Lisp as a Business Work Horse

Espen J. Vestre
Net Fonds ASA
Stenersgate 2
0184 Oslo, Norway
ev@netfonds.no

ABSTRACT

Net Fonds pioneered internet-based stock trading in Norway in 1997. Due to highly automatized processes, we are able to serve a large customer base with a small staff (11 people). We use Common Lisp for most of our in-house developed applications, which include both server applications which e.g. automate routing and matching of stock orders, and an end-user trading application with real time stock quote streaming. This paper gives a walk-through of our main Common Lisp applications and a few examples of technical solutions.

1. INTRODUCTION

Net Fonds was founded in 1997 and was the first Norwegian company to offer internet-based stock trading. We don't consider ourselves a stock broker in the traditional sense, since the customers do the trading almost entirely on their own, assisted by our computer systems. Most of these systems are written in Common Lisp and run on our many Linux servers.

The choice of Lisp as the main implementation language goes back to the startup of the company. Initially, the IT department consisted of one man: Emacs (gnus) developer Lars Magne Ingebrigtsen. So the natural choice of implementation language for the first in-house applications was Emacs Lisp. The web pages were done in PHP and some back office gui applications were written in Tcl.

As the customer base and the computing demands started to grow, the core database was moved to Oracle and the Emacs Lisp applications were gradually moved to Common Lisp (but one important application, the main interface for our brokers, is still in Emacs Lisp!).

The author joined Net Fonds in 2002, as the fourth member of the technical staff, all four doing both Lisp development and Linux/BSD system administrations of almost 100 servers.

2. SYSTEM OVERVIEW

A very coarse picture of our systems is given in figure 1 (the number of actual system components is much higher, but an accurate map would probably be almost unreadable).

2.1 Live Feed

An important part of our business is to provide our customers with "real time" stock quotes, making it possible for professional investors and traders to react very quickly to market changes: Several Lisp-based server applications connect to systems providing real time feeds from the different exchanges. They feed a dedicated Oracle database with live stock quotes, and they also, through two layers of TCP-based servers, feed PrimeTrader, an end user trading application. There are 4 different such feed servers, all written in Lisp.

2.2 End User Interfaces

The end user gets real time (or delayed, depending on his level of subscription) stock quotes through either conventional web interfaces or the PrimeTrader application. The web interfaces are implemented in PHP and communicate with the rest of the systems through the Oracle database. The trading application PrimeTrader is a Common Lisp application, developed with LispWorks with CAPI gui, and is offered in Windows, Linux and Mac OS X versions. It communicates with its own protocol, based on http, with its servers (also written in Lisp).

Both the web interfaces and PrimeTrader submit orders by inserting them into the core Oracle database.

2.3 Auto Router

For the main exchanges we are connected to, we offer *auto routing* of orders. The Auto Router immediately picks up the newly issued order and examines it. If the order looks OK (Order price limit is not deviating too much, the customer owns the shares to sell or has enough money to buy them, etc.), it is immediately forwarded to its destination exchange, but if there is some kind of deviation, it is queued for manual examination by our brokers.

2.4 Exchange Interfaces

The Auto Router sends the order to either one of the SAXESS (the protocol and system used by the Oslo and Stockholm exchanges) servers or one of the FIX servers (communicating with the american stock exchanges).

If the order is filled, information is returned through these servers, which updates their status in the core database.

