Turk J Elec Engin, VOL.9, NO.2 2001, © TÜBİTAK

BAG Distributed Real-Time Operating System and Task Migration[*]

Bekir Tevfik AKGÜN[†]
Computer Engineering Department, Faculty of Electrical and Electronic Engineering, Istanbul Technical University, Ayazağa, 80626, İstanbul-TURKEY

Abstract

BAG is a distributed operating system designed for real-time applications which is run on a distributed real-time system. The heterogeneously distributed BAG system consists of nodes which have VME-bus chassis, different types of processor modules, and an interconnection network. The operating system has three main parts having distributed properties: task migration, load balancing and a distributed file system. Heterogeneous task migration is based on the extended finite state machine (EFSM) programming model. The EFMS model has also eased the implementation of the migration mechanism. The load balancing algorithm is centralized in one node. But the overall system will be a multi-centered structure. Another objective of our work is to achieve a fast load balancing mechanism suitable for real-time systems. A file system supporting the task migration mechanism is also designed and developed. Users and processes all have the same view of this file system as a global tree. The file system uses a client/server approach and meets distributed file system requirements with real-time concepts such as priorities and time-out values.

Key Words: *Distributed systems, real-time systems, operating systems, task migration, load balancing, distributed file system.*

1. Introduction

The scope, quality and user-friendliness of computer systems depend on the services provided by their operating systems. A large number of operating systems have been implemented for centralized computing systems. Today, computers are spread out over long distances and they can be connected via LANs or WANs. These computers can operate in coordination with the supervision of a distributed operating system. These systems are called distributed systems and their coordination is managed by distributed operating systems. Distributed systems are subject to dynamic change of hardware and software components; moreover, they are reconfigurable and extensible.

Real-time applications such as computer integrated manufacturing, telecommunications, signaling, management and transaction processing can be executed on distributed architectures which are managed by real-time operating systems. The extended finite state machine (EFSM) programming model, which

[*]This work is supported by the Scientific and Technical Research Council of Turkey, project EEEAG-BAG2.

[†]Present Address: Communication Design Deartment, Faculty of Art and Design, Yıldız Technical University, Yıldız, 80750, İstanbul-TURKEY

is widely used in message based real-time environments, is highly suitable for those applications. As a consequence, specification and description languages like SDL have been defined and standardized. There are also methods for representing computation in real-world systems, such as the Actor model [1]. Actors encapsulate a state and a set of procedures that manipulate the state and a thread of control [2].

A real-time system is one whose basic specification and design correctness arguments must include its ability to meet its timing constraints as stated in [3]. If the timing requirements of hard *real-time* systems are violated, the results could be catastrophic. By contrast, there are applications that also have deadlines, but are noncritical. Thus, such failures will not be catastrophic in *soft real-time* systems. Real-time applications on distributed systems are becoming widespread by means of technological development. Studies on distributed real-time systems structured by clusters, or stations connected by an interconnection network are concerned mostly with improving the real-time constraints of communication systems. HARTOS provides services necessary to support time-constrained and fault-tolerant communication [4]. These include real-time channel service, deadline-based scheduling, clock synchronization, and group communication. The source code of the pSOS+ real-time kernel was modified for providing these services in this work. The three evaluation tools are a synthetic workload generator, a real-time monitor and a software fault injector, which collectively create a facility for various experiments on HARTS and HARTOS. Distributed system extensions for generalized rate monotonic scheduling are studied in [5]. It is assumed that in this work, sometimes both real-time and non-real-time applications must co-exist in some processor and share the network with real-time traffic to reduce the number of processors in a system. Three algorithms are presented in [6] to deal with independent jobs that are preemptable and migratable, or preemptable and nonmigratable, or nonpreemptable. It is often too costly to migrate jobs among processors in a distributed system. Consequently, time constrained jobs are not migrated. The main objective of a distributed operating system is to improve the overall system performance by using process migration. Processes are moved among the nodes of the system under control of the load balancing rules. Process migration requires a distributed file system, because a process can continue accessing its own opened file after moving it to another node.

This paper presents a message based distributed real-time operating system prototype called BAG [7]. The BAG distributed real-time operating system project (EEEAG-BAG2) is supported by TÜBİTAK (the Scientific and Technical Research Council of Turkey). We use EFSM as the BAG programming model. The task migration mechanism is based on user-level states which suit the EFSM model in BAG. Using the BAG model, we add distributed properties to the real-time system. The main concepts of of BAG model are also suitable to developing abstraction mechanisms to simplify the task of developing and maintaining open systems in Agha's work [2].

