Design and Implementation of a Handy Distributed KLIC System

K. Kakiuchi, M. Ohkawa, R. Kawabata, T. Hagino, K. Furukawa, T. Hattori

November 24, 1994

Abstract

We added distributed processing functions to KLIC and developed "Handy Distributed KLIC System" called HAYDN. It is useful for the research on distributed AI. HAYDN allows multiple KLIC programs to be executed rather independently on multiple UNIX WSs to solve problems cooperatively. We realized communication among processors using sockets. We adopted client-sever model as logical configuration among multi-processors. We designed protocols between a server and clients, and implemented this system on top of KLIC 1.410. We are planning to build several application programs running on HAYDN such as an airplane reservation system and an or-parallel MGTP.

1 Introduction

KL1, a concurrent logic programming language developed at the Institute for New Generation Computer Technology(ICOT), was originally executable on only few machines such as PS1, Multi-PDI and PIM[1]. KLIC,a new implementation of KL1, compiles KL1 programs into C programs so we are now able to use KL1 on most workstations and personal computers[2].

This paper describes design and implementation of a handy distributed KLIC system called HAYDN. It runs on UNIX worksations connected by network. HAYDN aims to provide a handy tool for writing distributed programs as well as concurrent programs on network-linked UNIX workstations.

To realize HAYDN, we adopted a client-server model for achieving communications among processors. We also extended KL1 by adding several commands for communications. The extended language is called Distributed KL1 (DKL1).

DKL1 makes it possible to realize such distributed systems that are hard to describe in KL1, e.g. on-line tickets-reservation systems and on-line banking systems. Thus, DKL1 is adequate for a larger problem domain than KL1. To show its ability, we will see a tiny airplane tickets reservation program in section 3.1.

It is also possible to modify KL1 programs into DKL1 programs by adopting message oriented programming style. We will try to convert an MGTP program written in KL1 into an equivalent program in DKL1 in section 3.2.

2 A Handy Distributed KLIC System HAYDN

2.1 Design philosophy of HAYDN

HAYDN aims to realize concurrent and distributed systems based on uni-processor KLIC systems. It allows multiple KLIC programs to be executed rather independently on multiple UNIX WSs to solve problems in cooperation. We designed HAYDN as simple as possible to minimize modification of KLIC and KL1.

As a logical architecture of HAYDN, we adopted a client-server model on communication among processors. This model makes communication protocols and corresponding language extentions very simple. The entire system consists of a server and several clients(Figure 1). Both of the server and the clients are written in KL1 and compiled by KLIC compiler. There is no limitation on the number of client. Clients never communicate with each other directly, but always through the server.

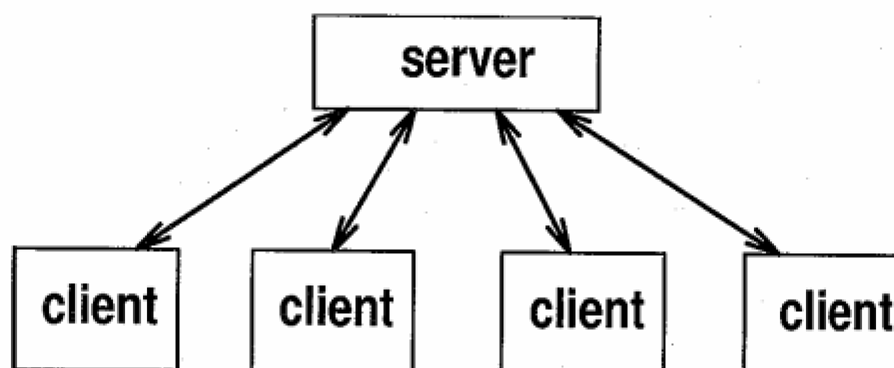

Figure 1: Outline of HAYDN — a client-server model

Communication between a server and a client is very simple. It is done using ground terms; i.e. variables cannot be used in a message to avoid problems about distributed unifications. The server should be already in execution before any client is invoked. Each client makes connection with the server in the first step. After connection is made, all the client can do is to send a ground term to the server, or receive a ground term from the server. The client sends a term with a *tag*, an arbitrary chosen symbol. The server add a pair of the tag and the term to a data queue. Clients must specify a *tag* in case of data retrieval. The server checks whether the tag exists in the data queue. If it exists, the corresponding term will be immediately returned. Otherwise, the server put the message into a wait queue.