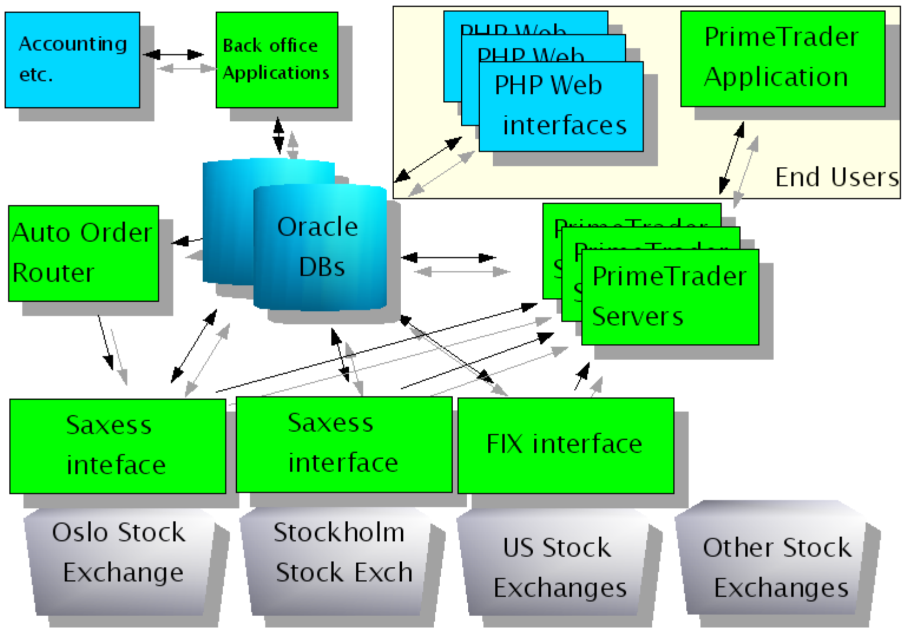

Figure 1: A partial and simplified overview of the Net Fonds System Architecture

From the database, the status is picked up by order status web pages and order status window of PrimeTrader.

2.5 "Back Office" Applications

After the orders have been executed, a whole portfolio of back office applications do all the bookkeeping work needed to ensure that ownership of the shares is properly transferred, that they are paid for (or credited for) by the customer, etc. Several of these applications have just been ported to Common Lisp.

3. THE LISP-BASED COMPANY

The advantages of using Common Lisp is probably well-known to most of the readers of this paper, and I like to sum them up as *having more fun while doing less work*.

What makes Net Fonds special, I think, is that we're using Common Lisp as our *work horse*, not just for the advanced "AI" applications. In fact, most of what we do is outside of the traditional application domains of Lisp. All the small tasks, the write-once scripts and small utilities, all the stuff that's usually done in e.g. Perl, we do in Lisp. So, ironically, what makes us special is how we do the boring stuff!

Before going more into detail, I'd like sum up our main advantages from using Common Lisp:

3.1 Flexible System Administration

Running server applications built with Common Lisp gives you the "dynamic" advantages of Lisp programming: New features can be loaded into running images, errors can be inspected and fixed without taking down the servers, etc. We use two different methods for communicating with running servers: Most of the applications are started inside an emacs shell buffer running under *screen*[1], i.e. we can always resume talking to the stdout/stdin of the Lisp processes. Some servers run completely in the background and use a different approach: They contain an "eval server" accessible with TCP.

[1]A "virtual terminal" utility for Unix

3.2 Writing Parsers

We communicate with external systems using a lot of different protocols. Using Lisp we're able to quickly write new parsers and adapt to changes in the complex protocols.

Several of these parsers are automatically generated from formal protocol specifications or C header files. This is probabably the closest we get to the more traditional symbol processing uses of Lisp!

3.3 High Reliability

With Lisp, we're able to get very reliable programs with less programming effort. Some of the server applications run for months non-stop, and minor errors are fixed by loading new code while they run.

3.4 More Fun

I don't think this should be underestimated. The fact that Common Lisp is such a powerful and useful tool for the programmers, making it easier and more fun for them to achieve their goals, is of course of great value to the company.

4. SOME SAMPLES

In the rest of the article, I will give you some samples of things we're doing with Common Lisp. The samples are taken from the trading application PrimeTrader and its server counterpart, which has been my main responsibility since I joined Net Fonds.

