ICS

TECHNICAL REPORT

Dynamic Object Migration in Logical Networks

Eugene Gendelman
Lubomir F. Bic
Michael B. Dillencourt

UCI-ICS Technical Report No. 99-52
Department of Information and Computer Science
University of California, Irvine, CA 92697-3425

November 4, 1999

Information and Computer Science
University of California, Irvine

Dynamic Object Migration in Logical Networks

Eugene Gendelman
University of California, Irvine 444
Computer Science Building, ICS
Irvine, CA 92697-3425
1 949-856-2719
egendelm@ics.uci.edu

Lubomir F. Bic
University of California, Irvine
Computer Science Building, ICS
Irvine, CA 92697-3425
1 949-856-2719
bic@ics.uci.edu

Michael Dillencourt
University of California, Irvine
Computer Science Building, ICS
Irvine, CA 92697-3425
1 949-856-2719
dillenco@ics.uci.edu

ABSTRACT

This paper presents a mechanism for migration of parts of computations in systems based on distributed objects or autonomous agents. In such systems load balancing and process migration can be achieved by migrating objects, or logical nodes, between participating computing nodes.

The presented migration mechanism uses minimal knowledge of the system topology and requires only a moved object to stop computing while in migration, which makes it usable in large distributed computations.

Keywords

Dynamic load balancing, agent system support, agent system architecture.

1. INTRODUCTION

The ability to migrate parts of the computation plays a very important role in today's distributed systems. It is used for such tasks as load balancing and process migration.

The computation migration can be accomplished by migrating the whole process [1], migrating a computation thread [2], or migrating objects [3]. The migration process can be broken down into two parts: capturing a state of the computation and the migration protocol. In this paper we discuss a novel migration protocol for distributed systems based on objects or the autonomous-agent paradigm [4]. The structure of such systems allows to balance the load of the processors by migrating a logical part of the computation; an object or a logical node. This type of load balancing can be implemented in an application-transparent, machine-independent way [1]. The logical topology can be modified during the computation, and therefore migration during such network modifications should be considered in the protocol.

The migration mechanism presented here induces a minimal overhead on the whole system, and requires minimal knowledge of the global state of the distributed application, which makes it useful for large distributed computations. This migration mechanism was implemented in the MESSENGERS [4] mobile-agent system.

The rest of this paper is organized as follows: section 2 gives an overview of the MESSENGERS system, section 3 describes the migration mechanism, section 4 presents related work and concludes this paper.

2. THE MESSENGERS SYSTEM

MESSENGERS is a distributed system based on the principles of autonomous objects. Autonomous objects in the MESSENGERS system are called Messengers. MESSENGERS target general purpose computing. The applications suitable for the MESSENGERS system include individual-based simulations [6-7], matrix multiplication [8], monte carlo simulations and others [5]. The agents in the MESSENGERS system are cooperating in solving one large problem. The computation is taking place on the trusted hosts that could be distributed through a network, but it assumes a common shared file system. The system is implemented in C. Unlike most mobile agent systems [13-17] agents in MESSENGERS are fully compiled into machine code. On migration, agents don't carry code with them, but just a pointer to the next executing function [8,9]. These qualities make MESSENGERS suitable for high speed computing.

2.1 MESSENGERS PARADIGM

MESSENGERS distinguishes three separate levels of network: the physical network, the daemon network, and the logical network, as illustrated on figure 1. The physical network is the underlying computational resource. The daemon network is a collection of processes, whose task is to manage Messengers, and interpret system commands. Examples of possible system commands are initiation of checkpointing, daemon failure notification, load balancing messages, and injection of a new Messenger. There is one daemon per physical node. The logical network is an application-specific computation network created at run time on top of the daemon network. Multiple logical network nodes may be created on the same daemon network node, and thus are running on the same physical node. Logical nodes may be interconnected by logical links into an arbitrary topology for the purposes of navigation.

Each link of the logical network has a name and several (optional) weights, which Messengers use for determining which links to route themselves along. Each logical node has a name and provides a memory space commonly accessed by all Messengers that gather on the node. This memory space, the Node Variable Area, functions as both a node-unique database and as an inter-Messengers communication channel.