HAYDN is realized by the *cs* module which contains only **connect/4**. The description of connect/4 is as follows:

cs:connect(Host,Port,Name,Stream)

An internet domain socket of the host specified by a string **Host** and a port number **Port** is opened. A client name should be given by an atom **Name**. Following functors are available to construct a term for a normal **Stream**, each of which instructs a certain communication with the server.

- **put(Tag,Term)**
  Send **Term** with an atom **Tag** to the server. We can use atoms, functors, numbers and strings as **Term**.

- **get(Tag,Term)**
  Send a request for an atom **Tag** to the server. If the server already has a term for that **Tag**, The result will be immediately put into **Term**. If not, wait for another client coming after to put the **Tag** into the server.

- **nget(Tag,Term)**
  Same as **get(Tag,Term)**, except for the behavior when the server has not received a term for the **Tag** yet. **nget(Tag,Term)** does not wait for another client, whereas **get(Tag,Term)** does.

- **dbcontrol(Message)**
  Send a control message **Message** to the server. There are two messages provided now.

  - **list**
    Show contents of the data queue in the server.
  - **clear**
    Initialize the data queue in the server, i.e. clear all data.

The client-server model makes it possible to designate a destination processor of communication. This ability can be used to realize distributed systems which are dominating in real life applications. See section 3.1 for detail.

Since we specify a logical symbol as a message destination, it is also possible to send a message without knowing physical address of the destination, provided both sides use the same logical symbol. See section 3.2 for detail.

2.2 The implementation of HAYDN

We wrote the entire HAYDN system in KL1 to make it portable. Communication among processors is implemented by UNIX sockets.

Let us illustrate how communication predicates shown in section 2.1 work in a program.

Predicate	Message from client to server	Return from server to client
cs:connect(Host,Port,Name,Stream)	connect [Name]\n	—
put(Tag,Term)	put [Tag] [Term]\n	—
get(Tag,Term)	get [Tag]\n	[Term]\n
nget(Tag,Term)	nget [Tag]\n	[Term]\n
dbcontrol(Message)	dbcontrol [message]\n	—
—	close\n	—

In sending process, **Term** is transformed in the following manner:

Functor_Name(Functor_Arg1 Functor_Arg2 Functor_Arg3 ...)

is transformed into

Functor_Name Functor_Arity Functor_Arg1 Functor_Arg2 Functor_Arg3

Of course, Functor_Arguments can be also a functor. For example,

"$functor_1(arg_1, functor_2(functor_3(arg_2), arg_3))$"

is transformed into

"$functor_1$ 3 arg_1 0 $functor_2$ 2 $functor_3$ 1 arg_2 0 arg_3 0"

In fact, terms are going through the socket stream in this transformed format.

A server accepts connection from clients, reads requests, and serves for each request, concurrently (Figure 2). Service of *get*, *nget* and *put* are always done through the database ('db' in Figure 2). All the results of 'service' will be merged and owned by database. The data queue mentioned in the earlier section is realized by this merged results. The database has one more queue, the wait queue. A tag might not exist in the data queue when clients specify the tag to retrieve a term. There is no problem if the message is *nget*. In case of *get*, however, the client must wait for the tag till it will be put into the queue by another client. The database memorizes such jobs by putting them in the wait queue. In service for *put*, the database checks the wait queue first whether there are hanging requests for the tag. If it finds any, it gives a term to the waiting jobs, in addition to putting it into the data queue.

