TWlib – a library for distributed search applications*

Jörg Denzinger and Jürgen Lind

Department of Computer Science
University of Kaiserslautern
Postfach 3049
67653 Kaiserslautern
E-mail: {denzinge, lind}@informatik.uni-kl.de

Abstract

We present TWlib, a library to support the knowledge-based distribution of certain search processes. The distribution concept, called teamwork, is intended for such search processes that use a set of facts as representation of the state and addition and removal of facts as transition between states. Teamwork has proven to be quite successful in different applications by generating synergetic effects.

Teamwork employs four types of components. Experts and specialists work independently on solving the search problem. Cooperation is achieved by periodically judging the new generated facts by referees and generating a new start state out of the best facts. A supervisor controls the whole system and can adapt it to the given problem by exchanging bad experts and specialists. TWlib provides communication functions (including a secure broadcast) and control skeletons for the teamwork specific flow of control.

1 Introduction

Knowledge-based search is the basic method employed in many AI systems. Planning systems, learning systems or deduction systems, all have to deal with very large search spaces and therefore try to determine the right (or at least a good) path through these spaces by using additional knowledge about the problems to solve and their domains. But even the use of this additional knowledge is sometimes not enough to meet the efficiency requirements of these systems.

One obvious way to speed up search is to do it in parallel. Unfortunately, knowledge-based search requires the ability to react quickly to new possibilities that are the result of the last search step done. Therefore parallelization methods that only try to achieve a higher number of search steps per time unit do very often not combine well with the quick evaluation of and reaction to evolving possibilities. As a result, the parallel system may even be less efficient than the knowledge-based sequential one.

Approaches to combine knowledge-based systems with the possibilities of distributed computation are one of the main foci of distributed AI. But unfortunately, the efficiency of the developed systems is not the main concern of many researchers. Since the development of an efficient system employing DAI techniques to solve problems in an application domain requires experiences in operating systems, DAI and the problem solving methods needed for the application, often compromises are made. Such a compromise very often is to neglect the operating system part by using either easy (and not very efficient) solutions to problems of this level or by employing development (see, for example, ARCHON [JW92]) and testbed (see, for example, DRESUN [CLL93] or MACE [GBH88]) systems for DAI applications. Although these systems offer much comfort for implementing various multi-agent systems, they have to support often quite different concepts. This results in not very efficient realizations of these concepts, again.

If we take a look at the developments in the field operating systems/ distributed systems, then we can observe that researchers of this field have encountered this efficiency problem also. Their solution is to propose operating systems with only a small fixed kernel (much smaller than UNIX) and various (specialized and efficient) extensions that serve different needs of the users of the system (see [NS95]). A user then chooses those extensions that optimally suit his requirements (or better those versions of the extensions that fit together). No compromises are made.

For DAI this concept means not building a testbed for all or many multi-agent concepts a user might be interested in, but building small libraries that opti-

*This work was supported by the Schwerpunktprogramm Deduktion of the DFG

mally support one (or perhaps a few very related) concept(s). In this paper we present such a library, called *TWlib*, for the *teamwork method* (see [De95]), a distribution method for a certain knowledge-based search process we call *search by extension and focus.*

Search by extension and focus can be briefly characterized as follows. The search state is represented by a set of so-called facts. The transition between search states is accomplished by using extension rules that may add new facts or change old ones. For large instances of search problems it is not feasible to try out all possible extensions. Therefore the search is focused on parts of the extensions by the use of guiding strategies and heuristics. Hence the name search by extension and focus. Examples for search problems that can be represented and solved by search by extension and focus are such different problems as automated theorem proving and optimization problems.

Typically, there are many different heuristics to provide a focus to the search and it is no trivial task to select a good one to a given instance of a search problem. Our teamwork method takes care of this problem by employing several guiding heuristics independently in parallel on different computers. In order to achieve a cooperation between these heuristics, the search states on each computer are periodically evaluated by so-called referees. These referees also select outstanding facts of the computers. A supervisor generates then a new start search state, using the reports of the referees, which is transmitted to all computers to start a new round.

Teamwork has proven to be quite successful for search by extension and focus in different applications (see [DF94], [De95]). The cooperation of the different heuristics results in *synergetic effects.* For example, our teamwork based equational theorem prover DISCOUNT ([ADF95]) was able to prove several theorems no heuristic could prove when working alone. But implementing a teamwork based search system requires a complicated communication and control structure in order to be efficient. Our TWlib, written in C++, provides the necessary procedures and procedure skeletons to allow a potential user of teamwork to concentrate on the specific questions of the search problem, for example what heuristics to use, how to build referees and a supervisor. So there is no need to deal with implementing communication and control primitives.