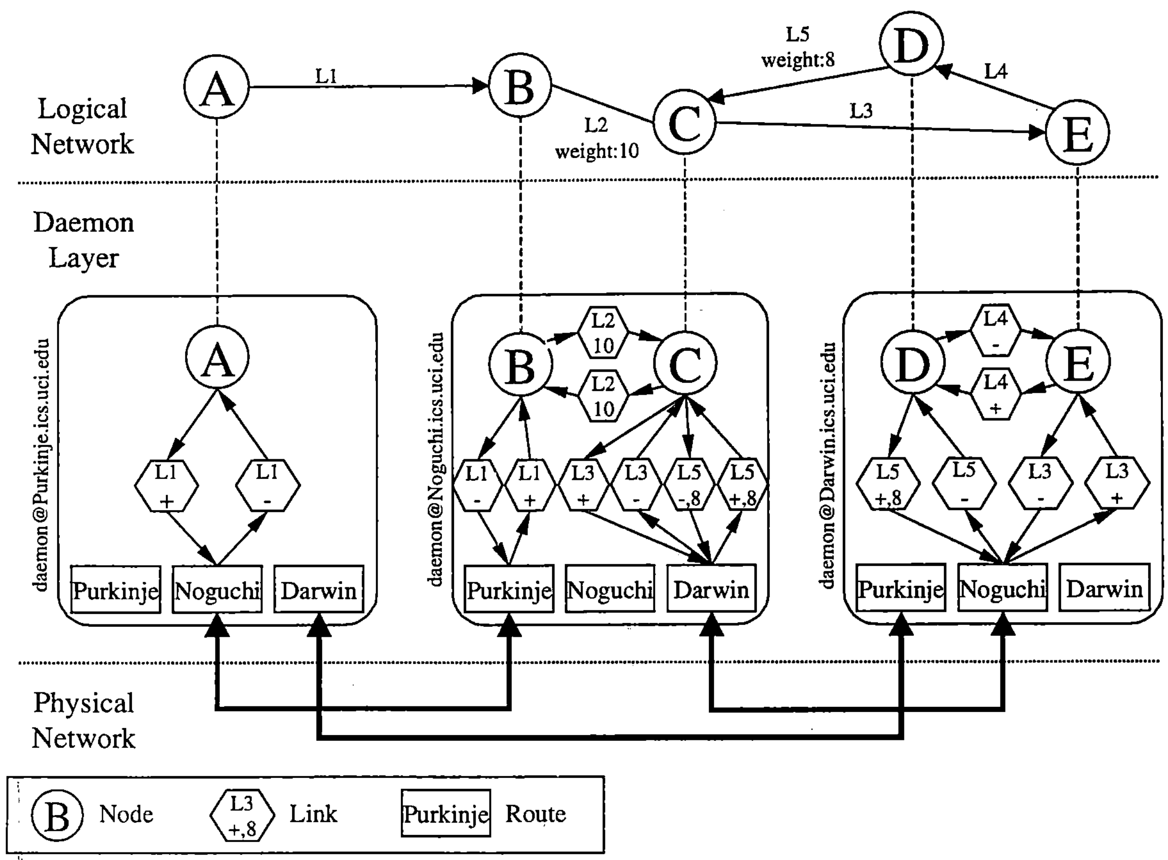

Figure 1

Each Messenger can access three types of variables: Messenger, node, and environmental variables. Messenger variables are local to and carried by the Messenger as it propagates through the logical network. Node variables are node-resident and are mapped to the Node Variable Area where the Messenger is currently running. There they are shared among Messengers running on the same node. Environmental variables provide information such as the current node name, and the name and weights of the last traversed link.

Messengers navigate through the logical network based on their own internal program and state. This is accomplished by explicit navigational statements, which also permit the creation or destruction of logical links and/or nodes. Messengers may also perform arbitrary computations in the nodes they visit. This can take two forms. First, Messenger internal program may contain computational statements, and second, Messengers may invoke ordinary C functions.

