Description of the

Parallel Toolbox and Related Topics

Version 1.0.0

http://www.ZPR.Uni-Koeln.DE/GroupBachem/VERKEHR.PG/RESEARCH.P/toolbox/

Marcus Rickert[1]

January 6, 1996

[1]ZPR, Universität zu Köln, Germany and TSA-DO/SA, Los Alamos National Lab NM, USA

Contents

Chapter 1

Introduction

1.1 General remarks

This manual replaces the paper *A First Draft on how to Integrate High Fidelity and Cellular Automata Approaches to Microsimulation in TRANSIMS on a Distributed Computer Network*. It is meant to describe the concepts of the Parallel Toolbox (chapter 2), and its usage (chapter 3). As of version 0.9.8 the old chapter 4 **TRANSIMS related topics** is obsolete. It will be replaced by seperate documentation provided by the two teams in Los Alamos and Cologne. Please refer to the appropriate WWW pages:

http://studguppy.tsasa.lanl.gov/
http://www.ZPR.Uni-Koeln.DE/GroupBachem/VERKEHR.PG/RESEARCH.P/

The reader should know that is *only a working paper* which tries to summarize all ideas that might lead to a functioning toolbox and application. The author strongly encourages critical comments on any part of this paper. Also this summary is inhomogeneous as far as the level of detail is concerned. Some aspects of the implementation are still very unclear so that in places only handwaving arguments or explanations are given.

This manual will be updated regularly as the project proceeds. New features and changes will be logged in 1.3.

1.2 Motivation for the Toolbox

High fidelity traffic simulations have been created by several scientific and engineering groups all around the world over the last decades. Due to its scale TRANSIMS will run into the same problems as any of its predecessors: computational speed provided by today's conventional[1] computers is still not sufficient to cope with the extremely large number of

[1]that is *normal* single node computers

individual objects that have to be simulated in real time for a realistic network (e.g. of a major city like Albuquerque or let alone the Los Angelos basin).

In this manual two methods will be described to increase the performance of the simulation and thus to make a large scale computation feasable:

- **Parallel computers** will be used to distribute the network onto several computational nodes. The toolkit PVM will provide a programming environment that allows porting an application on a large variety of modern supercomputers like the CM-5, the T3D, the Paragon, or the Parsytec. The Parallel Toolbox serves as an interface layer between PVM and the application level providing several services neccessary in a parallel computer environment.
- **Cellular automata** will serve as an alternative model for simulating the motion of the vehicles. In contrast to the high fidelity model based upon intelligent objects CAs can be regarded as a low fidelity solution which still captures the main characteristics of traffic flow while boosting computational speed by a factor of 10 to 100. There are limitations to the concept of CAs which will probably emerge as soon as first test runs are performed. Certain problems will deliver unsatisfactory (or even qualitively wrong) results when solved in an CA approach. One of the main objectives of a program which integrates both models will thus be
  - to compare results quantitively and qualitively, and
  - to find parameters by which to decide whether to use slow high fidelity or the fast low fidelity approach.

1.3 Change Log

This section summarizes the changes of the toolbox. A more detailed description can usually be found in `$CHOME/include/ChangeLog.h`.

1.3.1 Version 0.9.3 dated 02–28–95

- Classes `TIntelligentObject` and `TIntelligentParallelObject` are obsolete.
- The handling of error and warning messages has been completely revised: each class derived from `TBaseObject` can overload methods to describe the name of the class and the current contents of the instance of the class.
- There are three new timers: `GraphicsTimer`, `BoundaryTimer`, and `BalanceTimer`.

1.3.2 Version 0.9.4 dated 03–13–95

- `TSimulationSlave::SelectNestNodeToBeTransferred` now selects the node furthest away from the center of mass of the subnet. This is a very easy way to keep the subnets *nicely shaped.*

- First attempts with dynamic load balancing on rectangular grids successful. Not stable yet, though.
- Errors in talk mechanism fixed affecting collisions of talk requests.
- Coloring of graph edges implemented restricting load transfers between CPNs to *slots* defined by the color of the corresponding graph edge.
- Some changes in default functionality of methods of `TSimulationSlave` reducing the number of overloaded methods in `TApplicationSlave`.
- New template `TApplication<>` reducing the main application source file to several lines.
- First changes to comply with ANSI-C++ standards and to enable compilation on SunPro compilers.
- New methods `De/EncodeNodeData` of `TBaseNode` to de/encode additional node information.
- New Grid Extension demo `Dim2ca` simulating two dimensional traffic.

1.3.3 Version 0.9.5 dated 03–29–95

- First version running on the dedicated Intel Paragon.
- During spawning of slaves a version dependant magic number is checked to verify the compatibility of different binaries.
- It is now possible to dynamically insert a CPN having more than one processor.
- A new version method for simulation control called `SimulationControl2` simplifies programming since the operation is no longer based on callbacks and switch statements. The ancestor method of `TSimulationControl` must not be called anymore!
- The application `periodic` features a new multi lane CA. In future there will be two demo applications: `periodoc` will simulate a traffic circle with periodic boundary conditions, `network` will be restricted to displaying networks. The latter still uses the old CA. This will change.
- New classes `TView`, `TViewManager`, `TViewManagerSlave`, and `TViewDataHandler` to display excerpts (**`views`**) of the network at a high level of detail (individual vehicles). The class `TBaseEdge` has been added stubs to transfer and display the data. The new demo `periodic` supports this feature.

1.3.4 Version 0.9.6 dated 04-24-95

- The method `TTransferObject::Activate` is obsolete. In all descendant classes this method has to be replaced by `TTransferObject::ActivateLevel` (see B.2).
- Most inline and template methods have been moved to files of their own. The naming conventions are `TType.i.C` for include methods and `TType.t.C` for template methods. The files are located in the same directory as the corresponding `TType.h` files. The main file generating all template instances must define the conditional `TEMPLATEBODIES` before the first `#include`. See for example `MicroSimTemplates.h`.
- `TBoundary` has a new flag `External` which is used to differentiate between an external boundary from another CPN and an internal boundary. Moreover boundaries can be *reversed* which makes `ToBoundaries` behave linke `FromBoundaries` and vice versa.
- The ZPR micro simulation is available in a first (very simple!) version (see 3.9). It will eventually replace the old `network` demo.
- The `TView` mechanism is stable now.
- The class `TMultilaneEdge` now handles bidirectional traffic using two instances of `TMultilaneGrid`.
- The method `TBaseNode::PrepareTimeStep` now has default methods for nodes of valence 1 and 2: vehicles are reflected at nodes of valence 1 and simply propegated at nodes of valence 2.
- The network and topology windows are sized according to aspect ratio of the simulation network.

1.3.5 Version 0.9.7 dated 05–24–95

- The starting of applications is now managed by two shell scripts. Please see section 3.7.1.
- Master and slaves processes can be started with a nice value passed as command line option.
- Slave processes can be debugged.
- Load ratio history now depends on real time and no longer on time steps.
- Most nethods handling dynamic load balancing have been moved from `TSimulationSlave` to new class `TLoadBalance`.
- Dynamic load balancing for street network works for two CPNs.
- Toolbox is stable with only one CPN.

- MicroSim–Makefile has been completely revised. The subdirectories `common`, `toolbox`, and `CA` now have makefiles of their own. Their objects are kept in architecture dependant libraries which are located in `$CHOME/lib/$PVM_ARCH`.

1.3.6 Version 0.9.8 dated 06–05–95

- New architecture SGI5 on Silicon Graphics machines running Irix5.
- Relatively stable dynamic load balancing on street networks.
- Command line option `-X` excludes master from distribution of street network. This option is primarily intended for fast parallel computers with slow frontends (e.g. Intel Paragon).
- Command line option `-U<time steps>` activates the communication bench mark.
- All makefiles of `MicroSim` have been completely revised. They are located in the `script` subdirectory. A site dependant makefile named `$CHOME/site/SiteMakefile` can be supplied to define local settings. This makefile is not part of the archive and thus will never be overwritten by new versions.

1.3.7 Version 1.0.0 dated 01-06-96

This is the final version of the toolbox. It comprises most of the features it was originally designed for. All future versions will incorperate *shared memory* not only as supported PVM–architecture but also structurally: a simulation distributed in a heterogenous computer network will use shared–memory techniqes (e.g locking) *locally on CPNs* and message passing techniques as means of communication between CPNs.

Here's a list of the new features of version 1.0.0:

- New class *TEventHandler* offers faster event handling by passing events directly to the receiving class methods instead of going through long `case`–switches.
- Full speration of toolbox and graphics.
- Usage of declarator `const` with most class methods.
- Builtin performance (speed, load balancing, and communication) statistics available.
- Toolbox now works on heterogenous hardware networks with more than one binary data representation. The encoding method is switched automatically from `PVM_DATA_RAW` to `PVM_DATA_DEFAULT` and back.
- Source code (excluding `graphx`) compatible with new GCC version 2.7.2.
- New architectures `SUNMP`, `SGIx`, and `ALPHAMP`. The toolbox compiles on `RS6K` for IBM machines but still crashes. This is probably due to the fact that the GCC is buggy for the IBM risc architecture.

- New option `-C` to allow offload of not connected net parts (test version). This may improve load balacing for small networks even though communication is increased.
- Architecture independent signal handling. Source code should now compile on most unix machines.
- `TBaseNode` now has new field `AssociatedData`. The router's shortest path algorithm uses this field to store Dijkstra data.

1.4 Known bugs

Here's a list of some little (and some not so little) bugs that are well known and will hopefully be fixed in one of the upcoming versions.

- The caption of the topology display is not updated after deletion of insertion of CPNs.
- The dynamic load balancing on street networks still crashes once in a while. This might be due to the fact that under weird circumstances the offload algorithm still selects clusters in such a way that a subnet on a CPN is split into fractions.

1.5 Who's who

There are a lot of people involved in the development and programming of the toolbox and the TRANSIMS application. Here are a few:

Chris Barrett (roberts@tsasa.lanl.gov) Project leader, theory of simulation and dynamic load balancing methods.

Kathy Bergbigler (kbp@lanl.gov) Dynamic load balancing.

John Davis (jfd@lanl.gov) Intersection functionality, driver logic, route plans.

Stephen Eubank (eubank@tsasa.lanl.gov) Adaption of the toolbox to the requirements of TRANSIMS.

Kai Nagel (kai@tsasa.lanl.gov, kai@zpr.uni-koeln.de) Theory of high speed computation, cellular automata.

Rob Oakes (oakes@tsasa.lanl.gov) Configuration management, support.

Peter Oertel (poertel@zpr.uni-koeln.de) Programming of cellular automata.

Michael Olesen (michael@tsasa.lanl.gov) Graphics toolbox.

Steen Rasmussen (steen@tsasa.lanl.gov) Theory of simulation, cellular automata research.

Marcus Rickert (rickert@tsasa.lanl.gov, mr@zpr.uni-koeln.de) Parallel Toolbox, Grid Toolbox Extension, and dynamic load balancing methods.

Jay Riordon (jay@tsasa.lanl.gov) Cellular automata research.

Doug Roberts (roberts@tsasa.lanl.gov) Intersection functionality, driver logic.

Paula Stretz (stretz@agps.lanl.gov) Software team leader.

Here's the Cologne crew from Germany:

Achim Bachem (bachem@zpr.uni-koeln.de) Chairman of the Zentrum für Paralleles Rechnen in Cologne.

Christian Gawron (gawron@zpr.uni-koeln.de) Graphics toolbox, multi threaded version.

Christoph Moll (cm@mi.uni-koeln.de) Coordination, shortest path algorithms, vehicle routing.

Peter Oertl (poertel@zpr.uni-koeln.de) Programming.

Peter Wagner (pwagner@zpr.uni-koeln.de) Coordination, traffic research.

Chapter 2

The Parallel Toolbox

2.1 Requirements

2.1.1 Supported Platforms and Compilers

The Parallel Toolbox has been tested on the following platforms lately:

Operating System	PVM Architecture	Compiler	date
Sun OS 4	SUN4	gcc 2.6.3	05–25–95
Sun OS 4	SUN4	SunPro ?	
Solaris 2	SUN4SOL2	gcc 2.6.3	05–25–95
Solaris 2	SUN4SOL2	SunPro ?	
Paragon	PGON	gcc 2.6.3	05–26–95
Linux	LINUX	gcc 2.6.2	06–05–95
Irix5	SGI5	gcc 2.6.3	06–04–95
Alpha	ALPHAMP	gcc 2.7.2	12–15–05
Irix6	SGI(MP)64	gcc 2.7.2	01–06–96
Solaris 2	SUNMP	gcc 2.7.2	01–06–96

Table 2.1: *Platforms*

The toolbox compiles and links on the `SGI64` architecture but still crashes immediately after startup.

2.1.2 Parallel Virtual Machine (PVM)

The Parallel Toolbox requires a consistent installation of the PVM toolbox version 3.3.x. Most of the above architectures have been tested with version 3.3.6 and 3.3.7.

2.1.3 Files

Supposing `CHOME` to be the root directory of c source code the source code files of the Parallel Toolbox and the demos are arranged in the following combination of subdirectories:

`$CHOME/common` contains low level classes and tools which are independant of both the toolbox and the demos.

`$CHOME/scripts` contains shell scripts used for installing the toolbox and running toolbox applications. Most of the scripts are written for `csh` or `tcsh` which should be available on all platforms. It also contains default makefiles which are included by the application makefiles.

`$CHOME/toolbox` contains all classes of the toolbox. For a description of each individual class and the location in the source files see B.10.

`$CHOME/include` contains include files for both the `common` and the `toolbox` directories.

`$CHOME/include/ChangeLog.h` contains details about changes of the Parallel Toolbox.

`$CHOME/include/DEFINES.h` contains a list of all conditional defines used in the Parallel Toolbox.

`$CHOME/include/ToDo.h` contains a list of all the things that are still to be done.

`$CHOME/include/KnownBugs.h` contains a list of all known bugs which are known to the authors and will hopefully be fixed soon.

`$CHOME/include/Communication.h` contains communication benchmarks.

`$CHOME/include/Computers.h` contains a overview of the computer architectures used in the file above.

`$CHOME/network` contains overloaded classes of the network CA demo.

`$CHOME/periodic` contains overloaded classes of the CA demo with periodic boundary conditions. This demo is actually not a demo but an application used to examine different lane changing rules in a two lane CA. It uses the CA defined in `$CHOME/CA`.

`$CHOME/frame` contains dummy files that can easily be copied to define the application framework for a new application.

`$CHOME/CA` contains the new multilane bidirectional CA classes.

`$CHOME/CA/Rules` contains different lane changing algorithms for `TMultilaneGrid`.

`$CHOME/MicroSim` contains the classes of the ZPR micro simulation.

`$CHOME/TRANSIMS` contains overloaded classes of the TRANSIMS application including a simple single lane CA. Some of these files are used by the demo in `network`.

`$CHOME/grid` contains classes defining the toolbox extension for rectangular grid applications.

`$CHOME/life` contains classes of the *Game of Life* demo using the Grid Toolbox Extension.

`$CHOME/doc` contains `dvi` and `ps` images of this manual.

Versions of the toolbox will be made available in two tar files:

- `toolbox-<version>.tar.gz` containing all files in `common`, `toolbox`, `include`, `network`, `TRANSIMS`, `frame`, and `doc`.
- `grid-<version>.tar.gz` containing all files in `grid` and `life`.

2.2 Concepts

2.2.1 Parallelization

In microsimulations computational speed is one of the main objectives. Any simulation should run as fast or faster than the problem it tries to model takes in real time. For a large scale simulation of a traffic network this is only possible by distributing the network onto several computational nodes, called CPNs.

To achieve this aim the objects of the simulation are associated with nodes and segments which are inserted into a graph. In case of a traffic simulation this association is trivial since intersections directly correspond to nodes and segments to edges. In case of the toolbox extension for grids nodes are associated with subgrids of the grids while edges contain the boundary dependencies of the subgrids.

The major requirements of the toolbox are:

- The computation on the nodes and edges of the graph is **local**, that each object might access data residing on its neighbours, but not on objects randomly located somewhere else in the graph. This implies that there has to be maximum **range of causality** over which objects of the simulation can influence each other. On edges connecting two CPNs all objects residing within this range are transferred to the remote CPN through messages (see 2.3). In traffic simulation this range is defined by the driver logic.
- It must be possible to formulate the update logic of the simulation as **time step driven** with a constant time step.

If the above requirements are met the Parallel Toolbox automatically provides the following mechanisms:

- distribute the network onto several computational nodes,

- provide an easy programming interface for controlling the course of the simulation
- provide methods to gather statistics during run time of the simulation
- perform a *dynamic* load balancing during the progress of the simulation in a straight forward easy approach,
- remove or add computational nodes (except the master control node) during the progress of the simulation after an appropriate waiting period.

In future versions efforts can be made to

- optimize the load balancing so that the frequency and/or amount of communication between nodes is minimized,

2.2.2 Platforms

Roughly spoken at the moment there two major types of parallel computer hardware:

- The first type can be called the *high end* version because it usually includes especially designed hardware and/or software for both on node computation as well as communication between the nodes such as the CM-5 (Thinking Machines), the T3D (Cray Research) and the Paragon (Intel). On these systems programs are usually assigned to a partition of *dedicated* nodes; that is, they run in a multiple task single user environment.
- The second type of parallel systems is simply a cluster of workstations connected by a LAN such as Ethernet or better FDDI. Compared to their high end counterparts they are far less expensive but often suffer from poor performance as far as communication is concerned.

To keep the implementation as portable as possible the library PVM was chosen to take care of the communication between the computational nodes. As of the latest version of README files PVM exists in *native* ports for both the CM-5 and the Paragon so that a relatively high speed can be expected on these platforms. On the T3D PVM is the native communication library directly supported by the manufacturer. As for the workstation clusters PVM allows to combine different operating systems and/or hardware in a LAN or even globally via Internet.

2.2.3 Parallel Structure

Applications based upon the Parallel Toolbox will have a SIMD (Single Instruction Multiple Data) structure combined with master slave control (see figure 2.1). All CPNs will run the same compiled binary, but they will differ by the subnet that they handle during the simulation. There will be one CPN called **master** with special functions. All other CPNs are called **slaves**.

The master will perform all tasks of a slave plus the following special functions:

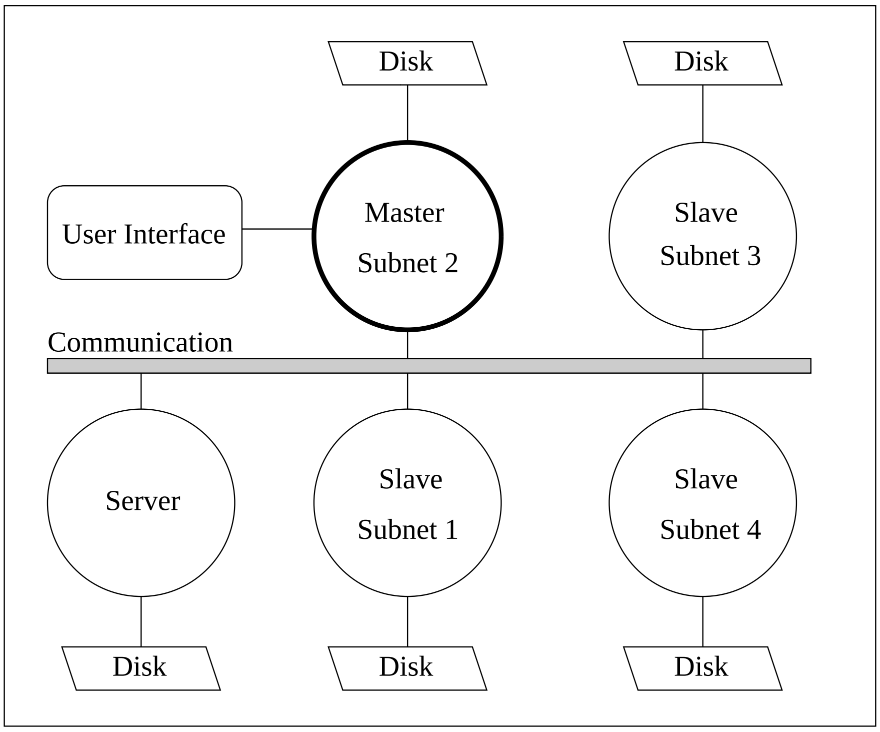

Figure 2.1: *SIMD Master Slave Structure*

- supply a text user interface or a GUI,
- control the PVM environment,
- control the simulation.

The CPNs will be connected through a communications network with access to file servers. All CPNs will have local storage capacities.

2.2.4 Load Balancing

It is the goal of an application running on a parallel computer network to load all available CPNs *equally*, that is equal simulation intervals should take the same wall clock time on the CPNs. The easiest approach is **static load balancing** in which the objects of the simulation are distributed onto the CPNs exactly *once* before the start of the simulation. This **initial distribution** depends on the number of available CPNs as well as their relative performances.

Since computational requirements on the individual CPNs may change during the course of the simulation one has to see to it that busy CPNs are relieved of some of their local subnet which is then transferred to CPNs running idle. This process is called **dynamic**

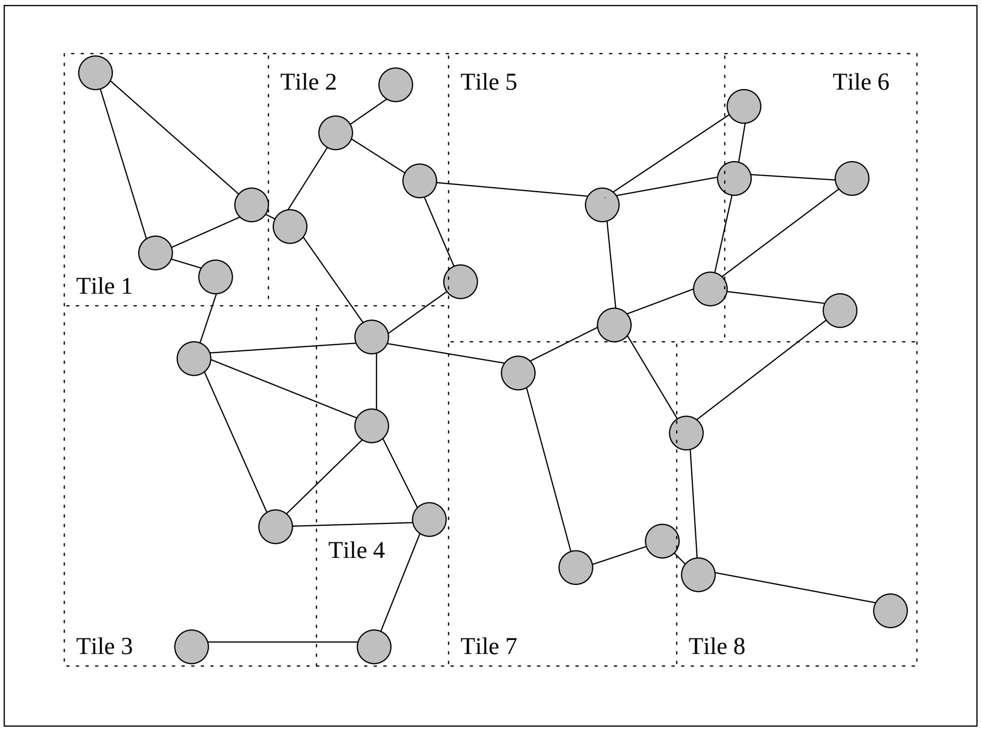

Figure 2.2: *Initial Distribution of Nodes onto eight CPNs*

load balancing. The Toolbox will take care of the load balancing if an estimated value for the load generated by each single object of the simulation has been made available.