This report is organized as follows: After this introduction we characterize search by extension and focus. Then we present the teamwork method. In section 4 we a give a kind of guided tour of the TWlib that includes presenting the class hierarchy and dependencies, the tasks that remain to do for an user of the library, and the code of the top-level methods (or skeletons). The later should give an implementer a good insight what to expect when using TWlib. In section 5 we compare our experiences of first implementations of teamwork (without TWlib) with the experiences when using TWlib. Finally we will discuss some future work on TWlib and teamwork.

2 Search by extension and focus

There are many very different search processes in literature. Nevertheless, they have in common the use of *states* to describe their progress and the use of *transition* rules to reach states from other states. A first, but very useful classification of search processes is based on the representation of states. There are two groups of search processes. Members of the first one need explicit information about the history of the process while members of the second one do not need to represent this information explicitly.

Search processes of the first group are often based on such principles as dividing a problem into subproblems and therefore use trees or directed graphs as representations of the search state. Search processes of the second group use sets of results as representations of the state. Note that there may be processes of both groups that can be used to solve a given search problem. Our TWlib is aimed at distributing search processes of the second group. Therefore we will give in the following a formal definition of these processes that we call search by extension and focus.

Definition 2.1 (Search by extension and focus)

Search by extension and focus *is described by a 4-tuple* $(\mathcal{B},\Omega,\mathcal{I},S_0)$. *The set of possible facts* $\mathcal{B}$ *defines the possible states* S *of the search by* $S \in 2^{\mathcal{B}}$. Ω *is a predicate defined on* $\mathcal{B}$ *and used to describe a legal state* S *of the search by* $\Omega(s)$ *is true for all* $s \in S$. $\mathcal{I}$ *is a set of extension rules* $A \to B$, $A,B \in 2^{\mathcal{B}}$. S_0 *is called the start state of the search and has to be a legal state. We write* $S \vdash_{\mathcal{I}} S'$ *for states* S *and* S', *if there is a rule* $A \to B \in \mathcal{I}$, *such that* $A \subseteq S$ *and* $S' = (S / A) \cup B$. *A sequence* $(S_0,S_1,\ldots,S_n)$ *with* $S_{i-1} \vdash_{\mathcal{I}} S_i$ *for* $i=1,\ldots,n$, *is called a search derivation.*

Typically, doing search means applying a search process, that is developed for a search problem, to an instance of the search problem. In our formal definition the search process is represented by $\mathcal{B}$ and $\mathcal{I}$, while the actual instance determines S_0 and Ω and also provides the goal of the search.

Definition 2.2 (Goal of a search)
Let $(\mathcal{B},\Omega,\mathcal{I},S_0)$ *be a search by extension and focus and*

$g \in \mathcal{B}$ with $\Omega(g)$ = true the goal *of the search. A state containing g is called a* goal state. *The goal is* reachable, *if there is a search derivation (S_0,S_1,...,S_n) such that S_n is a goal state.*

So, the main problem of a search is to find a (short) search derivation to a state that includes the goal. Since there may be many possible extensions to a given state (and also many that lead to legal states), there has to be a function that determines which extension should be chosen next, i.e. a function providing a focus. Typically, such a focus function associates with each pair (state, extension rule) a weight that rates the extension step.

Definition 2.3 (Focus function)
Let ($\mathcal{B}$,Ω,$\mathcal{I}$,S_0) be a search by extension and focus and g its goal. An injective function f:$2^{\mathcal{B}} \times \mathcal{I} \rightarrow Z$ is called a focus function *and the derivation (S_0,S_1,...,S_i,...) is produced by f, if for the extension $A_i \rightarrow B_i$ that produced state S_i we have $f(S_{i-1},A_i \rightarrow B_i) \leq f(S_{i-1},A \rightarrow B)$ for all $A \rightarrow B \in \mathcal{I}$.*

If one wants to allow only legal states in a search sequence (which is usually the case), then the focus function only has to rate pairs (S_{i-1},$A \rightarrow B$) that produce a legal state. In practise, very often the condition "injective" function is dropped and the decision between extension steps with equal f-value is made employing the FIFO-strategy. This way there are many focus functions that can be used, very often too many to allow an automated decision which one to choose for a given problem instance. Another problem is that very often none of the implemented focus functions are good enough to solve a given problem instance in an acceptable time. These problems are solved by our teamwork method.

3 The teamwork method

The teamwork method is our approach to distribute search by extension and focus. A system based on teamwork has four types of components: experts, specialists, referees and a supervisor. The interaction between these components is organized as a cycle consisting of three phases. In the **competition phase**, also called working phase, experts and specialists work independently on their tasks. In the **judgement phase**, the first part of a *team meeting*, referees judge the work of the experts and specialists and select outstanding results and in the **cooperation phase**, the second part of a team meeting, the supervisor generates a new start state for the further search.

Experts work on solving the given problem instance by using search by extension and focus. Each expert uses a different focus function, thus generating different search sequences.