2.2 LOGICAL NETWORK

The logical network consists of logical nodes, connected by logical links as shown in figure 1. Each link, although presented as a single structure to the user, consists of two parts. One part of the link is located on the daemon, where the first node resides, and another part of the link resides on the daemon hosting the destination node. Each link has a daemon-unique id. Every link also knows the id of its counterpart, and the daemon address where the destination node resides. The link pointer is stored in two data structures. One data structure is accessible from the logical node. It is the list of pointers to all the links connected to the logical node. Another data structure is a tree, accessible from the daemon. This tree contains the pointers to all the links residing on the daemon, sorted by the link id.

These structures are used to accommodate Messenger transfer from one node to the other as follows. The Messenger finds the link it wants to traverse from the link list of its current logical node. From the link information the Messenger knows on what physical machine the destination node resides, and the id of the link on the destination daemon. The Messenger is transferred to ~~the~~ destination daemon, and it searches for the link using the link id in the daemon link tree. From the link information the Messenger acquires the pointer to the destination logical node.

arrows indicate the direction in which Messengers can traverse the logical link.
Notation:

Node – migrating logical node.
Old Daemon – an old host of Node
New Daemon – a new host of Node
Connected Daemon – a host of logical nodes connected to Node by logical links.

1. Old Daemon: send *TransferNode* message to the New Daemon. This message includes the Node structure, node variables, links connected to Node, and all Messengers residing on the Node. After this step Messengers that arrive at the Node are not executed, but stored in the node queue instead. (Figure 3b).
2. New Daemon: send *UpdateLink* message to all Connected Daemons. This message contains information needed to update the information of connected links, such as new destination link id and destination link daemon. All Messengers arriving at the New Daemon are not executed, but stored in the node queue until the completion of the migration. (Figure 3c).
3. Connected Daemons: when daemon receives *UpdateLink* message, it redirects the appropriate links and sends *LinksUpdated* message to the Old Daemon. (Figure 3c)
4. Old Daemon: after all expected *LinksUpdated* messages are received, Old Daemon sends *ActivateNode* message to the New Daemon. This message contains the list of Messengers that arrived to the Node after step 1. Old Daemon destroys its copy of Node. (Figure 3d).
5. New Daemon: after receiving *ActivateNode* message, New Daemon attaches its node queue to the node queue received from the Old daemon, preserving FIFO order, then it activates all the Messengers on the New Node. (Figure 3d).

3.2 CONCURRENT ALGORITHM

Depending on the strategy used to decide when to move logical nodes, multiple nodes can be moved at the same time. In the situation when two nodes, connected by logical link are migrating simultaneously, the algorithm presented in section 3.1 may not work correctly.

Consider the example in figure 4. Nodes "a" and "b" are connected by a logical link. They are simultaneously being migrated (figure 4a). Node "a1" is created first, and it sends *UpdateLink* message to node "b0" (figure 4b). "b0" updates the link and sends *LinksUpdated* message to "a0" (figure 4c). Node "a0" is deleted from its Old Daemon. At the same time "b1" is created, and it sends an *UpdateLink* message to node "a0", but there is no record of such a node (figure 4d).

The algorithm has to be adjusted so that the node on the Old Daemon is not deleted before any *UpdateLink* message that may be coming to it. To solve this problem, in the *LinksUpdated* message, the Connected Daemon also includes the address of any local node that is being moved. This will prevent the first daemon to complete step 4 of the algorithm before receiving the *UpdateLinks* message from the second daemon.

3.3 HANDLING LOGICAL NETWORK MODIFICATIONS

Section 2.3 described how Messengers can modify the logical network. The node migration algorithm needs to be adjusted for these operations. To lessen the number of possible interleaving combinations between the node migration protocol and protocols that modify the logical network, the node that initiates the modification of the logical network can not migrate before the network modification protocol is completed. This means that node "a" in figure 2 can not migrate during modification of the link, (only node "b" can).

