Functional vs Object-Oriented Distributed Languages*

R.F. Pointon[1], S. Priebe[2], H-W. Loidl[1], R. Loogen[2], and P.W. Trinder[1]

[1] Department of Computing and Electrical Engineering,
Heriot-Watt University, Scotland;
{rpointon,hwloidl,trinder}@cee.hw.ac.uk

[2] Department of Computer Science,
Marburg University, Germany;
{priebe,loogen}@informatik.uni-marburg.de

1 Introduction

Improved network technology and a speedily growing infrastructure of interconnected computer systems have generated a resurgence of interest in distributed languages. Java is a good example of modern object-oriented distributed language design, with automatic dynamic memory management, strong types to enhance safety and inheritance to facilitate code reuse.

Theoretically non-strict functional languages offer additional benefits for constructing distributed systems. These languages have sophisticated, e.g. polymorphic, type systems to further enhance the safety of distributed systems. They also have a highly-dynamic model of distribution with data transfer on-demand, a relatively high degree of distribution transparency with the runtime-system managing details of the data transfer, and the potential to safely abstract over distribution-control primitives with higher-order polymorphic functions.

These benefits of distributed functional languages are bought at the price of an elaborate implementation. Although there have been many distributed functional language designs, few have robust long-lived implementations, although notable exceptions include Erlang [Cla94] and Haskell with Ports [HN00]. The authors represent groups who have implemented two non-strict distributed functional languages based on Haskell [PH$^+$97]: Eden [BLOP97] and Glasgow distributed Haskell (GdH) [PTL00]. The full paper evaluates the distributed language features of Eden and GdH in comparison to Java using two case studies: a factory chatroom and a multiuser game.

2 The Distributed Languages

Java [DSMS98] supports distribution via the Remote Method Invocation (RMI) API, which allows Java programs to invoke methods of remote objects.

* Supported by research grant GR/M 55633 from EPSRC, British Council/DAAD travel grant (Project No. 1097), and APART fellowship 624 of the Austrian Academy of Sciences.

Dynamic class loading allows to load code on-demand, in particular, the code can be loaded across the internet, facilitating client/server style distribution.

GdH [PTL00] supports both parallel and distributed computation using two classes of threads: purely-functional threads and side-effecting I/O threads. Thread creation is explicit: a remote fork primitive creates an I/O thread on a named processing element (PE). Communication and synchronisation are implicit: threads on one PE can share variables with threads on other PEs. Data transfer is lazy: only the immediately required portion of a data structure is transferred when required by another thread. I/O threads can explicitly communicate and synchronise using polymorphic semaphores (`MVars`) and abstractions built on top of them, such as channels and buffers. Thread location is exposed to the program level so that a program can use resources unique to a PE.

Eden [BLOP97] uses explicit processes together with implicit communication via stream-based channels. A pre-defined process `merge` allows many-to-one communication. Lean communication is achieved by an eager sending protocol and communication connections can be built up flexibly by using dynamic reply channels. The memory is distributed: there is no global sharing between PEs but only local sharing within one PE. Complex process nets can be constructed to realize transformational and reactive systems.

3 Factory Chatroom

A distributed chatroom enables many users, and one privileged superuser, in different locations to cooperate in solving factory scheduling problems using a simulator [Kit92]. The Factory Chatroom is distributed with multiple clients connecting to a central server, which maintains central configuration information. Figure 1 is an interaction diagram showing the communication between clients and the server. We have investigated two different implementations of the system, depicted in Figure 2: one with server represented as shared synchronised state; and one with a server as a process managing its internal state and operating on a queue of messages from the clients.

The Java implementation [Wil97] uses shared synchronised state to maintain the list of users and simulation state. Several client applets connect via RMI to operate on the server state and within the server information is passed back to the clients using RMI to the client-side message queues.

The GdH implementation also uses shared synchronised state, protecting the server state with an `MVar` on one PE to ensure mutual-exclusion. Clients on other PEs use remote fork to manipulate the server's state. In contrast to Java the GdH implementation has to instantiate the exact number of clients when the program starts.

In Eden the server is implemented as a process. The client processes are connected via streams over which the chat protocol transactions are communicated. In contrast to GdH, communication is implicit (reducing code size) and independent of the central I/O monad. For embedding side-effects, such as setting up the frontend to the user, into the computations the low-level primitive `unsafePerformIO` is used to set up a new state thread.

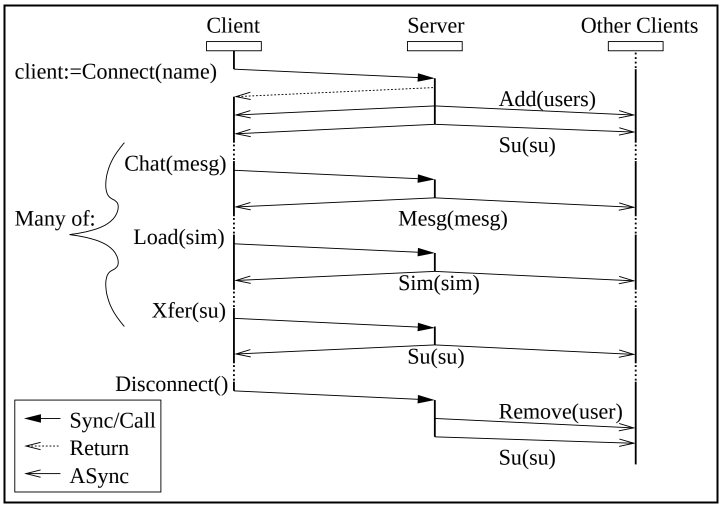

Fig. 1. Communication Overview of the Factory Chatroom.

