PALATIN: A Platform for Interactive Algorithms

Christian Scheideler
Department of Computer Science
Johns Hopkins University
3400 N. Charles Street
Baltimore, MD 21218, USA
Email: scheideler@cs.jhu.edu

Andreas Terzis
Department of Computer Science
Johns Hopkins University
3400 N. Charles Street
Baltimore, MD 21218, USA
Email: terzis@cs.jhu.edu

***Abstract*— We outline the design of a platform, called PALATIN, that allows easy development and reliable and efficient execution of concurrent algorithms in a distributed, potentially unreliable environment. The platform consists of three parts: a basic philosophy of how to write code for concurrent data structures and algorithms, a C++ library called Spheres supporting this philosophy, and the design principles of the PALATIN platform itself that allows programs written in the Spheres environment to interact with each other across multiple sites. This is done by organizing the PALATIN platforms of these sites into a peer-to-peer system allowing programs to find and communicate with each other via remote method invocations.**

I. Introduction

This paper outlines the design of a platform for interactive algorithms, called PALATIN. By "interactive algorithms" we mean algorithms that form logical structures spanning multiple sites that allow sites to interact with each other. Typical Internet applications for interactive algorithms are information sharing, distributed computing, or instant messaging. The goal of PALATIN is to provide a decentralized infrastructure that allows multiple interactive applications to be run on top of it in an efficient and secure way.

The first wave of interactive applications was implemented following the client-server paradigm. Today, we are witnessing a second wave of applications built on a different paradigm often dubbed "peer-to-peer". Peer-to-peer computing, which has become extremely popular in the recent years, is the sharing of computer resources and services by direct exchange between client systems. It is anticipated that many services will emerge in the future that are based on the peer-to-peer paradigm, and we see PALATIN as a first universal and structured approach towards providing a scalable platform for these services.

II. Environments for distributed programs

Writing distributed programs is a challenging task. Many researchers have tried to overcome this problem with the help of a suitable programming environment, using message passing in the 1970s [1], [5], [10], remote procedure calls in the 1980s [4], [9], [17], and distributed shared memory (DSM) in the 1990s [2], [3], [8], [12]. Still, the percentage of distributed applications based on these environments is very small. One possible explanation for this is that each round is an evolutionary stage for the distributed programming paradigm, and each attempt brings us closer to a methodology that will ultimately be easy to use and efficient. A less optimistic explanation of the failure of each attempt is that no matter what methodology people will come up with, ease of use and efficiency are two concepts that cannot be achieved at the same time. We believe that this is in fact true for the direction in which the design of universal platforms for distributed programs is currently heading. The basic problem is that DSM is the *wrong* approach for providing a universal platform for distributed programs. This belief is also shared by the JXTA designers [16], which is one of the reasons why they designed JXTA. Although the DSM approach tremendously simplifies the development of distributed applications, it suffers from the inherent problem that a universally applicable distributed shared memory cannot be implemented efficiently. But even if it could, the mere attempt of hiding networking issues may create the illusion to the programmer that there is a single natural design for a given application, regardless of the context in which that application will be deployed. This assumption can lead to the design of parallel programs that perform poorly in a distributed environment by using, for example, a parallelism that is too fine-grain to hide the network latency.

So if a distributed shared memory is not the right way to go to provide a universally applicable interface for efficient distributed programs, what is the right way?

We believe the problem with the DSM approach is that its level is too high. A platform that wants to be universally applicable and efficient *must* provide a much lower level of functionality. This level of functionality should be high enough to hide low-level networking issues, but it should be low enough that it allows to write efficient code for whatever middleware is necessary to support a particular class of distributed applications. If this platform is easy enough to use so that middleware can easily be developed on top of it, then people can just write their own middleware to support, for example, a particular class of distributed computing applications, or a class of multimedia streaming applications. Middleware implementations can then be shared among people like C++ libraries so that a middleware only has to be implemented once. In fact, one of our future plans for PALATIN will be that we will build a publicly available repository of various middleware implementations for PALATIN so that people can then easily write code for their particular applications.

III. Our approach

A. Use a purely object-oriented approach

Our basic approach to enable interaction between sites is purely object oriented. Objects will be responsible for maintaining a logical structure between the sites as well as managing shared data. That is, we require all shared data to be encapsulated in objects. To access data encapsulated in an object, a method in this object must be called, i.e. all object variables are private. Forcing all accesses to an object's data to go through one of its methods simplifies concurrent access to the data and helps structure the program in a modular way.