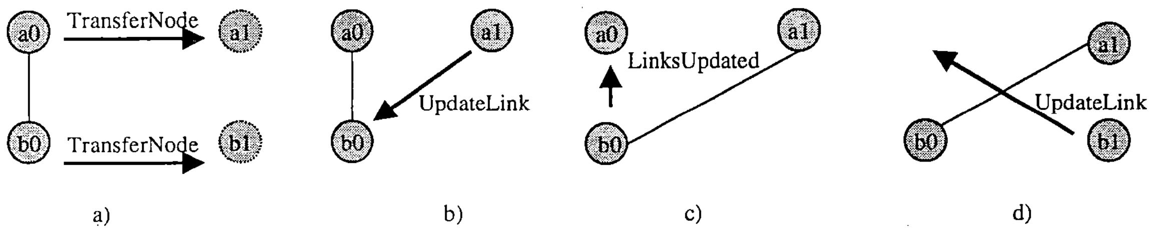

Figure 4

This algorithm avoids loosing or duplicating Messengers. Before step 3 Messengers that travel to Node will go to the Old Daemon. After step 3 these Messengers will go to the New Daemon. Since Connected Daemons send *LinksUpdated* messages to the Old Daemon, if the system uses FIFO channels, receiving *LinksUpdated* message guarantees that there are no more Messengers traveling from that daemon to the Node. After *LinksUpdated* message is received from every Connected Daemon, the node on the Old Daemon can be deleted.

When a link is created to the node "b", first a *FindNode* message is broacasted to all daemons. Both Old Daemon and New Daemon register this message. Old Daemon replies, and does not complete step 4 of the node migration protocol until the Messenger that creates the link arrives. Then, in step 4, this Messenger, together with others, and together with the list of *FindNode* messages of the Old Daemon is transferred to the New Daemon. At the New Daemon the Messenger is processed and a new link is created.

There is a short period between steps 4 and 5 when the Node at the Old Daemon does not exists, and the Node at the New Daemon is not activated yet. If the *FindNode* message arrives in this interval, it will be ignored by the Old Daemon. When *ActivateNode* message arrives at the New Daemon, the attached list of *FindNode* messages is compared with the local list, and if this contains extra messages, the corresponding replies are sent to the senders of these messages.

Another situation that has to be considered during creation of the link to the existing node is when node "b" replies to the *CreateLink* message, specifying the id of the local link, and *then* starts migrating. Then a new node "b" is created, and a message is sent to the node "a" (figure 2) updating the link ids. If the *CreateLink* message from the old host of "b" comes *after* the *UpdateLink* message sent from the new host of "b", it should be discarded.

When a Messenger that deletes or modifies a link arrives to the Old Daemon, it is transferred to the New Daemon with other Messengers and executed there. So these modifications of the logical network do not affect node migration.

4. CONCLUSION

Load balancing based on the migration of logical nodes provides an application-independent approach to load balancing. The decision on when and what to move can aim at balancing the amount of computation per processor, at minimizing interprocessor communications, or both. The parameters that could be considered to decide on node migration include the number of logical nodes per daemon, the number of Messengers per daemon, the number of remote logical links, and traffic on each logical link. This information is collected independently of the user application, allowing reuse of the same load balancing strategies for different applications.

Such a load balancing mechanism was proposed by Brunner and Kale in [3]. The authors implement and evaluate several policies that can trigger migration in the Charm++ system [11]. Even though the Charm++ system is based on what they refer to as data-driven objects, while MESSENGERS is based on autonomous agents operating in the logical network, the same load balancing strategies can be used to balance both systems. A load balancing algorithm for dynamic grid applications was presented in [12].

The Charm++ system, however, uses a different object migration mechanism, presented in [1]. In Charm++ each processor has a map of locations of every object participating in the application. As objects migrate, these maps become out of date. The solution is to forward messages to each element's (object's) original host processor. Since each processor is kept informed of the location of each element it originally hosted, it can forward the message to its new destination.

A description of the thread migration mechanism was presented in [2]. Their strategy for migration of computation threads is similar to the one presented here. In their strategy it is the responsibility of the connected threads to establish the connection with the new thread. To do this each connected thread (corresponding to a neighboring node in MESSENGERS implementation) spawns a child thread to establish that connection whenever a new thread (corresponding to a new node) is created. By allowing the new node to establish the connections with the neighboring nodes (steps 2 and 3 of the algorithm) the need for such a child thread is eliminated. The system in [2] also does not consider the situation when thread migration is happening while connections are created, deleted or modified. Our approach, as described in this paper, permits migration in a dynamically changing network.