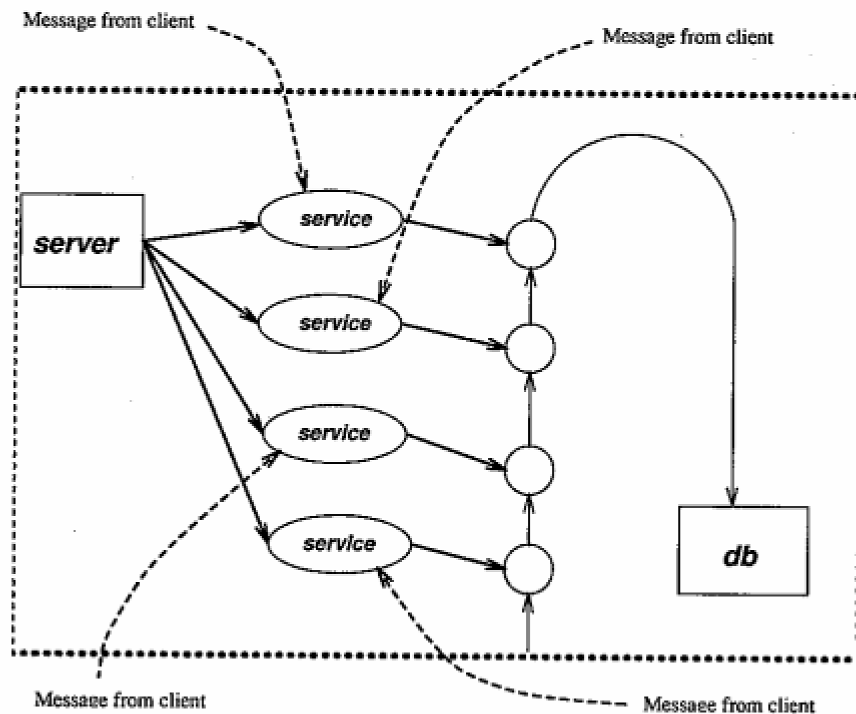

Figure 2: Mechanism of server

Let us see a very simple KL1 program as an example to send some terms to the server and receive them back from the server.

```
:- module main.
main :- cs:connect(string#"soukoku",10003,test,[dbcontrol(clear)|S]),
        put(S,S1),
        get(S1,[]),
        io:outstream([print(S),nl]).

put(S,S1) :- S = [put(a,b(c)),put(b,c(d)),put(c,e(f))|S1].
get(S,S1) :- S = [get(a,X),get(c,Y)|S1].
```

This program initializes the data queue and sends three terms with tag **a**, **b**, **c**. Then it retrieves terms with tag **a** and **c**.

2.3 Efficiency

Following are two programs for calculating $\sum_{i=1}^{1500} i$. One is using HAYDN, and the other is not.

- PROGRAM 1 (Using HAYDN)

  We use four clients. The first, second and third clients calculate $\sum_{i=1}^{500} i$, $\sum_{i=501}^{1000} i$ and $\sum_{i=1001}^{1500} i$, respectively. The forth client gathers the result of other three clients through the server and outputs sum of them.

- PROGRAM 2 (Not using HAYDN)

  Simply calculate $\sum_{i=1}^{1500} i$ by the single process.

When calculating $\sum_{i=1}^{1500} i$, no difference exists between execution time of **PROGRAM 1** and **PROGRAM 2**. However, in case of calculating $\sum_{i=1}^{6000000} i$, there is a big difference between them; 1 seconds for the former and 3 seconds for the latter.

So, we can say HAYDN programs can be much faster than equivalent KL1 programs if each of divided jobs is large enough compared to communication, though we should be careful to estimate gain and loss when writing programs.

3 Application of DKL1

3.1 A simple airline reservation system

DKL1 is designed to make it possible to realize such distributed systems that has some difficulties when written in KL1.