The main objective of this work is to incorporate the properties of a distributed operating system mentioned above in one existing heterogeneously distributed real-time system. The hard real-time components of an application must be assigned as nonmigratable. The properties of the distributed operating system will be valid for non-real-time or soft real-time components of the application program or application specific system tools. When the number of nonmigratable tasks is increased in a particular node, these tasks receive a great amount of CPU power with their higher priorities. The same situation will be encountered while the nonmigratable tasks need CPU power when the external actions are increased, resulting in real-time behavior of the controlled system. Thus, it will be convenient to move migratable tasks to other lower loaded nodes in order to enable the CPU to be used for these tasks, giving more resources to nonmigratable tasks. One step of load balancing can be initiated by the user for migratable tasks. Therefore, task migration and load balancing are useful tools for system users for tuning the soft real-time or non-real-time portions of the real-time system. A distributed file system with real-time properties is provided for migratable tasks. Although nonmigratable tasks can still use the local file system as in the original system, these tasks can also use the distributed file system. In this case, nonmigratable tasks gain control of file access by using higher priority.

The source code of the pSOS+ real-time kernel is not modified; the kernel is only extended with additional software. The Ethernet based interconnection network is available during this work. The FDDI based interconnection network provided time-constrained real-time communications, and a new interconnection structure is proposed for another TÜBİTAK project (EEEAG-BAG3) [8]. But the scope of this paper is limited to distributed services for a distributed real-time system.

2. The System Architecture

The platform which forms the basis of BAG is an experimental multiprocessor system that consists of nodes connected by an interconnection network. The node configuration is based on the VME common bus standard because it is fast and commonly used in real-time applications. Each node has a VME-bus chassis, a host processor, and target processors which run application tasks. The host processor performs the coordination of system management and monitoring actions. Application development tools are located on the host.

The host processor is an MVME 167 (Motorola) board with a 68040 32 bit processor, which is 16 MB RAM accessible from the VME-bus [9]. A hard disk, a diskette drive, a tape cartridge, an Ethernet interface and an ASCII terminal are also connected to the MVME 167 board. The host runs under the UNIX operating system and executes the VMEexec real-time software development package.

Two kinds of MVME 162 boards, with and without arithmetic coprocessor, that have 68040 microprocessors [10] with 4 MB RAM, and MVME 187 boards that have 88100 RISC microprocessors [9] with 4 MB RAM are used as target processors. A hard disk, an Ethernet interface and an ASCII terminal are also connected to the target processors. The target processors run under the pSOS+ real-time kernel. Application programs for the target processors are written in C language.

3. The Software Platform

The BAG real-time operating system is based on the pSOS+ Real-Time Executive [11], which is a part of VMEexec software. VMEexec [12] is a real-time software development system which runs under UNIX. VMEexec is used to develop real-time applications in C language which make use of the VME bus. The kernel of VMEexec is based upon the real-time executive interface definition (RTEID), which defines a core set of operating system services. The RTEID serves as a complete definition of real-time executive external interfaces, ensuring that the application source code conforming to these interfaces operates as defined in all real-time executive environments. Multiprocessing is medium independent, supports the concept of location-transparent objects, and adheres to decentralized, peer-only architecture.

The pSOS+ specifications define several functions, known as directives or services, and group them into a set of resource managers. The pSOS+ uniprocessor kernel serves as the executive on each target processor. A task is the basic unit for sequential execution, resource ownership and scheduling. The kernel uses a pre-emptive, priority-based scheduler that lets tasks of equal priority cycle in round-robin fashion. Since pSOS+ is not a kernel of a distributed operating system, it does not provide or support task migration and load balancing over a multiprocessor system. We have enhanced pSOS+ with task migration and load balancing mechanisms. We have also developed a distributed file system on our platform.

In a loosely coupled distributed environment, a parallel application consists of several cooperating tasks that communicate with each other through message passing. The pSOS+ kernel provides message queues so that tasks communicate through message exchange in a single target processor. VMEexec software supports the multiprocessing, as the message queues are location-transparent objects in one VME chassis. Therefore, of it, two tasks located on different target processors can communicate with each other via a global message queue. However, neither of these tasks is required to know whether this message queue is global or not. In this step of developing the BAG distributed real-time operating system, we also use message queues of pSOS+ in a single VME chassis.

4. The Task Migration

In process-based distributed operating systems, process migration consists of moving a process from the processor on which it is executed by another processor. The need for process migration has been emphasized by Solomon and Finkel [13]. It has been shown by successful implementations that process migration is possible in pre-emptive systems [14],[15]. There are two basic groups of issues in a process migration mechanism. The first group consists of the determination of the process state, the detachment of the process from its current environment, and the transferring and connecting of the process to the new environment. The state of the process consists of text, stack and data segments, register contexts, information about some internal parameters and system queues, heap area, message queues, and communication information with other processes. The second group of issues relates to the continuity of communication directed to the migrating process and removal of side-effects of the process migration on the remaining processes.

The BAG distributed real-time operating system is based on message-passing processes like DEMOS/MP [16], Charlotte [17], and V-System [18]. In these three operating systems, the the process to be migrated is suspended with its current state, the necessary operations for state transfer and the continuity of message communication are accomplished and the process is resumed on the destination processor. At the beginning of the migration operation, the process is marked as “in migration” and extracted from the queue in which it was placed. On the completion of migration, the process is resumed on the destination processor within the state it was in before the operation (i.e., ready, waiting or suspended). The V-system uses a state-change-driven information policy [19]. Each node broadcasts its state whenever its state changes significantly. State information consists of expected CPU and memory utilization. The Sprite system is targeted toward a workstation environment [20]. Sprite uses a centralized state-change-driven policy. Each workstation, on becoming a receiver, notifies a central coordinator process.