Definition 3.1 (Expert)
An expert *X is characterized by a focus function f_X: $2^{\mathcal{B}} \times \mathcal{I} \rightarrow Z$. An expert starts cycle i with start state S_i. During cycle i an expert may be active, i.e. running on a processor, or not. By $\mathcal{E}xp$ we denote the set of all experts.*

Specialists can also work on solving the problem instance, without being limited to using search by extension and focus, or they can generate data that helps controlling the search or they can combine these two tasks.

Definition 3.2 (Specialist)
A specialist *Sp is a function f_{Sp}: $2^{\mathcal{B}} \rightarrow 2^{\mathcal{B}} \times$ message-set. It starts cycle i with the set S_i of facts and returns a set Res_{Sp} of facts with $\Omega(s)$ = true for all $s \in Res_{Sp}$ and it can also return a message out of a set of messages for the supervisor. During a cycle a specialist may be active or not. By $\mathcal{S}pec$ we denote the set of all specialists.*

Note that a specialist can produce its set Res_{Sp} by any (correct) means, i.e. it need not use search by extension and focus. The set of possible messages *message-set* has to be defined by the user. Each message of *message-set* has to be interpreted by the supervisor.

Referees have two tasks: computing a measure of success for experts and some specialists and selecting outstanding results of experts and specialists. The results of both tasks are passed on to the supervisor. A referee of a specialist also reports the message of the specialist (if there is any).

Definition 3.3 (Referee)
A referee *R consists of two functions*

$$meas_R: (2^{\mathcal{B}})^* \rightarrow Z \text{ and}$$
$$selres_R: 2^{\mathcal{B}} \rightarrow \mathcal{B}^{\leq k},$$

where $(2^{\mathcal{B}})^*$ is a sequence of states and k the maximal number of results that may be selected by the referee. By $\mathcal{R}ef$ we denote the set of all referees.

If a referee judges the success of an expert then it uses the whole search sequence produced by this expert as basis for its judgement. Since specialists can use totally different representations for their search states, the referee can only use the set Res_{Sp} as basis. This is also the set $selres_R$ chooses from, while a referee of an expert uses its last search state as input for $selres_R$.

The supervisor achieves the cooperation of experts and specialists by generating a new start state for the next cycle out of their results. But it also compares the success of the experts and specialists and selects the members of the team of the next cycle using the measures of success of the experts and specialists in prior cycles and further information provided by a long-term memory (see [DK94]). Also the messages of the specialists are used. As third task the supervisor determines the length of the next cycle, also based on the measures of success, the long-term memory and the messages of the specialists. Therefore the formal definition is a little bit complicated.

Definition 3.4 (Supervisor)
The supervisor *computes after a cycle i the following three functions:*

$$Comp{:}(2^{\mathcal{B}} \times \mathcal{B}^k \times \mathbb{Z})^n \to 2^{\mathcal{B}}$$
$$Sel_i{:}(\mathcal{E}xp \cup \mathcal{S}pec) \times \mathcal{R}ef \to \{0,1\}$$
$$Time_i{:}\ 2^{\mathcal{B}} \times ((\mathcal{E}xp \cup \mathcal{S}pec) \times \mathcal{R}ef)^n \to N$$

where n is the number of available processors.

While the function $Comp$ that computes the new start state remains the same throughout a whole distributed search run –it simply uses the state produced by the expert with the best measure of success and adds the selected results (function $selres_R$) of the other experts and the specialists to generate the start state– the functions Sel and $Time$ change from cycle to cycle, indicated by index i, because they have to take the results of the prior cycles and the messages of the specialists into account. Since there are only n processors available, we demand $\sum_{x \in \mathcal{E}xp \cup \mathcal{S}pec} Sel_i(x, \_) = n$, so that only n experts and specialists (with their referees) are active in each cycle.

Using the definitions from above it is quite easy to write procedures for experts, specialists, referees and the supervisor, provided one has a good understanding about the search problem one wants to solve. Typically, different experts and specialists can be taken from sequential programs for solving the problem. If one uses a fixed team for all cycles and, for example, a linear growth of the cycle time, then also implementing the supervisor (or a first version of it) is no problem. Only the referees represent for many search problems a new point of view, since an assessment of the results of transitions that have been made in the past is not usual. But we found that simple statistical criteria produce already good results for both $meas_r$ and $selres_R$ (see [DF96]).

Implementing the control cycle and the communication between the components is the part that requires more than knowledge about the search problem one wants to solve. It is possible to avoid some interprocessor communication by allowing components of different type to share a processor. Since an expert or specialist, a referee or the supervisor never are active during the same time, one can implement them as one process having different modi (or an agent having different roles).

So, at the beginning of a distributed run, the process on one processor is in supervisor mode, receiving the problem instance. After transforming the instance into a search state, the process sends this state to all other processors whose processes are in expert or specialist mode. Until the next team meeting the work of the supervisor is done and this process changes to expert mode.