In KL1, the processor for each goal is automatically determined by KLIC system. Therefore it is impossible, in general, to specify a processor for a certain goal. This is reasonable when programs run on a multi-processor machine, but the situation will change when we deal with distributed systems. Considering a simple request-and-reply system, for example, the

reply should be processed at the same host the request comes from. In DKL1, programmers can write such a designation by specifying tags which represent proper names of hosts.

Following is a simple airline tickets reservation program called SATR to show the expressive power of DKL1 as a programming language for distributed systems [1].

SATR consists of two kinds of clients: a SATR manager and SATR windows. It is possible to have more than one SATR windows so that we can make reservations in different places at the same time. A client configuration of SATR is shown in Figure 3.

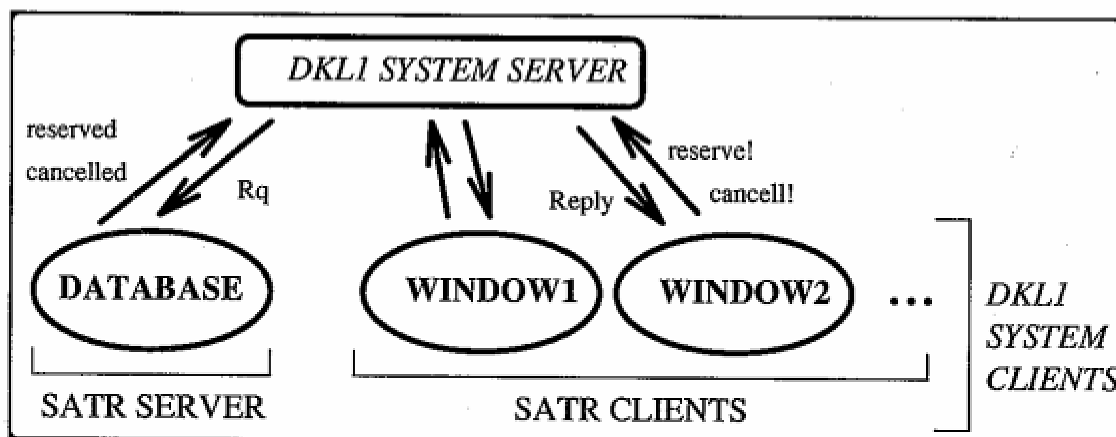

Figure 3: A client configuration of SATR

The manager program shown in Figure 4 consists of `eor/2` and `record/5`. The process `eor/2` plays a role of generating new database entries. The process `record/5` plays a role of managing a reservation condition for each request. These processes construct an object-oriented database where each entry is an individual object.

Asyncronous i/o is essential to these problems. Assume that asyncronous i/o was not supported. Then, the `get_requests/2` program shown in Figure 5 would suspend, even if there were other ready goals, until the HAYDN server received a message with a 'db' tag. DKL1 supports asyncronous i/o because the latest version of KLIC supports it.

3.2 A theorem prover MGTP

MGTP is a theorem prover developed at ICOT[3][4]. It tries to build models in a bottom-up fashion. The original MGTP is written in KL1 and runs in parallel on PIM, the dedicated parallel processors for KL1. To investigate expressive power of DKL1, we selected MGTP as a target program to be converted into DKL1.

At first, we describe the structure of original MGTP program briefly. Figure 6 shows its program structure.

In "mgtp" predicate, if no more candidate exists, the answer for that problem is "satisfiable" (model is found). If it fails, that is, the current model satisfies antecedent of a negative problem clause, its branch is "closed" and if all branches are "closed", the answer is "unsatisfiable". If the consequent of a selected clause is satisfied (i.e., it is redundant with