Today, one software design tendency is to model application programs for real-time systems as extended finite-state machines (EFSM)[21]. The EFSM model consists of user defined states and state transitions. On entering a new state, execution is suspended until the receipt of a message whose arrival initiates a set of operations called state transition. Results, including outgoing messages, are produced and the transition is completed with the determination of the next state. In real-time systems, it is desirable that transition operations between application program states be executed as fast as possible, with minimum interruptions.

Two levels of states are defined for an application process: kernel-level process states and user-level program states. In the process based operating systems mentioned above, processes are migrated while in a kernel-level state. The operating system starts the migration process in a preemptive manner whenever it is found necessary.

In the pSOS+ context, processes are called tasks. Therefore, the word “task” is used to refer to a “process” in the following sections.

The task migration mechanism of the BAG operating system is based on user-level states. A task is not migrated when it is active, executing the operations of a state transition. A task is migrated only when it is waiting for a message at a user-level state where it is passive. This model improves performance and also simplifies problems related to the continuity of message handling. Issues that appear before and after migration in other operating systems, such as message redirection, message loss prevention and message loss recovery, do not to exist in our model.

The user may also specify tasks which are not to migrate. These tasks may be dedicated hardware drivers or user manager programs, etc.

Task migration mechanisms can be implemented at three different levels [22]:

1. at the operating system kernel level
2. outside the kernel
3. embedded in a compiler and runtime package.

The task migration mechanism of BAG is implemented through manager programs which run on target modules and the library functions written to support the mechanism [23]. These library functions are embedded in the application program via a preprocessor which is also implemented as a part of the mechanism. This preprocessor helps in the achievement of transparency in user code generation. Our implementation of the task migration mechanism can be regarded as a combination of the second and third levels mentioned above. The BAG programming model and task migration mechanism are explained in the following sections.

5. The Load Balancing

The primary advantages of distributed systems are high performance, availability, and extensibility. To realize these benefits, system designers must overcome the problem of allocating the available processing capacity so that the system is used to its fullest advantage. Several working load-distributing , including V-system [19], Sprite [20], Condor and Stealth, are reviewed in [24]. The performance comparison of the algorithms is also given. The V-systemrs selection policy selects only newly arrived tasks for transfer. Sprite's selection policy is primarily manual. Tasks must be chosen by users for execution. Condor is concerned with scheduling long-running CPU-intensive tasks only. Condor's selection and transfer policies are similar to Sprite's in that most transfers are manually initiated by users. The Stealth Distributed Scheduler differs from V-system, Sprite and Condor in the degree of cooperation.

In the following paragraphs, an application of a load balancing algorithm on BAG will be presented. This is a centralized algorithm if one observes only one node of the system. But the overall system will be extended to do load balancing in a multi-centered manner by migrating tasks form one node to another. The center for this implementation is the host of the node. One of the primary objectives of this application is to achieve a fast load balancing mechanism suitable for real-time systems.

The implementation load balancing algorithm is based on the measurement of CPU utilization [23]. The V-system's load index is the CPU utilization at a node [19]. To measure CPU utilization, a background process that periodically increments a counter is run at the lowest priority possible. The counter is then polled to see what proportion of the CPU has been idle. A CPU utilization monitor task called *monitor*, of the lowest priority level, measures CPU utilization in this work. This task runs on every target. Each time the *monitor* receives the CPU, it increments its local *counter* by one and preempts itself. The lower the counter value, the higher will be the CPU utilization factor of the CPU. The terms "load" and "CPU utilization" are used interchangeably in this paper.

The basic components of the load balancing algorithm implemented in the BAG distributed system are transfer policy, selection policy, location policy and information policy. These policies and their realization are described below for a system which has only one node and one center (Fig. 1). Some experimental results are shown in Figure 2.

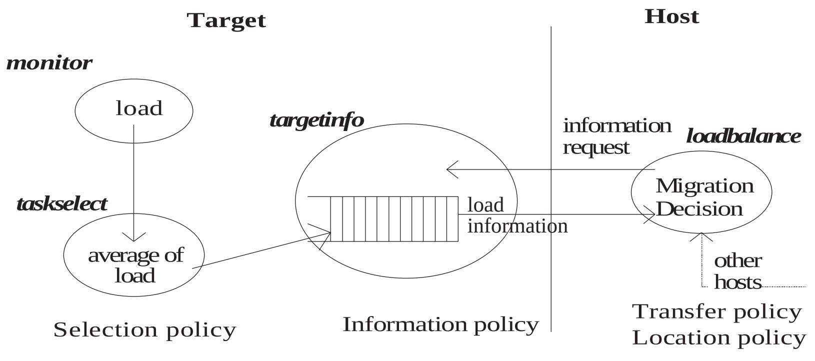

Figure 1. Policies and the realization of the algorithm

	target1	target0
LOADING app1, app2		
task name:	app1	app2
load time:	3 sn	2 sn
load of targets:	52%	58%
LOADING app3		
task name:	app3	
load time:	3 sn	
load of targets:	92%	34%
LOADING BALANCING		
transferred task:	app1 $\longrightarrow$	
transfer time:	4 sn	
load of targets:	83%	74%