When the end of a working phase is reached, each processor changes into referee mode. This way the referees can access all data of their experts/specialists without expensive communication. After the judgement phase the process that was in supervisor mode at the end of the last meeting changes back into supervisor mode and receives the measures of success of all experts and specialists (function $meas_r$). After the best expert is determined, its process changes to supervisor mode while the process that was supervisor so far changes back to referee mode. For this change only little information has to be exchanged between the two processes.

Now, the new supervisor receives the full reports of the referees. The selected results of the other experts and the specialists can be integrated directly into the actual search state of the process in supervisor mode. After determining the members of the team in the next cycle the new start state is transmitted to the other processes and a new cycle begins.

By carefully choosing point-to-point connections between processors or broadcasting to all processors (the later when transmitting the new start state), one is able to achieve communication and control without much overhead. Then the following synergetic effects occur, that allow a team to be much more effective than single experts that employ only one focus function.

- An expert that is able to select many necessary extensions on the way to a goal state but not all the required extensions can receive the results of the missing extensions from the other experts. This effect is based on the cooperation aspect of teamwork.
- An expert that is able to generate a very good start search sequence but is not able to continue it until a goal state is reached transmits the end

of this sequence to the other experts. One of these experts may not be able to produce the sequence it got from the other one, but it can continue it, thus producing larger good sequences. This change of focus functions produces search sequences that are much better for a problem instance than all sequences produced by one expert alone and it is based on the competition aspect of teamwork. One can see these sequences as the product of a "Hyper-strategy".

While a potential user of the teamwork method still has to write procedures for experts, specialists, referees and the supervisor that allow for synergetic effects, our TWlib provides the necessary communication and control primitives and also skeletons (with respect to these primitives) for the components. Please note that the communication structure of teamwork has requirements on the system software level that are typically neither directly provided by operating systems nor by development packages for distributed systems as for example PVM (see [GB+93]). Such a requirement is, for example, a secure broadcast.

4 The TWlib

In order to allow potential users of teamwork, i.e. implementers of systems employing search by extension and focus, to get rid of the system software part of an implementation of a teamwork based system we developed TWlib. It offers a set of classes with associated methods, some of which are only skeletons that have to be extended according to the search problem that has to be solved. In order to provide this object-oriented view without much loss of efficiency and also to use a widely known programming language we have chosen C++ as implementation language. As network architecture and operating system for our first version of TWlib we have chosen a net of workstations running UNIX (Sun OS 4.1.4) using the TCP/IP protocol suite for interprocess communication.

In the following we will begin our presentation of TWlib with some remarks about the class hierarchy and dependencies. Then we will characterize the tasks an implementer using TWlib has to do which also characterizes the tasks TWlib takes care of. Then we take a deeper look at the main class of TWlib, class teamworkApplication, which form the application programmers interface (API) of TWlib.

4.1 The class hierarchy and dependencies of TWlib

The main purpose of TWlib is to support the necessary communication between the objects involved in a teamwork application and their internal scheduling tasks. These objects can be (and some usually are) located on different computers and therefore, object communication includes network communication as well. In addition, the teamwork specific data and control flow is provided by skeletons of methods which have to be expanded by the user. Additionally, error handling and logging of events are also supported. These tasks together lead to the class hierarchy and dependencies that are depicted in Figure 1.

An implementer employing TWlib will only have to deal with classes Xpert, Specialist, Referee and teamworkApplication. These classes form the *application layer* of TWlib.