[1]We took no account of the problem concerning authenticity.

```
% eor( +MsgToDB, -PutMsgToSvr )
eor( [insert(Num,Name)|RestMsg],PutMsgToSvr ):- true |
        record( RestMsg, MsgToNext, Num, Name, PutMsgToSvr1 ),
        eor( MsgToNext, PutMsgToSvr2 ),
        merge({PutMsgToSvr1,PutMsgToSvr2},PutMsgToSvr).
eor( [reserve(Mado,Name)|RestMsg],PutMsgToSvr ):- true |
        PutMsgToSvr=[put(Mado,full)|RestPutMsgToSvr],
        eor( RestMsg, RestPutMsgToSvr ).
eor( [cancell(Mado,Name,Num)|RestMsg],PutMsgToSvr ):- true |
        PutMsgToSvr=[put(Mado,notfound )|RestPutMsgToSvr],
        eor( RestMsg, RestPutMsgToSvr ).
eor( [], [] ).
% record( +MsgToDB,-MsgToNext,+Num,+Name1,-PutMsgToSvr )
record([reserve(Mado,Name)|RestMsg],MsgToNext,Num,Name1,PutMsgToSvr ):-
        Name1  = empty |
        PutMsgToSvr = [put(Mado,reserved(Num))|RestPutMsgToSvr],
        record( RestMsg, MsgToNext, Num, Name, RestPutMsgToSvr ).
record([reserve(Mado,Name)|RestMsg],MsgToNext,Num,Name1,PutMsgToSvr ):-
        Name1 \= empty |
        MsgToNext = [reserve(Mado,Name)|RestMsgToNext],
        record( RestMsg, RestMsgToNext, Num, Name1, PutMsgToSvr ).
record([cancell(Mado,Name,Num)|RestMsg],MsgToNext,Num,Name,PutMsgToSvr):-
        PutMsgToSvr = [put(Mado,cancelled)|RestPutMsgToSvr],
        record( RestMsg, MsgToNext, Num, empty, RestPutMsgToSvr ).
record([cancell(Mado,Name,Num)|RestMsg],MsgToNext,Num1,Name1,PutMsgToSvr ):-
        (Num1 \= Num) |
        MsgToNext = [cancell(Mado,Name,Num)|RestMsgToNext ],
        record( RestMsg, RestMsgToNext, Num1, Name1, PutMsgToSvr ).
record([insert( Num,Name )|RestMsg],MsgToNext,Num1,Name1,PutMsgToSvr ):-
        MsgToNext = [insert( Num,Name )|RestMsgToNext ],
        record( RestMsg, RestMsgToNext, Num1, Name1, PutMsgToSvr ).
record([], MsgToNext, Num, Name, PutMsgToSvr  ):-
        MsgToNext=[].
```

Figure 4: SATR manager program

```
% get_requests( -GetMsgToSvr,-RqMsgToDB ).
get_requests( GetMsgToSvr,RqMsgToDB ):- true |
        GetMsgToSvr = [get(db,Rq)|RestGetMsgToSvr],
        RqMsgToDB = [Rq|RestRqMsgToDB],
        gotten_requests( Rq,RestGetMsgToSvr,RestRqMsgToDB ).
gotten_requests( Rq, RestGetMsgToSvr, RestRqMsgToDB ):- Rq \= finish |
        get_requests( RestGetMsgToSvr, RestRqMsgToDB ).
gotten_requests( Rq, RestGetMsgToSvr, RestRqMsgToDB ):- Rq  = finish |
        RestGetMsgToSvr=[],
        RestRqMsgToDB=[].
```