Figure 2. Test Results for the Algorithm

Transfer policy: Transfer policy determines whether a node is in a suitable state to participate in a task transfer. The host performs this policy within the *loadbalance* program. Many proposed transfer policies are threshold policies [25]. According to the threshold policy realized in the system, targets which are loaded more than 50% are chosen as senders whereas targets which are loaded less than 50% are chosen as receivers. But this condition is not sufficient for the stability of the algorithm; thus, the transfer policy works in conjunction with the location policy.

Selection policy: After the transfer policy decides that a target is a sender, a selection policy selects a task for transfer. This policy is performed by the *taskselect* task which runs on each target. In BAG, each target selects the most suitable task to be transferred. This is the task with the least memory requirement. This guarantees that the time needed for the transfer of the task will be the shortest.

Location policy: The location policy is responsible for finding a suitable "transfer pair". The *loadbalance* program that runs on the host determines the most suitable sender-receiver pair according to the information calculated and sent by each target. As this information is retrieved, the most loaded and the least loaded targets are determined. The most loaded target with a load greater than the threshold value is chosen as a sender, while the least loaded target with a load less than the threshold value is chosen as a receiver. In order to achieve the stability of the algorithm, the following strategy is adopted: if the load difference of a chosen sender and receiver pair is less than half of the threshold value, there is no task transfer between them. Location policy tries to find a suitable pair of the same type of target because homogeneous migration is performed between these types of target and this kind of migration has a lower cost than the heterogeneous migration process.

Information policy: The load information of each target is calculated by the targetinfo task periodically in our system. This task runs on each target and stores the load information in a circular queue. The host gathers the information at predetermined intervals and terminates the load balancing operation. The intervals of information gathered by the host are determined according to the performance criterion of the system.

6. The Distributed File System

As in all operating systems, one of the most important features of distributed systems is their support for distributed file operations. A distributed file system enables its users to access a file without knowing the exact location of it in the network. Ideally, users should not suffer from any performance degradation regardless of whether the file to be accessed is local or remote.

The BAG file system is based on the hierarchical tree principle. All files and directories in the system are located on a global tree. Users and processes of the system all have the same view of this tree. In this way, any file in any point of the file tree can be shared in a system-wide manner. The file system uses a client/server approach for providing distributed file service. In this structure, clients on the nodes of the distributed system send their service requests to the servers that are dedicated system nodes for performing file operations.

The file system meets distributed file system requirements in respect of real-time concepts. Thus a client requesting file service sends its request by assigning it a priority level and a time-out value to be used in the event that the priority levels of requests from contending clients for the same file are equal. Servers provide service first to the request having the highest priority level. If there is more than one request with the same priority level, the request having the least time-out value receives service first. File sharing throughout the system is achieved by locking the whole file. This implies that one or more processes can read the same file at the same time without any conflict. In the general case when a group of processes wants to write to a file and another group wants to read from the same file, only one of the writer processes is allowed to perform the operation, the file being completely locked to the others. Other processes wait for the completion of the operation. Our prospective objective is to enable record locking in the system. Some implementations such as Newcastle Connection [26] and Sun Microsystems' NFS [27], LOCUS file system [28] are based on block (record) sharing.

Like other implementations, this file system provides a naming mechanism. Any object (user, process) accesses a file by its name without knowing its location. This implies that, when a task is migrated from one node to another, the same accessing mechanism is kept valid. Since directories can be apprehended as special files, operations such as creating new directories, deleting the existing ones, and adding new files to the directories are all file-based operations and are supported. All distributed system designs have the objective of high scalability. The BAG file system enables the expansion of the system with new storage media and corresponding file servers in order to enhance performance. The file system enables its users to declare that a new file will be created on the local disk of the workstation instead of being created on a remote file server, although this contrasts with transparent access. This feature is designed to increase system performance by eliminating redundant network traffic. This sort of file creation can be performed on some distributed file system implementations like Cedar file system [29] and Andrew file system [30]. Users can experience high performance by using this feature, for instance, for tasks that are unlikely to be migrated. The Sprite kernel calls are very similar to those provided by the Berkeley versions of UNIX [31]. Although the Sprite file system is implemented as a collection of domains on different server machines, it appears to users as a single hierarchy that is shared by all the workstations.

Fault tolerance and crash recovery in distributed systems are realized by keeping multiple copies of particular files on several file servers. This approach requires the employment of a system-wide file consistency protocol. This means that all copies of an existing file must be consistent. The BAG file system does not provide this feature currently.