5. REFERENCES

[1] Robert K. Brunner and Laxmikant V. Kale. Adapting to Load on Workstation Clusters. The Seventh Symposium on the Frontiers of Massively Parallel Computation.

[2] A. Haidt, P. Stellmann. Novel Migration Mechanism for Load Balancing of Parallel Applications. Mannheim SuParCup, 1999.

[3] Robert K. Brunner and Laxmikant V. Kale. Handling Application-Induced Load Imbalance using Parallel Objects. Publication Information Not Available. http://charm.cs.uiuc.edu/.

[4] L. F. Bic, M. Fukuda, M. B. Dillencourt, F. Merchant. MESSENGERS: Distributed Programming Using Mobile Autonomous Objects. Journal of Information Sciences, 1998.

[5] Hairong Kuang, Lubomir F. Bic, Michael B. Dillencourt. Paradigm-Oriented Distributed Computing Using Mobile Agents. Technical Report No99-38, Department of Computer Science, University of California, Irvine.

[6] Eugene Gendelman, Archana Mulay. Bio-Net Simulator" http://www.ics.uci.edu/~egendelm/prof/publications.html.

[7] Susan L. Mabry, Lubomir F. Bic, Kenneth M. Baldwin. CVSys: A Coordination Framework for Dynamic and Fully Distributed Cardiovascular Modeling and Simulation. Int'l Biomedical Optics Symposium (BIOS'98), special section on Biomedical Sensing, Imaging and Tracking Technologies, San Jose, CA, Jan. 1998.

[8] Christian Wicke, Lubomir F. Bic, Michael B. Dillencourt, Munehiro Fukuda. Automatic State Capture of Self-Migrating Computations in Messengers. ICSE98 Int'l Workshop on Computing and Communication in the Presence of Mobility, Kyoto, Japan, April 1998. http://www.ics.uci.edu/~bic/messengers/messengers.html.

[9] Christian Wicke, Lubomir F. Bic, Michael B. Dillencourt. Compiling for Fast State Capture of Mobile Agents. Parallel Computing 99 (ParCo99). TU Delft, The Netherlands. August 1999.

[10] Eugene Gendelman, Lubomir F. Bic, Michael B. Dillenourt. An Application-Transparent, Platform-Independent Approach to Rollback-Recovery for Mobile-Agent Systems. In submission. http://www.ics.uci.edu/~egendelm/prof/publications.html.

[11] L. V. Kale, S. Krishnan. Charm++: Parallel Programming with Message-Driven Objects. In G. V. Wilson and P. Lu, editors, Parallel Programming using C++, pages 175-213. MIT Press, 1996.

[12] Andreas D. Haidt. Dynamic Load Balancing with Self-Organizing Maps, Parallel Computing '99 (ParCo99), Delft, The Netherlands, Aug. 1999.

[13] Introduction to the Odyssey API. http://www.genmagic.com/technology/odyssey.html.

[14] Bill Venners. Under the Hood: The architecture of aglets. JavaWorld, April 1997 http://www.javaworld.com/javaworld/jw-04-1997/jw-04-hood.html.

[15] H. Peine and T. Stolpmann. The Architecture of the Ara Platform for Mobile Agents. In Kurt Rothermel, Radu Popescu-Zeletin, editors, Proc. of the First International Workshop on Mobile Agents MA'97. (Berlin, Germany), April 1997. http://www.uni-kl.de/AGNehmer/Projekte/Ara/index_e.html.

[16] Robert Gray, George Cybenko, David Kotz, and Daniela Rus. Agent Tcl. In William Cockayne and Michael Zypa, editors, Itinerant Agents: Explanations and Examples with CDROM. Manning Publishing, 1997. ftp://ftp.cs.dartmouth.edu/pub/kotz/papers/gray:bookchap.ps.Z.

[17] Johansen, D., van Renesse, R. and Schneider, F. B. An Introduction to the TACOMA Distributed System, Technical Report 95-23, Dept. of Computer Science, University of Tromco, Norway, 1995. http://www.cs.uit.no/Localt/Rapporter/Reports/9523.html.