Figure 5: A program which needs asynchronous i/o

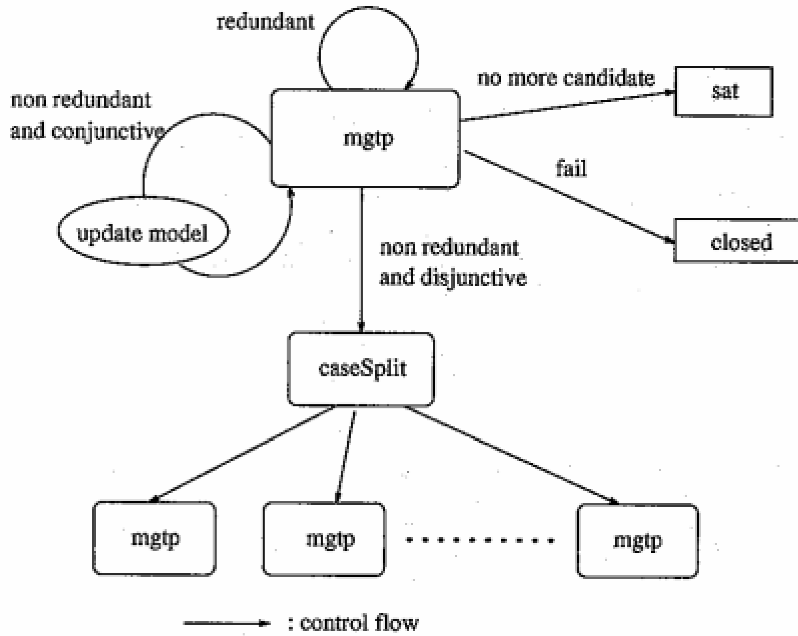

Figure 6: Structure of original MGTP program

respect to the current model), it neglects the clause and repeats this process by calling itself recursively. If the consequent is unsatisfied (non-redundant) and the clause is conjunctive, the model is updated. If the state is non-redundant and the clause is disjunctive, it calls "caseSplit" predicate. This predicate is defined recursively. It adds disjunct case to the new model expansion candidate, and calls "mgtp" predicate, taking the new candidate as its argument.

The feature of this program is that all activations of all "mgtp"s are done by recursive calls. In program code, "mgtp" and "caseSplit" predicates are defined recursively.

To convert KL1 programs into DKL1 ones, we need to divide them into modules each of which is to be allocated to a (logical) processor. Then, we need to design messages among those modules.

Figure 7 shows an image of the module configuration of distributed MGTP, which we are implementing on HAYDN, and messages flowing among them.

In this figure, $\langle\cdots\rangle$ show conditions to select that message-sending actions. The major difference of these two programs is control mechanism. While recursive calls are mainly used in KL1 version, message passings are used in DKL1 version.

We prepare a "manager" module for collecting answers from each branch. An "mgtp" module sends a message to a "manager" module, or to another "mgtp" module. For example, if the result of a certain "mgtp" module is "fail", it sends a "closed" message to the "manager" module. There are two more messages, "case_splited", "sat (*model*)", to be sent to the "manager". If the state is non-redundant (either conjunctive or disjunctive cases), it sends the message to ask for solving sub-problems to another "mgtp". When an "mgtp" receives a new sub-problem, it repeats the same process described above.

Unlike the original program, the DKL1 program has a fixed number of "mgtp" modules, and does not create additional "mgtp"s in the process of solving problems. On the other