The distributed file system is composed of host nodes, target nodes with disk devices, and diskless target nodes. Distributed file service is carried out by file server tasks and agent tasks. File server tasks are loaded to all the nodes having attached disks. All target nodes run agent tasks which actually represent the server tasks on the targets. Client tasks running on the targets receive distributed file service in response to their requests via agents.

In the file system all file server tasks are located around a logical ring. Each member of the ring knows the identity of its successor in the ring. A file server in the ring, after receiving a file service request, checks whether the file in the request is present in its disk. If the file does not exist in its disk, it forwards the request to its successor. Thus the file is opened if it is present in an intermediate node of the ring. Otherwise, that is if the request returns to the first server, this means that the requested file does not exist in the system and is to be created in this node. In either case, file servers produce a unique file identifier (UFID). From this point on, all communications take place between the client and the file server by means of the UFID. A unique file identifier is a long word (for our system 32 bits) which corresponds to one and only one file throughout the system. A UFID comprises of two sections: a number designating the identity of the home file server and a number designating the count of files created on this server.

In the BAG file system, the maximum number of files that can be open in an instant of time can be given to the file servers. Since 16 bits are reserved for this number, at any time we can have 2^{16} files, an excessive number intended for future use. The file system locates all the files of the system on a global tree, and all the processes in the system have the same view of this tree. Since this tree is viewed to be the same by all processes, file sharing is made easy. This structure is also appropriate for the migration of processes. The aim of the distributed file system is to keep all the storage media balanced throughout the system. This is achieved by delivering storage limits to each file server. If a server detects that it has exceeded the limit for storing data, it forwards the request to its successor in the logical ring. Also, local storage areas are subject to limit. Local limits for storing data are selected to be greater than those for global storage in order to make vast amounts of free space in the local disk, thus reducing the overall network traffic.

7. The Programming Model

The EFSM model has been selected as the real-time programming model. Multiple tasks can be created in a nested manner, and in this case each child task has to conform to the EFSM model.

The "main" block of the sample program in Figure 3 has three basic sections: a section for the declaration of local variables, a section for initializations, and a section in which program codes for user-level states are specified by "switch" and "case" program control statements. The initialization section includes a primary and a generic initialization segment.

```
/*header files*/
/*global variables*/
main()
{
unsigned int state;
unsigned int qid;
/*other local variables*/
/*primary initialization program code*/
        if(initial_exec())
        {
                create(NAME("q000"),count,EXPORT,&qid);
        /*other codes          */
                state=first_state;
        }else
        /*generic initialization program code*/
        }
        /*user level states*/
        for(EVER)
        {
                switch(state)
                {
                        case first_state:
                                receive(qid,WAIT,timeout,buffer);
                                /* first state transition code
                                ...          */
                                state=next_state;
                                check();
                                break;
                        case next_state:
                                receive(qid,WAIT,timeout,buffer);
                                /* next state transition code
                                ...          */
                                state=state_i;
                                check();
                                break;
                        /*other transition codes for other states
                        ...          */
                }
      }

}/*end of main*/
```

Figure 3. A sample program in the BAG real-time programming model.

The primary initialization segment is the code segment that is executed only once, when the task is activated for the first time. In this segment, initialization operations on message queues are performed. The primary initialization segment is not executed after migration. The BAG library function "initial_exec()" returns a logical value indicating whether the task is being executed for the first time or not. The generic initialization segment is the code segment that is activated after each migration. This segment performs initialization operations related to child tasks. A temporary segment of data called the "safe segment" is defined in our implementations. This segment is needed due to the differences observed in the data and stack segments of the heterogeneous system. We have observed that compilers use different locations of memory for variables. Separate compilers must be used for the targets (MVME 162, MVME 187) in the heterogeneous system. A memory area of an abstract structure type is allocated for the safe segment to relocate the variables. The standard data representation is called *external data representation*, and its designer must successfully handle the problem of different representation for data, such as characters, integers and floating-point numbers. These issues were discussed by Maguire and Smith [32] for handling floating-point numbers in external data representation. We do not encounter this problem, because the same producer (Motorola) supplies our different types of microprocessor. But our mechanism will handle this problem by means of as safe segment. Global and local data of the task are transferred into the safe segment before the migration operation starts. When the migration operation is completed, the contents of this segment are restored to the data and stack segments of the migrated task on the destination target. For the transferring and restoring of the safe segment, an additional source code part is injected into the user code while preprocessing. Thus, the user task performs these transfers itself.

A task executes the library function "check" at the end of each "case" block, i.e., on each state exit. This function first checks the existence of a migration request on the related task. If no migration request is present on the task and there is a message on its own message queue apprehended by polling, it simply continues execution with the next state transition code. Conversely, if there is no migration request and no message, it informs the manager by sending its a message including own task i.d. and message queue i.d. and suspends itself for rechecking again. When a new message arrives or there is a new migration request present, the manager resumes this task. However, if a migration request is present, the task is suspended. The programming model supports child tasks and requires them to be migrated with their creator. It is necessary for every child task to execute its "check" function for the migration operation to be started. The greater the number of child tasks waiting for their execution on the ready system queue, the higher the possibility of delays introduced by waiting for the child tasks to reach their "check" functions. Such delays may be unacceptable in real-time applications. For this reason, the number of child tasks may also be used as a task selection criterion in the selection policy of the load balancing algorithm.