A detailed description of the load balancing techniques can be found in 2.5. For now it is sufficient to know that load balancing requires both an initial distribution and the ability to transfer network topology from one CPN to another.

2.2.5 Distribution

As mentioned above the objects of the simulation have to be distributed in such a way that the execution times needed by the CPNs are more or less equal. But that in itself is not sufficient since cutting the network into subnets causes generation of boundaries which themselves lead to communication. It is therefore a goal to minimize the amount (total message length) and frequency (number of messages) of communication. An easy and yet efficient approach is to do a **geometric distribution** in which each CPN handles a part of the area defined by all nodes of the network (see figure 2.2).

One has to destinguish between the *initial* distribution onto a given number of computational nodes and the *dynamic* distribution that takes place to keep the load balanced. Of course, if the *dynamic* load balancing is sophisticated enough it should be sufficient to start with the whole network residing on a single CPN and simply add the other CPNs one after another hoping that the system will adapt accordingly. But this would result in a starting phase in with significant load disparities and very weak performance. So in order to decrease the time neccessary to reach a load equilibrium the Toolbox does a coarse initial distribution which is then refined by the dynamic load balancing.

The initial distribution tries to split the network in such a way that

- the number of subnets is equal to the number CPNs (which is the minimal number of subnets),
- the ratios of estimated loads assigned to the CPNs are equivalent to to ratios of the computational performances of the CPNs,
- the shapes of the subnets are such that the number of boundaries is small.

Figure 2.2 contains a network that is equally distributed onto 8 CPNs each taking care of the nodes residing on its associated tile. In this example the number of nodes was used to determine load. A detailed description of the initial distribution can be found in 2.4.

2.2.6 Events and Messages

In a parallel computer environment CPNs communicate through events and messages. In the Parallel Toolbox an event can be regarded as a package of information which contains a type, an addressee, and data. One or more events can be combined into a message which contains a type and a destination CPN. Each object class that may be transferred across CPN boundaries has to provide methods for encoding and decoding of data contained in the instance of that object.

In the Parallel Toolbox events are used for:

- controlling the course of the simulation,
- transferring network from one CPN to another,
- transferring boundary information
- transferring statistical data.

The way messages are handled has a strong impact on the performance of the simulation. It is the goal of the toolbox to minimize the number of messages sent and the sum lengths of the messages[1]. Whenever possible the Toolbox combines as many events as possible into a single message.

2.2.7 Object Hierarchy

[1]Of course, the toolbox has little influence on the complex data structures of the simulation. Therefore if the data structures are large, so will be the messages.

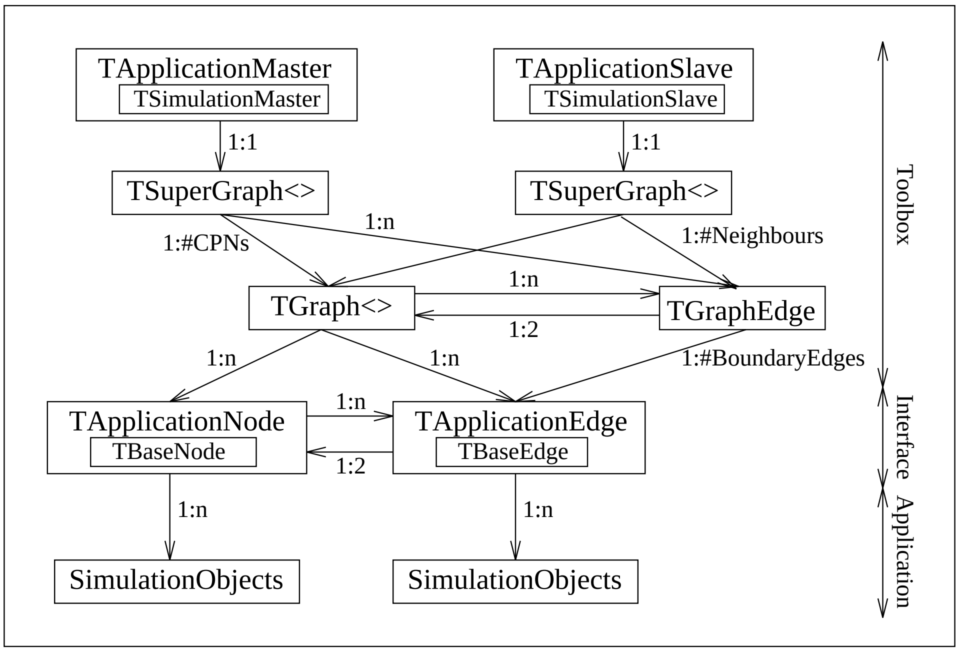

Figure 2.3: *The Object Hierarchy*

The objects of the simulation are arranged in a hierarchy which is displayed in 2.3. Note that the arrows denote a *has-a* relationship going from owner to dependant and the expressions $1 : x$ denote the number x of dependant instances per owner instance. The top levels CPN topology (`TSimulationSlave`, `TSuperGraph`, `TGraph`, and `TGraphEdge`) and network topology (`TGraph`, `TBaseNode`, `TBaseEdge`) are provided by the toolbox. The control of the application will be achieved by defining descendant classes `TApplicationSlave` and `TApplicationMaster` as well as overloading some of their methods. The topology of the simulation network (e.g. segments and intersections in a street network) is built by defining descendant classes `TApplicationNode` and `TApplicationEdge` of the interface level classes `TBaseNode` and `TBaseEdge`. Instances of `TApplicationNode` and `TApplicationEdge` are inserted into the `TGraph`[2] that takes care of distributing the nodes and edges onto the available CPNs. The hierarchy representations on the slaves differ from the representation of a the master as follows (see 2.4): the master has an instance of `TSuperGraph<>` for *all* CPNs in the CPN topology. This is neccessary for graphics, some control matters and later global load balancing. The slaves only have instances of their own subnet (which contains complete lists of all nodes and edges residing on the slave) and one instance for each neighbouring CPN it has at least one boundary edge in common with. The node lists and edge lists on those `TGraphs`, however, are restricted to the boundary edges themselves

[2]This is actually wrong. According to the diagramm the nodes and edges are handled by an instance of `TGraph`. The user, however, will insert the objects into instance of `TSuperGraph`, which itself will create an instance of `TGraph` and insert the objects into that. During distribution more instances of `TGraph` (and instances of `TGraphEdge`) will be created to eventually match the number of CPNs. At the same time nodes and edges will be moved between `TGraphs` accordingly.

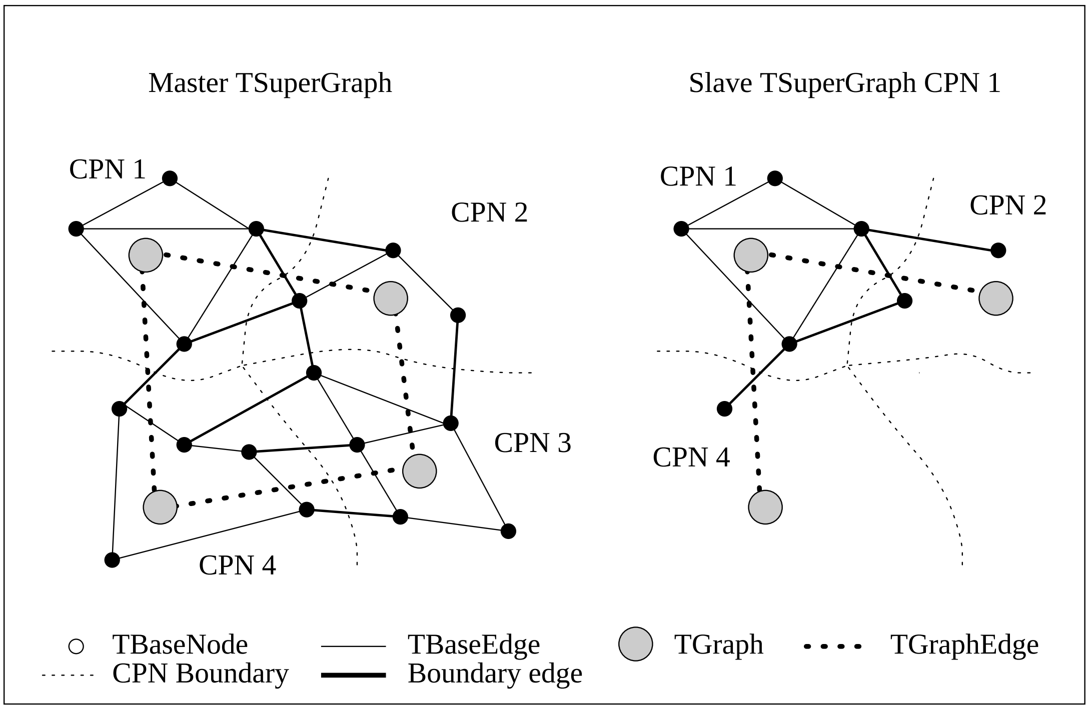

Figure 2.4: *Hierarchy Representations on Master and Slaves*

and the nodes incedent to the boundary edges. Note that the master functioning as a slave also has a restricted instance of `TSuperGraph`.

2.2.8 Objects and References

Objects will be the main means of storing and handling data in the simulation. On the one hand the are control classes that provide tools and algorithms. Some of the methods of the predefined **control classes** `TSimulationSlave`, `TSimulationMaster`, `TBoundary` have to be overloaded to define the functionality of the simulation.

On the other hand there are the predefined *network data classes* `TBaseNode` and `TBaseEdge` which also have to be overloaded, but also have to be generated and combined in such a way to exactly represented to given network. This is described in 2.8.

Depending on the scope of an object its class will be derived from one of the four base classes described in 2.8.1. There are two types of references between objects: the first covers references that are not transferred to other CPNs in which case the reference is represented by a pointer. In case the object is transferred a pointer to another object loses its validity because on the remote CPN the referenced object will be located in a different memory location.

To solve this problem all classes that have to be transfered and referenced must be derived from the base class `TObjectID`. Each instance of that class used in the simulation will be assigned a unique ID. On each CPN an AB-tree called **ID-tree** will be maintained containing all pairs of (ID, pointer to object) currently residing on that CPN. Whenever an entity moves from one CPN to another it will keep its ID but its entry in the old ID-tree will be deleted and inserted in the tree on the new CPN.

All *references* to objects will be handled through classes of template `TReference<>`. The first time a `TReference` is accessed it will search the local ID-tree, retrieve the corresponding pointer to the object and return this pointer. On all subsequent calls it will simply return the stored pointer.

2.2.9 Context Objects

Warning: The following indented feature has not been implemented yet!

> As mentioned in 2.2.7 all objects are organized in a strict hierarchy. Simulation control uses this hierarchy in a top down manner: control events are passed to the topmost level which forwards them to all objects of the level below and so on until they reach the bottommost level of objects. Such an object on the lowest level might need information about each intermediate object lying on the path from the topmost level. The combination of all these pieces of information is called **context** and stored in an class derived from `TContext`. For each execution of the simulation the toolbox will create an instance of that context class and pass it down. On its way to the bottom of the hierachy each level stores data into it. After the execution the instance is disposed.
>
> Beside their function as a means of passing information between the layers of the object hierarchy context objects are also useful to pass a huge sets of parameters to a function or an object method avoiding modifications in the declaration of object methods in case the number and/or type of parameters has changed. Normally context objects they do not have methods although it might be appropriate in some occasions to supply some for extended data conversion and consistency checking.

2.2.10 The Timing of a Simulation Run

The following enumeration is supposed to convey an idea of the main steps executed during a simulation run. Also see figure 2.5.

1. The user initializes PVM on the future master CPN by calling `pvm` and adding all CPNs manually or starts a script to do it automatically.

2. The user starts the program on the master CPN with an initial configuration of slave CPNs.

3. The master starts all other instances of the program on the slave CPNs. They will enter the main message loop and wait for messages.

4. The master reads the network structure from an input data file or creates an artificial network.

5. The master distributes the network and sends out messages to the slave CPNs with encoded network elements.

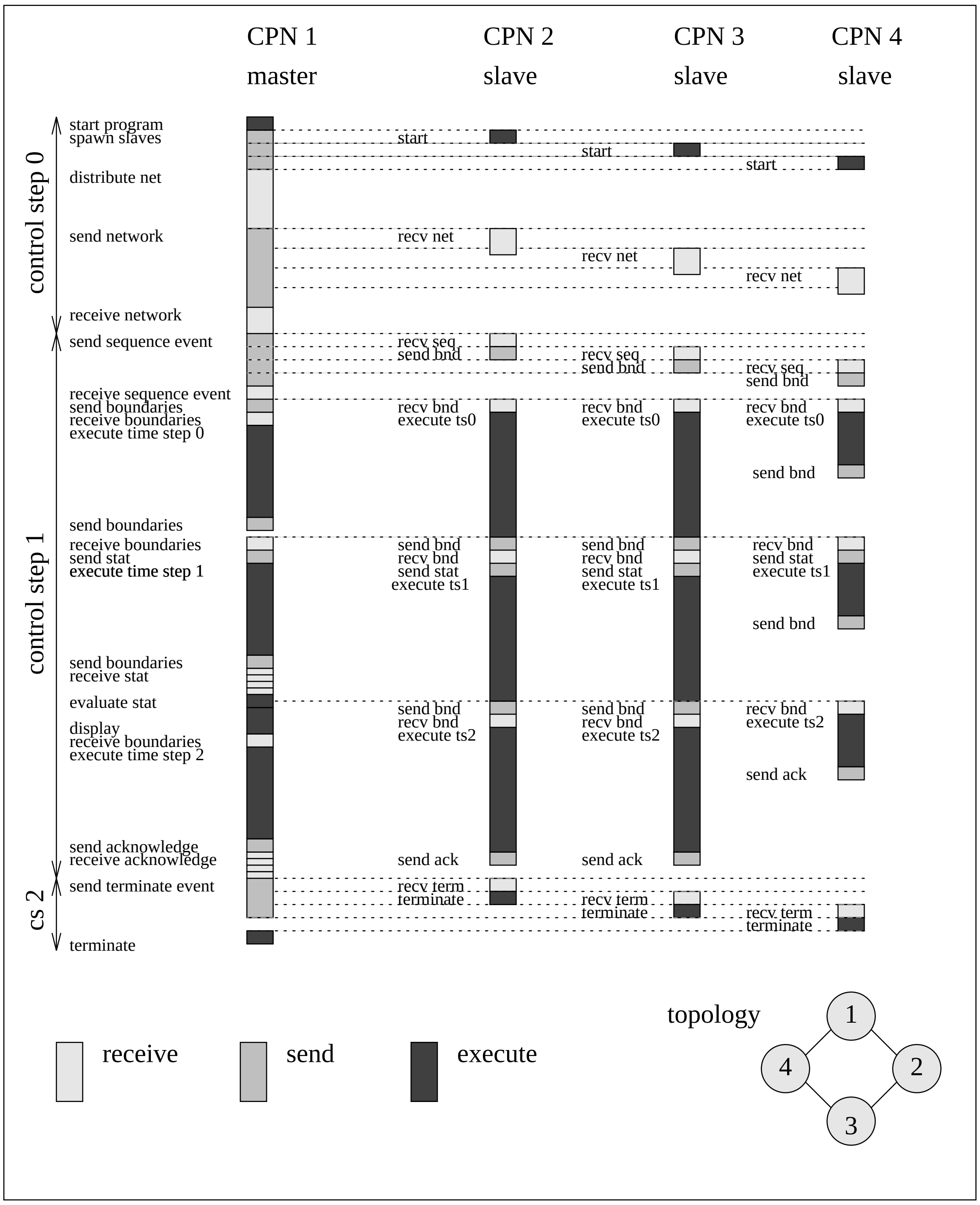

Figure 2.5: *Timing of a Simulation Run*

6. The master sends an event to all CPNs to start a simulation sequence of a given number of time steps. All slaves send out the boundaries for the first time step to the neighbouring CPNs. During that sequence the master mainly functions as a slave.
7. The slaves enter the main loop:
   - They wait for the arrival of all boundaries from their neighbours.
   - If neccessary, statistical data (e.g. idle time statistics) is sent out.
   - If idle time statistics are available local load balacing is done.
   - They execute a time step.
   - Unless the end of the sequence is reached they send out the boundaries for the next time step.

   The master has several additional functions:

   - If available, statistical data from the slaves is processed and displayed.
   - The X-Windows event queue is checked if there is need to update the graphics output.
   - The PVM environment is checked whether CPNs have to be removed from the topology or new CPNs can be added to the topology.
8. The master stops the simulation by sending an event to the slaves.
9. The master and the slaves stop execution.

2.3 Boundaries

After the distribution of the nodes of the network there will be edges crossing CPN boundaries. Those are called **boundary edges**. Currently they are cut exactly in the middle so that each associated CPN computes half of the edge. Note that therefore boundary edges exist twice (see 2.6). Before either CPN can execute a time step it has know about the objects on the remote CPN: not all but at least those that are within the **range of causality**, that is about all objects that might have in impact on the decision making of objects residing in the local part of the edge. Therefore on both CPNs the edge is prompted to provide boundary imformation encoded in a descendant of class `TBoundary`. This boundary information is transferred to the remote CPN and over there given to the duplicate of the edge which appends this information to the local data stored on the edge and thus makes the execution of the time step possible. This type of boundary will be called **external boundary** since it is meant to be transferred to another CPN.

Related to this problem is the handling of boundaries between nodes and edges of the network: the execution of a timestep on segment might depend on the object configuration on the node and vice versa. It seems very likely that these **internal boundaries** are very similar if not identical to the external ones so that they could actually be handled by the same methods.

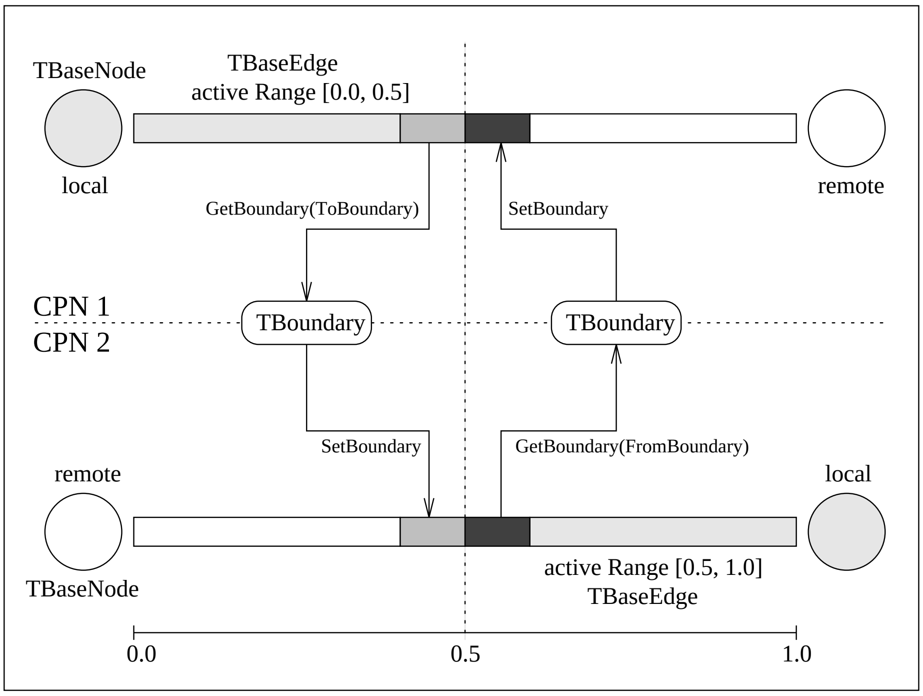

Figure 2.6: *External Boundaries*

2.3.1 External Boundaries

As mentioned before, boundaries edges connecting nodes on different CPNs exist on *both* CPNs (see 2.6). This might cause a problem because the objects residing on the edge have to updated in a consistent manner. Either they have to be updated by only one CPN or if they are updated by both CPNs their updates have to be *identical*. The boundaries edges are split into halves[3]. Each half is linked to a local node and the objects on this half are handled by the CPN that the local node resides on. Around the transition between the halves there is a certain width in which objects on one half might have an impact on decisions made by objects on the other half. So the toolbox has to see to it that all data neccessary for these decisions is passed to the remote CPN. To do this it makes the edge encode the data into a boundary objects `TBoundary` by calling the method `GetBoundary`. Then this objects is sent to the remote CPN and decoded there by calling the method `SetBoundary`. Since each edge has two potential boundaries the values `FromBoundary` and `ToBoundary` are used to differentiate between them.

2.3.2 Internal Boundaries

Internal boundaries are boundaries handled by the nodes `TBaseNode` of the network (see

[3]This is true for the current version. In future it might be possible to split the edge at almost any point and to dynamically shift this location.

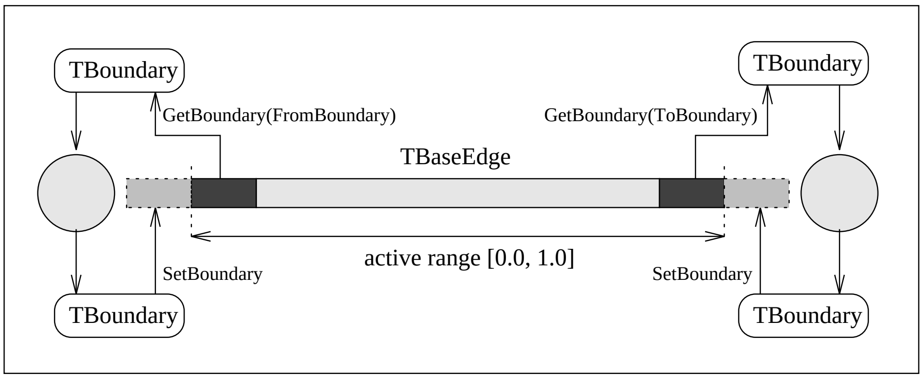

Figure 2.7: *Internal Boundaries*

2.7). In order to simplify programming if would advantageous if the boundaries returned by the edges through `GetBoundary` were equivalent to the boundaries required by the nodes. Vice versa it should be possible to supply node methods that generate boundaries which could in turn be used by the edges to do an update.