B. Do not use shared memory addressing

An object is identified by its physical location, which may consist of the IP address of the machine holding it and the memory location within that machine. Using this representation, we avoid the indirection created by using a virtual shared memory location, which is expensive to resolve in a scalable way. Also, we avoid the danger of two sites allocating the same shared memory location. Furthermore, the programmer has an influence on where data is located, avoiding the mistake that the location and management of the objects is handled automatically by the platform.

C. Instead, connect objects by an explicit link structure

Every object has a global identifier (ID) that represents the physical location of an object, which we plan to be encoded in an unsigned integer of (at least) 128 bits so that the physical location is not directly accessible to the programmer. This prevents the programmer from bypassing PALATIN for establishing connections to other remote objects. Instead, PALATIN requires the programmer to use this ID to establish platform-aware links to objects. Once such a link is established, an object can call methods of the other object. Enforcing platform-aware links allows PALATIN to provide availability services for the link structure and therefore to notify objects if certain links do not work any more, which relieves the programmer from periodically checking the liveness of a link by him/herself. It also allows PALATIN to notify an object when another object tries to establish a link to it, which is useful for security purposes.

The explicit link approach can be used to organize objects in an object structure suitable for the particular application. When adding suitable methods to the objects, any object can be a handle into the object structure so that any piece of information in that structure is accessible to any process through any of its objects, like this is the case for DHT-based peer-to-peer systems.

D. Allow objects to find each other

If there is no virtual shared memory, then it may appear to be hard for objects generated at different machines to find each other so that they can organize in a link structure. Here, our solution is relatively simple. There is a space of group IDs (or GIDs, represented by unsigned integers of at least 128 bits) under which objects can register. PALATIN organizes all objects that have been registered so far in a searchable data structure. Thus, as soon as an object has registered for a particular GID, it can be found by other objects by requesting from PALATIN the ID of any object that has registered under a particular GID.

E. Distinguish between local and remote method invocations

In our view, trying to enforce a unified object view as done in several distributed shared memory systems, i.e. to make local and remote objects indistinguishable, is a big mistake. Thus, in our approach we make sure that local and remote objects as well as local and remote method invocations are clearly distinguishable from each other. This allows the programmer to have full control on whether an object should be local or remote, and whether a call should be local or remote.

F. Only allow non-blocking method invocations

Conventional techniques for implementing access to concurrent objects typically rely on critical sections, ensuring that only one process at a time can operate on the object. However, critical sections are poorly suited for asynchronous systems: if one process is halted or delayed in a critical section, other, non-faulty processes will be unable to progress. By contrast, a concurrent object implementation is *non-blocking* if it always guarantees that some process will complete an operation in a finite number of steps, and it is *wait-free* if it guarantees that *each* process will complete an operation in a finite number of steps.

To enforce non-blocking and wait-free object implementations, we require that the following rules be fulfilled:

- Every remote method invocation must be non-blocking, which implies that remote method invocations are not allowed to have return values. If a return value is needed, then this has to be done in a way that the destination object of a call invokes a new method in the source object with its reply as a parameter.
- Every method in an object must be guaranteed to terminate. That is, while loops or other constructs waiting for an event to occur before terminating are *not* allowed in methods. If a while loop is needed, then this has to be realized by repeating invocations of a special method representing the body of the while loop.

G. Use strictly sequential execution inside objects

It is very difficult to design correct programs for objects in which methods within an object are allowed to run concurrently. To avoid these problems, our approach requires that every object can only run one method at a time, i.e. method invocations are executed in a strictly sequential way. This does not cause any problems with guaranteeing a non-blocking and wait-free object implementation because methods are required to guarantee termination. So every method invocation will eventually be executed.

H. Build on a well-accepted programming language

We believe that a distributed programming environment will only be successful if it builds on a well-accepted language that allows to write efficient programs. Since our distributed programming approach requires object-oriented programming, we chose the C++-programming language as our basic language platform. Certainly, Java is more secure than C++, but Java is slow. Also, security risks with C++ are limited because we do not allow PALATIN to move objects itself. Thus, only applications on top of PALATIN can move objects, and movements can only be realized through remote method invocations, limiting the ability of a site to introduce malicious code at another site.