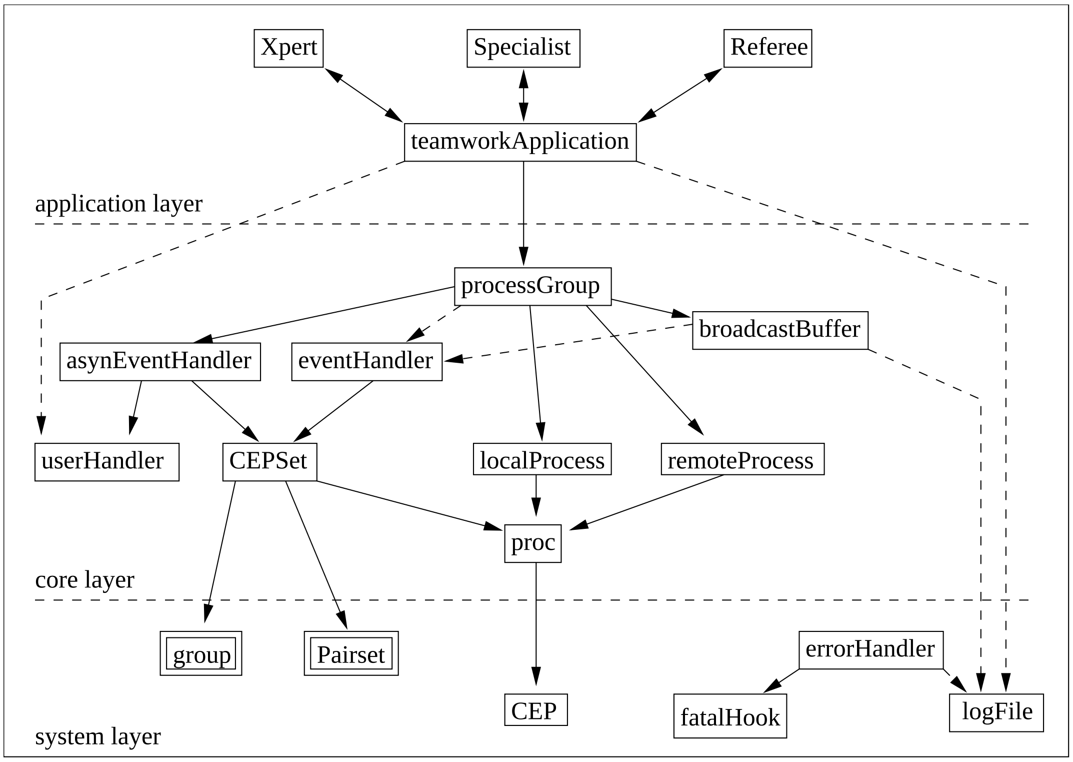
The other classes form the *core layer* and the *system layer* which encapsulates the underlying operating system. Class processGroup is an administrative class holding information about the environment (e.g. the other processes) of a process. Another administrative class is the class CEPSet which controls all the connections between a process and the other processes. These connections are used to transmit asynchronous events, point-to-point messages like referee reports and broadcast messages like the new start state of the search. Please note that we implemented a secure broadcast (no simulation) that uses point-to-point messages only for acknowledgments and missing information. All this functionality is provided by the classes CEP (communication end point) and broadcastBuffer.

Classes group and Pairset provide data structures that are used for process control and finally class errorHandler contains TWlib's error I/O and a hook for a user function to be called on a fatal error.

4.2 Tasks of an implementer using TWlib

Since TWlib is intended as basis for many very different applications, all functions and program segments that deal directly with the application or its data structures have to be provided by the implementer. More precisely, an implementer has the following tasks:

- He must provide the code for experts, specialists and referees and the data structures they use.
- He must provide initial data about the computers to use and must also announce all experts, specialists and referees that can be used to the system in a predefined format. Typically this information is read from a file as is the problem instance to solve.
- He must implement send and receive functions for referee reports and search states, because these data formats are typically application specific. The library offers transfer routines for character



streams (allowing secure broadcast), so that converting application data formats into character streams (and backwards) is sufficient.

- He must provide the function that chooses the new supervisor during a team meeting.
- He must provide the supervisor functions Comp, $\mathrm{Sel_i}$ and $\mathrm{Time_i}$. The resulting schedule for the next working phase must be given to the system in a predefined format.

The TWlib takes care of all other tasks. The most important of these tasks are:

- providing administrative data structures for experts, specialists and referees
- process management
- efficient communication via
  - secure broadcast
  - simulated broadcast (for short messages only)
  - point-to-point connections

In addition, the classes of the application level include skeletons that simply can be completed for:

- initializing the whole system
- experts, specialists and referees
- the main loop of a process including the course of events of a team meeting
- error handling and normal and abnormal program termination

4.3 Class teamworkApplication

The classes of the application level of TWlib map the theoretical concepts of section 3 to an actual implementation. The flow of control in a teamwork-based system consists of one loop that is preceded by an *Initialization*. The loop contains two parts: *Work* and *Meeting*. The loop can only be left by calling an exception (an asynchronous event). If no error occurs the exception which is called is the so-called *End* sequence.

The whole flow of control is implemented in class teamworkApplication. An implementer using TWlib should derive his own classes (e. g. UserXpert, UserSpecialist, UserReferee and UserteamworkApplication) from the respective base classes of TWlib and override the methods which deal with the specific application area. In the following we will describe the class teamworkApplication by tracing the execution of the method `main` of this class. An excerpt of this method is depicted in Figure 2. When the teamwork root process is called, an object of (User)teamworkApplication is instantly generated and control is passed to the new object's `main` method.

```
void teamworkApplication::main( int argc, char** argv )
{
  //------------------------------------------------
  // role independent initialization
  //------------------------------------------------
                     :
                     :
  if( !thisProcess->isInitial( argc, argv )){
      // we are not the teamwork root process...
      role = XP;
      initXpert( argc, argv );
      initUserCode();
      receiveInitialSystem();
      while( true ){
          work();
          meeting();
      };
  }
  else{
      // we are the teamwork root process...
      role = SV;
      initSupervisor( argc, argv, conf_files );
      initUserCode();
      sendInitialSystem();
      while( true ){
          work();
          meeting();
      }
  }
}
```

Figure 2: Extract of method `main`

Note that this method is used by all processes on the different computers. This means, that each process has exactly one object of class teamworkApplication and, since teamworkApplication also includes the supervisor part of the system, that in several places of `main` and subsequent methods there is a distinction whether the object is the actual supervisor or an expert/specialist.