Note that this is only a suggestion. In constrast to the external boundaries which *have* to be provided through overloading to guarantee a consistent simulation, internal boundaries are not automatically handled by the toolbox except in the trivial case of a circular network with no functionality defined on the nodes.

2.3.3 Consistent Handling of Boundary Objects

Both in internal and external boundaries there is a region in which the same objects are handled by both CPNs. In order to guarantee consistence of the simulation it is neccessary that both instances of the same object behave exactly *identically.* In case of a deterministic simulation this is, of course, no problem. In case that decisions of objects depend on random number it is neccessary to make the random generator[4] *part of the object* and to pass it together with the object data to the remote CPN.

During a time step the objects on the edge usually change positions, that is, some will probably leave the remote boundary and enter the normal active part of the edge whereas others will do vice versa (see 2.8). After the time step all objects have to be deleted that still remain in boundaries supplied by remote CPNs (object 244 for CPN 1 and objects 332 and 567 for CPN 2). Likewise all objects that have entered the local active part of the edge have to be permanently inserted (object 567 for CPN 1 and object 244 for CPN 2). Note that object 332 has to be deleted, too, although it would be at the correct location for the next time step. Leaving it in the edge, however, would lead to collision with the boundary for the next time step which will contain another copy of that object.

[4]that is all values neccessary to reproduce the random sequence on the remote CPN, which could be for example the current seed of the generator

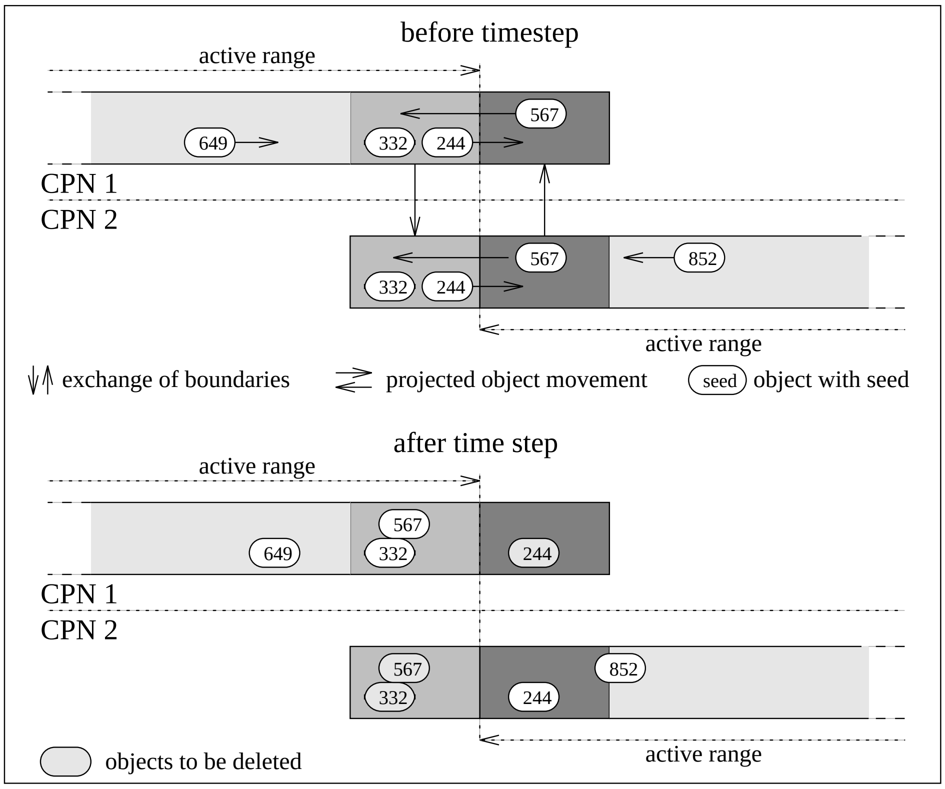

Figure 2.8: *Consistent Handling of Boundary Objects*

2.3.4 Timing

The Toolbox uses the boundaries for driving the simulation. Every CPN will send its boundaries to the neighbouring CPNs as soon as possible at the beginning of a timestep. After that it will start waiting for its neighbours' boundaries to arrive. As soon as all boundaries have arrived the motion part of the timestep is executed. Due to this interaction of the CPNs through boundaries they operate weakly synchronized. Between neighbouring CPNs there may be a difference in time steps of $\Delta t = \pm 1$ as displayed in figure 2.9: due to slow execution of time step 1 on CPN C, CPN B has received boundaries from CPN A for time step 2 *and* already for time step 3. The toolbox buffers those early boundaries automatically.

2.4 Initial distribution

For the initial distribution of a network with nodes $n_1 \ldots n_N$ onto CPNs $C_1 \ldots C_C$ we have to make three assumptions:

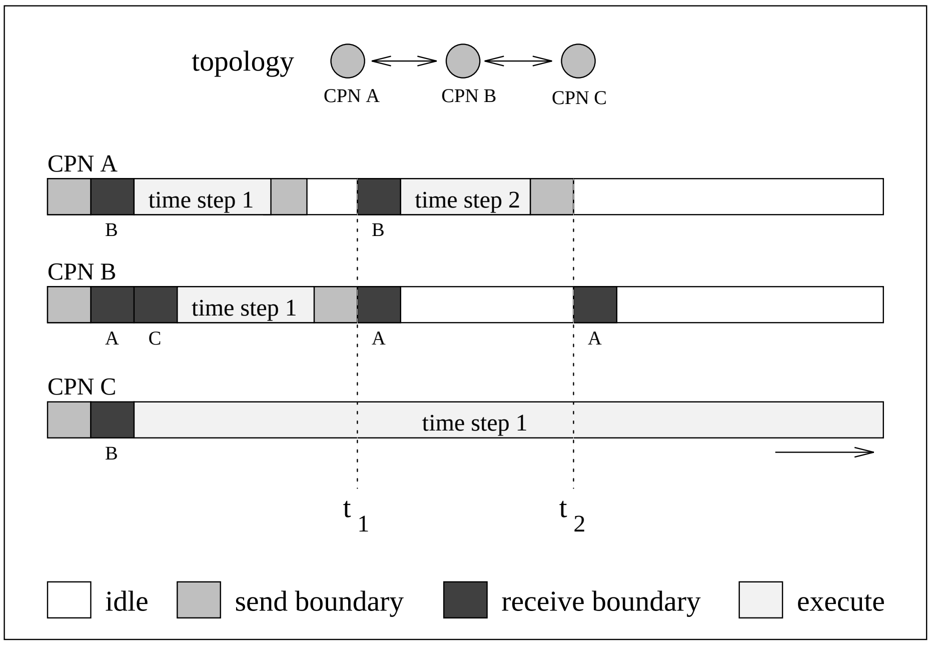

Figure 2.9: *Timing of Boundaries*

- We have performance values for each CPN given in arbitrary but equivalent values $S_1 \ldots S_C$ where larger values denote better performance.
- Each node n_i has an estimated load l_i associated with it which is derived from the complexity of the node itself and all its incedent edges.
- Each node n_i has an euclidic location (x_i, y_i).

2.4.1 Recursive algorithm

The algorithm is defined recursively:

1. If the number of CPNs is one, assign nodes to CPN.
2. Split CPNs in halves $C_1 \ldots C_{\lfloor C/2 \rfloor}$ and $C_{\lceil C/2 \rceil} \ldots C_C$ with sum performances $\sum_{i=1}^{\lfloor C/2 \rfloor} S_i$ and $\sum_{i=\lceil C/2 \rceil}^{C} S_i$.
3. Sort nodes according to their x (even depth) coordinates or y (odd depth) coordinates respectively.
4. Split nodes in such a way at node j that the ratio of the load of the node sets is equivalent to the ratio of the performance values:

$$\frac{\sum_{i=1}^{j-1} l_i}{\sum_{i=j}^{N} l_i} \overset{!}{\simeq} \frac{\sum_{i=1}^{\lfloor C/2 \rfloor} S_i}{\sum_{i=\lceil C/2 \rceil}^{C} S_i}$$

5. Split recursively

2.4.2 Correction

The node sets generated by the above algorithms usually cover rectangular tile of the plane. The alternating sort according to X/Y coordinates favours similar X/Y ratios for the edges of the rectangle. The problem is that despite this advantageous shape the nodes of a set are often not connected, so that some subsets split into even more subsets. Because of the assumption that each CPN will be assigned a connected subnet (see 2.2.5) this distribution has to be corrected in the following way:

1. Determine the number of subsets of each CPN and the associated estimated load of each subset.
2. If the number is greater one assign all subnets *but* the largest to neighbouring CPNs. For a given subnet choose only a recepient CPN with which it has a common edge.

2.5 Load Balancing

2.5.1 Why Load Balancing?

There are two major reasons for dynamic load balancing. The first is due the simulation itself since the load that is imposed by the simulation itself changes during the run of the simulation. In case of a microsimulation this very likely to happen since the computation is more or less proportional to the number of objects residing on a CPN and as the objects travel through the network the load moves accordingly.

The second reason refers to local area networks. In contrast to dedicated machines the nodes in a LAN are often used by other users decreasing the optimally available computational performance in a time dependant and usually random way.

2.5.2 Running Idle Versus Overloaded

The goal of load balacing for a given set of CPNs is to optimize the *overall sum performance* of all CPNs. Therefore the load balancing has to favor a single CPN or some CPNs running idle over a *single* CPN running loaded to maximum capacity since in the latter case this single CPN slows down *all others.* In figure 2.10 4 CPNs are shown once with CPN 1 running idle (50%) and once overloaded (100%). If 4 is the maximum performance of the perfectly balanced system the performance for CPN 1 idling is 3.5 and for CPN 1 overloaded 2.5. Note that this ratio becomes the worse the more CPNs are in the CPN topology.

2.5.3 Measuring the Load

During the simulation each CPN keeps track of the time it spends on different tasks:

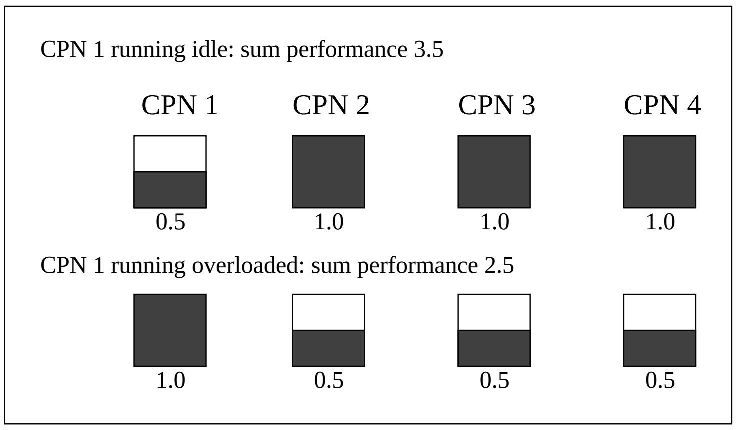
CPN 1 running idle: sum performance 3.5
CPN 1
CPN 2
CPN 3
CPN 4
0.5
1.0
1.0
1.0
CPN 1 running overloaded: sum performance 2.5
1.0
0.5
0.5
0.5

Figure 2.10: *Running Idle Versus Overloaded*

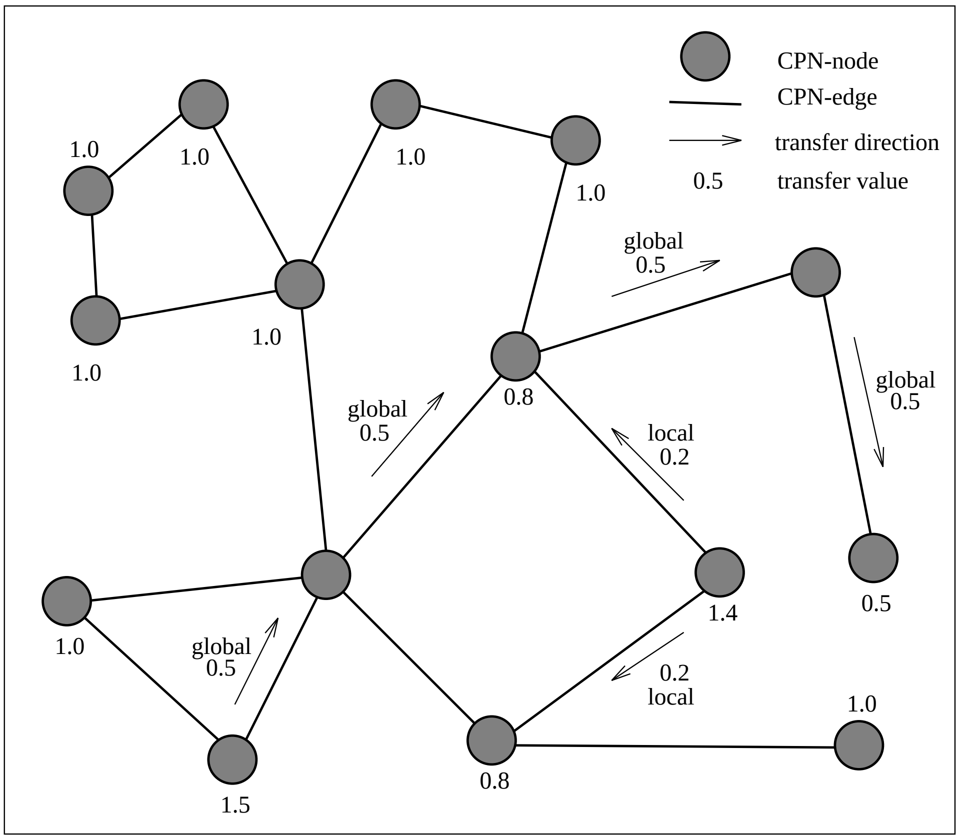

Figure 2.11: *Local and global dynamic load balancing*

execute time is the time used for execution of the `PrepareTimeStep` and `ExecuteTimeStep` methods of the simulation.

idle time is the time the CPN spends idle, that is: waiting for messages to be processed.

graphics time is the time spent on retrieving and displaying graphics. In many cases this value will be non-zero only for the master.

boundary time is the spent on retrieving, sending, and receiving boundaries.

work time is anything else.

Beside those five timers each CPN also computes the estimated load of the residing subnet l in arbitrary units and a performance P:

$$P = \text{corrected load} = \frac{\text{estimated load}}{\text{execution time}} = \frac{l}{t}$$

This value is stored in a queue of constant length handled by `TStatistics`[5]. The minimum value

$$P^{min} = \min_{Q} P$$

of all values in the queue Q is used as a measure of realistic performance. In certain intervals each CPN uses the neighbour statistics mechanism to propegate this P^{min} information to its neighbours together with l .

2.5.4 Local Load Balancing

As soon as a CPN has a complete set from its neighbours it evaluates the P and compares them to its own. Assume a CPN having n neighbours $N_1 \ldots N_n$ with corrected performance minimum $P_1^{min} \ldots P_n^{min}$ and estimated loads $l_1 \ldots l_n$. The local execution time is t_0 and the local load l_0. If the performance with exceeds that of neighbour N_j by a more than minimum percentage t_{min} the CPN will try to offload a part of its network to that neighbour.

To compute the exact amount neccessary let us assume that after the transfer the execution times of the two CPNs t_0 and t_j should be equal. If l_t is the amount of load to be transferred one obtains

$$t_0 := \frac{l_0 - l_t}{P_0^{min}} \overset{!}{=} \frac{l_j + l_t}{P_t^{min}} =: t_j$$

After isolation of l_t one obtains

$$l_t = \frac{l_0 P_j^{min} - l_j P_0^{min}}{P_0^{min} + P_j^{min}}.$$

This l_t is the optimal amount of load that should be transferred. Unfortunately the CPN does not know about the other neighbours of N_j which might offload to N_j, too. So l_t is

[5]currently the length is constant in time steps, but will soon be constant in wall clock time

corrected by a factor c which should depend on the number of neighbours n_j of N_j, for example $c = \frac{1}{n_j}$, so that the effective load amounts to

$$l_t^{eff} = \frac{1}{n_j} \frac{l_0 P_j^{min} - l_j P_0^{min}}{P_0^{min} + P_j^{min}}.$$

2.5.5 Global load balancing

Warning: The following indented feature has not been implemented yet!

term	meaning
n	number of CPN–nodes
l_i	current load on CPN–nodes i
v_i	valence of CPN–node i
$l_{opt} = 1$	optimal load of a CPN
A_i	set of neighbours of CPN i

Table 2.2: *Terms used in descriptions of dynamic load balancing*

`TLoadBalancer` will look for the CPN j having greatest load l_j. Then it will look for k CPNs $n_1 \ldots n_k$ having least loads so that just

$$\sum_{i=1}^{k} (1 - l_{n_i}) > l_j - 1$$

A simple Dijkstra will be run to determine the shortest paths (where the weight of each edge will be assumed as one) $p_1 = e_{11} \ldots e_{1l_1}$ through $p_k = e_{k1} \ldots e_{kl_k}$. The edges on paths p_m for $m = 1, \ldots, k-1$ will be assigned the transfer values

$$\bigwedge_{i=1}^{l_m} transfer(e_{mi}) := 1 - l_{n_m}.$$

On the last path the edges will be assigned

$$\bigwedge_{i=1}^{l_k} transfer(e_{ki}) := l_j - 1 - \sum_{p=1}^{k-1} (1 - l_{n_p}).$$

2.5.6 Simultaneous Transfers

$\Longrightarrow$ **new in 0.9.4**

During load balacing actitivy there might be several simulatenous transfers from or to one CPN. We destinguish between three major cases for three CPNs A, B, and C:

A **receives from** B **and** C**:** This should not impose any problems since the insertion of network topology works independent of the order insertion. Therefore it is safe to *mix* transfers events from both CPNs.

A **sends to** B **and** C**:** This is no problem either since A handles the offloads to B and C sequentially.

A receives from B, but sends to C: This is tricky because due to a certain slack in execution A might have started to receive topology from A before it start sending topology to C. What A might have is an inconsistent subnet which might result in run time errors.

It might be possible to structure sending and receiving of events in such a way that even third case does not result in errors. For the time being the problem is solved by *restricting transfers between neighbouring CPNs to time slots.* To achieve this all graph edges of the CPN topology are colored at the beginning of the simulation using a certain number of colors. The coloring is done in such a way that the each node has as few edges with duplicate colors as posible. The method `ColorEdges` of `TSuperGraphPrimitive` does this job.

In the course of the simulation as CPN boundaries start shifting with respect to each other graph edges may be deleted and new ones created. In case of a new graph edge the corresponding CPNs start negotiating about the new color via the talk mechanism. Until a color is found all transfer across the edge is blocked.

The mechanism is defined by the following parameters: t_{gap} is the gap in time steps between transfer of different colors, t_{delay} is the talk delay for the first transfer, n_c the number of colors, and t_{poll} the polling interval for idle time statistics. Since a complete series of transfers has to be finished before a new is started through new polling there is the following dependency:

$$t_{poll} \quad \geq t_{delay} + n_c t_{gap}.$$

The maximum number of colors neccessary depends on the maximum valence of all CPNs which might be higher than four. Since an increase of colors over four is not desirable because of the implied derease in transfer frequency it will be neccessary to use some colors more than once. In such a case in which more than one edge of the same color have scheduled a transfer one of the edges will be chosen randomly all others will be blocked. $\Longleftarrow$ **new in 0.9.8**

2.6 Transfer of Topology

In the previous section the concepts of dynamic load balancing were described, which determine how much load is to transferred from which CPN to which of its neighbours. Transferring the load itself, namely part of the network topology, will be described in this section.

2.6.1 Synchronization

As mentioned in 2.3.4 neighbouring CPNs may have a difference in time steps $\Delta t = \pm 1$. So in case that such a Δt exists a transfer of topology is not trivial, since the objects residing on the transferred net elements belong to different time steps, too. The soluation of this problem is obtaining local synchronicity of all CPNs involved in the transfer. Of course this synchronicity cannot neccessarily be achieved for the same time that the transfer values are determined because that might make a slower at time step t wait for its faster neighbour at time step $t + 1$ to reach t which will never happen, since there is no roll back mechanism[6].

The parallel toolbox solves this problem by delaying the actual transfer for one time step. This is done through the talk mechanism (see 2.9.1) synchronizing the CPNs at $t + 1$.

2.6.2 Optimizing Communication through Load Balancing

As soon as synchronicity between two CPNs is reached the offloading CPN determines which part if its local subnet it will transfer to its neighbour. As communication is more or less linear to the number of boundary edges which in turn is linear to the circumference of a subnet it will be the objective of this selection to keep the subnet convex and the number of boundary edges as small as possible. Summarized the objectives are as follows:

- Minimize the number of subnets: since each CPN has one subnet to start with this is equivalent with maintaining the connectivity of the given subnet during selection and offloading. This point will be explained in detail in 2.6.5.
- Minimize the number of neighbours: to each neighbour the CPN has to send a message encoding all common boundary edges. So reducing the number of neighbours reduces *the number of messages*. Also it reduces the dependency of the CPNs on each other allowing more slack in communication.
- Minimize the number of boundary edges: this measure mainly reduces the *total message lengths* of the messages sent to the neighbours.
- Optimize the mapping of subnets onto the communication topology: in contrast to local area networks in which communication is usually sequential dedicated systems have a specific underlying communication topology. One could define a transformation which associates locations of the simulation network with locations in the communications network. It would be an additional goal of the load balancing to minimize for example the sum distances of the centers of gravity of the subnets projected into the communication topology to the locations of their corresponding CPNs. That way it would be likely that local commnication of CPNs with their neighbours would actually result in local — and therefore parallel — communication on the dedicated system.

