Battlefield report: Bittorrent protocol implementation

Analysis of using Erlang and Haskell

Jesper Louis Andersen
jesper.louis.andersen@gmail.com

Sep 27, 2010
Overview

Overview

**Goal:** Tell a story. Give insight.
**Priming:** What is it, really?

Priming: What is it, really?

Actors! You have hundreds of independent processes ...
Overview

Priming: What is it, really?
Actors! You have hundreds of independent processes ...
War diary: Musings over the implementations.
History

*Etorrent* - A bittorrent client implemented in Erlang
  - Erlang/OTP implementation
  - Initial Checkin, 27th Dec 2006
  - Had first working version around early 2008
  - 5 KSLOCs

*Combinatorrent* - A bittorrent client in Haskell
  - GHC (Glasgow Haskell Compiler) implementation
  - Initial checkin: 16th Nov 2009
  - First working version less than 2.5 months after
  - Implements an actor-like model on top of STM (Software Transactional Memory)
  - 4.1 KSLOCs
Acknowledgements

This is joint work; try to make it easy to contribute:

*Etorrent*: Tuncer Ayaz, Magnus Klaar

*Combinatorrent*: Alex Mason, Andrea Vezzozi, “Astro”, Ben Edwards, John Gunderman, Roman Cheplyaka, Thomas Christensen
Why?

Several reasons:

- "To fully understand a programming language, you must implement something non-trivial with it."
  – Jespers Law
- A priori
- A posteriori
- Gauge the effectiveness of modern functional programming languages for real-world problems.
- BitTorrent is a good "Problem Set"
Why?

Several reasons:

- “To fully understand a programming language, you must implement something non-trivial with it.” – Jespers Law
  - A priori
  - A posteriori
Why?

Several reasons:

- “To fully understand a programming language, you must implement something non-trivial with it.” – Jespers Law
  - A priori
  - A posteriori
- Gauge the effectiveness of modern functional programming languages for real-world problems.
Why?

Several reasons:

- “To fully understand a programming language, you must implement something non-trivial with it.” – Jespers Law
  - A priori
  - A posteriori
- Gauge the effectiveness of modern functional programming languages for real-world problems.
- BitTorrent is a good “Problem Set”
KSLOCs

<table>
<thead>
<tr>
<th>Software</th>
<th>KSLOCs</th>
</tr>
</thead>
<tbody>
<tr>
<td>wgo</td>
<td>0e+00</td>
</tr>
<tr>
<td>combinatorrent</td>
<td>1e+05</td>
</tr>
<tr>
<td>bittornado</td>
<td>2e+05</td>
</tr>
<tr>
<td>rtorrent</td>
<td>3e+05</td>
</tr>
<tr>
<td>ktorrent</td>
<td>4e+05</td>
</tr>
<tr>
<td>transmission</td>
<td></td>
</tr>
<tr>
<td>deluge</td>
<td></td>
</tr>
<tr>
<td>Vuze</td>
<td></td>
</tr>
</tbody>
</table>
**HTTP vs BitTorrent**

BitTorrent is about *Content distribution*. Some key differences:

<table>
<thead>
<tr>
<th>HTTP</th>
<th>BitTorrent</th>
</tr>
</thead>
<tbody>
<tr>
<td>▶ Simple</td>
<td>▶ Complex</td>
</tr>
<tr>
<td>▶ Stateless</td>
<td>▶ Stateful</td>
</tr>
<tr>
<td>▶ One-to-many</td>
<td>▶ Peer-2-Peer</td>
</tr>
<tr>
<td>▶ “Serial”</td>
<td>▶ “Concurrent”</td>
</tr>
<tr>
<td>▶ Upstream bandwidth</td>
<td>▶ Upstream bandwidth scales</td>
</tr>
<tr>
<td>heavy</td>
<td>proportionally with number of</td>
</tr>
<tr>
<td></td>
<td>consumers</td>
</tr>
</tbody>
</table>

In BitTorrent everything is sacrificed for the last point.
Key concepts

*One:* A stream of bytes is split into *pieces* and exchanged among peers with a message-passing protocol.
Two: Swarm intelligence

Beehives, Ant colonies, wasps.
Two: Swarm intelligence

Beehives, Ant colonies, wasps.

Each client acts *independently* with a 10 second memory, only evaluates downstream bandwidth; unless it is *seeding*.

Mantra: *Be friendly to your established friends, but be optimistic about gaining new ones*  
Mimics human interaction.
Actor models

“Island model”
Actor models

“Island model”

- Cheap processes (green, userland based)
- Fast CTX switch
- Process Isolation, message pass is persistent or a copy
Communication (Link)

Peer #1
- Peer #2
- P1
- Socket
- P2
- P3
- Tracker
- PeerMgr
- ChokeMgr
- Status
- PeerP
- Peer SendQueue
- Peer_Receiver
- Peer Sender
- Main
- FS
- Console
- Timer
- PieceMgr
- Listener
- HTTP

Peer #2
- Peer_Receiver
- PeerP
- Peer SendQueue
- Peer Sender
Process Hierarchy (Location)

S0
- Main
- Timer
- Console
- PeerMgr
- ChokeMgr

S1
- S2
- FS
- Tracker
- Status
- PieceMgr

S2
- SPeer1
  - P1Receiver
  - P1SendQ
  - P1PeerP
  - P1Sender
- SPeer2
  - P2Receiver
  - P2SendQ
  - P2PeerP
  - P2Sender
Bigraphs

\[ \text{Bigraph} = \text{Hypergraph} + \text{Tree} \]

Do not confuse with bipartite graphs.