PrimeTrader provides our customers with real time stock quotes from several exchanges and easy-to-use order forms. It's a pure Lisp application, written with LispWorks with CAPI and compiled for Windows, Linux and Mac OS X. It's a self-contained application which even uses its own RSA and Blowfish encryption code, written entirely in Common

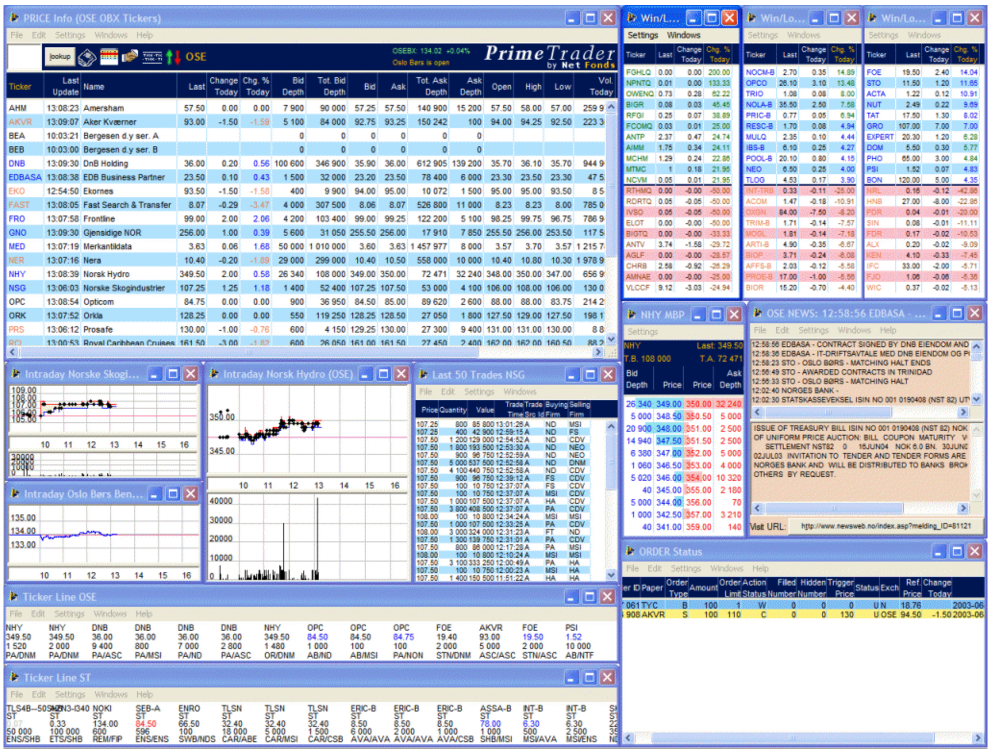

Figure 2: PrimeTrader Screen Shot

Lisp. Figure 2 shows a screen shot from the Windows version.

4.1 Automatic Patch Downloads

When PrimeTrader connects to the server, it will get a list of available patches, and if any of them are applicable, it will ask the user for permission to download the patch. The core of the patch download code looks like this (the function that calls this code uses **available-patches** to find suitable patches and then, after asking the user for permission, uses **install-patch** to install each of them.

```
(defun available-patches ()
  (remove-superseded-patches
   (remove-if-not
    #'patch-useful
    (trader:collect-from-exchange-table
     '(nftp:patch :all :all)))))

(defmethod install-patch ((patch nftp:patch))
  (unless (find patch *installed-patches*)
    (set-status
     "Downloading ~a"
     (nftp:patch-name-of patch))
    (handler-case (download-patch patch)
      (error (cond)
             (error
              "Error during patch download: ~a"
              cond)))
    (set-status "Verifying ~a"
                (nftp:patch-name-of patch))
    (verify-sha1 patch)
    (set-status "Loading ~a"
                (nftp:patch-name-of patch))
    (load-patch patch)
    (set-status "Updating patch info")
    (push patch *installed-patches*)
    (recompute-active)
    (save-patch-file)))

(defmethod download-patch ((patch nftp:patch))
  (http:get-url (nftp:url-of patch)
                (patch-local-pathname patch)))
```

In English, what happens is:

- The patch object (which is transferred from the server after login), contains information on location, required version etc., and also a SHA1 fingerprint to ensure integrity.
- The download is performed with an ordinary http get from one of our main apache servers.
- The SHA1 fingerprint of the file is computed and compared to the fingerprint given by the server.
- The patch itself, which is an ordinary fasl-file, is loaded into the running image.
- The patch is kept in PrimeTrader's configuration directory, and a Lisp file containing loading instructions is updated. This file is loaded automatically during the next startup.