2.6.3 Selecting Topology

After the load balancing routines on CPN A have determined *how much* topology is to be transferred to its neighbour CPN B , it is now neccessary to select *what* topology is to be transferred. The toolbox takes an interativ approach (see 2.12):

[6]a roll back mechanims stores all information neccessary the assume any previous time step back to the time step defined by the minimum of all time steps in the simulation

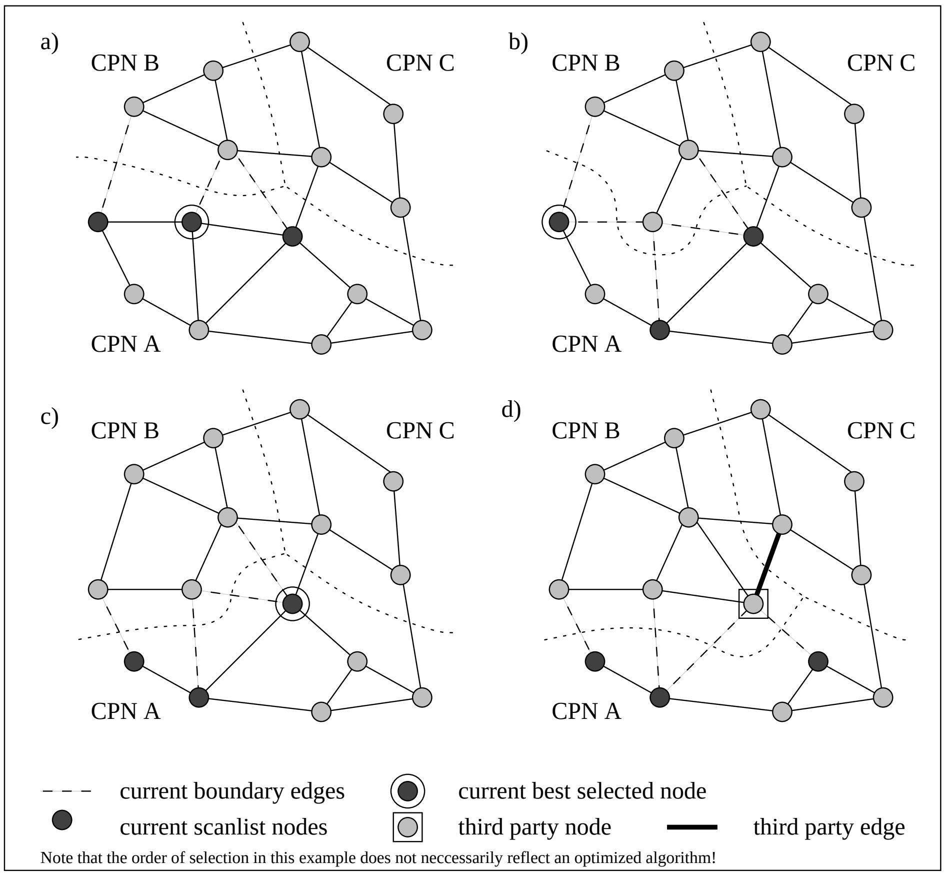

Figure 2.12: *Selecting Topology*

1. CPN A goes through all boundary edges it has in common with CPN B[7]. All local nodes that are reached by those boundary edges are added to a scanlist[8].

2. The method `SelectBestNodeToBeTransferred` is called which selects the best node of current scanlist according to the criteria described in the previous section. For the time being the default functionality is to select the node which is furthest away of the current center of mass of the subnet. $\Leftarrow$ **new in 0.9.4**

3. All edges that lead from a local node to the selected node are marked to be transferred. All local nodes reached by the incedent edges of the selected node are added to the scanlist unless they are already part of it. Then selected node is removed from the scanlist.

4. Repeat steps 2 and 3 until the desired amount of topology is reached.

[7]If there is at least one boundary edge between A and B then there is a CPN-edge between the CPN-node of A and the CPN-node of B. This CPN-edge contains a list of all boundary edges that those two CPNs have in common.

[8]Some nodes may be reached by more than one boundary edge. They are only added once.

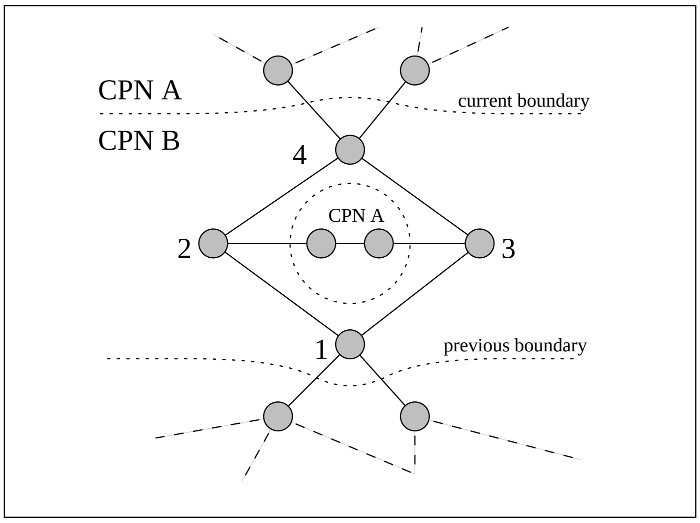

Figure 2.13: *Isolated Subnet after Selection*

2.6.4 Transferring Topology

After selectionon on CPN A the topology has to transferred to the remote CPN B (see 2.6.3). There are two problems that might occur:

- Some nodes which transferred may be referenced by other CPNs which will be called **third party nodes**. The same applies to edges one node of which id references by another CPN. Those are called **third party edges**. In step d) of the figure there is one node on CPN C referencing one of the nodes to be transferred from A to B and one edge between A and C that will connect B and C after transfer. In such a case two measures are necessary:
  - CPN C has to be informed about the change of ownership of the third party node.
  - CPN A has to see to it that boundaries sent by CPN C for the third party edge will be forwarded to CPN B. Note that this neccessary since there is no synchronicity between CPN C and CPN A or B. So if CPN C is fast compared to CPN A it might have already sent out boundaries for topology that is no longer residing on CPN A.
- During selection the method `SelectBestNodeToBeTransferred` might have chosen nodes in such a way that isolated subnets would remain on the CPN (see 2.13). In the example CPN A selects four nodes for transfer (denoted by the order of selection). The two center nodes within the circle are *not* selected so that they remain on CPN A, but isolated from the subnet. A solution for this problem will be described in the section.

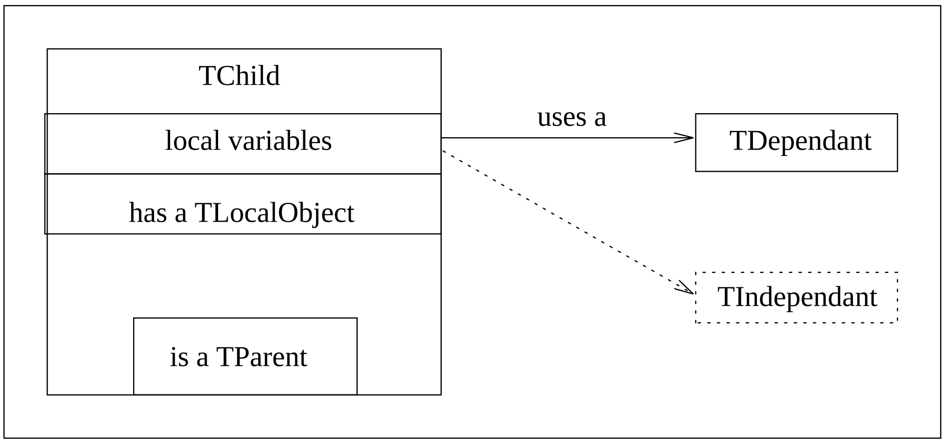

Figure 2.14: *Order of encoding*

2.6.5 Connectivity

The problem of connectivity of subnets has been solved using node clusters as smallest transfer unit[9]. The clustering algorithm will be described in later a version of this handbook. ⇐ **new in 0.9.8**

2.6.6 Encoding and Decoding Data

In case a complex object (e.g. `TApplicationSegment`) is transferred to another CPN, a single event only containing the object itself is not enough since it might contain both contain local objects and references to dependant objects that will have to be transferred as well. Consider an object like the one in figure 2.14. `TChild` is an object derived from object `TParent`. It has local objects such as `TObject` and references to dependants like `TDependant`. The following order will be used to encode the complex object recursively:

1. Encode the parent object `TParent`.
2. Encode all local variables that are not objects and have global scope and encode heap variables referenced by local variables.
3. Encode all local objects such as `TLocalObject`.
4. Encode depandant objects like `TDependant`.

Note that every object is only a depandant of exactly *one* other object although it may be references by several others. Independant objects that are primarily handled by other objects are not encoded. Otherwise they would be multiply created on the destination CPN.

[9]...although the algorithm does not work perfectly yet as of version 0.9.8

On the destination node the object is decoded in exactly the same order. In fact it is the `decoding` and not the `encoding` that determines the order. Part A of the object may have to encoded before part B because part A may contain information about how to decode part B.

2.7 Parallel Environment

2.7.1 Dynamic Insertion of CPNs

The toolbox allows the dynamic insertion of a CPN during the run time of the simulation. This option is advantageous in a local area network since the complete set of workstatios in the network may not be available at the start of the simulation. For time being it is the responsibility of the user to prompt the insertion of CPNs. But it may be possible to write a small utility scanning the LAN for idle CPNs and integrating those CPNs automatically. The following enumeration summarizes the most important steps of an insertion.

1. The user adds another CPN through the PVM interface.
2. The master receives a predefined PVM message containing information about the new CPN CPN_N added and initiates a global synchronization for all CPNs at the end of the current control step.
3. The master uses the idle time data of the most recently time steps to determine the CPN_A that has least idle time.
4. The master which neighbout CPN_B of CPN_A has least idle time.
5. The master prompts CPN_A to transfer a *single* node having a common boundary edge with CPN_B to CPN_N.
6. CPN_A informs CPN_B about the transfer with the new node having the status of a *third party node* and possibly *third party edges* described in 2.6.4.
7. The master initiates the next control step and the simulation continues.
8. Over the next few time steps load balancing is blocked for the new CPN. After that its neighbours will start offloading through relular load balancing until its load is balanced with is those of its neighbours.

2.7.2 Dynamic Deletion of CPNs

As the insertion of a CPN the deletion of a CPN CPN_D will probably only be neccessary in a local area network. This time, however, the event will probably be triggered by the shutdown mechanism of the UNIX operating system. In a normal shutdown on a CPN each process receives the defined signal `SIGHUP` prompting the process to terminate gracefully before the signal `SIGKILL` is issued after after a certain delay. The toolbox uses the signal to remove the CPN from the CPN topology before the CPN is completely shut down. Note that this will probably require a longer delay period between the two signals than defined by default. This can be changed by the system manager.

1. CPN_D receives the signal SIGHUP.
2. CPN_D refuses to accept any road network from its neighbours, but instead tries to offload everything down to a *single* node.
3. CPN_D informs the master if node count has reached one.
4. At the end of the current control step the master initiates a global synchronization.
5. The master picks out one neighbour CPN_N of CPN_D and prompts CPN_D to transfer the remaining node to CPN_N.
6. The master terminates the slave process on CPN_D.
7. The master initiates the next control step and simulation continues.
8. Over the next time steps the neighbours of the deleted CPN will offload to their neighbours (and so on) through regular load balancing until the excess load is distributed over the whole CPN topology.

2.7.3 Parallel Filesystem

Warning: The following indented feature has not been implemented yet!

> The PIOUS toolbox (see [?]) provides a distributed file system based upon PVM. It would be advantageous if the main classes of the Parallel Toolbox provided an interface to PIOUS.

2.8 Important Classes

This section contains a short description of the most important classes of the Parallel Toolbox. A more detailed description of the methods can be found in B.

2.8.1 Base Classes

Four base classes are declared which all other objects will be derived from. They are abstract objects; that is, there will not be any instances of the base objects but only instances of derived objects although none of the object methods will be formally declared abstract to allow for incomplete virtual redefinition.

Class `TBaseObject`

`TBaseObject` is the lowest level base class. Every class that is used with the basic container class `TObjectArray` should be derived from this class. It declares method stubs for class naming `GetClassName` and contents description `GetInstanceName`. Those names are used to build a descriptive prefix prepended to messages (normal, warning, or error) printed to the terminal.

Class `TObject`

`TObject` defines all virtual methods stubs for high level simulation objects like encoding, decoding, event handling, message handling, and active referencing.

Class `TObjectID`

The base class `TObjectID` defines the ID by which each object will be identified during the simulation. A new ID will be assigned upon creation of an object which will be kept until the final destruction of the object. An ID once assigned will never be reused. In case an object is moved from one CPN (source) to another CPN (destination) it will be destroyed on its source CPN and recreated on its destination CPN with the same ID it had before. That way all references to its ID will remain valid although its physical location in the cluster may change.

Class `TTransferObject`

Simulation objects are central to the design approach described here. They represent those entities of the simulation that can be transferred to another CPN in the parallel computer topology. For the computation of the load of subnet consisting of transfer objects the method `GetLoadEstimate` has to be overloaded which returns a value proportional to the load of this computational requirements of this object in arbitrary units.

Classes derived from `TTransferObject` will also take an active part in the simulation update. Therefore the class provides virtual methods `PrepareTimeStep` and `ExecuteTimeStep`.

2.8.2 Classes `TSimulationSlave` and `TApplicationSlave`

`TSimulationSlave` is the topmost object in the simulation hierarchy. The class must be overloaded by an application dependant class called `TApplicationSlave`. There is one instance on each CPN that has pointers to one instance of `TSuperGraph` defining the local subnet computed by the CPN and `TMessageControl` providing the communication interface to PVM. After startup it will take care of initialization of local data structures. Then it will enter the main loop which checks `TMessageControl` whether new messages have arrived and process them. It will only leave this loop at the end of the simulation.

The duties of `TSimulationSlave` can be summarized as follows:

- Handle the topology of the local subnet through `TSuperGraph`.
- Handle the simulation objects residing on the local subnet (also through `TSuperGraph`).
- Provide statistics of the subnet.
- Perform local load balancing.

2.8.3 Classes `TSimulationMaster` and `TApplicationMaster`

The class `TSimulationMaster` is a child of `TSimulationSlave`. It must be overloaded by an application dependant class called `TApplicationMaster`. There is only CPN in the parallel topology that has a `TSimulationMaster` instead of a `TSimulationSlave`. In addition to the latter it has another pointer to an instance of `TSuperGraph` comprising the *whole* simulation network.

The duties of `TSimulationMaster` can be summarized as follows:

- Read in network data or create a artificial network.
- Distribute the network onto available CPNs.
- Provide a interface to control the course of the simulation.
- Gather global statistics.
- Provide a graphical user interface.
- Control deletion and insertion of CPNs.
- Later: perform global load balancing.

2.8.4 Classes `TBaseNode` and `TApplicationNode`

The overloaded *methods* of `TApplicationNode` define the functionality of a node of the simulation network. The *instances* of `TApplicationNode` (together with those of `TApplicationEdge` combined in a graph represent the simulation network itself. `TBaseNode` is derived from `TTranferObject` to enable it to be transferred and compute part of the simulation.

`TBaseNode` has a geometric location of type `TLocation` and handles a list of all its incedent edges in a `TDLList<>`.

Application	Node Functionality	Edge Functionality
CA demo (circle)	none, except local boundaries	simple CA
CA demo (net)	random distribution	simple CA
Grid Extension	subgrid	none, except except definition of boundary dependencies
TRANSIMS	traffic lights + queues	intelligent CA

Table 2.3: *Node And Edge Functionality*

2.8.5 Classes `TBaseEdge` and `TApplicationEdge`

The function of `TBaseEdge` and `TApplicationEdge` is equivalent to the that of `TBaseNode` and `TApplicatioNode`: the overloaded *methods* of `TApplicationEdge` define the functionality of an edge of the simulation network. The *instances* of `TApplicationEdge` (together with those of `TApplicationNode` combined in a graph represent the simulation network itself. Note that there does not have to be any real edge functionality as well as there does not have to be any node functionality[10]. The table 2.3 shows an overview of the classes in different applications.

`TBaseEdge` is derived from `TTranferObject` to enable it to be transferred and compute part of the simulation.

But in one respect edges are special: in case an edge goes from one CPN to another it will be split in halves. For that reason each edge has an **active range** $[a, b]$ where a is associated with its first (from) node and b with second (to) node. The table B.1 gives an overview over the different combinations of a and b. For the time being edges are only cut in the middle so that the split point s_p is always 0.5.

`TBaseEdge` provides method stubs to define the behaviour of the edge in case it functions as a boundary edge. This also includes a method to generate an object of type `TApplicationBoundary` into which all boundary data is encoded. The boundary ranges for from–boundaries and to–boundaries with a given boundary width of b_w are also contained in the table.

Note that although `TBaseEdge` is said to have *from–node* and a *to–node* this does not imply that object movement (is there is any) is only meant to follow that direction. If the edge represents a street segment it is the reponsibility of `TApplicationEdge` to provide both logical directions.

2.8.6 Class `TGraph(Primitive)`

The class `TGraphPrimitive` handles a graph constisting of instances of `TBaseNode` and `TBaseEdge` defining the simulation network. It is derived from `TBaseNode` enabling it to

[10]of course, either has to have at least some functionality

be node in a supergraph itself. Moreover it has a geometric location which is computed as the center of mass of all its nodes. Methods like `PrepareTimeStep` or `ExecuteTimeStep` result in the successive calls of the equivalent methods of all its nodes and edges.

The template `TGraph<>` is used with the user supplied classes `TApplicationNode` and `TApplicationEdge` in order to provide automatic generation of nodes and edges.

2.8.7 Classes `TSuperGraph(Primitive)` and `TGraphEdge`

The class `TSuperGraphPrimitive` operates on graphs of type `TGraphPrimitive` derived from `TBaseNode`) and graph edges (of type `TGraphEdge` derived from `TBaseEdge`) instead of the base classes themselves. It represents the CPN topology as a graph of graphs since the CPNs are arranged in a graph and each CPN holds a local subnet which is a graph in itself.

The class `TGraphEdge` connects graphs; that is, nodes in a supergraph. The instance of `TGraphEdge` between CPNs A and CPN B holds a list of all boundary edges of type `TApplicationEdge` going from A to B. The existance of graph edge is equivalent to the existance of at least one boundary edge.

The template `TSuperGraph<>` is used with the user supplied classes `TApplicationNode` and `TApplicationEdge` in order to provide automatic generation of nodes and edges.

2.8.8 Classes `TBoundary` and `TApplicationBoundary`

The class `TBoundary` serves as a container to transfer boundary information between CPNs in case of external boundaries or between edges and node in case of internal boundaries.

2.9 Advanced Topics

2.9.1 Talk Mechanism

The talk mechanism provides a means to perform a locally synchronized exchange of data. This is mainly needed for network transfer during dynamic load balancing since the time steps of the sending and the receiving CPNs have to match. A *talk* between CPN A from which the it **originates** and CPN B **answering** it goes like this:

1. A calls the method `RequestTalk` of `TSimulationSlave` passing three parameters:
   - the ID of the other CPN,
   - the `TalkID`, a unique number by which the 'topic' to be talked about is identified,
   - a ΔT relative to the current time step T_{now} after which the talk is to be established.

2. A continues execution until the requested time step $T_{talk} = T_{now} + \Delta T$ is reached. In between B has received the talk request. It also continues execution.
3. A and B have reached the same time step T_{talk}. The method `Talk` of `TSimulationSlave` on A is called with the chosen `TalkID` and an instance of `TTalkInfo`.
4. A starts talking to B in a ping pong manner until either of them cancels the talk.
5. Both continue their normal operation.

There are some aspects which need special attention:

- If both CPNs request equivalent talk requests with the same `TalkID` and for the same effective time steps a collision occurs. In that case one of the talk requests is canceled by the system using the IDs of the CPNs and the time step to determine which one.
- A CPN might issue more than one request for the same time step with more than one CPN using one or more talk Ids. In such a case no assumption can be made in which order the talks are initiated except that requests for the same time step with the same CPN and different talk IDs are handled exactly in the order they were requested.
- There is only one active talk at a time.
- During the talk neither of the CPN performs any computation. Therefore it should be the goal of the talk protocols to be as short as possible.

2.9.2 Memory Management

For all objects that are likely to change CPNs (such as `TDriver`, `TVehicle` or `TRoute`) optimized allocation (`getmem`) and deallocation methods (`free`) will be supplied to keep administrive overhead as small as possible. Free memory for instances of these objects will be handled in arrays with lists linking the free entries. Dynamic memory allocation will only be neccessary if the number of objects of a given type temporarily exceeds the estimated maximum number of array entries.

2.9.3 Changing Load Balancing Behaviour

A change of load balancing behaviour of the simulation can be achieved by

- overloading the methods `GetLoadEstimate` in `TBaseNode` and `TBaseEdge` and
- overloading the method `SelectLoadToBeTransferred` in `TSimulationSlave`.

Whereas the first part should always be done by any application to garantee a decent load balancing, the second part is considerably more difficult and requires a new load balacing algorithms to replace the ones builtin.

2.9.4 Changing Topology Transfer Behaviour

A change of topology transfer behaviour is done by overloading either the method `SelectBestNodeToBeTransferred` or the method `SelectTopologyToBeTransferred` of `TSimulationSlave`.

In the first case the selection of topology would still be iterative; that is, one node to be transferred is selected at a time and after each selection the configuration has to be evaluated again.

In the second — more complicated — case it is possible to provide a completely different selection method possibly selecting the whole subnet to be transferred *at once*.

2.10 Problems

2.10.1 Granularity

Boundaries will be located in the middle of segments to avoid the complex behaviour and references close to nodes. Nevertheless the point where a segment is split is only *indepedent* of the associated nodes if the segment is twice as long as vision range. This reduces the potential number of segments that are suited for the placement of a boundary considerably and thus creates a coarser grid for load balancing.

2.10.2 Synchronization

Weak synchronicity between CPNs is neccessary in a time step driven system, meaning that almost equal execution times are desirable on all CPNs, but no master clock forcing *strong* synchronicity.

During the first tests in turns out that at least in a local area network synchronicity should only be reached to a certain exactness since the more synchronous the CPN run the more focussed communication becomes, which can only be handled sequentially. So it might be advantageous to actually force a little *slack* of synchronicity between the CPNs in order to smoothen network communication.

2.10.3 Scalability

Although the microsimulation part itself scales well with the number of CPNs provided[11] there are several aspects that scale only poorly:

- Processing of the input data is done by the master CPN. This also results in a severe disparity of memory requirements since the whole network has to be stored to execute the load balancing.

[11]This only applies to parallel computer hardware that provides simultaneous local communication scaling with the number of CPNs

- Graphics output is handled by a single CPN.
- Global actions such as gathering statistics or command dispatch triggers dense cascades of local and/or global communication. This will especially affect workstation clusters where communication between any pair of CPNs is always *sequential*.

2.10.4 Fault tolerance

For the time being the toolbox does *not* provide any mechanism to recover or even resume a simulation after a fatal crash of one of the CPNs and/or other hardware components.

It should be possible to provide a `Dump` method prompting all objects to dump their current state to a disk file. Together with a dumped representation of the network structure and distribution handled by the toolbox the state of the simulation could be restored again. The maximum possible loss would be the computation done between two subsequent dumps which will probably in the order of minutes versus a computation time of hours.

2.11 Coming Up Soon

These features should be implemented pretty soon:

Local load balancing on complex networks This will be done and tested in several steps:

- planar homogenous nets
- non–planar homogenous nets
- autobahn network
- city network

Port to Paragon The Paragon will serve as the first dedicated parallel system that the toolboy will be tested on.

Time dependence of effective load history The mechanism used to determine the effective performance of a node in a local area network is still based upon a history measured in time steps. This will be changed to wall clock time. At the same time the syntax extension of PVM hostfile will be adapted to allow for different history length for individual CPNs.

2.12 Outlook

2.12.1 PMI

It might be possible to port the Parallel Toolbox to the new standarized Parallel Message Passing Interface. Since it is supposed to provide an equivalent super set of all features of PVM and since the toolbox only uses a subset of all PVM features the port should be pretty easy[12].