I. Use as few new constructs as possible

The famous British physician Stephen Hawking once said that in order to successfully write scientific literature accessible to a broad audience, one has to keep in mind that every formula cuts the number of readers in half. We believe that the same is also true for any new programming environment offered to users. In this context, Hawking's recommendation can be rephrased as one should use as few new language constructs as possible. If the new programming environment is significantly different from existing environments (e.g. C++) early adopters will face a steep learning curve and adoption will be hindered. For example, it has been suggested that one of the reasons why Java has been successful is that it was very similar to C++ making it easy for programmers to learn it and quickly start programming in it. We follow the same principle in our research.

J. Spheres and PALATIN

Taking all of the requirements above into account, we intend to design a distributed programming platform that consists of two components: Spheres and PALATIN. The Spheres environment is an extension of C++ that allows programmers to implement distributed algorithms. C++ programs written in Spheres handle the remote information exchange via the PALATIN platform. The task of the PALATIN platform is to organize together with other PALATIN platforms in a decentralized system providing lookup service for registered objects so that objects at different machines can find each other. Also, PALATIN checks the state of remote objects as well as the connections between objects.

IV. The Spheres programming environment

The Spheres programming environment is based on C++ and requires only a few reserved data types and methods.

A. Reserved data types

Spheres uses three reserved data types:

- **Sphere**: this is the base class for all remote objects. That is, any definition of a remote class must have the form class $\langle$Name$\rangle$: public Spheres $\{\ldots\}$;.
- **SphereID**: is an unsigned 128-bit integer containing information about the physical location of a remote object. This information is encrypted so that it is not accessible to the programmer, but it can be used to establish links to other objects through the SphereLink object. Whenever a new Sphere object is created, an automatic SphereID is created for this object that is accessible via thisID.
- **SphereLink**: objects of this class provide links to other remote objects. More details on how this works are given below.

B. The Sphere class

The Sphere class provides several methods that allow remote objects to get in contact with each other.

- **void Register(uint128 GID)**: this registers the object calling the operation under GID so that other remote objects can find it (uint128: unsigned 128-bit integer).
- **void Unregister(uint128 GID)**: this removes the object from the registry for GID.
- **SphereID Any(uint128 GID)**: returns the SphereID for some remote object that has registered under GID.

Every created Sphere object automatically registers locally under PALATIN so that PALATIN can keep track of the currently available objects.

C. The SphereLink class

The SphereLink class uses the following methods to control links between remote objects:

- **void Open(SphereID ID)**: this opens a link to the remote object identified by ID. Alternatively, a link is also opened when creating the SphereLink object by calling "$\ell =$ **new** SphereLink(ID)".
- **void Close()**: this closes the link. The close method will automatically be invoked when calling "**delete**(ℓ)" for some SphereLink$*$ ℓ.
- **void Call($\langle$class$\rangle$::$\langle$method$\rangle$, void* p)**: this calls **method** of the remote object of type **class** reachable via the given link with parameter list p.

D. Starting the Spheres environment

In order to start the Spheres environment, the "RunSpheres()" command has to be executed.

E. Exceptions

Several exceptions can be created by PALATIN that can be caught by objects, such as

- LINK_FAILURE: a link to a Sphere object is broken
- LINK_REQUEST: another object wants to establish a link to that object

These exceptions are accompanied with further information so that an object can react to them appropriately. More details and example programs are available at `http://www.cs.jhu.edu/~scheideler/courses/600.348_F03`.

V. The PALATIN platform

Besides providing services to the Spheres programs, PALATIN platforms have to be administered and organized in a scalable overlay network.

A. Administration

PALATIN will provide an interface that allows the user of the machine to set the maximum bandwidth and the amount of memory PALATIN is allowed to use, so that its use of resources is under the control of the user.

B. Overlay network

The PALATIN overlay network offers the following operations:

- $p.\text{JOIN}(q)$: a new platform q joins the PALATIN network by invoking JOIN in platform p already in the system.
- $p.\text{LEAVE}()$: platform p leaves the network.

When a PALATIN platform p is started at some computer, it will contact a PALATIN server to get in contact with another platform q that is already part of the PALATIN network. p then calls $q.\text{JOIN}(p)$ at q. Also, upon being started, the PALATIN platform provides a server socket that allows local Spheres programs to interact with it. PALATIN is planned to use fair queueing to treat multiple Spheres programs in a fair way when handling remote method invocations. We also plan to extend the two operations with further operations that allow to adapt to a changing bandwidth of a node.