4.3.1 The Initialization

The initialization usually begins by reading information about the configuration of the system from so-called configuration files. This is normally the first action of method `initSupervisor` of the supervisor process (see Figure 2). Please note, that the startup sequence for the remote processes has some user hooks to customize the start parameters for every single host. For more information we refer to the documentation that is included in the distribution package of TWlib. Then, the expert processes are started on the remote machines and the communication network between all processes of a teamwork application is set up.

After the system initialization, method `initUserCode`, which should be overridden by the implementer, allows to transfer information about experts, specialists and referees into the internal database of the teamworkApplication object of each process. The source of this information can be the configuration files read in during the system initialization. Since a teamworkApplication object includes the administration of all experts, specialists and referees (i.e. the sets $\mathcal{E}xp$, $\mathcal{S}pec$ and $\mathcal{R}ef$ of section 3), all those components that should actually be used have to be announced to the object. The class teamworkApplication provides methods `announceX` (where X stands for Xpert, Specialist or Referee) for this task. The result of these methods is always a unique id that in the following is used by the teamworkApplication object to refer to this component. These announcements should be made to each teamworkApplication object in the same order to guarantee that each component has the same id in all processes.

In order to compose a starting team (for a working phase) an initial schedule (i. e. an Xpert/Specialist and a referee assignment to each process) must be composed. An individual assignment is made by calling method `assignX` with the id of the component and the id of the process it should be assigned. Before a working phase only one referee and either an expert or a specialist may be assigned to a process. The methods associated with the respective ids will be called when the `work` or `assess` methods of the teamworkApplication objects are invoked. Since typically the next team is formed by the supervisor (function $\mathrm{Sel_i}$) TWlib provides methods, called `sendAssignment` and `receiveAssignment`, to inform the other processes about their next actual components.

Finally, the description of the actual search problem to solve has to be sent to all remote processes (methods `sendInitialSystem`, resp. `receiveInitialSystem`). This problem is also read from a configuration file. In order to demonstrate the methods for communication provided by the classes of the system level, we included the code of these two methods from an example user application class sampleApp in Figure 4. The object pg is of class processGroup.

```
void sampleApp::sendInitialSystem( void )
{
  //----------------------------------------------------------
  // send the goal specification to all team members (virtual
  // broadcast is better for short messages then physical
  // broadcast)
  //----------------------------------------------------------
  pg->write( ALL, (void*)&goal, sizeof( unsigned long ) );
  pg->write( ALL, (void*)&eps_env, sizeof( unsigned long ) );
  //----------------------------------------------------------
  // send a full problem description to all team members:
  // 1. send the size of the description via virtual broadcast
  // 2. send the description via physical broadcast
  //----------------------------------------------------------
  pg->write( ALL, (void*)&maxIndex, sizeof( int ));
  for( int i = 0; i < maxIndex; i++ ){
      if( pg->write( BROADCAST, (void*)&(data[i]),
                     sizeof( unsigned long )) < 0 ){
          ERR.fatalLog( __FILE__, "write error, exiting..." );
```

```
      }
  }
  //------------------------------------------------------------
  // flush the broadcast buffer explicitly
  //------------------------------------------------------------
  pg->flush();
}

void sampleApp::receiveInitialSystem( void )
{
  //------------------------------------------------------------
  // read the goal specification from the supervisor
  //------------------------------------------------------------
  pg->read( supervisor, (void*)&goal, sizeof( unsigned long ));
  pg->read( supervisor, (void*)&env, sizeof( unsigned long ));
  //------------------------------------------------------------
  // read a full problem description:
  // 1. read the size of the description via virtual broadcast
  // 2. read the description via physical broadcast
  //------------------------------------------------------------
  pg->read( supervisor, (void*)&size, sizeof( int ));
  for( int i = 0; i < size; i++ ){
      if( pg->read( BROADCAST, (void*)&(data[i]),
                    sizeof( unsigned long )) < 0 ){
          ERR.fatalLog( __FILE__, "read error, exiting..." );
      }
  }
}
```

Figure 4: Examples for methods `sendInitialSystem` and `receiveInitialSystem`

4.3.2 The main loop

In the loop of method `main` one can observe the two main phases of the work of a team: the working phase and the team meeting. As already stated, invoking method `work` calls the expert/specialist that is assigned to a process for the actual working phase. Only the actual supervisor has to invoke an additional method: `setWorkingParams`. This method installes a timer object which interrupts all processes after the time given by the supervisor (function $\mathrm{Time_i}$) (see Figure 5).

```
void teamworkApplication::work( void )
{
  if( role == sv){
      setWorkingParams();
  };
  // call the expert according to the schedule given by the user
  Xperts->getByFirst( xp )->work();
}
```