We deliver most of our upgrades this way, but sometimes a major rewrite makes a complete install necessary.

4.2 Some Useful Tools for Servers

When Lisp Servers run for weeks and months, they need to do some internal system maintenance. The PrimeTrader server code contains some modules that support this:

- *An eval server* makes it possible to connect to a Lisp listener in order to inspect the state of the server and modify its code. For security reasons, the eval server only listens to localhost and generates a password only readable by a certain userid.
- *A cron.lisp* module implements a process similar to the *cron* facility familiar to unix users.
- *An at.lisp* module implements an analogue to unix's *at.*
- *A logger* module implements a facility for logging important events, including rotating and compression of log files.

Of these components, the *cron.lisp* is probably the most important, since the only alternative would be to have a lot of specialized processes doing all these tasks:

- Possibly do a global GC (every hour).
- Remove state of aborted sessions (every minute).
- Refresh some stock exchange info (every morning).
- Regenerate "watch lists" of stocks (every hour).
- Log the number of logged-in users (every minute).
- Rotate and compress logs (every midnight).
- Regenerate the eval-server password (every hour).

4.3 GC Considerations

Each PrimeTrader delivers real time stock quotes to 50-80 simultanous users, and the number of packets processed is quite large. So the server creates a large amount of data that lives for a few seconds, long enough for the LispWorks GC to move it to generation 2, where it isn't (with the default settings) garbage collected. Doing a GC of generation 2 lasts a little too long (up to 3 or 4 seconds), so we try to avoid that it happens during opening hours. What we do, is to

- run a GC ((mark-and-sweep 2) in LispWorks) every morning after the feed has been restarted, and
- run a job each hour that checks whether the allocated memory has reached a threshold, and then possibly do a GC.

We let the allocated memory grow with 150-300MB before a GC is triggered. For most of the PrimeTrader servers, this means that the morning GC run is sufficient (our machines have 1GB of RAM installed).

4.4 Slave Subprocesses

The PrimeTrader servers use our database to pick up order status information and to enter new orders. Initially, our servers communicated directly with Oracle. This didn't work out quite well, since LispWorks blocks during Oracle FFI calls. Such blocking was unacceptable in a server that served a large number of users with time-critical data, each of them would get a delay at least as long as the duration of the database call.

The solution to this problem was a very simple program called *lispslave*, which is started as a subprocess from the main lisp server. Each time the main server wants to call the database code, it writes the call to a pipe to one of the subprocesses, which does the actual call and writes it back again.

For each *lispslave* process, there is a corresponding *lispmaster* thread inside the PrimeTrader server. Each master communicates with its slave through a conventional unix pipe. A configurable number of such masters are started as the PrimeTrader server starts up, and they each start their own slave subprocess.

The master threads communicate with the rest of the PrimeTrader server through a *queue.* The main interface to the *lispmaster* code looks like this:

```
(defun slave-eval (form)
  (let ((pair (list form)))
    (lq:enqueue pair *eval-queue*)
    (unless
        (mp:process-wait-with-timeout
          "waiting for result"
           *slave-timeout*
           #'rest pair)
      (error "No response from slave subprocess"))
    (let ((result (rest pair)))
      (if (first result)
          (error
            (format nil "~a error in lispslave: ~a"
                    (second result)
                    (third result)))
        (values-list (second result))))))
```

This code works like this: The caller of **slave-eval** wants to evaluate *form* in one of the slaves. **slave-eval** starts by queueing a list containing the form into the **eval-queue**, where an idle lispmaster process will find it, evaluate it, and put a result or an error message into the *cdr* of the list.

So **slave-eval** just has to wait (with process-wait-with-timeout) for the result to appear in the *cdr*, and then return it (or possibly signal an error).

5. CONCLUSION

The fact that Net Fonds operates a profitable Internet-based business with very low IT costs, indicates that using Common Lisp for all kind of tasks, even the most trivial ones, can be good for your business.

In short, you get not only *more fun and less work*, but even *more reliable services.*