2.12.2 Shared Memory

In contrast to the port to PMI which only replaces an existing interfaces with another maintaining the old restrictions a port to shared memory could be a real enhancement of the Toolbox: one of the major disadvantages of message based simulations is a **restricted horizon**; that is, a restricted availability of data residing on a remote CPN. In case of a simulation with a sometimes large *range of causality* (see 2.3) like a traffic simulation this horizon leads to definitive decrease of fidelity.

On shared memory systems a global address space by all processors executing the simulation in well defined **threads**. All requests referring to all other objects can be resolved with a delay which is almost equivalent with conventional single processors memory access.

It would be a good idea to port the Toolbox and the Toolbox based applications to a shared memory system in three steps:

1. In the first step the shared memory implementation would be implemented in such a way that it *emulates* the message based system. The application would not have to be modified.
2. Then additional features are added which are only possible in global address space. This mainly refers to application itself. It should be possible to run the simulation in two modes and compare performance. This also permits to investigate the interesting question whether the limited horizon due to message passing *really* produces different result or whether this is simply an assumption of users feeling uneasy about *a limited view* restricting their potential to reason about how to make objects react in the simulation.
3. If shared memory systems are widely available the application and possibly the Toolbox could be freed of the old message passing aspects.

[12]Any volunteers? ⌣̈

Chapter 3

How To Use The Parallel Toolbox

3.1 Building an Application Framework

To write and run a new application based upon the Parallel Toolbox follow these steps:

1. Install PVM if neccessary.
2. Install the Parallel Toolbox and the Grid Extension.
3. Create a subdirectory for your application called `myapp`. Create a subsubdirectory `myapp/PVM_ARCH`. Use `touch` to create the file `.depend` in that directory. Copy the AppMakefile from the subdirectory `frame` to `myapp`. Create a logical link from `Makefile` to `../MyAppMakefile`.
4. Copy the following dummy files from the `frame` subdirectory into the `myapp` subdirectory and rename `App` into `MyApp` in each filename. Note that all filenames in table 3.1 are relative to `$CHOME`.
5. Modify the files and adapt them to your needs by supplying methods overloading the builtin default functionality.

frame file	application file
`frame/AppMakefile`	`myapp/MyAppMakefile`
`frame/AppTemplates.C`	`myapp/Templates.C`
`frame/App.C`	`myapp/MyApp.C`
`frame/Slave.C`	`myapp/MyAppSlave.C`
`frame/Master.C`	`myapp/MyAppMaster.C`
`frame/Node.C`	`myapp/MyAppNode.C`
`frame/Edge.C`	`myapp/MyAppEdge.C`
`frame/Boundary.C`	`myapp/MyAppBoundary.C`

Table 3.1: *Files of the Grid Application Framework*

6. Compile and link your program by entering the `myapp/$PVM_ARCH` directory and typing `Make`.
7. Create a logical link from `/pvm3/bin/$PVM_ARCH/MyApp` to `$CHOME/myapp/$PVM_ARCH/MyApp` or create a shell script file `/pvm3/bin/$PVM_ARCH/MyApp` calling `$CHOME/myapp/$PVM_ARCH/MyApp`. If your do the latter you have the option to call your application using `nice -level`. Don't forget to give the script execute permission by typing

   chmod +x /pvm3/bin/$PVM_ARCH/MyApp

8. Run your application (see 3.7).

3.2 Overloading Methods

The main idea of defining an interface using an object oriented language is offering virtual method stubs which can/must be overloaded by methods provided by the user. With respect to traditional programming in which the user both influenced *the in which* functions were called and the functionality, the user now only provides functionality for a task and leaves calling of the methods to the toolbox. Depending on the desired effect there are different options on how to *overload* a method:

- A method is not overloaded: This usually implies that the method stub has a builtin default functionalilty which meets the needs of the user.
- A method is overloaded, the ancestor method is not called: the user completely replaces the functionality of the builtin method.
- A method is called and the ancestor method is called: this is a very frequent case in which the user *extends* the functionality of original method. There might be restrictions as to *when* the ancestor method is called: at the beginning of, at the end of, or at anytime during the execution of the overloaded method.

In the appendix you will find descriptions of the most important methods of the toolbox. If there are any restrictions or requirements as far as the overloading is concerned they will be mentioned in the appropriate *Overload* subsection of the description.

3.3 Simulation Control

The method `SimulationControl` in class `TSimulationMaster` is the central method to influence the execution of the simulation. As mentioned before the master also functions as a slave *during* a simulation sequence. Before the first sequence and between two sequences the master is the only acting CPN (see 3.1). This is used to initialize the simulation, start statistics and start simulation sequences. It is also used to terminate the simulation.

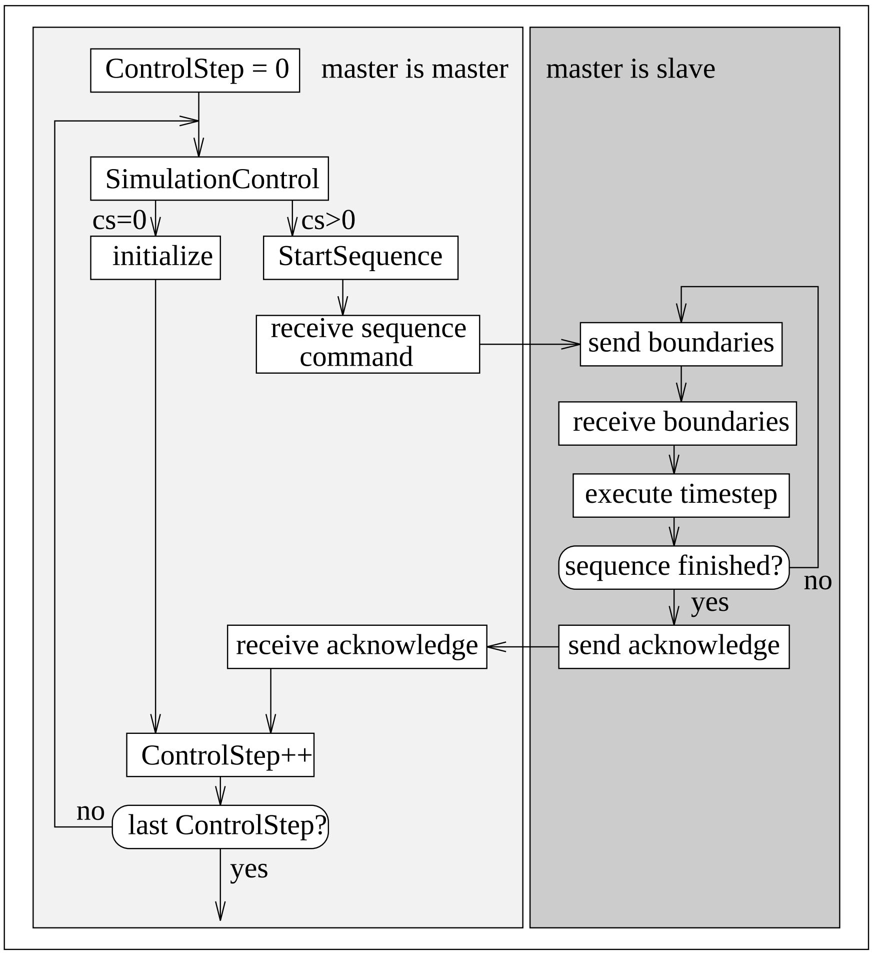

Figure 3.1: *Master and Simulation Control*

3.3.1 Initializing

All initialization of the slave and the master should be done in the constructors `TMyAppSlave(TMessageControl &theMessageControl)` and `TMyAppMaster(TMessageControl &theMessageControl)` which have to be provided. Initialization that cannot be done in a constructor should be done by overloading `StartSlave` or `StartMaster` respectively.

3.3.2 Generating a Network

A simulation network consisting of instances of `TApplicationNode` and `TApplicationEdge` is generated by instantiating the nodes and edges and adding them to the `GlobalNet` provided as local variable in `TSimulationMaster`. The class `TSuperGraph` provides to methods `Add(TBaseNode *aNode)` and `Add(TBaseEdge *anEdge)` to add the elements. Note that an edge can only be inserted into the net *after* both adjacent nodes have already been added. Otherwise there will be error during the update of the incedence list of either node.

Note that during the creation of the simulation network you do not have to consider the upcoming distribution of the network. You are allowed to regard the simulation a *single* net without boundaries. Afterwards the method `DistributeAndActivateNet` should be called taking care of distribution and similar tasks to prepare the system for the first simulation sequence.

3.3.3 Activating Load Balancing

The method `ActivateLoadBalacing` is used to activate the load balancing mechanism.

3.3.4 Activating Statistics

If a statistics object has properly been defined and handled through overloading methods in `TSimulationSlave` and `TSimulationMaster` (see 3.4) the master method `StartStatistics` can be used to activate the polling mechnism for a specific `StatID`.

3.3.5 Initiating a Simulation Sequence

The master method `StartSimulationSequence` can be used to initiate a sequence of time steps in which all slaves (and the master as slave) compute the simulation. The main parameter is the length of the sequence. When choosing this number one has to consider the following tradeoff:

- Keeping the number large reduces the frequency of global synchronizations which usually to not scale very well. Thus large numbers probably improve the *performance* of the simulation.
- Keeping the number small improves the response time of the simulation. Some incidents (e.g. the deletion of a CPN) *require* global synchronization which is needed very fast after the occurance. Thus small numbers probably improve *stability* of the simulation.

There is an additional parameter `SequenceID` by which the slave can distiguish between different types of sequences.

3.3.6 Terminating the Simulation

The simulation is terminated by setting the `Quit` parameter in `SimulationControl` to `TRUE`.

3.4 Gathering Statistics

The Parallel Toolbox supplies a mechanism to request statistitical data from each CPN. The user can define as many statistics as desired by differentiating them through an integer ID called `StatID`. There are two criteria describing the way the statistics are gathered:

local statistics versus global statistics In case of **local statistics** statistical data is requested from each CPN and broadcast to all its neighbours. As soon as a CPN has a complete set of statistics from its neighbours either of the methods `HandleStatResult` or `HandleMultiStatResult` is called with the `StatID` and the statistical infos as parameters.

In the case of **global statistics** all slaves (including the master functioning as slave) are prompted to send their statistical data to the master. As soon as the master has a complete set of statistics from its neighbours either of the methods `HandleStatResult` or `HandleMultiStatResult` is called with the `StatID` and the statistical infos as parameters.

multi statistics versus single statistics In case of **multi statistics** each info received from the CPNs (either all slaves or all neighbours) is stored *individually* in an array structure which is passed through the method `HandleMultiStatResult`. There is no automatics aggregation done. The idle time information, for example, is a multi statistics because the individual idle time of each neighbour is important.

In case of **single statistics** the first info received will stored. Any further info will be combined with the first info using the overloaded method `Add` which has to be supplied by the user. After aggragation the second info is disposed. The result is a single statistics info which is passed through `HandleStatResult`.

Because of order of aggregation is not known to the user the operation `add` has to be both *associative* and *commutative*. If A, B, and C represent statistic infos and $\circ$ the operation this is equivalent to:

$$(A \circ B) \circ C = A \circ (B \circ C) \qquad \text{and} \qquad A \circ B = B \circ A$$

In order to use the statistics feature follow these steps:

1. Declare a descendant `TMyStatObject` of the class `TStatObject` provided in `TStat.h` and overwrite the methods `Decode` and `Encode`, and in case of a single statistics also the method `Add`. Provide a trivial contructor `TMyStatObject(void)` and `TMyStatObject(int StatID)` which both initialize all local variables of the instance. *Initialized* means in particular that the instance should act the null-object with respect to the operation `Add`.
2. Declare an ID for this statistics by choosing an integer value starting at `SC_StatID_FirstFreeID`. In case of the Grid Extension Application choose a value starting at `Grid_StatID_FirstFreeID`.

3. Overload the method `TSimulationSlave::CreateStatObject` creating instances of the statistics object.

4. Overload the method `TSimulationSlave::Handle(Multi)StatResult` to handle the statistics results.

5. Insert a call to `ActivateStatistics` in your `SimulationControl` method in your derived class of `TSimulationMaster`.

6. Check the detailed description of these methods in B.

3.5 Graphics

3.5.1 Network Windows

The network window is activated by command line option `-x` together with the method `OpenGraphics` of `TSimulationMaster`. It displays all edges and nodes in the global graph. The distribution onto the CPNs is denoted by different shades of grey. The display of the nodes can be deactivated through command line option `-u`.

3.5.2 Topology Window

The network window is activated by command line option `-x` together with the method `OpenGraphics` of `TSimulationMaster`. It displays CPNs as nodes (filled circles). The area of the circle of a CPN corresponds to the share of the network that the CPN is computing. An edge between two nodes is equivalent with the existence of at least one boundary edge between the subnets on the two CPNs. The figure printed beside the edges denote the *color* of the edge in the context of transfer slots.

3.5.3 Views

$\Longrightarrow$ **new in 0.9.5**

A view represents an excerpt of the network activity at a high level of detail. At the moment there are two ways to activate a view:

- a **static view** is fixed at a certain *location* to monitor the simulation activity in that region. It is started on the master by creating a new instance of class `TView` and registering it with the master view manager through `TViewManager::AddView`. The master view manager can be retrieved from the graphics manager `TSimulationMaster::GraphicsManager` by calling `TGraphicsManager::GetViewManager`. The view is active until it is removed through `TViewManager::RemoveView`.

- a **dynamic view** is fixed to a certain *object* and will follow this object as the simulation proceeds. It can be started on any slave (including the master) by calling the

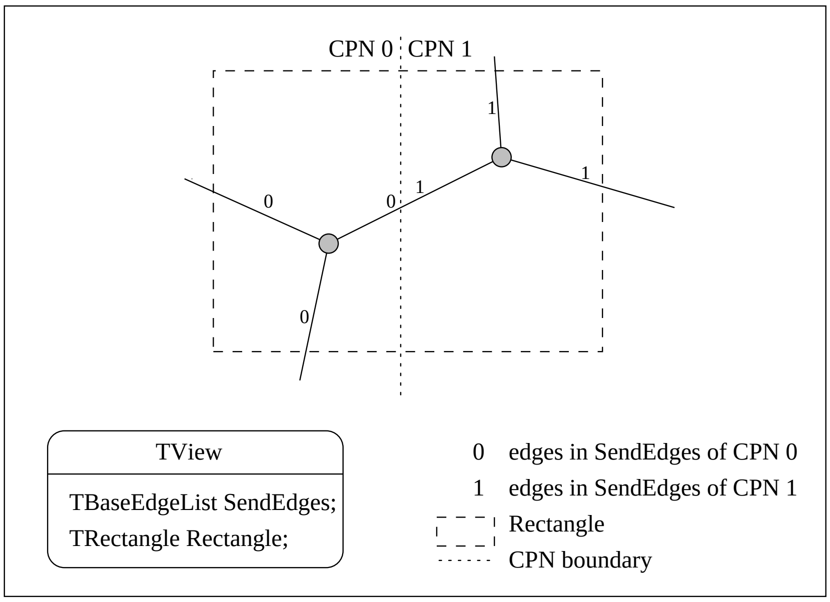

Figure 3.2: *TView On Slaves*

method `TViewManagerSlave::TraceObject` of the local view manager slave which can be retrieved from the slave through `TSimulationSlave::GetViewManagerSlave`. The view is active until the method `TViewManagerSlave::UntraceObject` is called. Note that this can be on a different slave than the one the trace was started on.

To use the view facility several class methods have to be overloaded. These shall be described by looking at the steps neccessary to retrieve the draw data from the slaves and display it on the master.

Retrieving Data on the Slaves

- The slave calls `TViewManagerSlave::SendViewData` for each view.
- `SendViewData` goes through the local list of edges[1] that contain data visible in the current view rectangle.
- For each edge the view calls the overloaded method `EncodeViewData` which encodes all data of the current time step into the message which is passed as a parameter. Note that the data format is completely free since there is a symmetric overloaded method `DecodeViewData` which is called on the master.

[1]So far the current view facility only works properly for edges. Nodes will be added later.

Receiving Data on the Master

- The master receives view data from a slave.
- It decodes the view ID and passes the event to the selected view.
- The view decodes the time step `ReceiveTimeStep` of the data. Then it compares the `ReceiveTimeStep` to the current `DrawTimeStep` and increases `DrawTimeStep` so that it is `VIEW_DRAW_DELAY` time steps behind behind the maximum time steps stored for that view. This slack is neccessary to allow for slower CPNs contributing to the display to send their data.
- The view decodes a reference for an edge, retrieves an pointer to the edge and calls `DecodeViewData`. It passes itself and the incoming message as parameters. Moreover it checks whether is has to add the add to its list `DrawEdges`.

  The edge has a local variable of class `TViewDataHandler` through its decendency from `TViewDrawObject`[2]. This data handler can be used to store pointers to the decoded data.

  - First the edge uses the method `TViewDataHandler::GetViewData` to check whether there has already been data decoded for that time step. Note that the method `DecodeViewData` is called twice if the edge is a boundary edge receiving data from two CPNs.
  - If so (pointer not 0) it adds the new data to the existing using the old pointer. If not it should create an instance of a class caplable of holding the new data. This class should be derived from `TBaseObject` with a virtual destructor so that automatic destruction (and thus memory deallocation) is possible.
  - If the pointer has changed or the old pointer was null the edge uses the method `TViewDataDandler::SetViewData` to store the pointer.

Displaying Data on the Master

- The master calls (through the `GraphicsManager`) the method `TViewManager::Draw` passing a pointer to a customized instance of `TViewDrawContext`.
- The view manager walks through its views and calls `TView::Draw` for each view passing the draw context.
- Each view adds a reference to itself to the context and walks through the edges in `DrawEdges` and calling first `DrawViewBackground` and second `DrawViewForeground`. Moreover it checks whether an edge has not received any draw data for more than `VIEW_DRAW_DELAY` time steps. If so that edge is removed from the list `DrawEdges`.
- Each edge uses the method `TViewDataHandler::GetViewData` to retrieve the data. If the pointer is 0 it simply returns.

[2]which also supplies the stubs for encoding and decoding

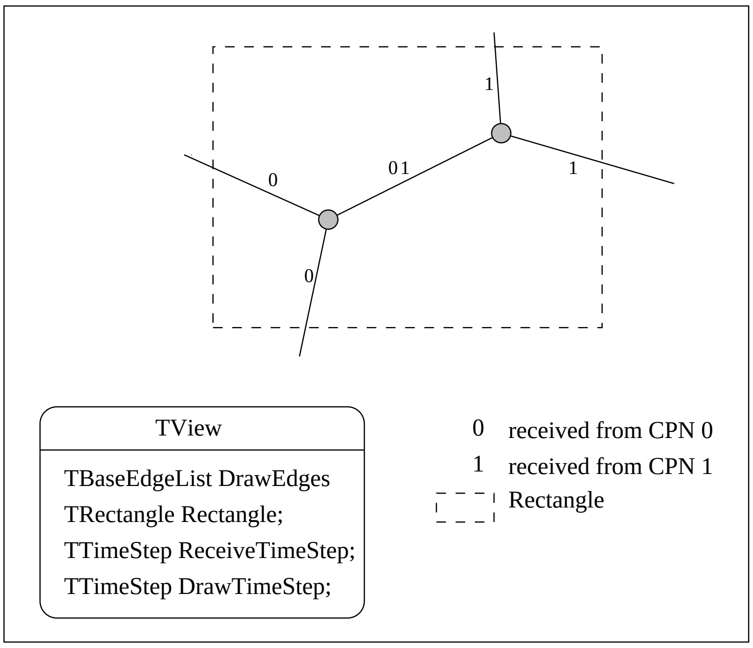

Figure 3.3: *TView on the Master*

3.6 Using Command Line Options

There are up to four classes trying to interpret the options provided on the command line: `TMessageControl`, `TSimulationMaster`, `TGridMaster` and the used supplied `TApplicationMaster`. The user can use any option letter which is not used by any other of the classes involved[3] by doing the following:

- Overload the method `InterpretOption` of `TSimulationMaster` in which you are asked to interpret options.
- Pass a list of your potential options as parameter to the `StartMaster` method of `TSimulationMaster`. The format of the option string is described in the manual page of `getopt(3)`.

3.7 Running the Simulation

3.7.1 Configuring Your System

Since version 0.9.7 it is required to install two scripts `TExecute` and `TExecute.tcsh` which are available in the `$CHOME/scripts` directory. This is usually done by creating the logical links ⟸ **new in 0.9.7**

[3]If the Grid Extension is not used its options are available

Option	Class	Description
`-b<number>`	GM	Sets interval between updates of the bitmap to `number` time steps. If `number` is zero the bitmap is deactivated.
`-c<number>`	MC	Sets number of dedicated nodes to `number`.
`-g<number>`	GM	Sets the number of grid node rows and columns to `number`
`-G<name>`	MC	Activates debugging for CPN `name`
`-h<filename>`	MC	Sets filename of PVM–hostfile.
`-m<kb>`	MC	Sets maximum message length to `kb` kilobytes.
`-N<number>`	SM	Sets nice level to `number`
`-o<number>`	GM	Sets the number of grid point rows and columns per grid node to `number`
`-p<filename>`	SM	Playback load transfer data stored in file `<filename>`[4]
`-u<sec>`	SM	Suppresses plotting of nodes
`-U<timesteps>`	SM	Activates communication check and sets interval to `timesteps`
`-r<mode>`	SM	Sets PVM routing mode to `mode`
`-v<number>`	SM	Sets maximum number of views
`-w<sec>`	SM	Sets communication timeout
`-x`	SM	Activates X-Windows graphics
GM: `TGridMaster` in Grid Extension		
MC: `TMessageControl`		
SM: `TSimulationMaster`		

Table 3.2: *Predefined Toolbox Commandline Options*

```
ln -s $CHOME/scripts/TExecute $HOME/pvm3/bin/$PVM_ARCH
ln -s $CHOME/scripts/TExecute.tcsh $HOME/pvm3/bin
```

Note that the first link has to be duplicated for each PVM architecture being used. The script `TExecute` contains a list of possible locations containing the shell `tcsh`. If the path to the shell on your system is different, please add it to the list.

3.7.2 Using and Troubleshooting PVM

3.7.3 Dedicated Parallel Machine

3.7.4 Architecture of the Intel Paragon

$\Longrightarrow$ **new in 0.9.5**

The nodes on an Intel Paragon mainly are of two types: the **service nodes** and the **compute nodes**. Although both types feature the same processor hardware and the same amount of memory (at least in Jülich (FRG)) their performances and duties differ considerably. The service nodes serve as the connection to the LAN of the institute and supply the main storage media. They also have a fullfledge UNIX operating system capable of supplying an interactive user environment. The compute nodes on the other hand only have a restricted UNIX operating system for running applications linked with special proprietary libraries. They supply the actual computational power of the system since they are interconnected via a highly sophisticated communication network.