Figure 5: Method `work`

Then method `meeting` is invoked (see Figure 6). Method `assess` calls the referee that is assigned to a process. The following methods form a skeleton that has to be expanded by overwriting the respective methods according to the intended application. The methods `sendShortReport` and `receiveShortReport` transmit the results of the $\mathrm{meas_R}$-function of the referees. Note that both methods have no arguments. Instead the implementer has to put the results of the referee-functions into memory areas which are read by these methods. Figure 7 shows the code for these two methods (for sampleApp, again).

```
void teamworkApplication::meeting( void )
{
  assess();
  if( role == sv){
      receiveShortReports();
      newSupervisor = chooseNewSupervisor();
  }
  else{
      sendShortReport();
      newSupervisor = getNewSupervisor();
  }
  if( newSupervisor == thisId ){
      receiveFullReports();
      integrateResults();
      changeTeamSetup();
      sendNewSystem();
  }
  else{
      sendFullReport();
      changeTeamSetup();
      receiveNewSystem();
  }
}
```

Figure 6: Method `meeting`

```
void sampleApp::sendShortReport( void )
{
  //------------------------------------------------------------
  // send a short report to the supervisor
  //------------------------------------------------------------
  pg->write( supervisor, (void*)&rep->assess, sizeof( double ));
}

void sampleApp::receiveShortReports( void )
{
  //------------------------------------------------------------
  // read the reports created by the referees running on the
  // remote processes and store it in an application specific
  // 'report' structure
  //------------------------------------------------------------
  for( int i = 0; i < mem.groupSize(); i++ ){
      pg->read( &mem, &from, (void*)&assess, sizeof( double ));
      report = new Report;
      report->from = from;
      report->assessment = assessment;
      Reports->append( report );
  }
}
```

Figure 7: Examples for methods `sendShortReport` and `receiveShortReports`

The method `chooseNewSupervisor` has to select the best expert and its processor and has to transfer this information to all other processes (`getNewSupervisor`) Then the referees of the experts/specialists send their full reports, i.e. the results of their $\mathrm{selres_R}$-function, (`sendFullReport`) to the new supervisor (`receiveFullReports`) that integrates these results (`integrateResults` realizes the function Comp). The method `changeTeamSetup` is responsible for selecting the next team (function $\mathrm{Sel_i}$) and assigning to each processor its expert/specialist and referee. Then the supervisor transmits the new start state (`sendNewSystem`) to all other experts/specialists (`receiveNewSystem`).

4.3.3 The End Sequence

The last stage of a teamwork application is entered when an expert or specialist has found the goal of the search. This expert/specialist invokes the method `goalReached` that issues an asynchronous event `GOAL_REACHED` which causes all processes to invoke method `cleanup` of their teamworkApplication object. The method `cleanup` can use the predicate `reachedGoal` to test whether the local process has found the goal. An implementer can add to `cleanup` the necessary presentation functions for the user of his system. The last operation of `cleanup` is to terminate its process.

5 The TWlib in use

The main goal of the TWlib is to take away from an implementer the system software part while implementing a teamwork based system. Also, by providing skeleton methods and the main methods of some classes, the implementer gets some guidance how to structure his system and where to make his application dependent extensions. This should lead to a substantial gain with respect to the time needed for developing and implementing a teamwork based system.

Since there are many search problems that can be solved using search by extension and focus, we can not provide experiences for all possible applications here; but we can compare our experiments in implementing teamwork based systems before we had TWlib and after. Besides some small (and even silly) dummy applications we have developed and implemented three teamwork based systems: the DISCOUNT system ([ADF95]), the DOTT system for solving the traveling salesman problem ([De95]) and the DiCoDe system, a theorem prover based on Condensed Detachment (see [Ta56]) that is currently intensively tested.

For implementing the first version of DISCOUNT we needed 1.5 man years starting with an already implemented unfailing completion procedure (that even included several focus functions). This first version did not include broadcasting abilities. Adding these abilities took another 0.5 man years.

For implementing DOTT we needed 1.5 man years, again. But in this case we did not have the code for experts already available. Although we could reuse large parts of the code for broadcasting from DISCOUNT and also could employ many experiences made with DISCOUNT (although we had a new implementer), the stated 1.5 man years resulted in a system with only one type of referees and a very crude implementation of the supervisor (due to our limited experiences with the traveling salesman problem, really completing the DOTT system is currently the task of a new student who is at work since half a year).

