm none cp \* $PBS_O_WORKDIR
```

Figure 4.1: The PBS/Torque job file to execute a simulation on 4, 3, and 2 CPUs

```
4
real 0m1.098s
user 0m0.002s
sys 0m0.006s
=====
3
real 0m0.637s
user 0m0.000s
sys 0m0.007s
=====
2
real 0m0.553s
user 0m0.002s
sys 0m0.003s
```

Figure 4.2: The Timing results for a simulation with 15 LPs on 2, 3, and 4 CPUs

```
4
real 0m0.532s
user 0m0.002s
sys 0m0.006s
=====
3
real 0m0.522s
user 0m0.004s
sys 0m0.003s
=====
2
real 0m0.631s
user 0m0.001s
sys 0m0.009s
```

Figure 4.3: The Timing results for a simulation with 500 LPs on 2, 3, and 4 CPUs

```
4
real 0m0.669s
user 0m0.000s
sys 0m0.001s
=====
3
real 0m0.678s
user 0m0.001s
sys 0m0.003s
=====
2
real 0m0.895s
user 0m0.003s
sys 0m0.004s
```

Figure 4.4: The Timing results for a simulation with 1000 LPs on 2, 3, and 4 CPUs

```
4
real 0m4.599s
user 0m0.003s
sys 0m0.004s
=====
3
real 0m6.065s
user 0m0.001s
sys 0m0.002s
=====
2
real 0m9.513s
user 0m0.003s
sys 0m0.003s
```

Figure 4.5: The Timing results for a simulation with 5000 LPs on 2, 3, and 4 CPUs

## Chapter 5

# Proposal for Future Work

As the concern of this project was to build a modular engine there are plenty of opportunities for future developments by students who are interested in distributed simulation. We give you a couple hints what might be done next to improve the BGTW engine.

**GVT Calculation** At the moment the global virtual time computation is done not very efficiently. The GVT calculation is a critical part in respect of performance in a Time Warp system. To improve the performance algorithms like the one of Mattern[11] might be implemented.

**Different Approaches** As you know Time Warp is just one of many approaches to synchronize your distributed simulation. An interesting and worthy effort would be to implement a different module which uses whether a well know algorithm or simply just use your own thoughts and test your implementation against the existing time warp module.

**Simulation Application** Yet we haven't had the time to write a reasonable simulation which exploits the power of distributed computation. A very experimental and valuable project would be to write and test an application which takes advantage of parallel computing. An analysis might be to determine the threshold where the breakpoint of your application according to performance gains due to parallel computation is. Another point would be to write and test applications which run specifically well due to the Time Warp approach and such which don't.

**Performance improvements** There are some points in the engine which are for improvement. For example: message acknowledgment, or different MPI calls in the message handler.

**User Interface** The interface to our engine is very small and clean but it's still not very easy to use.

**LP addressing** At this point, LPs are addressed by their id number. For improved usability it should be possible to specify the destination of a message by the LP name. Each LP has already a member variable `name`, but it isn't really used.

**Portability** The engine is written for a homogenous parallel environment. It should be possible to port it to different architectures and to run on a heterogenous system.

**HLA** At the moment the engine is not HLA compliant. A major improvement would be to implement the HLA interface.

# List of Figures

3.1	Complete class diagram . . . . .	10
3.2	Details of the Process and the Simtime class . . . . .	12
3.3	Details of the Memory Manager . . . . .	13
3.4	Details of the Msg class . . . . .	14
3.5	Details of the Message_Handler class . . . . .	15
3.6	Details of the GVT and ThreadedGVT class . . . . .	17
3.7	Details of the LP and TWLP class . . . . .	18
3.8	Details of the TWEvent class . . . . .	19
4.1	The PBS/Torque job file to execute a simulation on 4, 3, and 2 CPUs	27
4.2	The Timing results for a simulation with 15 LPs on 2, 3, and 4 CPUs	27
4.3	The Timing results for a simulation with 500 LPs on 2, 3, and 4 CPUs	28
4.4	The Timing results for a simulation with 1000 LPs on 2, 3, and 4 CPUs . . . . .	28
4.5	The Timing results for a simulation with 5000 LPs on 2, 3, and 4 CPUs . . . . .	29

# Bibliography

- [1] A. Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley Pub Co, 2001. 10, 13
- [2] R. Bagrodia, R. Meyer, M. Takai, Y. an Chen, X. Zeng, J. Martin, and H. Y. Song. Parsec: A parallel simulation environment for complex systems. *IEEE Computer October 1998*, 31(10):77 – 85, 1998. 6
- [3] R. E. Bryant. Simulation of packet communication architecture computer systems. Technical Report TR-188, MIT Laboratory for Computer Science, 1977. 4
- [4] K. M. Chandy and J. Misra. Distributed simulation: A case study in design and verification of distributed programs. *IEEE Transactions on Software Engineering*, pages 440 – 452, 1979. 4
- [5] J. Dahmann, R. M. Fujimoto, and R. M. Weatherly. The dod high level architecture: an update. In *Proceedings of the 30th conference on Winter simulation*, pages 797–804. IEEE Computer Society Press, 1998. 7
- [6] J. S. Dahmann, R. M. Fujimoto, and R. M. Weatherly. The department of defense high level architecture. In *Proceedings of the 29th conference on Winter simulation*, pages 142–149. ACM Press, 1997. 7
- [7] S. Das, R. Fujimoto, K. Panesar, D. Allison, and M. Hybinette. Gtw: a time warp system for shared memory multiprocessors. In *Proceedings of the 26th conference on Winter simulation*, pages 1332–1339. Society for Computer Simulation International, 1994. 4, 6
- [8] R. M. Fujimoto. *Parallel and Distribution Simulation Systems*. John Wiley & Sons, Inc., 1999. 4
- [9] R. M. Fujimoto and M. Hybinette. Computing global virtual time in shared-memory multiprocessors. *ACM Trans. Model. Comput. Simul.*, 7(4):425–446, 1997. 6

- 
- [10] D. R. Jefferson. Virtual time. *ACM Trans. Program. Lang. Syst.*, 7(3):404–425, 1985. 3, 4, 6
- [11] F. Mattern. Efficient algorithms for distributed snapshots and global virtual time approximation. *Journal of Parallel and Distributed Computing*, 18(4):423–434, 1993. 30
- [12] H. Rajaei. Sima: an environment for parallel discrete-event simulation. In *Proceedings of the 25th annual symposium on Simulation*, pages 147–155. IEEE Computer Society Press, 1992. 7
- [13] H. Rajaei, R. Ayani, and L.-E. Thorelli. The local time warp approach to parallel simulation. In *Proceedings of the seventh workshop on Parallel and distributed simulation*, pages 119–126. ACM Press, 1993. 5
- [14] X. Zeng, R. Bagrodia, and M. Gerla. Glomosim: a library for parallel simulation of large-scale wireless networks. In *Proceedings of the twelfth workshop on Parallel and distributed simulation*, pages 154–161. IEEE Computer Society, 1998. 7