Communication between service nodes and compute nodes is supposed to be rather slow. Moreover the service nodes are shared by all users running applications on the paragon so that load level is very high at certain peak levels during the day. The problem is now that the current version of PVM start the *master* process on a *service node*, but subsequently spawns all *slaves* on *compute nodes*. This is very unfortunate since the toolbox usually assigns some topology to the master process, since it would be running idle otherwise most of the time. Therefore there is a need to supply a switch for the Paragon which prohibits the master to take part in the computation of the network. This switch will be implemented in one of the next versions of the toolbox.

Running the Application on the Intel Paragon

It is not quite clear whether the steps described here apply to *all* Paragon systems in general or whether they are restricted to the installation at the KFA in Jülich (FRG).

1. Insert the definitions for the `PVM_ARCH=PGON` and `PVM_ROOT= /pvm3` environment variables into your `.cshrc` or `.profile`. Note that the architecture path does `not` point to the system directory sub tree where the PVM binaries like `pvm` or `pvmd3` reside, but to a local directory in *your* directory tree.

2. Compile the application. The makefile for the Paragon implementation is a somewhat more complicated than the workstation version since running a applications under PVM on the Paragon requires *two* binaries: the master binary linked with `libpvm3.a` and the slave binary linked with `libpvm3pe.a`. The name of the master binary will be the same as that of the application, the slave binary will have the suffix `.slave`.

3. Create directories `/pvm3` and a script `/pvm3/<application>.slave` which calls the slave binary.

4. Create a partition of `N` nodes with name `partname` by calling[5]

   ```
   novpart -sz N partname
   ```

   You will get a subshell which is associated with the new partition.

5. Start the PVM deamon by typing

   ```
   pvmd3 -sz N -pn partname &
   ```

   Together with the previous command and a temporary call of the PVM shell (not neccessary) you will get output looking similar to this:

[5]Allocating a partition probably works differently on every system. So be careful!

```
hkf444@zam158-a.473: novpart -sz 4 pvm
Creating non-overlapping partition "pvm"...
Spawning subshell...
novpart_pvm: pvmd3 -sz 4 -pn pvm &
[1]     4654660
[pvmd pid4654660] using 4 nodes
7f000001:0475
novpart_pvm: pvm
pvmd already running.
pvm> conf
1 host, 1 data format
                    HOST     DTID     ARCH   SPEED
                zam158-a    40000     PGON    1000
pvm> quit
pvmd still running.
novpart_pvm:
```

6. Start your application supplying the mandatory option `-c` possibly followed by other options:

   ```
   myapp -cN <other options>
   ```

   Note that `N` is again the number of nodes in the allocated partition. The reason for this is somewhat strange: PVM only has a *host* configuration which assumes that every host (even the Paragon!) has exactly one processor. There is no function in PVM allowing to query the number of processors of a given host. Therefore although the deamon knows about the partition size you still have to supply the option! There is no way to find out about the number of processors by causing a controlled error while spawing slave number $N+1$ since the system alledgedly *crashes* immediately while doing so. And on the Paragon a crash in one partition might corrupt the whole system!

3.7.5 Local Area Network

Do the following steps to prepare your local area network environment for running the application:

1. If not done yet, create a `hostfile` in your home directory and configure it for all machines in your LAN.
2. Start the PVM shell.
3. Add hosts to your CPN network.
4. Leave the shell.

5. Change directory to `$CHOME/myapp/$PVM_ARCH` and start the application using all desired command line options.

3.7.6 Dynamic Insertion of CPNs

Follow these steps to dynamically insert a new CPN:

1. Enter the PVM shell.
2. Add the CPN.
3. Leave the shell.

3.7.7 Dynamic Deletion of CPNs

At the moment there is no simple way to delete a CPN from the CPN topology. Deleting a CPN in the PVM shell kills the deamon on that CPN which causes the application to crash! Instead to the following:

1. Log into the machine which corresponds to the CPN in question.
2. Use the unix command `ps` to determine the process ID `PID` of the application process. Note that you usually can use `grep` to filter out entries which contain the program name.
3. Type `kill -SIGHUP PID`.

3.8 Debugging the Simulation

3.8.1 Debugging the master process

To debug the master process simply start the application using a debugger. For example:

```
mr@casimir [~/c/periodic/LINUX]$ gdb periodic
GDB is free software and you are welcome to distribute copies of it
 under certain conditions; type "show copying" to see the conditions.
There is absolutely no warranty for GDB; type "show warranty" for details.
GDB 4.13 (i486-slackware-linux), Copyright 1994 Free Software Foundation, Inc...
(gdb) break TPeriodicMaster::Simulate
Breakpoint 1 at 0x5fbd7: file ../TPeriodicMaster.C, line 458.
(gdb) run -s5000
Starting program: /vol/home/mr/c/periodic/LINUX/periodic -s5000
Program::Main() START
| TMessageControl(HOST?): master display is 'casimir:0.0'
| TPeriodicMaster(HOST? TS:0): systemlength set to 5000.000000 [km]
| TPerformance(): statistics file 'PerfStat' opened (300.000000 second intervals)
| THardwareInfoManager(/home/mr/hostfile): reading hostfile...
| THardwareInfoManager(/home/mr/hostfile): 4 info(s) read
| TMessageControl(HOST?): This is the Parallel Toolbox Version 0.9.6 dated 04-24-95 compil
| TMessageControl(HOST?): Retrieving initial PVM topology...
| TMessageControl(HOST?): PVM reports 1 architecture(s) on 2 host(s)
| TMessageControl(4001C, casimir): Spawning 1 slave(s) 'periodic' on speedy.
[t180009] BEGIN
[t180009] TExecute periodic casimir:0.0
[t180009] Program::Main() START
| TMessageControl(4001C, casimir): Hello world!
| TPeriodicMaster(casimir TS:0): This is the Periodic Multi Lane CA demo dated 04-27-95 ..
| TPeriodicMaster(casimir TS:0): SystemLength: 5000.000000 [km]
[t180009] | TMessageControl(180009, speedy): Hello world!
[t180009] | Warning(0) in TMessageControl(180009, speedy): timeout after 10.000000 [second
[t180009] | Warning(0) in TPeriodicSlave(speedy TS:0): TSimulationSlave timeout!
[t180009] | TPeriodicSlave(speedy TS:0): waiting for 2 boundary edges.
[t180009] | TPeriodicSlave(speedy TS:0): OK: message received!

Breakpoint 1, TPeriodicMaster::Simulate (this=0xa0800) at ../TPeriodicMaster.C:458
458         FILE           *StatFile = 0;
(gdb)
```

3.8.2 Debugging a slave process

⟹ **new in 0.9.7**

The command line option `-G<hostname>` activates debugging for all slave processes started on the host specified by `hostname`. So far there is no mechanism to select a specific process on a specific slave. There may be more than one `-G` option given on the command line.

Make sure that the environment variable `HOSTDISPLAY` is set to the XWindows display that the debugging session is being run on! If there are hosts in the PVM topology which do not belong to the local domain the whole name has to be given (e.g.):

```
export HOSTDISPLAY=faure.tsasa.lanl.gov:0.0
```

After start of the application the master process will begin spawning slave processes. Whenever a slave process is on one of the specified hosts an instance of the selected[6] debug tool will appear on the screen. Start the application by typing `run` (or whatever neccessary) after optionally setting break points. Please, note that the master process is *blocked* until you start the execution of the slave process. This is due to the protocol between master and slaves.

[6]so far only xterm with gdb/dbx is available

Option	Description
`-A<speed>`	sets maximum velocity of slow vehicles to `speed`.
`-B<speed>`	sets maximum velocity of fast vehicles to `speed`.
`-D<prob>`	sets deceleration probability to `prob`.
`-L`	activates dynamic load balancing **BEWARE!**.
`-R<ratio>`	sets ratio of slow vehicles to `ratio`.
`-S<ratio>`	sets input scale to `ratio` (default is 1.0).
`-d<density>`	sets overall density of vehicles to `density`
`-l<number>`	sets number of lanes (both directions) to `number`.
`-t<length>`	selects thesis input format (default is TRANSIMS input format).

Table 3.3: *ZPR Micro Simulation Command Line Options*

During a begugging session all output from a debugged slave process that would usually go to the master process will appear on the appropriate debugging tool.

After the application has been terminated on the debugged slave processes the debugging tools *will remain active*. You will have to close them manually, usually by klicking or typing `quit`.

3.9 The ZPR Micro Simulation

3.9.1 Command Line Options

The syntax of the ZPR micro simulation is:

```
MicroSim [options] mapbasename
```

where `mapbasename` will be appended the suffices `.nod` for the node file and `.edg` for the edge (segment) file. The following command line options can be used in addition to those described in 3.6.

3.10 The CA Demo With Periodic Boundary Conditions

3.10.1 Command Line Options

The following command line options can be used when invoking the CA demo in addition to those described in 3.6.

Option	Description
`-A<speed>`	sets maximum velocity of slow vehicles to `speed`.
`-B<speed>`	sets maximum velocity of fast vehicles to `speed`.
`-C<prob>`	sets lane changing probability to `prob`.
`-D<number>`	sets number of decay steps per density `number`.
`-E<bitmask>`	activates reduced gap for lanes given by `bitmask`
`-L<number>`	sets number of lanes (clockwise) to `number`.
`-O<number>`	sets number of sites to look back.
`-R<ratio>`	sets ratio of slow vehicles to `ratio`.
`-S`	activates symmetric lane changing rule set.
`-T<number>`	sets number of simulation steps per density `number`.
`-a<density>`	minimum density used for initial random distribution of vehicles
`-b<density>`	maximum density used for initial random distribution of vehicles
`-e<length>`	edge length in kilometers
`-i<number>`	number of density intervals
`-l<number>`	sets number of lanes (anti clockwise) to `number`.
`-o<name>`	activates statistics and sets statistics filename to `name`
`-p`	activates multi statistics (for each density)
`-q`	activates scatter statistics
`-s<length>`	length of the circular demo network in kilometers

Table 3.4: *Periodic CA Demo Command Line Options*

Option	Description
`-d<density>`	density used for initial random distribution of vehicles
`-n<filename>`	use network stored in TRANSIMS data file `<filename>`
`-t<filename>`	use network stored in thesis data file `<filename>`

Table 3.5: *Network CA Demo Command Line Options*

3.11 The Network CA Demo

3.11.1 Command Line Options

The following command line options can be used when invoking the Network CA demo in addition to those described in 3.6.

Chapter 4

Toolbox Extension for Rectangular Grids

4.1 Concept

As described in the previous chapters the Parallel Toolbox can distribute a certain class of simulations defined on a graph. Looking at classical problems defined on a grid one sees that in fact these problems can be transformed onto a graph in a trivial way:

- The *grid points* of the grid are the *nodes* of the graph. The iteration rules are equivalent with the update rule of one timestep in the graph simulation.
- The *dependencies* between grid points are the *edges* of the graph. A grid point *depends* on another one if it needs the state of point to execute the next time step.

This sounds pretty easy! But wait! Looking at it twice it turns out to be quite an overkill of structural information: a single grid point with usually 4 or 8 bytes would have one associated node and up to 4 associated edges[1]. Estimating the memory requirements for an element at approximately 64 bytes the structural information would outweigh the grid information by a factor of 50!

So we have to find another way to represent the grid in our graph. The solution is quite simple: instead of using grid points as nodes we use *subgrids* of a given size. In figure 4.1 there is a system of 4 by 4 subgrids. Each subgrid is associated with a node and the edges between the nodes exactly represent the dependencies of the subgrids[2]. Note that after distribution most of the edges will handle the exchange of internal boundaries while some will handle the inter–CPN external boundaries. The structures of the internal and external boundaries, however, are identical.

As far as the size of the subgrid is concerned the user has to consider the following trade off:

[1]There are 8 neighbours with eight edges, but each edge is shared by two grid points.

[2]which are exactly equivalent with the corresponding dependencies of a single grid point

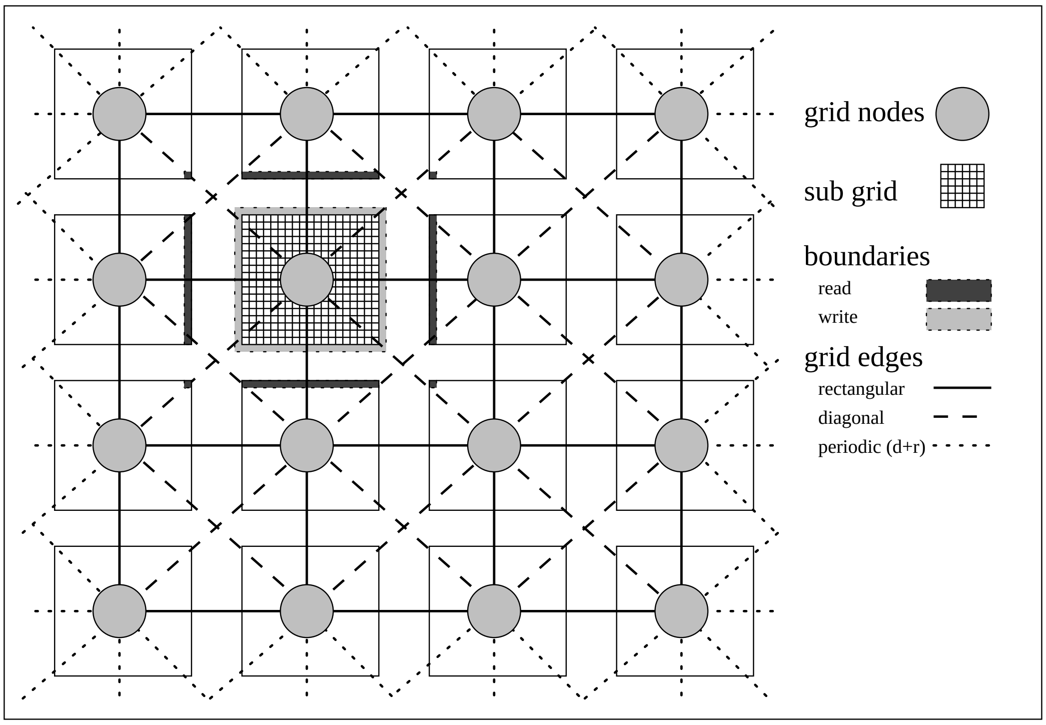

Figure 4.1: *Definition of a Subgrid*

- On the one hand the subgrid should be as large as possible because computation is very efficient on large data segments. Moreover structural overhead is reduced and thus the relative number of boundaries.
- On the other hand load balancing will be done in units of subgrids. So choosing the subgrids small will actually enhance performance on a system in which dynamic load balancing is neccessary.

4.2 Structure of The Grid Extension

The idea of the Grid Extension is to predefine as many descendants of Parallel Toolbox classes as possible to relieve the user of as much programming work as possible. The figure 4.2 shows all predefined classes. Note that only `TMyAppGridSite` and `TMyAppGridMaster`[3] have to be provided by the user.

`TMyAppGridSite` is class which contains all methods to define the functionality of the grid point including:

Initialization All initialization that has to be done before the start of the simulation.

[3]Actually, if the application does not gather any statistical data, but only creates graphics output, `TMyGridAppMaster` is not necessary, either.

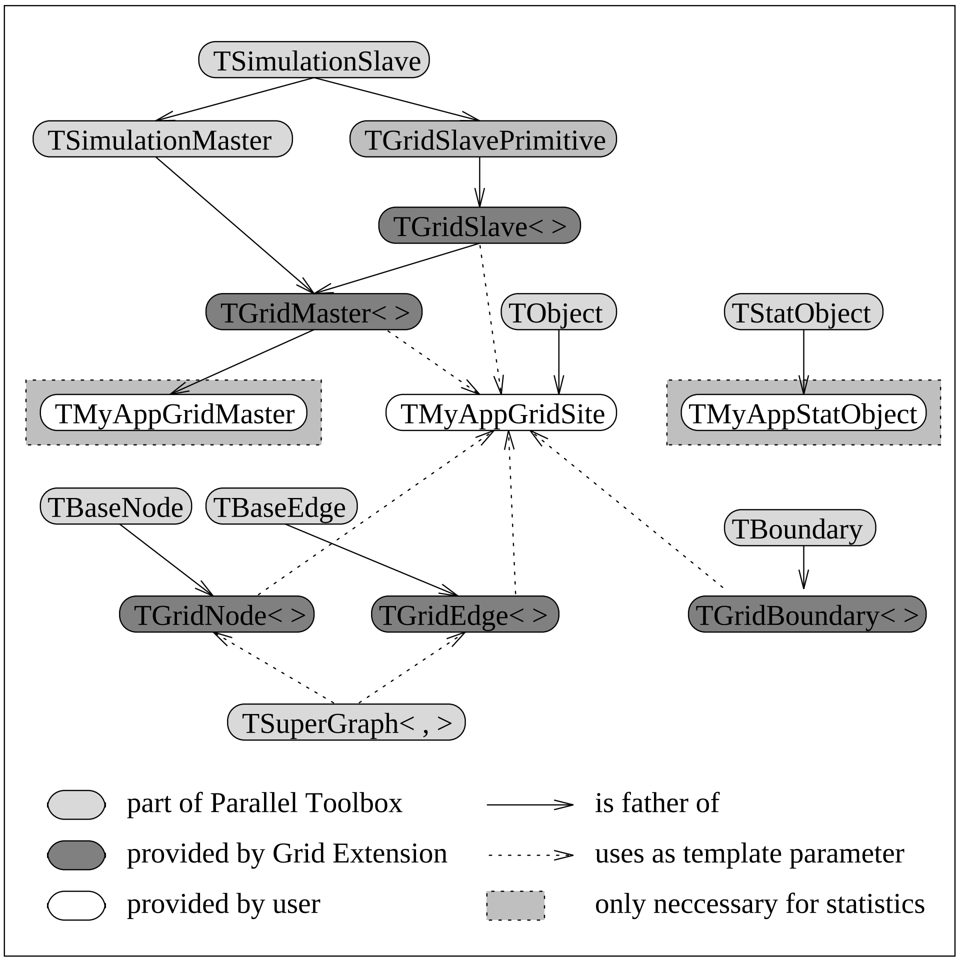

Figure 4.2: *Predefined Decendant Objects Of The Grid Extension*

Prepararing a time step We assume that the update step is devided into two sub steps: first each grid point looks at its current state and those of its neighbours and determines its new state for time step $t + 1$. But it does not update its state because otherwise its neighbours might already use the new state for their update $t \to t + 1$. The new state is stored in a scratch variable.

Executing a time step Now the new state is read from the scratch variable and the update step is complete.

Gathering statistics There are methods to aggregate statistics defined on the characteristics of the grid point.

Defining graphics A very simple graphical display is provided in which each grid point can be assigned a color depending on its current state.

frame file	application file
`frame/GridMakefile`	`myapp/MyAppMakefile`
`frame/GridTemplates.C`	`myapp/Templates.C`
`frame/GridApp.C`	`myapp/MyApplication.C`
`frame/TGridAppGridSite.[C,h]`	`myapp/TMyAppGridSite.[C,h]`
`frame/TGridAppGridMaster.[C,h]`	`myapp/TMyAppGridMaster.[C,h]`

Table 4.1: *Files of the Grid Application Framework*

4.3 How To Use The Grid Extension

4.3.1 Building An Application Framework

You should be vaguely familiar with chapter 3 before you start using the grid extension. After that follow these steps:

1. Install PVM.
2. Install the Parallel Toolbox and the Grid Extension.
3. Create a subdirectory for your application called `myapp`. Create a subsubdirectory `myapp/PVM_ARCH`. Use `touch` to create the file `.depend` in that directory. Copy the GridMakefile from the subdirectory `frame` to `myapp`. Create a logical link from `Makefile` to `../MyAppMakefile`.
4. Copy the following dummy files from the `frame` subdirectory into the `myapp` subdirectory and rename `GridApp` into `MyApp` in each filename. Note that all filenames in table 4.1 are relative to `$CHOME`.
5. Modify the files according to comments contained in those files and the guidelines described in the next section. Most modifications consist of replacing text strings by other text strings. In two cases, however, additional functionality has to be provided:
   - `TMyAppGridSite.[C,h]` is used to define the functionality of a single grid point.
   - `TMyAppGridMaster.[C,h]` is used to define statistics and simulation control.

4.3.2 Defining Grid Point Functionality

The main class of the grid simulation is called `TMyAppGridSite` which is to be derived from the base class `TObject`. Although some methods in `TObject` have to be overloaded by `TMyAppGridSite` most methods are **not** declared `virtual` but `inline` for reasons of performance: due to the simple structure of the simulation topology — namely all grid points are equivalent — there is no need to implement grid point functionality through class overloading. Instead the class `TMyAppGridSite` is used as parameter to most predefined decendant classes of the Grid Extension (see 4.2). To garantee proper compilation and

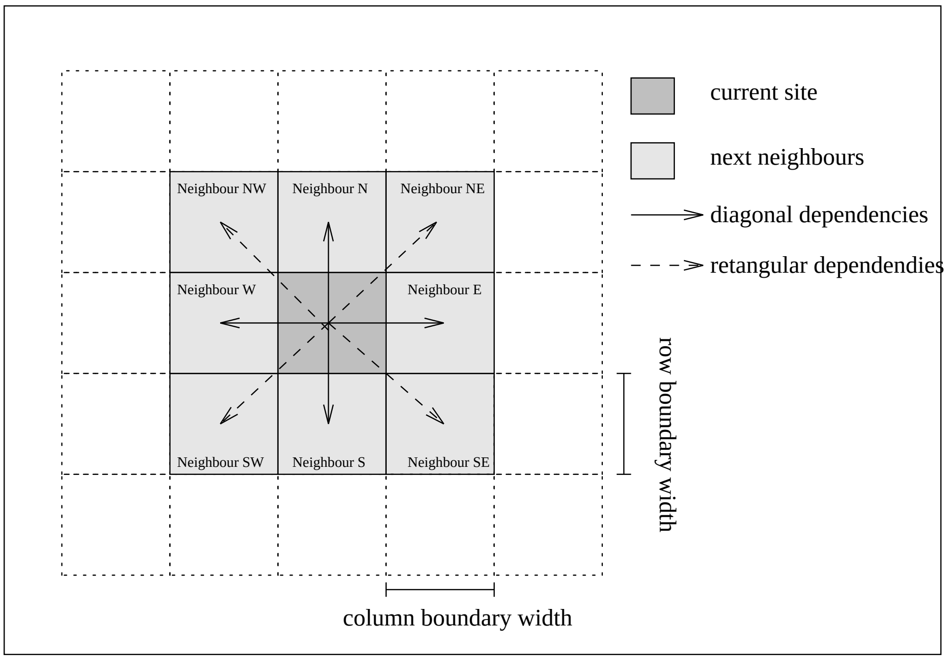

Figure 4.3: *Accessing Nearest Neighbours of a Grid Point*

functioning of the grid point the following methods have to be declared and non-trivially[4] defined for `TMyAppGridSite`.

```
TMyAppGridSite(void)
```

Description The trivial contructor can be empty or can do some basic initialization.

```
TMyAppGridSite operator = (TMyAppGridSite &OtherSite)
```

Description This assignment operator should be used to transfer grid point data from `OtherSite`) to the local instance.

```
bool Decode(TMessage &aMessage)
```

Description `Decode` decodes the grid point data from message `aMessage`.