8. The Task Migration Mechanism

The migration process begins after the selection of a task to migrate and proceeds in three phases: *negotiation, transfer* and *establishment.* Three different processor modules contribute to the migration process: host, source target module (STM), and destination target module (DTM). The migration mechanism has two submechanisms: *homogenous* and *heterogeneous migration mechanisms.* The homogeneous migration mechanism is applied when two processors of the same type take part in task migration. If both of the processors are of different types, then the heterogeneous migration mechanism is employed. The host decides on which migration action to be taken. The main role of the host in migration mechanism is control and coordination. The task migration mechanism is implemented through extensions to the pSOS+ kernel. The manager programs that have been developed for both host and targets are responsible for the execution of the task migration protocol. Because of the BAG programming model, there is not much difference between the homogeneous and the heterogeneous migration mechanisms. The manager program on the host starts the migration process on receiving a task name, a source target module name, and a destination target module name. At the completion of the migration operation, the host receives the results (Figure 4).

Negotiation phase: The manager program designates the task to be migrated by transferring its parameters to the STM. The STM marks the specified task as "in migration" mode. When this task executes the "check" primitive, it is blocked. If it has child tasks, every child task of the migrating task executes the "check" statement and they are also marked as "in migration". As the last child task executes its "check" statement, the transfer phase of migration begins.

Transfer phase: It is in this phase where the difference appears between the homogeneous and heterogeneous migration mechanisms. In the homogeneous migration mechanism, the code of the task is transferred directly from STM to DTM without the need for recompilation, whereas in order to transfer the code of the task in the heterogeneous migration mechanism the task is loaded from the host as if it were a new task that had never been executed before. After all the data related to the task –safe segment and dynamically allocated memory segments– is transferred, the task is resumed to be executed by the system.

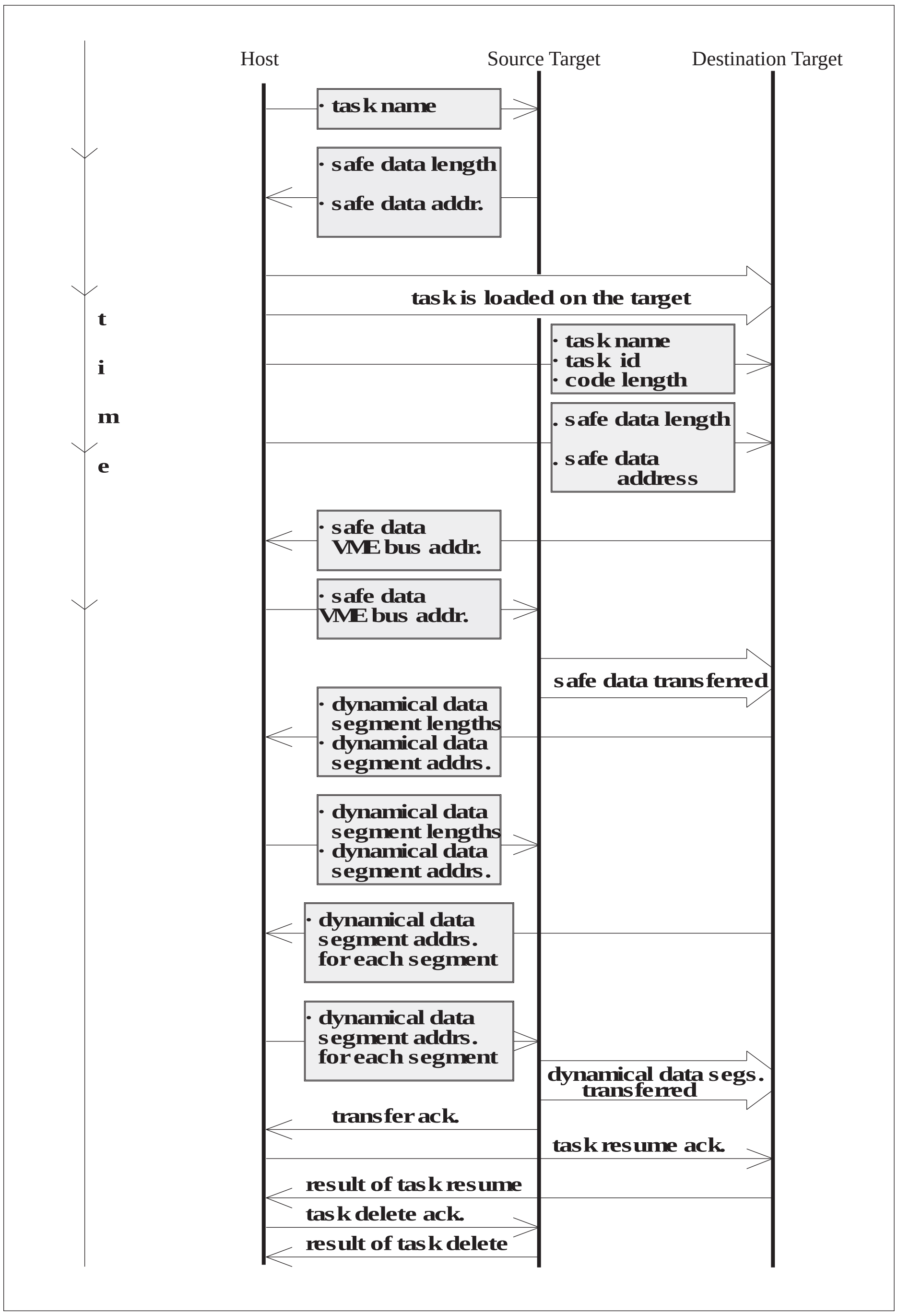
Host
Source Target
Destination Target
· task name
· safe data length
· safe data addr.
task is loaded on the target
t
i
m
e
· task name
· task id
· code length
. safe data length
. safe data address
· safe data VME bus addr.
· safe data VME bus addr.
safe data transferred
· dynamical data segment lengths
· dynamical data segment addrs.
· dynamical data segment lengths
· dynamical data segment addrs.
· dynamical data segment addrs. for each segment
· dynamical data segment addrs. for each segment
dynamical data segs. transferred
transfer ack.
task resume ack.
result of task resume
task delete ack.
result of task delete

