15-744 Project Report

Carlo Angiuli   Michael Sullivan

December 12, 2011

1 Motivation

Modern network applications need to support many simultaneous clients—this is known as the C10K ("10,000 client") problem [1]. But to maintain these simultaneous connections, an application must track each client's state independently. A straightforward way to do this is to spawn a thread for each client, but this approach does not scale as the number of clients grows.

A lightweight approach is to use kernel mechanisms for multiplexed, non-blocking I/O, such as `select`, `poll`, `epoll`, or `kquery`; unfortunately, to maintain separate state for each client, application programmers must manually structure their code as state machines, which is tedious and error-prone. Unsurprisingly, there exist many thin C/C++ wrappers for these facilities, but none avoid major code refactoring.

To maintain the illusion of threads (in particular, client-specific state), languages like Erlang and Haskell provide lightweight threading via user-space schedulers and non-blocking IO. While this affords better performance, the full generality of this approach still imposes unnecessary overhead on networking applications which don't require preemptable, independently scheduled threads [2].

2 Proposal

Hence, we intend to produce an embedded domain-specific language in Haskell to describe concurrent network applications. This language will provide composable, high-level network operations, in which programmers may express their application logic naturally, as if using threads.

However, this high-level description will be automatically synthesized into a state machine representation usable in a C loop invoking `epoll`, using no threads whatsoever. We will additionally output a graphical representation of this underlying state machine.

The back-end of this system can be implemented via several different options which we are still evaluating. One option is to directly interface with the `epoll` hooks in the Haskell runtime system, bypassing the native lightweight threads. Another option is to generate C source code which itself implements the network application, optionally linkable with native C code.

3 Milestones

We will first collect a representative sample of Linux network applications and examine the range of high-level primitives required to support them. We will then decide on a specific back-end for our language, and design the front- and back-end in tandem. To evaluate the utility of this language, we will build a number of representative applications in our language (e.g., an echo server, simple HTTP server) and compare them to traditional implementations, with respect to both code simplicity and performance.

References

[1] Dan Kegel. The C10K problem. `http://www.kegel.com/c10k.html`, September 2006.

[2] Steve Vinoski. Process bottlenecks within Erlang web applications. *IEEE Internet Computing*, 15:86–89, March 2011.