Comparing the three implementations, the amount of code for the communication is similar: seven remote calls. This is unsurprising given the equivalence of RMI to remote fork. Furthermore, the additional calls to connect to the server in Java directly correspond to explicit calls for client instantiation at program startup in GdH and Eden.

4 Multiuser Geographical Game

We are constructing a multiuser geographical game in GdH, that allows users to navigate through a virtual world and to interact with other users. This game is an example of a class of distributed applications, including multiuser distance learning environments, games, and map or architecture exploration.

In essence the game is very similar to the classic game of *battleships* where in the first phase each player places their troops and in the second phase players take turns searching for their opponent's troops. Multiple players are allowed, each interacting through a GUI client on their PE. In the implementation a client explicitly communicats with a central server where the game state is coordinated. All players must make a move within a given time period, introducing soft realtime limits on the application.

As players place troops and search, they interact with a world map. The data model itself is based on standard multi-layered Geographic Information Systems (GIS) maps [BM98]. Several layers, representing different aspects of the same geographic area, are used in an overlayed fashion to represent the world. The data is stored in a spatial data structure, a quad tree [Sam90], and therefore the data transferred to a players machine reflects their interest in specific layers and regions. In the implementation this data is distributed amongst the PEs to

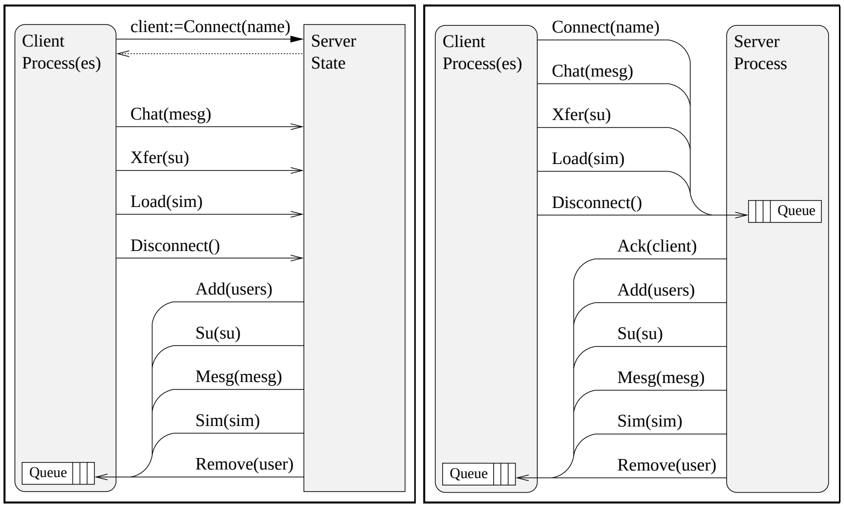

Fig. 2. A: Shared Synchronised State. B: Server as a Process.

avoid the bottleneck of centralised data and to improves scalability. The communication necessary to access the data is implicit, i.e. information is transfered on demand as the GUI renders a particular view of the world.

5 Discussion

We discuss GdH and Eden, two of only a few implemented distributed functional languages, and evaluate the usefulness of their distinctive language features on two distributed applications, one already implemented in both languages and another one under construction. For simple distributed programs with explicit communication, functional and object-oriented programming models are remarkably similar, although communication in Eden is less explicit. More sophisticated distributed programs benefit from advanced features of functional languages such as implicitly-shared data and lazy dynamic data transfer between PEs.

References

[BLOP97] S. Breitinger, R. Loogen, Y. Ortega-Mallén, and R. Peña. The Eden Coordination Model for Distributed Memory Systems. In *Workshop on High-level Parallel Programming Models (HIPS'97)*, pages 120–124. IEEE Computer Science Press, Geneva, Switzerland, April 1997. URL: http://www.mathematik.uni-marburg.de/inf/eden/.

[BM98] P. Burrough and R. McDonnell. *Principles of Geographical Information Systems*. Oxford University Press, 1998.

[Cla94] Claes Wikstrom. Distributed Programming in Erlang. In *PASCO'94 - First International Symposium on Parallel Symbolic Computation*, Linz, September 1994.

[DSMS98] M.C. Daconta, A. Saganich, E. Monk, and M. Snyder. *Java 1.2 and JavaScript for C and C++ Programmers.* John Wiley & Sons, New York, 1998.

[HN00] F. Huch and U. Norbisrath. Distributed Programming in Haskell with Ports. In *Draft IFL'00 - 12th International Workshop on the Implementation of Functional Languages*, Aachen, Germany, September 2000.

[Kit92] P. Kiteck. Analysis of Component Interaction in a Distribution Facility using Simulation. In *EUROSIM*, 1992.

[PH+97] J. Peterson, K. Hammond, et al. *Report on the Programming Language Haskell (Version 1.4)*, April 1997. URL: http://www.haskell.org/.

[PTL00] R.F. Pointon, P.W. Trinder, and H-W. Loidl. The Design and Implementation of Glasgow distributed Haskell. In *Draft IFL'00 - 12th International Workshop on the Implementation of Functional Languages*, Aachen, Germany, September 2000. URL: http://www.cee.hw.ac.uk/~dsg/gdh/.

[Sam90] H. Samet. *Applications of Spatial Data Structures.* Addison-Wesley, 1990.

[Wil97] P.A. Wilcox. Object-Oriented Factory Simulation in Java. Master's thesis, Department of Computer Science, Edinburgh University, 1997.