Figure 4. The flow diagram of a successful task migration.

Establishment phase: If the migration process proceeds without error up to this point, the task first executes generic initialization operations which recreate child tasks if they exist to enable the completion of their migration. After the termination of generic initializations, the "switch" statement passes control to the last state the task was in before migration.

9. Results and Conclusion

BAG is a distributed operating system designed for real-time applications which suit the EFSM programming model. The EFSM model has eased the implementation of the migration mechanism and has eliminated several problems that are present in other similar systems that support migration. The main objective of the BAG model is to enable heterogeneous task migration. By means of the BAG model, after recompilation the source code of a particular task, the code segment of this task is easily changed with the heterogeneous task migration mechanism.

A simple yet efficient load balancing algorithm is implemented. Its simplicity arises from the load measuring strategy and implementation of the policies. Its efficiency results from this simplicity. Fast response of load balancing is also of much importance in real-time systems, and the load balancing algorithm realized for BAG yields promising results on this subject. This algorithm uses information about the system state at a particular time; thus, it does not need any pre-calculated information about the tasks. Therefore, it is a dynamic algorithm.

The BAG file system introduces distribution into the system by using system-wide operations provided by the VMEexec software package. Agents and file servers all utilize queues for inter-task communication. In the BAG file system, all users share a global name space. Thus any user can access any file at any point of the name space.

Both homogeneous and heterogeneous migration mechanisms have been successfully tested over the BAG distributed system. As expected, heterogeneous migration takes more longer time than homogeneous migration. For small sized tasks, such as 10 kB or 50 kB, the time difference between the mechanisms is almost of the same order (Figure 5). However, it is observed that, as the task size increases, the time needed for migrating by heterogeneous migration mechanism increases dramatically. Therefore, the load balancing algorithm that we have designed takes this point into account and selects the task with the least memory requirement, for migration.

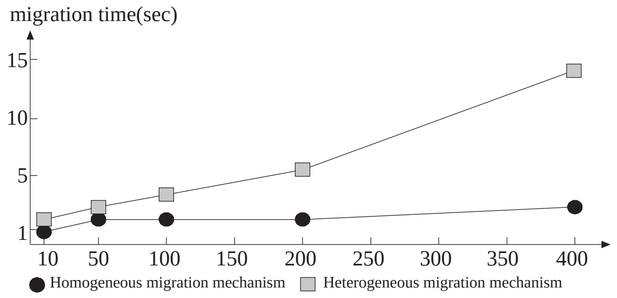

Figure 5. Migration time vs. length of task.

For future work, we are studying the implementation issues of task migration between CPUs of various families which possess different data representation and byte ordering. Since BAG is a distributed real-time operating system, we are trying to make the migration facility meet real-time requirement induced by the migrating tasks. Work is being carried out to enable a task to specify a migration duration in order to determine whether a delay is acceptable for the task under real-time conditions. If the migration operation cannot be completed by the specified deadline, the operation will be aborted and the task will resume execution on its original CPU.

10. Acknowledgments

I would like thank Emre Harmancı for the BAG project. Bülent Örencik and Nadia Erdoğan provided suggestions on designing the BAG project. Ersoy Pekşen implemented most of the mechanisms of migration and load balancing. Gürhan Menderes wrote the program of the file system.

References

[1] G.A. Agha, *Actors: A model of Concurrent Computation in Distributed Systems,* MIT Press, Cambridge, Mass, 1986.