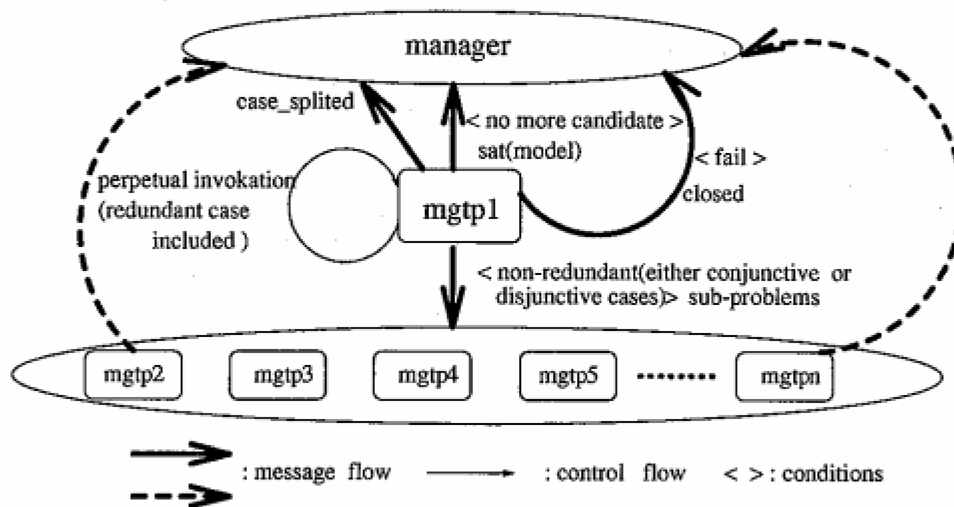

Figure 7: Image of distributed MGTP which we are implementing

hand, each "mgtp" performs perpetual invokation. Another feature of this program is the separaiton of invoking for perpetual process and processing sub-problems, while "mgtp" invokation for perpetual process is done by recursive call, sub-problem solving is activated by message passing.

Next, we explain implementation details of distributed MGTP on HAYDN. Figure 8 shows the client-server model for it. In figure 7, the "manager" module is a kind of server collecting answers of all branches. However, in relation to the "DKL1 system server", it is one of clients. So, when message passing occurs, the "DKL1 System Server" mediates between the "manager" module and "mgtp" modules (dotted lines in the figure). On the other hand, the "DKL1 System Server" itself plays the role of "sub-problems pool". If the state of executing "mgtp" module is "non-redundant", it puts a sub-problem to the server with infomation of D and M, where D means model expansion candidates, and M means model candidates.

At the beginning, an "mgtp" module gets a sub-problem from the server through message "get(to_mgtp, $\langle D, M\rangle$)", and if it expands a branch, it puts a sub-problem to the server through message "put(to_mgtp, $\langle D, M\rangle$)". If it does not expand a branch, that is, the branch is terminated, it puts the result of the branch to the server through message "put(to_manager,message(closed or model*(M)* or case_splited))". In this way, communication among modules is realized by message passing.

To clarify the difference, we show the essential parts of the two program codes below.

```
<original>                               |<distributed>
mgtp(D,M,Cn,Cg,S,...........,Os-Oz):-    | mgtp(S,Cn,Cg,D,M):-
 .........                               |  ......
 .........                               |  ........
 .........                               |  ........
 mgtp(NewD,[Delta|M},...........Oy-Oz))|  S=[put(to_mgtp,(NewD,[Delta|M]))|S1]
```

Another difference is the method for collecting answers. Answers are collected via message passing to the "manager" module. Three types of messages, "closed", "case_splited" and