[4]trivial in this context would mean empty function bodies for `void` methods or `return TRUE;` statements for `bool` methods.

```
bool Encode(TMessage &aMessage)
```

Description `Encode` encodes the grid point data into the message `aMessage`.

```
bool ExecuteTimeStep(TTimeStep TimeStep, TGridNodeInfo &GridNodeInfo)
```

Description `ExecuteTimeStep` executes one time step (iteration) of the grid point possibly using scratch data obtained from the immediately preceding `PrepareTimeStep`. With the help of `GridNodeInfo` the current grid point can access its nearest neighbours through the pointers `Neighbour_Direction(GridNodeInfo)` where `Direction` is one out of `N`, `NE`, `E`, `SE`, `S`, `SW`, `W`, or `NW` (see 4.3).

```
bool Fill(TGridNodeInfo &GridNodeInfo)
```

Description `Fill` initializes the grid point at (and only at) the beginning of the simulation.

```
bool PrepareTimeStep(TTimeStep TimeStep, TGridNodeInfo &GridNodeInfo)
```

Description `PrepareTimeStep` gives the grid point the chance to evaluate its neighbours and store the information neccessary to actually execute — or rather assume the state of — the current time step.

The following methods have to be declared and non-trivially defined if statistics are desired. Otherwise they can be declared and trivially defined.

```
AddToStat(int StatID, TStatObject *StatObject)
```

Description `AddToStat` adds the current grid point to the statistics identified by `StatID` accumulated in `StatObject`.

```
bool static bool CreateStatObject(int StatID, TStatObject *&StatObject,
                                  bool &Handled)
```

Description `CreateStatObject` creates an empty but initialized instance of the user defined statistics object class `TMyAppStatObject` derived from `TStatObject`. The type of statistics is identified by `StatID`. If the ID could be handled `Handled` is to be set to `TRUE`.

The following method has to be declared and non-trivially defined, if a graphical display of the grid is desired.

```
TGridPixel GetPixel(void)
```

Description `GetPixel` returns a value in the range of $[0, \ldots, 255]$ representing the color in which this point shall be displayed.

4.3.3 Simulation Control

For a grid simulation with graphics the class `TGridMaster` need not be overloaded. In case statistics are desired, however, there has to be a derived class `TMyAppGridMaster` overwriting the methods `SimulationControl`. See 3.3 for a detailed description of this method.

The first control step used by the overloaded method should be `GridMaster_CS_StartSequence`. Note that the ancestor method supplies control steps for general initialization so that only the statistics are left to be started.

4.4 Predefined Grid Extension Command Line Options

See table 3.2.

4.5 Game of Life Demo

The *Game Of Life* is a classical cellular automata (CA) defined on a rectangular grid. Its behaviour is simular to the growth of cells on a plane. Each grid point is either empty or occupied by a cell. The update rules are simple:

- If the grid point is empty and exactly three of the eight next neighbours are occupied a new cell is *born*.

- If the grid point is occupied and the cell has either 2 or 3 neighbours it survives. Otherwise it dies.

The demo simulates the CA with a grid of 512 by 512 grid points. The subgrids on the grid nodes are 128 by 128. Instead of simply using a flag whether a point is occupied or not, the *age* of the cell is stored in each grid point. For each generation the cell survives this counter is incremented. An age of zero represents an empty cell. On the graphics screen the grid point will be colored according to its age: the older it is the darker it will appear.

The statistics feature is used to determine the number of cells on the grid and their sum age. This values are printed to the screen every 5 time steps.

4.6 Two Dimensional Traffic CA Demo

$\Longrightarrow$ **new in 0.9.4**

The two dimensional traffic demo simulates traffic of a mixture of vehicles going horizontally and vertically in a grid plane. A certain ratio of grid points are occupied with blocks.

Appendix A

Glossary

Here are some of the technical terms used in this draft just to make sure writer and readers have the same understanding of the matter. The class handling the mentioned object type is given in parentheses.

- **Node (TBaseNode):** Node of the traffic network to which segments are incedent. These are mainly junctions and intersections.
- **Edge or Segment (TBaseEdge):** Part of the traffic network representing a street, road or highway segment between two nodes.
- **Boundary edge** Edges that have nodes in neighbouring CPNs.
- **Network (TGraph):** Entity of all nodes and segments usually provided as ASCII input file.
- **Computational Node (CPN)** One unit in a parallel computer system. Each CPN has a local network which is a connected subnet of the global network.
- **CPN-Edge (TGraphEdge):** An edge which has TGraphs as nodes. It is part of a TSuperGraph.
- **Topology (TSuperGraph):** The way CPNs are arranged and accessed in a parallel computer system. TSuperGraph has nodes which are TGraphs connected by TGraphEdges. The existence of a TGraphEdge means that the subnets on the two associated CPNs are connected by at least one boundary edge of type TBaseEdge.
- **Tile** Part of the geometric area covered by the traffic network that will be assigned to a CPN.

Appendix B

Description of Classes

B.1 Class `TObject`

```
bool DecodeSelf(TMessage &aMessage)
```

Description `DecodeSelf` decodes all data in the local instance of the derived class using the message `aMessage`. See 2.6.6 for the recommended order of decoding.

Overloading The corresponding method of the ancestor class has to be called first before any other decoding is done.

```
bool Encode(TMessage &aMessage)
```

Description `Encode` Encodes all data in the local instance of the derived class using the message `aMessage`. See 2.6.6 for the recommended order of encoding.

Overloading The corresponding method of the ancestor class has to be called first before any other decoding is done.

B.2 Class `TTransferObject`

```
bool Activate(bool FirstTime = FALSE)
```

⟹ **obsolete since 0.9.6**

Description Overload this method in order to be informed about a change of state of a transfer object. It is common to allocate local data structures to accomodate the incoming additional data structures received in `DecodeNodeData` for `TApplicationNode` or `DecodeRange` for `TApplicationEdge`. Upon call of this method it can be assumed that `DecodeSelf` has already been called so that basic object data is available.

The very first time this method is called the parameter `FirstTime` is set to `TRUE`. This can be used to initialize the additional data structures since in this case no subsequent call to `DecodeNodeData` or `DecodeRange` can be expected.

Overloading The ancestor method must be called at the beginning of this method.

```
bool ActivateLevel(int Level)
```

⟹ **new in 0.9.6**

Description Overload this method in order to be informed about a change of state of a transfer object. There are 7 states: `INACTIVE_LEVEL`, `ACTIVE_LEVEL` and `ACTIVE_LEVEL_INIT1` through `ACTIVE_LEVEL_INIT5`.

It is common to allocate local data structures for level `ACTIVE_LEVEL` to accomodate the incoming additional data structures received in `DecodeNodeData` for `TApplicationNode` or `DecodeRange` for `TApplicationEdge`. Upon call of this method it can be assumed that `DecodeSelf` has already been called so that basic object data is available.

The level `ACTIVE_LEVEL` is always called whenever the object isactivated. The other init levels are only called if they are requested by passing the topmost init level to method `TSimulationMaster::DistributeAndActivateNet`. Whenever `ActivateLevel` is called for a certain level L it can be assumed that all other elements have been activated at least to level $L-1$.

The method `TTransferObject::IsFirstActivation` returns `TRUE` if this method is called for the first time. This can be used to initialize the additional data structures since in this case no subsequent call to `DecodeNodeData` or `DecodeRange` can be expected.

The obsolete `Activate` methos can easily be transformed into the new format. Suppose the old method `Activate` looked like this:

```
bool
TDescendent::Activate(bool FirstTime)

{
   if (!ParentClass::Activate(FirstTime))
      return FALSE;
   ...general actions...
   if (FirstTime)
      {
      ...special actions...
      }
   return TRUE;
}
```

The new version with `ActivateLevel` looks like this:

```
bool
TDescendent::ActivateLevel(int Level)

{
   if (!ParentClass::ActivateLevel(Level))
      return FALSE;
   switch (Level)
      {
      case ACTIVE_LEVEL:
         ...general actions...
         if (IsFirstActivation())
            {
            ...special actions...
            }
         break;

      case ACTIVE_LEVEL_INIT1:
      ...
      }
   return TRUE;
}
```

Overloading The ancestor method must be called at the beginning of this method.

```
bool Deactivate(void)
```

Description Overload this method in order to be informed about a change of state of a transfer object. It is common to free local data structures which have just been encoded through `EncodeNodeData` for `TApplicationNode` and `EncodeRange` for `TApplicationEdge`.

Overloading The ancestor method must be called at the beginning of this method.

```
double GetLoadEstimate(void)
```

Description Overload this method to return a value which will be used as the computational requirement imposed by this transfer object class. The value should depend on local parameters of the individual instance like number of elements handled by this instance.

Overloading No call of ancestor method neccessary.

B.3 Class `TStatObject`

A class `TApplicationStat` handling statististical data must be derived from class `TStatObject`. Beside `Encode` and `Decode` which have to be overloaded anyway the most important methods of this class are:

```
constructor TApplicationStat(int StatID)
```

Description This destructor must be supplied for the derived class. It is usually called in `CreateStatObject` of `TApplicationSlave`. All local variables must be initialized to the null object of the operations used for aggregating the statistics.

Overloading The `StatID` must be passed to the constructor of `TStatObject`.

```
bool Add(TStatObject &StatObject)
```

Description The method `Add` adds the statistical data contained in `StatObject` to the local instance. This overloading is only required if the statistics is not started with mode `STAT_FLAG_MULTI_STAT`.

Overloading The ancestor method must be called.

```
bool Evaluate(void)
```

Description Overload this method to make secondary computations based upon the data stored in the info before it is passed to either `HandleStatResult` or `HandleMultiStatResult`.

Overloading It is recommended to call the ancestor method.

```
TTimeStep GetTimeStep(void)
```

Description This method returns the time step that these statistics were aggregated at.

B.4 Class `TBaseNode`

B.4.1 Encoding and Decoding of Nodes

There are two types of nodes: *local active* nodes participating in the simulation and *remote inactive* nodes which only reside on a CPN so that a boundary edges have valid references for both of their adjacent nodes. The toolbox will take care of this difference by calling the methods `Activate` and `Deactivate` for *local* nodes.

Moreover it is assumed that active nodes have local dynamic data structures that have to encoded and decoded in case of a transfer to another CPN. Therefore the methods `EncodeNodeData` and `DecodeNodeData` should be overloaded for `TApplicationNode`.

As a rule of thumb the data de/encoded in `Encode` and `DecodeSelf` on the one hand and `EncodeNodeData` and `DecodeNodeData` on the other hand should be destinguished as follows:

- `Encode` and `DecodeSelf` should take care of very basic information like IDs and flags which reflect the *state* of the node.
- `EncodeNodeData` and `DecodeNodeData` should take care of extended dynamically allocated data structures. The above mentioned flags could be used to determine which data structures to allocate or deallocate.

The calling conventions of the encoding and decoding methods are graphically displayed in figure B.1.

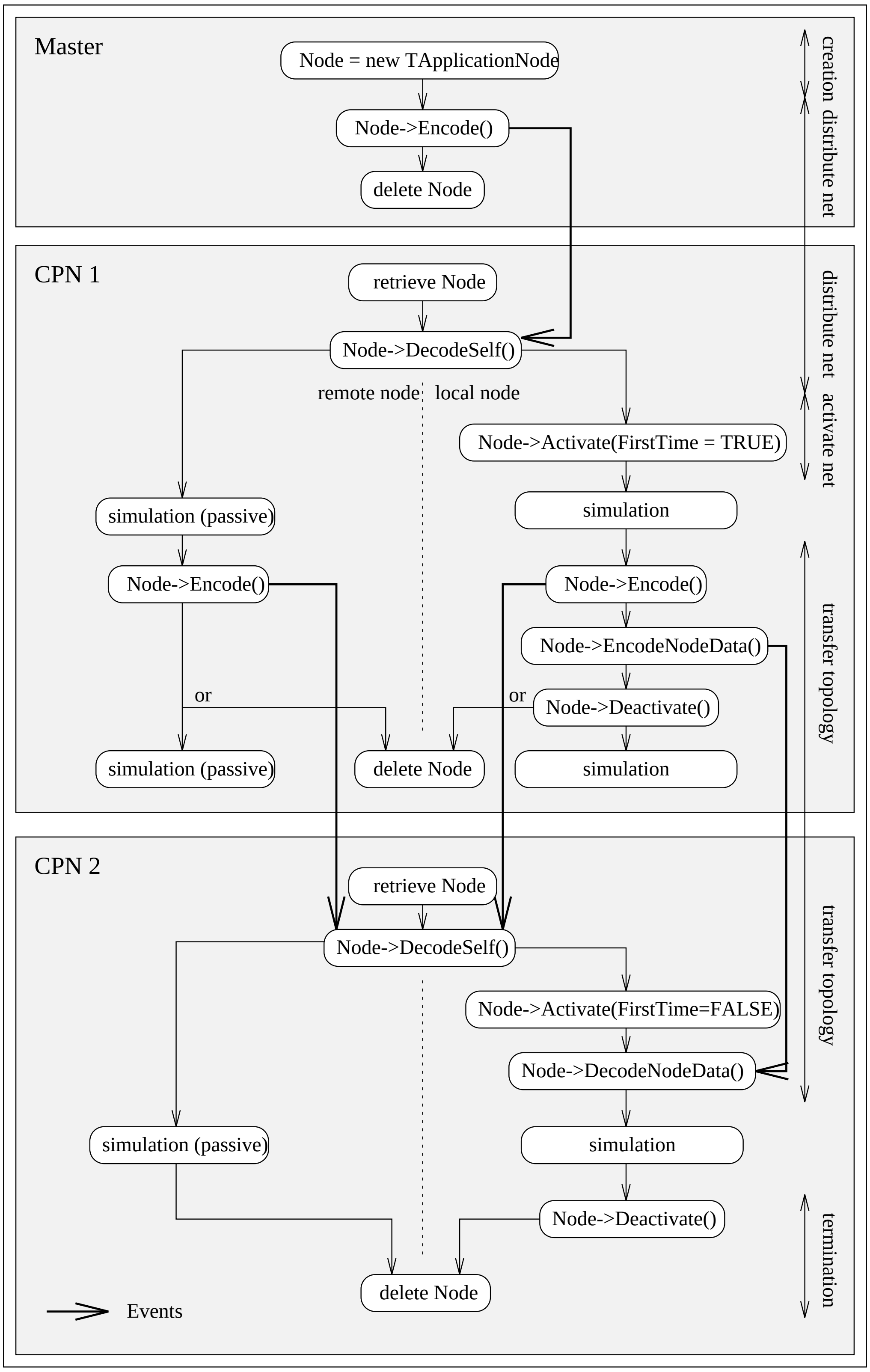

Figure B.1: *Coding and Decoding of* `TApplicationNode`

B.4.2 Methods defining transfer behaviour

```
bool DecodeNodeData(TMessage &aMessage)
```

Description Overload this method to decode additional node data structures from message `aMessage`. Upon call of this method it can assumed that the methods `DecodeSelf` and `Activate` have already been called. ⇐ **new in 0.9.4**

Overloading It is recommended to call the ancestor method at the beginning of this method.

```
virtual bool DecodeSelf(TMessage &aMessage)
```

Description Overload this method to decode basic node information from message `aMessage`.

Overloading The ancestor method must be called at the beginning of this method.

```
bool Encode(TMessage &aMessage)
```

Description Overload this method to encode basic node information into message `aMessage`.

Overloading The ancestor method must be called at the beginning of this method.

```
bool EncodeNodeData(TMessage &aMessage)
```

Description Overload this method to encode additional node information into message `aMessage`. ⇐ **new in 0.9.4**

Overloading It is recommended to call the ancestor method at the beginning of the method.

B.5 Class TBaseEdge

B.5.1 Encoding and Decoding of Edges

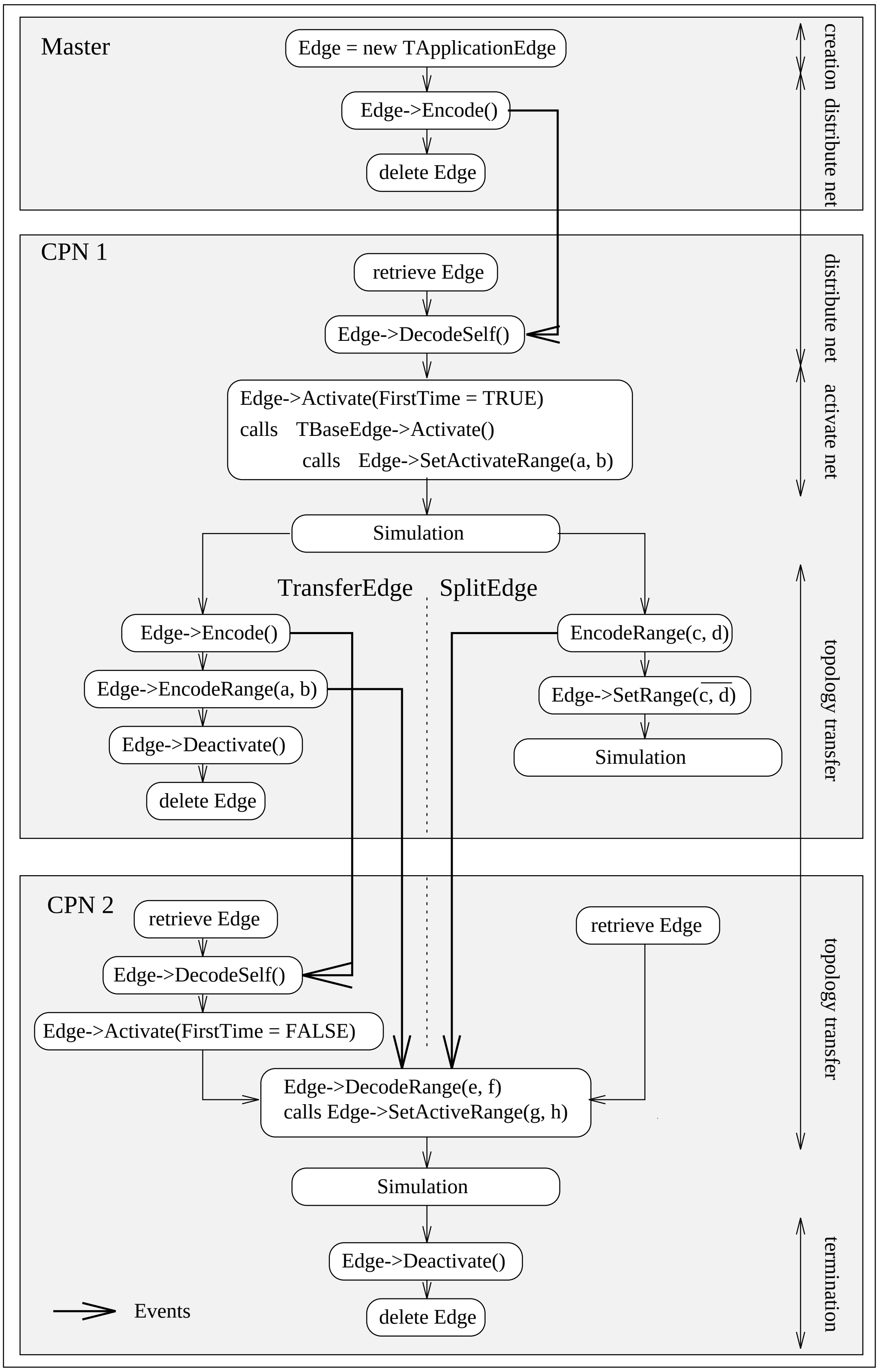

Figure B.2: *Coding and Decoding of* `TApplicationEdge`

In contrast to nodes all edges are *active*. But since some of them are boundary edges only a certain *range* extending from the `FromNode` with value 0.0 to the `ToNode` with value 1.0 is valid. The edge is informed through `SetValidRange` about changes of this range.

As with the nodes it is assumed that the data stored on the edge can be split into *basic* data de/encoded through `DecodeSelf` and `Encode` and *additional* data de/encoded through `DecodeRange` and `EncodeRange`.

The calling conventions of the encoding and decoding methods are graphically displayed in figure B.2.

B.5.2 Methods defining transfer behaviour

```
bool DecodeRange(TMessage &aMessage, double RangeBegin,
                 double RangeEnd)
```

Description Use this method to decode the range [`RangeBegin`, `RangeEnd`] from message `aMessage`. Upon call of this method it can be assumed that methods `Activate`, `DecodeSelf`, and `SetActiveRange` have already been called.

Overloading It is recommended to call of the ancestor method at the beginning of this method.

```
bool DecodeSelf(TMessage &aMessage)
```

Description Overload this method to decode basic edge data from message `aMessage`.

Overloading The ancestor method must be called at the beginning of the method.

```
bool DisposeRange(double RangeBegin, double RangeEnd)
```

Description Overload this method in order to be informed about ranges that are not needed anymore and can thus be deallocated.

Overloading It is recommended to call the ancestor method at the end of this method.

When	range $[a, b]$	from–boundary	to–boundary
edge is local	$[0.0, 1.0]$	$[0.0, b_w]$	$[1.0 - b_w, 1.0]$
edge is boundary, from–node is local	$[0.0, s_p]$	$[0.0, b_w]$	$[s_p - b_w, s_p]$
edge is boundary, to–node is local	$[s_p, 1.0]$	$[s_p, s_p + b_w]$	$[1.0 - b_w, 1.0]$

Table B.1: *Active Ranges of Edges*

```
bool Encode(TMessage &aMessage)
```

Description Overload this method to encode basic edge data into message `aMessage`.

Overloading The ancestor method must be called at the beginning of this method.

```
bool EncodeRange(TMessage &aMessage, double RangeBegin,
                 double RangeEnd)
```

Description Overload this method to encode the range [`RangeBegin`, `RangeEnd`] into message `aMessage`. Note that you are allowed to deallocate the range after encoding. Overload `DisposeRange` for this purpose.

Overloading It is recommended to call the ancestor method at the beginning of this method.

```
bool SetActiveRange(double RangeBegin, double RangeEnd)
```

Description Overload this method to determine the active range of the edge which participates in simulation. The values passed here also define the boundary areas for `GetBoundary`.

Overloading It is recommended to call the ancestor method at the beginning of the method.

B.5.3 Methods defining boundary behaviour

```
bool GetBoundary(TTimeStep TimeStep, TBoundary *&Boundary,
                 TDirection BoundaryDirection)
```

Description Overload this method to fill the boundary `Boundary` which has previously been allocated through `CreateBoundary` of `TSimulationSlave`. You are allowed to type cast `Boundary` into `TApplicationBoundary`. Use the values set by `SetActiveRange`, the parameter `BoundaryDirection` and table B.1 to determine the proper data to encode.