[2] G.A. Agha, "Abstracting Interaction Patterns: A Programming Paradigm for Open Distributed Systems," Formal Methods for Open Object-based Distributed Systems, IFIP Trans., 1997.

[3] S. H. Son, *Advances in Real-Time Systems,* Prentice-Hall, Inc., 1995.

[4] K. G. Shin, et al., "A Distributed Real-Time Operating System," IEEE Software, Vol. 9(5), pp. 58-68, 1992.

[5] L. Sha and S. S. Sathaye, "A Systematic Approach to Designing Distributed Real-Time Systems," IEEE Computer, Vol. 26(9), pp. 68-78, 1993.

[6] R. Ha and J. W. S. Liu, "Validating Timing Constraints in Multiprocessor and Distributed Real-Time Systems," Technical Report UIUCCS-R-93-1833, University of Illinois at Urbana-Champaign, 1993.

[7] B.T. Akgün, et al., "BAG Real-Time Distributed Operating System," 3rd International Conference on High Performance Computing HiPC, India, pp. 120-125, 1996.

[8] B. Orencik, "A Hierarchical Interconnection Network for Real-Time Systems," Elektrik, Vol 6(2), pp. 131-166, 1998.

[9] Motorola, *MVME166/167/187 Single Board Computers Programmer's Reference Guide,* 1992.

[10] Motorola, *MVME 162 Embedded Controller User's Manual,* 1993.

[11] Motorola, *pSOS+/MC68040 Real-Time Executive with MMU User's Manual,* 1992.

[12] Motorola, *VMEexec Reference Manual,* 1991.

[13] M. H. Solomon and R. A. Finkel, "The Roscoe Distributed Operating System," 7^{th} Symposium on Operating Systems Principles, California, pp. 108-113, 1979.

[14] M.M. Theimer, et al., "Preemptable Remote Execution Facilities for the V-system", 10^{th} Symposium on Operating Systems Principles, ACM, pp. 2-12, 1985.

[15] E.R. Zayas, "Attacking the Process Migration Bottleneck," 11^{th} Symposium on Operating Systems Principles, ACM, pp. 13-24, 1987.

[16] M.L Powell and B.P. Miller, "Process Migration in DEMOS/MP," 9th ACM Symposium on Operating System Principles, New Hampshire, pp. 110-119, 1983.

[17] Y. Artsy and R. Finkel, "Designing a Process Migration Facility. The Charlotte Experience," IEEE Computer, Vol. 9, pp. 47-56, 1989.

[18] D.R. Cheriton, "The V Distributed System," Comm. of the ACM, Vol. 31(3), pp. 314-333, 1988.

[19] M. Stumm, "The Design and Implementation of a Decentralized Scheduling Facility for a Workstation Cluster," Proc. Second Conf. Computer Workstations, IEEE CS Press, pp. 12-22, 1988.

[20] F. Douglis and J. Ousterhout, "Transparent Process Migration: Design Alternatives and the Sprite Implementation," Software Practice and Experience, Vol. 21(8), pp. 757-785, 1991.

[21] CCITT, *Blue Book, Annex D to Recommendation Z.100 SDL User Guidelines,* 1988.

[22] A. Goscinski, *Distributed Operating Systems The Logical Design,* Addison-Wesley Publishing Company, 1992.

[23] B.T. Akgün, et al., "An Implementation of a Load Balancing Algorithm for the BAG System," 14^{th} IASTED Int. Conference on Applied Informatics, Austria, pp. 43-45, 1996.

[24] N. G. Shivaratri, P. Krueger, and M. Singhal, "Load Distributing for Locally Distributed Systems," IEEE Computer, pp. 33-44, 1992.

[25] T. Kunz, "The Influence of Different Workload Descriptions on Heuristic Load Balancing Scheme," IEEE Trans. on Software Engineering, Vol. 17(7), 1991.

[26] D. Brownbridge, et al., "The Newcastle Connection - Or UNIXes of the World Unite!," Software Practice and Experience, Vol. 12, pp. 1147 1162, 1982.

[27] D. Walsh, et al., "Overview of the Sun Network File System," Usenix Winter Conference, pp. 117 124, 1985.

[28] B. Walker, et al., "The LOCUS Distributed Operating System," ACM SIGOPS Op. Sys. Rev., Vol. 17(5), pp. 49-70, 1983.

[29] D. Gifford, et al., "The Cedar File System," Commun. of the ACM, Vol. 31, pp. 288 298, 1988.

[30] J. H. Morris, et al., "A Distributed Personal Computing Environment," Commun. of the ACM, Vol. 29, pp.184 201, 1986.

[31] J. K. Ousterhout, et al., "The Sprite Network Operating System," IEEE Computer, Vol. 21(2), pp. 23-26, 1988.

[32] G. Q. Maguire and J. M. Smith, "Process Migration: Effects on Scientific Computation," ACM-SIGPLAN Notices, Vol. 23(3), pp. 102-106, 1988.