Hypergraph is the \textit{link}-graph
Tree is the \textit{location}-graph
Robustness

Robustness is key to good programming:

- Semantics (segfault, Null, of-by-one, ...)
- Proactive: Haskell
  - Type system
- Reactive: Erlang
  - Crashes, restarts
  - Supervisors
  - Redundancy

Ideas from both areas are needed in robust software!
Process Hierarchy (Location)
Strings in Haskell and Erlang

- Single linked lists of runes

Solution: Use ByteString for binary data in Haskell, binaries/iolists in Erlang.
Strings in Haskell and Erlang

- Single linked lists of runes
- Simple
- Unicode is trivial
- List operations are string operations
Strings in Haskell and Erlang

- Single linked lists of runes
- Simple
- Unicode is trivial
- List operations are string operations
- It *is* fairly fast
- Extremely memory heavy (16+ bytes per char in Erlang!)
Strings in Haskell and Erlang

- Single linked lists of runes
- Simple
- Unicode is trivial
- List operations are string operations
- It is fairly fast
- Extremely memory heavy (16+ bytes per char in Erlang!)

Solution: Use ByteString for binary data in Haskell, binaries/iolists in Erlang.
Some cool things in Haskell

- Haskell is king of abstraction (sans Proof assistants)
- Type system is *expressive* almost to the point of program proof
- Strong *Type Zoo*
Some cool things in Haskell

- Haskell is king of abstraction (sans Proof assistants)
- Type system is *expressive* almost to the point of program proof
- Strong *Type Zoo*
- Combinators run at full speed in Haskell
Some cool things in Haskell

- Haskell is king of abstraction (sans Proof assistants)
- Type system is *expressive* almost to the point of program proof
- Strong *Type Zoo*
- Combinators run at full speed in Haskell
- Close to being clay: you can model actors easily
Some cool things in Haskell

- Haskell is king of abstraction (sans Proof assistants)
- Type system is *expressive* almost to the point of program proof
- Strong *Type Zoo*
- Combinators run at full speed in Haskell
- Close to being clay: you can model actors easily
- Excellent community - vibrant; practitioners and academics.
- QuickCheck - (John Hughes, Wednesday)
The bad in Haskell

- Lazy evaluation - space leaks
The bad in Haskell

- Lazy evaluation - space leaks
  - Heap Profile – Use strictness annotations,
The bad in Haskell

- Lazy evaluation - space leaks
  - Heap Profile – Use strictness annotations,
  - Peak Mem:
  - Productivity:
  - CPU/Mb:
The bad in Haskell

- Lazy evaluation - space leaks
  - Heap Profile – Use strictness annotations,
  - Peak Mem:
  - Productivity:
  - CPU/Mb:

- Academic compilers, stability suffer
- Some libraries are extremely complex type-wise
Some cool things in Erlang

- Crash-oriented programming is bliss, an error might not be fatal
Some cool things in Erlang

- Crash-oriented programming is bliss, an error might not be fatal
- **OTP** - Actor abstraction: Servers, event drivers, finite state machine, supervision, logging, ...
Some cool things in Erlang

- Crash-oriented programming is bliss, an error might not be fatal
- **OTP** - Actor abstraction: Servers, event drivers, finite state machine, supervision, logging, ...
- Processes are individually garbage collected (isolation)
Some cool things in Erlang

- Crash-oriented programming is bliss, an error might not be fatal
- **OTP** - Actor abstraction: Servers, event drivers, finite state machine, supervision, logging, ...
- Processes are individually garbage collected (isolation)
- Interpreted language, but implementation is heavily optimized
Some cool things in Erlang

- Crash-oriented programming is bliss, an error might not be fatal
- *OTP* - Actor abstraction: Servers, event drivers, finite state machine, supervision, logging, ...
- Processes are individually garbage collected (isolation)
- Interpreted language, but implementation is heavily optimized
- Again, excellent community!
The bad in Erlang

- Not suited for number crunching (have to choose right algorithm, data structure)
The bad in Erlang

- Not suited for number crunching (have to choose right algorithm, data structure)
- No way to do imperative code (Deliberate choice by the Erlang developers, have to fake it)
The bad in Erlang

- Not suited for number crunching (have to choose right algorithm, data structure)
- No way to do imperative code (Deliberate choice by the Erlang developers, have to fake it)
- Dynamic typing (Dialyzer project helps, processes are small (< 500 lines))
The Ugly

Haskell:
- Take laziness seriously from the start
- Be careful when choosing libraries

Erlang:
- Be careful about messaging large data between processes
- Mnesia has optimistic conflict resolution

Both: Expect to manipulate your process model quite a bit.
The Ugly

Haskell:
► Take laziness seriously from the start
► Be careful when choosing libraries

Erlang:
► Be careful about messaging large data between processes
► Mnesia has optimistic conflict resolution
The Ugly

Haskell:
- Take laziness seriously from the start
- Be careful when choosing libraries

Erlang:
- Be careful about messaging large data between processes
- Mnesia has optimistic conflict resolution

Both: Expect to manipulate your process model quite a bit.
We use github for all code:

http://www.github.com/jlouis

Look for etorrent and combinatorrent