C. Group registry

Registry operations in the Spheres environment will be handled with the following operations in PALATIN:

- p.INSERT(ID, GID): this adds the identifier ID to the group with identifier GID.
- p.DELETE(ID, GID): this removes ID from the group with identifier GID.
- p.SEARCH(GID): platform p searches for any object in the PALATIN network that is currently registered under GID. The return value is of type SphereID.

We plan to extend these operations with further operations that allow to react to changing storage capacities in the nodes.

D. Status check

Since all objects created (resp. deleted) by a Spheres program and all links established (resp. removed) to other objects will be locally registered (resp. deregistered) at the PALATIN platform of the machine, the platform always has a consistent view of the current objects and connections. This allows platforms to continuously check whether connections (resp. the objects representing the endpoints of the connections) are still working. If not, an exception is created at the object representing the origin of the connection. Also, it allows platforms to check whether the registry information is up-to-date. If an object that has registered is not available any more, it is removed from the registry.

VI. Related work

There is a vast amount of projects on platforms for parallel computing. See `http://www.computer.org/parascope/` or `http://www.aspenleaf.com/distributed/` for a comprehensive list.

Several systems for peer-to-peer applications have already been developed in recent years. Among them are Chord [15], CAN [13], Pastry [14], and Tapestry [18] as well as more open architectures such as PeerWare [6], OpenHash [11], and KBR [7]. The closest to our approach is probably KBR (or Key-Based Routing). However, KBR does not appear to have services for the registration of processes or objects or for maintaining logical connections between these. Thus, although the KBR API could potentially be used in a universal way, it would reduce to a mere message passing system that is harder to use than our approach.

Also outside of the research community, middleware and middleware platforms have been developed for peer-to-peer systems, such as JXTA. PALATIN compared to the JXTA core can be seen as Chord compared to Gnutella. Overlay maintenance, routing, and the management of group IDs is rather ad hoc in JXTA whereas our goal for PALATIN is to develop a platform based on formally analyzed methods.

References

[1] *Message Passing Interface Forum*. `http://www.mpi-forum.org`.

[2] *The Orca Parallel Programming Language*. See `http://www.cs.vu.nl/orca/`.

[3] *Split-C*. See `http://www.cs.berkeley.edu/projects/parallel/castle/split-c/`.

[4] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2, 1978.

[5] R. Cook. MOD - A language for distributed processing. In *Proc. of the 1st International Conference on Distributed Computing Systems*, 1979.

[6] G. Cugola and G. Picco. PeerWare: Core middleware support for peer-to-peer and mobile systems. Technical report, Politecnico di Milano, May 2001.

[7] F. Dabek, B. Zhao, P. Druschel, J. Kubiatowicz, and I. Stoica. Towards a common api for structured peer-to-peer overlays. In *Proceedings of the 3rd IPTPS*, 2003.

[8] T. O. M. Group. *Common Object Request Broker: Architecture and Specification*, omg document number 91.12.1 edition, 1991.

[9] N. C. Hutchinson, L. L. Peterson, M. B. Abott, and S. O'Malley. RPC in the x-Kernel: Evaluating new design techniques. In *Proc. of the 12th Symposium on Operating Systems Principles (SOSP)*, 1989.

[10] M. Jaayeri, C. Ghezzi, D. Hoffman, D. Middleton, and M. Smotherman. CSP/80: A language for communicating sequential processes. In *Proc. of the Distributed Computing CompCon*, 1980.

[11] B. Karp, S. Ratnasamy, S. Rhea, and S. Shenker. Spurring adoption of DHTs with OpenHash, a public DHT service. Technical report, submitted for publication, Carnegie Mellon University, 2003.

[12] G. D. Parrington, S. K. Shrivastava, S. M. Wheater, and M. C. Little. The design and implementation of Arjuna. *Computing Systems*, 8(2):255–308, 1995.

[13] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *SIGCOMM '01*, 2001.

[14] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, 2001.

[15] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. In *SIGCOMM '01*, pages 149–160, 2001.

[16] J. Waldo, G. Wyant, A. Wollrath, and S. Kendall. A note on distributed computing. Technical report SMLI TR-94-29, Sun Microsystems Laboratories, November 1994.

[17] L. Zahn, T. Dineen, P. Leach, E. Martin, N. Mishkin, J. Pato, and G. Wyant. *Network Computing Architecture*. Prentice Hall, 1990.

[18] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz. Tapestry: A global-scale overlay for rapid service deployment. *IEEE Journal on Selected Areas in Communications*, 2003. Special Issue on Service Overlay Networks.