Overloading No call to ancestor method neccessary.

```
bool SetBoundary(TBoundary *Boundary)
```

Description Overload this method to decode boundary information from boundary `Boundary`. You are allowed to type cast `Boundary` into `TApplicationBoundary`.

Overloading No call to ancestor method neccessary.

B.6 Class `TSimulationSlave`

B.6.1 Public Utility Methods

B.6.2 Methods Defining Simulation Behaviour

```
bool BeginSimulationSequence(TTimeStep SequenceStart,
                             TTimeStep SequenceEnd, long SequenceID)
```

Description Overload this function in order to be informed about the beginning of simulation sequences if special preparations are neccessary. The parameter `SequenceID` is the passed in `StartSimulationSequence`.

Overloading No call to ancestor neccessary.

```
bool EndSimulationSequence(void)
```

Description Overload this method in order to be informed about the end of the current sequence.

Overloading No call to ancestor neccessary.

```
bool ExecuteTimeStep(TTimeStep TimeStep)
```

Description Overloading this method defines the action performed for each time step of the simulation sequence. It is the key method of `TSimulationSlave`. Upon call it can be assumed that all neccessary boundaries from the neighbours for the time step `TimeStep` have been received and have been made available.

Overloading Call of the ancestor method is recommended to support the view window graphics option.

B.6.3 Methods Defining Statistics Behaviour

```
bool CreateStatObject(int StatID, TStatObject *&StatObject)
```

Description This method is overloaded to create instances of user–defined statistics objects using the `TMyStatObject(int StatID)` constructor. The function returns a pointer to the new instance in `StatObject`. The return value is `TRUE` in case of success, `FALSE` otherwise.

Overloading In case the `StatID` does not belong to the user–defined IDs the corresponding ancestor method must be called.

```
bool GatherStatistics(int StatID, TStatObject *&StatObject)
```

Description Upon call of this method the slave must retrieve the statistics identified by ID `StatID` from the local subnet. It can be assumed that the instance of `TMyStatObject` pointed at by `StatObject` has been properly initialized!

Overloading In case the `StatID` does not belong to the user–defined IDs the corresponding ancestor method must be called.

```
bool HandleMultiStatResult(int StatID,
                           TMultiStatObjectPrimitive *StatObject)
```

Description This method is equivalent to `HandleStatResult` except for the fact that `StatObject` points to an array of pointers to instances of `TMyStatObject`. A type cast to `(TMultiStatObject<TMyStatObject> *)` may be practical. Note that the array pointed to by `StatObject` must be disposed after use!

Overloading In case the `StatID` does not belong to the user–defined IDs the corresponding ancestor method must be called.

```
bool HandleStatResult(int StatID, TStatObject *&StatObject)
```

Description This method is overloaded to handle the results of user–defined statistics. The pointer `StatObject` points at a instance of the user–defined statistics class containing the aggregated statistics of type `StatID`. To access methods and variables local a type cast to `(TMyStatObject *&)` may be practical. Note that the object pointed to by `StatObject` must be disposed after use! The return value is `TRUE` in case of success, `FALSE` otherwise.

Overloading In case the `StatID` does not belong to the user–defined IDs the corresponding ancestor method must be called.

B.6.4 Methods Defining Miscelleneous Behaviour

```
TBoundary *CreateBoundary(void)
```

Description This method has to be overloaded if a user defined `TApplicationBoundary` is used instead of the ancestor class `TBoundary`. In that case an instance of `TApplicationBoundary` has to be created and returned.

Overloading No call to the ancestor method neccessary.

```
TSuperGraphPrimitive *CreateNet(int GraphNr)
```

Description This method must be overloaded to supply an instance of the template class `TSimulationGraph<>` which has been applied to `TApplicationNode` and `TApplicationEdge`. Pass `GraphNr` to the constructor.

Overloading No call to ancestor method neccessary.

```
bool DecodeInitInfo(TEvent &anEvent)
```

Description Overwrite this method to decode the init info encoded by `TApplicationMaster` in `EncodeInitInfo`.

Overloading The ancestor method has to be called at the beginning of the method.

B.7 Class `TSimulationMaster`

B.7.1 Public Control Methods

```
bool StartSimulationSequence(long NrOfTimeSteps,
                             long SequenceID = 0)
```

Description Use this method to start a simulation sequence of `NrOftimeSteps` time steps in method `SimulationControl`. The parameter `SequenceID` will be passed to `BeginSimulationSequence` of `TSimulationSlave`.

```
bool StartStatistics(int StatID, TTimeStep Start, TTimeStep End,
                     long Interval, long Flags,PvmID Destination)
```

Description `StartStatistics` activates the gathering of statistics for the ID `StatID` starting at time step `Start` and ending at (not including) time step `End`. The statistics will be polled every `Interval` time steps. `Flags` is a bitwise combination of the following constants:

`STAT_FLAG_MULTI_STAT` will declare the statistics as a multi statistics instead of single statistics (default).

`STAT_FLAG_NEIGHBOUR_STAT` will declare the statistics as local statistics instead of global statistics (default).

The parameter `Flags` can be omitted if the defaults are ok. The parameter `Destination` should always be omitted. The method returns `TRUE` in case of success, `FALSE` otherwise.

B.7.2 Methods Defining Simulation Control

```
bool EncodeInitInfo(TEvent &anEvent)
```

Description Overwrite this method to send an init info to a new slave. This info should contain all information the slave needs to start simulation immediately afterwards. It will be decoded in `DecodeInfoInfo` of `TSimulationSlave`. The built in method transfers data retrieved from the command line to the slaves.

Overloading The ancestor method has to be called at the beginning of the method.

```
bool ExecuteMasterTimeStep(TTimeStep TimeStep)
```

Description Overwrite this method to execute tasks which have to be done by the master. The built in method takes care of the graphics if activated.

Overloading Call to the ancestor method is neccessary.

```
bool HandleMasterEvent(TEvent &anEvent, bool &Handled)
```

Description Use this method to be informed about incoming events. The parameter `Handled` has to be set to `TRUE` if the event could successfully be handled.

Overloading If the event type is unknown a call to the ancestor method is neccessary.

```
bool SimulationControl(TControlStep &ControlStep, bool &Quit)
```

Description This is the key method of `TSimulationMaster`. It is used to define the course of the simulation. The first time this method is called `ControlStep` will be set to zero. Before each subsequent call `ControlStep` will be incremented enabling the user to destinguish between the different calls in a `switch`–statement.

Upon call of this method it can be assumed that all slaves have finished their latest simulation sequence and are globally synchronized. At the end of `SimulationControl` the method `Synchronize` should be called unless the method `StartSimulationSequence` is used which implies an automatic synchronization.

It is explicitly allwowed to change the value of `ControlStep`. This can used to call the same step several times.

By setting `Quit` to `TRUE` the simulation is terminated. This method will not be called again.

Overloading **!!!!! Do not call the ancestor method !!!!!**

B.8 Abstract data structures

In the implementation, five abstract data structures will be used to handle sets of objects 'in an orderly fashion'. They will either be taken from a library (if available) or be written especially fitted for this simulation. All should be provided as templates to allow for maximum programming comfort and to enhance readability of the source code.

Non intrusive singly linked list

This is just a regular singly linked list that handles type cast (through templates) pointers of objects.

Non intrusive doubly linked list

Equivalent to the singly linked list.

Binary tree

The binary tree will be used to organize message passing between the CPNs[1] especially for *broadcasting* and gathering statistical data by *reducing*.

Heap

The heap will be used to find the maximum or minimum of a characteristic of a dynamically changing set of objects. An example is the scheduler which sorts the received events by the time stamp of each scheduled event. The event with the minimum in time will be executed first.

AB-tree

An AB-tree is defined as a tree in which all non leaves have at least A sons and at most B sons. It can be shown that AB-trees have always a depth complexity of $O(\log n)$ and therefore all dynamic insertions and deletions can be done in $O(\log n)$ time complexity. Usually an infix order is applied to the AB-tree to allow for quick managing of sorted sets.

In this simulation the ID-tree which links IDs with pointers will be organized in an AB-tree.

B.9 Trees

B.9.1 Object inheritence tree

See figure B.3. Arrows point from parent to child.

B.9.2 Object dependency tree

See figure B.4. Arrows represent relationship *uses a*. Labels like $1:n$ denote the number of objects used by another object. n may vary in each case.

[1]computational nodes

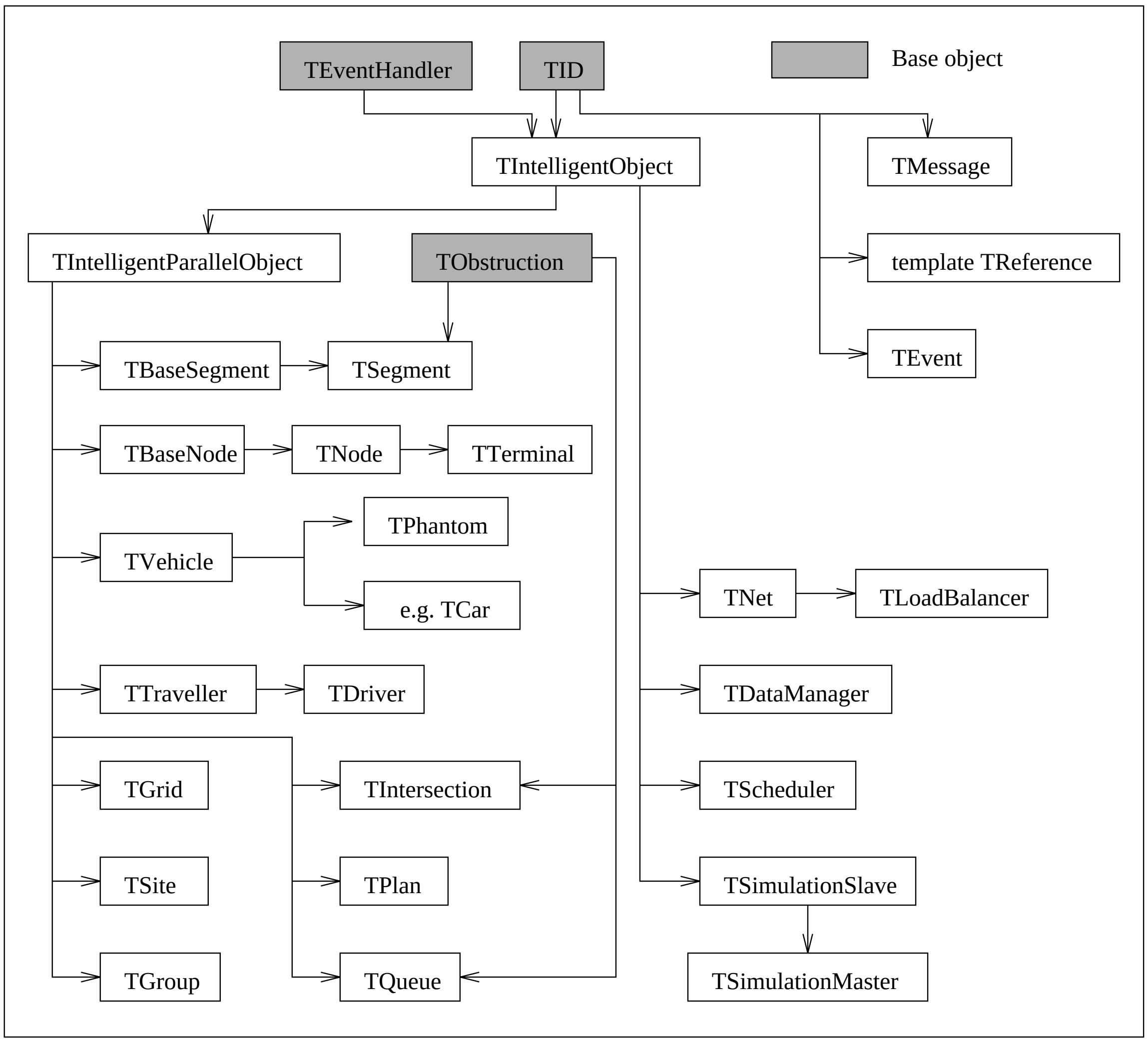

Figure B.3: *Object inheritence tree*

B.10 Location of Classes

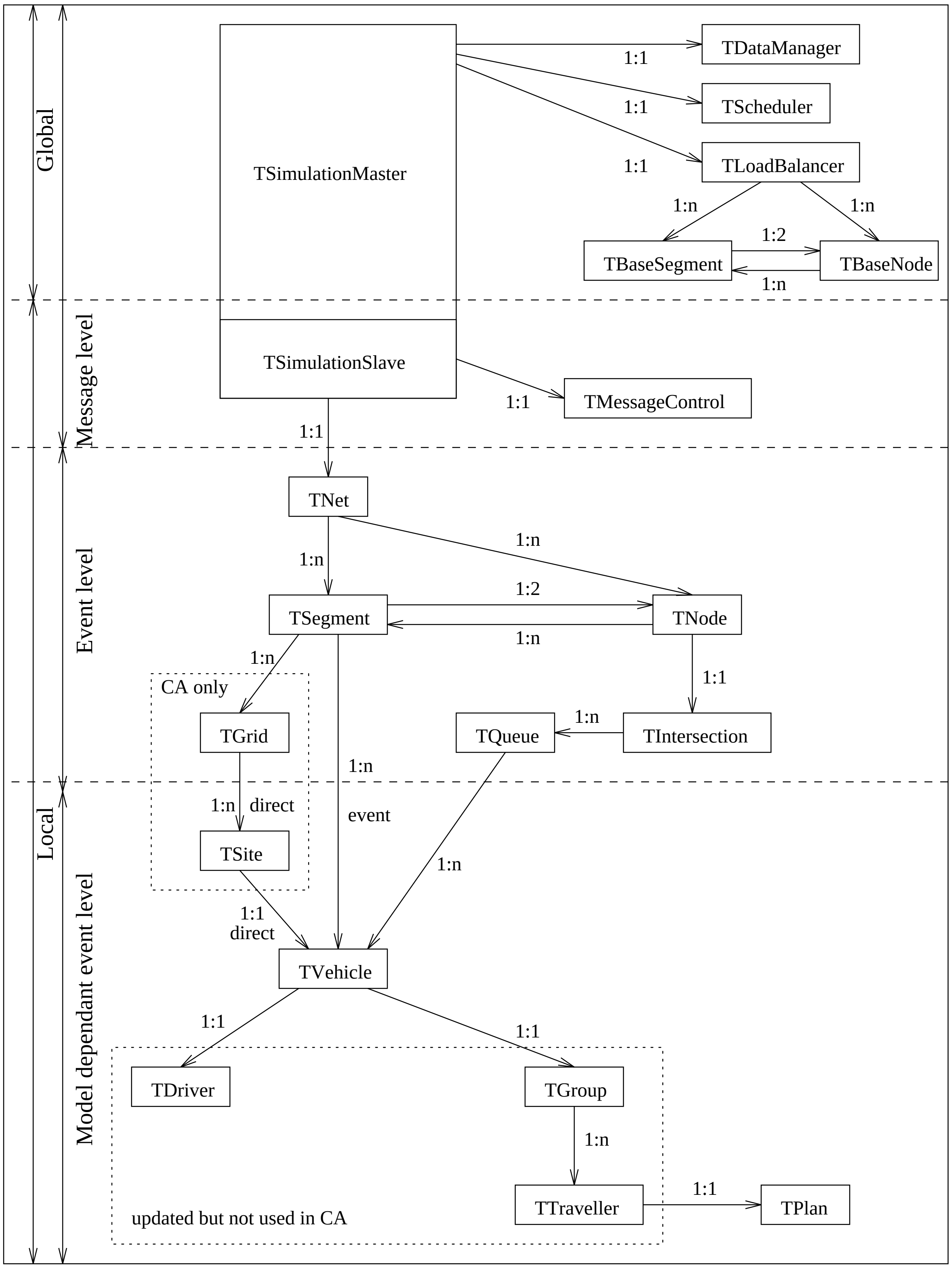

Figure B.4: *Object dependency tree*

Class	Location	Who
`EdgeIO`	TRANSIMS/EdgeIO.H	RO
`GrWindow`	michael/Graphx.h	MO
`NodeIO`	TRANSIMS/NodeIO.H	RO
`TABTree<>`	include/TABTree.h	MR
`TABTreeElement`	include/TABTree.h	MR
`TABTreePrimitive`	include/TABTree.h	MR
`TAdjacence`	obsolete (06–16–95)	MR
`TAlarm`	include/TSignal.h	MR
`TApplication<>`	include/TApplication.h	MR
`TArray<>`	include/TArray.h	MR
`TArrayPrimitive`	include/TArray.h	MR
`TBaseEdge`	include/TBaseEdge.h	MR
`TBaseNode`	include/TBaseNode.h	MR
`TBaseObject`	include/TBaseObject.h	MR
`TBoundary`	include/TBoundary.h	MR
`TColorInfo`	include/TColor.h	MR
`TControlInfo`	include/TSimulationSlave.h	MR
`TCPNCount`	include/TCPNInfo.h	MR
`TCPNInfo`	include/TCPNInfo.h	MR
`TDataManager`	TRANSIMS/TDataManager.h	MR
`TDebug`	include/TDebug.h	MR
`TDLList<>`	include/TDLList.h	MR
`TDLListEnumPrimitive`	include/TDLList.h	MR
`TDLListEnum<>`	include/TDLList.h	MR
`TDLListPrimitive`	include/TDLList.h	MR
`TDLListReference`	include/TDLList.h	MR
`TEnumerator<>`	include/TEnumerator.h	MR
`TEnumeratorPrimitive`	include/TEnumerator.h	MR
`TEvent`	include/TEvent.h	MR
`TForwardInfo`	include/TForward.h	MR
`TForwardManager`	include/TForward.h	MR
`TGraph<>`	include/TGraph.h	MR
`TGraphEdge`	include/TGraphEdge.h	MR
`TGraphPrimitive`	include/TGraph.h	MR
`TGraphicsManager`	include/TGraphicsManager.h	MR
`TGrid`	TRANSIMS/TGrid.h	MR
`TGridBoundary<>`	grid/TGridBoundary.h	MR
`TGridEdge<>`	grid/TGridEdge.h	MR
`TGridInitInfo<>`	grid/TGridSlave.h	MR
`TGridMaster<>`	grid/TGridMaster.h	MR
`TGridNode<>`	grid/TGridNode.h	MR
`TGridNodeInfo`	grid/TGridSite.h	MR
`TGridNodePrimitive`	grid/TGridNode.h	MR

Table B.2: *Location Of Toolbox Classes (part 1)*

Class	Location	Who
`TGridSlave<>`	grid/TGridSlave.h	MR
`TGridSlavePrimitive`	grid/TGridSlave.h	MR
`THardwareInfo`	include/THardware.h	MR
`THardwareInfoManager`	include/THardware.h	MR
`TID2Table<>`	include/TID2Table.h	MR
`TID2TableEntry<>`	include/TID2Table.h	MR
`TID2TablePrimitive`	include/TID2Table.h	MR
`TIdleTimeInfo`	include/TIdleTime.h	MR
`TIDTable<>`	include/TIDTable.h	MR
`TIDTableEntry<>`	include/TIDTable.h	MR
`TIDTablePrimitive`	include/TIDTable.h	MR
`TLifeGridSite`	life/TLifeGridSite.h	MR
`TLifeGridStat`	life/TLifeGridSite.h	MR
`TLifeMaster`	life/TLifeMaster.h	MR
`TLoadBalanceInfo`	include/TLoadBalance.h	MR
`TLoadBalanceNeighbourInfo`	include/TLoadBalance.h	MR
`TLoadTransferInfo`	include/TLoadBalance.h	MR
`TLocation`	include/TLocation.h	MR
`TMessage`	include/TMessage.h	MR
`TMessageControl`	include/TMessageControl.h	MR
`TMultilaneBoundary`	CA/TMultilaneBoundary.h	MR
`TMultilaneEdge`	CA/TMultilaneEdge.h	MR
`TMultilaneGrid`	CA/TMultilaneGrid.h	MR
`TMultilaneStat`	CA/TMultilaneStat.h	MR
`TMultilaneViewDrawContext`	CA/TMultilaneViewDrawContext.h	MR
`TMultiStatObject`	include/TStat.h	MR
`TMultiStatObjectPrimitive`	include/TStat.h	MR
`TObject`	include/TObject.h	MR
`TObjectArray<>`	include/TObjectArray.h	MR
`TObjectID`	include/TObjectID.h	MR
`TPositionHandler`	MicroSim/TPositionHandler.h	MR
`TPartitionInfo`	include/TGraph.h	MR
`TRandom`	include/TRandom.h	MR
`TRectangle`	include/TLocation.h	MR
`TReference<>`	include/TReference.h	MR
`TReferencePrimitive`	include/TReference.h	MR
`TSignal`	include/TSignal.h	MR
`TSimulationGraph<>`	include/TSimulationGraph.h	MR
`TSimulationInfo`	include/TSimulationSlave.h	MR
`TSimulationMaster`	include/TSimulationMaster.h	MR
`TSimulationSlave`	include/TSimulationSlave.h	MR
`TSimpleVehicleStat`	TRANSIMS/TSimpleVehicleStat.h	MR
`TSingleTransferInfo`	include/TLoadBalance.h	MR

Table B.3: *Location Of Toolbox Classes (part 2)*

Class	Location	Who
`TStatDelayInfo`	include/TSimulationSlave.h	MR
`TStatInfo`	include/TSimulationSlave.h	MR
`TStatistics`	include/TStatistics.h	MR
`TStatObject`	include/TStat.h	MR
`TStatResultInfo`	include/TSimulationSlave.h	MR
`TString`	include/TString.h	MR
`TSuperGraph<>`	include/TSuperGraph.h	MR
`TSuperGraphPrimitive`	include/TSuperGraph.h	MR
`TTalkInfo`	include/TTalk.h	MR
`TTimer`	include/TTimer.h	MR
`TTransferObject`	include/TTransferObject.h	MR
`TTransferRandom`	include/TTransferRandom.h	MR
`TTransimsBoundary`	TRANSIMS/TTransimsBoundary.h	MR
`TTransimsEdge`	TRANSIMS/TTransimsEdge.h	MR
`TTransimsNode`	TRANSIMS/TTransimsNode.h	MR
`TVehicle`	MicroSim/TVehicle.h	MR
`TVehicleDrawEntry`	CA/TMultilaneViewDrawContext.h	MR
`TView`	include/TView.h	MR
`TViewDataHandler`	include/TViewDataHandler.h	MR
`TViewDrawContext`	include/TViewDrawContext.h	MR
`TViewDrawObject`	include/TViewDrawObject.h	MR
`TViewManager`	include/TViewManager.h	MR
`TViewManagerSlave`	include/TViewManagerSlave.h	MR
`TWindow`	include/TWindow.h	MR

Table B.4: *Location Of Toolbox Classes (part 3)*

List of Figures

List of Tables