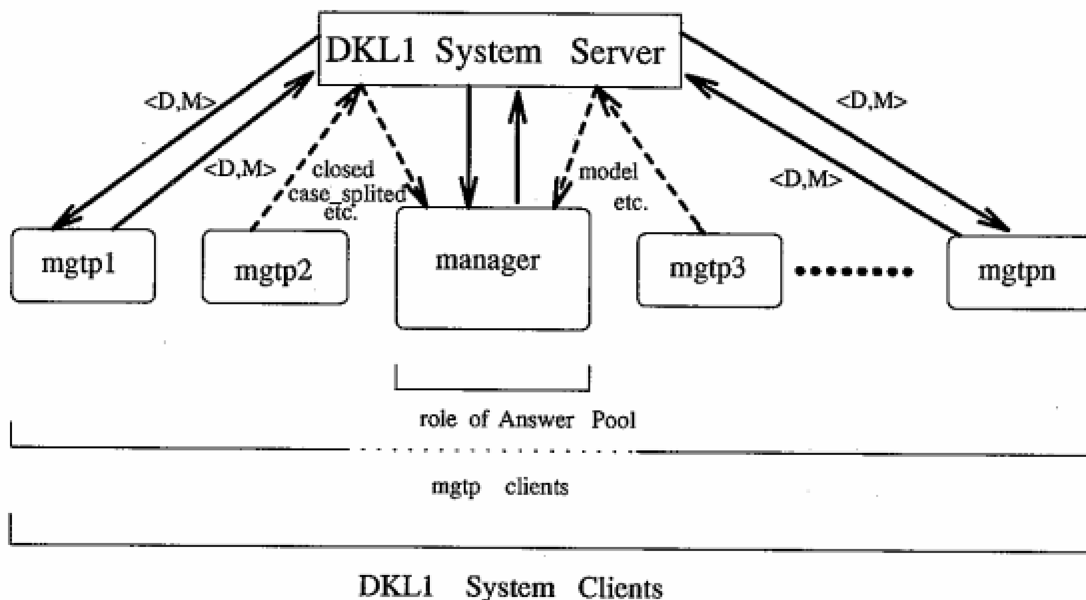

Figure 8: Client server model for the MGTP on HAYDN

"model(M)" are used to inform solutions to the "manager". The final answer is decided by the set of answers for all branches. If a model exists, the final answer is "satisfiable", otherwise (if no model exists, that is, all answers are "closed"), the final answer is "unsatisfiable". Precisely speaking, in our program code, the module of finding the final answer is not the "manager", but "checkSatUnsat". This module decides the final answer by investigating "Models", generated by the "manager" module.

Next, we explain termination detection in the "manager" module. We cannot use difference list and the short circuit technique because of the limitation on message passing that only ground terms can be used. The "manager" module counts visited terminal nodes, and when the count becomes equal to 0, it detects termination. When it receives a message of "case_splited", the number of open nodes increases by one, but when it receives a message of "closed" or "model(M)", the number decreases by one since one of open nodes is lost. Therefore, "manager" module not only collects answers of all branches but also detects termination by counting the number of visited terminal nodes.

To make the entire "mgtp" module as a perpetual process, we added a recursive call to itself. This recursive call allows multiple theorems to be proved concurrently in a single tail recursive process. This strategy solves the problem of handling indefinite number of sub-theorems in proving a theorem by a fixed number of processors.

4 Conclusion

We designed a language DKL1 as an extension of KL1 to write distributed programs. It is a very slight extension of KLIC so that it can be easily installed on any UNIX machines connected with networks. We also developed a run-time supporting system HAYDN on which

DKL1 programs can run in parallel and distributed way. The entire HAYDN system is written in KL1 to make it readable, portable and maintainable.

We adopted client-server model to realize communications among prosessors. All messages are sent to other (logical) processors via a central server located at some workstation.

We showed two programs to prove the feasibility of DKL1 and HAYDN. The first one is a simplified airline reservation system, which makes each UNIX workstation as a terminal for ticket reservation. We selected this example to show the expressive power of DKL1/HAYDN as a distributed programming language/system.

The second one is a concurrent theorem prover, MGTP. We succeeded to convert original MGTP in KL1 to a distributed version with relatively few efforts by adopting message oriented programming.

There are several research problems to be done in the future. First, we need to improve our current implementation of HAYDN. We have not fully evaluated the preformance of HAYDN architecture yet. We recognize that there are two major problems to be solved in order to attain a reasonable performance on HAYDN; one is an overhead of ground term communication via socket and the other is a communication bottle neck in a single server.

Second, our experiences of writing DKL1 programs are not enough and we need to write programs extensively in order to develop programming techniques for DKL1.

Finally, we want to find applications for real use to show the feasibility of our approach in the fields of both distributed and concurrent systems.

Acknowledgements

The authors want to express their special thanks to Mr. Hiroshi Fujita who kindly gave deliberate advises in designing distributed MGTP. They also want to thank Dr. Chikayama who suggested detailed programming for the realization of communications via socket in UNIX environment.

References

[1] Kazuo Taki, ed., *Parallel Processing in the Fifth Generation Computers,* bit, Special Issue, July 1993.

[2] Takashi Chikayama, *KLIC User's Manual,* ICOT,May 1994.

[3] Rainer Manthey and Francois Bry, *SATCHMO: a theorem prover implemented in Prolog,* Proc. of CADE 88, Springer Verlag, May 1987.

[4] Ryuzo Hasegawa and Masayuki Fujita, *Parallel Theorem Provers and Their Applications,* Proc. FGCS'92, 1992.