The implementation of DiCoDe, based on an already implemented sequential system CoDe (see [Fu96]) and using the TWlib, took only 0.5 man years (again with a new implementer). In fact, in DiCoDe also two small variations of the teamwork method can be used that were obtained by simply altering the method `meeting` of the TWlib. Although one might argue that DiCoDe as a theorem prover can profit much more from the experiences with DISCOUNT than DOTT, nevertheless the 0.5 man years development time led to a system that already includes more referees and a better supervisor than the first version of DISCOUNT. Obviously, the TWlib significantly reduced the implementation time.

We are currently using the TWlib in a reimplementation of DISCOUNT and for teamwork based distributed systems employing genetic algorithms and genetic programming (and trying to achieve cooperation with more conventional search paradigms).

Although the TWlib is already successfully used in a system (and part of systems in development) we still see our current version as only a first beta version. Since we expect several changes after other implementers use TWlib, we want to be able to inform everyone using TWlib about necessary changes. Therefore we have chosen to distribute TWlib by request (e-mail to one of the authors) and not via an anonymous ftp-side (The latter will be employed when the actual version is not beta anymore).

The distribution package of TWlib contains the library, documentation in form of a postscript-file and as a directory of html-pages, and a dummy application. The problem solved by this dummy application is quite silly, but the application demonstrates all necessary steps an implementer using TWlib has to do.

6 Conclusion and Future Work

With TWlib we presented a library to support distributed search applications that are based on the teamwork method. TWlib provides communication functions and control skeletons in order to allow an implementer to concentrate on the specific problems of the intended application. Also some data structures for administrative purposes are included in TWlib.

A high priority on our future schedule has the porting of the TWlib on other UNIX-like operating systems in order to use workstation clusters with heterogeneous operating systems. Future conceptual work centers on extending the TWlib to also include a graphical user interface. Such an interface has proven to be very useful in developing more and better components for our DISCOUNT system. A generic version of it for all teamwork applications is possible and would be much appreciated.

Finally, we want to use our experiences with the TWlib to provide similar libraries for other distribution concepts for search processes. Again, the main goal has to be to provide both an optimal efficiency and an easy use by implementers.

References

[ADF95] **Avenhaus, J. ; Denzinger, J. ; Fuchs, M.**: *DISCOUNT: A system for distributed equational deduction*, Proc. 6th RTA, Kaiserslautern, 1995, pp. 397-402.

[CLL93] **Carver, N. ; Lesser, V.R. ; Long, Q.**: *Resolving global inconsistency in distributed sensor interpretation: Modelling agent interpretations in dresun*, Proc. 12th Intern. WS on DAI, Hidden Valley, 1993, pp. 19-33.

[De95] **Denzinger, J.**: *Knowledge-Based Distributed Search Using Teamwork*, Proc. ICMAS-95, San Francisco, AAAI-Press, 1995, pp. 81-88.

[DF94] **Denzinger, J. ; Fuchs, M.**: *Goal oriented equational theorem proving using teamwork*, Proc. 18th KI-94, Saarbrücken, LNAI 861, 1994, pp. 343-354; also available as SEKI-Report SR-94-04, University of Kaiserslautern, 1994.

[DF96] **Denzinger, J. ; Fuchs, D.**: *Referees for Teamwork*, Proc. FLAIRS-96, Key West, 1996, pp. 454-458.

[DK94] **Denzinger, J. ; Kronenburg, M.**: *Planning for distributed theorem proving: The team work approach*, to appear Proc. KI-96, Dresden, 1996; also available as SEKI-Report SR-94-09, University of Kaiserslautern, 1994.

[Fu96] **Fuchs, M.**: *Experiments in the Heuristic Use of Past Proof Experience*, to appear Proc. CADE-13, New Brunswick, 1996.

[GBH88] **Gasser, L. ; Braganza, C. ; Herman, N.**: *Implementing distributed ai systems using mace*, in Bond, Gasser (eds.): Readings in DAI, Morgan Kaufmann Publishers, 1988, pp. 445-450.

[GB+93] **Geist, A. ; Beguelin, A. ; Dongarra, J. ; Jiang, W. ; Manchek, R. ; Sunderam, V.**: *PVM 3 user's Guide and Reference Manual*, Oak Ridge National Laboratory, 1993.

[JW92] **Jennings, N.R. ; Wittig, T.**: *ARCHON: theory and practice*, in Avouris, Gasser (eds.): DAI: Theory and Praxis, Kluwer Academic Publishers, 1992, pp. 179-195.

[NS95] **Nehmer, J. ; Sturm, P.**: *Generating dedicated runtime platforms for distributed applications - a generic approach*, Proc. 5th IEEE Computer Society Workshop on Future Trends of Distributed Computing Systems (FTDCS), Cheju Island, Korea, 1995, pp. 50-55.

[Ta56] **Tarski, A.**: *Logic, Semantics, Metamathematics*, Oxford University Press, 1956.