Go-style concurrency

Øyvind Teig
@embedded.TRD 26. March 2014

CSP and process-oriented programming

Even after a long history of channel-based concurrency it is not mainstream

In the light of Google's new programming language Go, this lecture will search for shores around the Channel Islands

CSP and process-oriented programming

Even after a long history of channel-based concurrency it is not mainstream

In the light of Google's new programming language Go, this lecture will search for shores around the Channel Islands

I would appreciate questions and dialogue during the presentation!

- Autronica Fire and Security AS (AFS)
- Owned by United Technologies Corporation (UTC)
  - Part of UTC Building and Industrial Systems
- Employs more than 380
  - 25 R&D (Trondheim)
    - Considering some internationalisation
- Headquarter in Trondheim
- Mainly fire detection
- NOK 825 mill

Disclaimers

In this lecture I present my own interests and views
(But the themes would be relevant to some reality at Autronica)

This lecture does not reveal any Autronica-sensitive information


I am..
{
// an embedded programmer (since 1978)
// not a computer scientist
// a blogging «coder" @ http://www.teigfam.net/oyvind/home/
}

// NTH 1976, Autronica 1976-, engine systems, fluid level, fire detection, HW, SW,
// Asm, MPP Pascal, PL/M, Modula 2, occam, (Java, Perl), C, small RTX systems, published, blogging

Goal

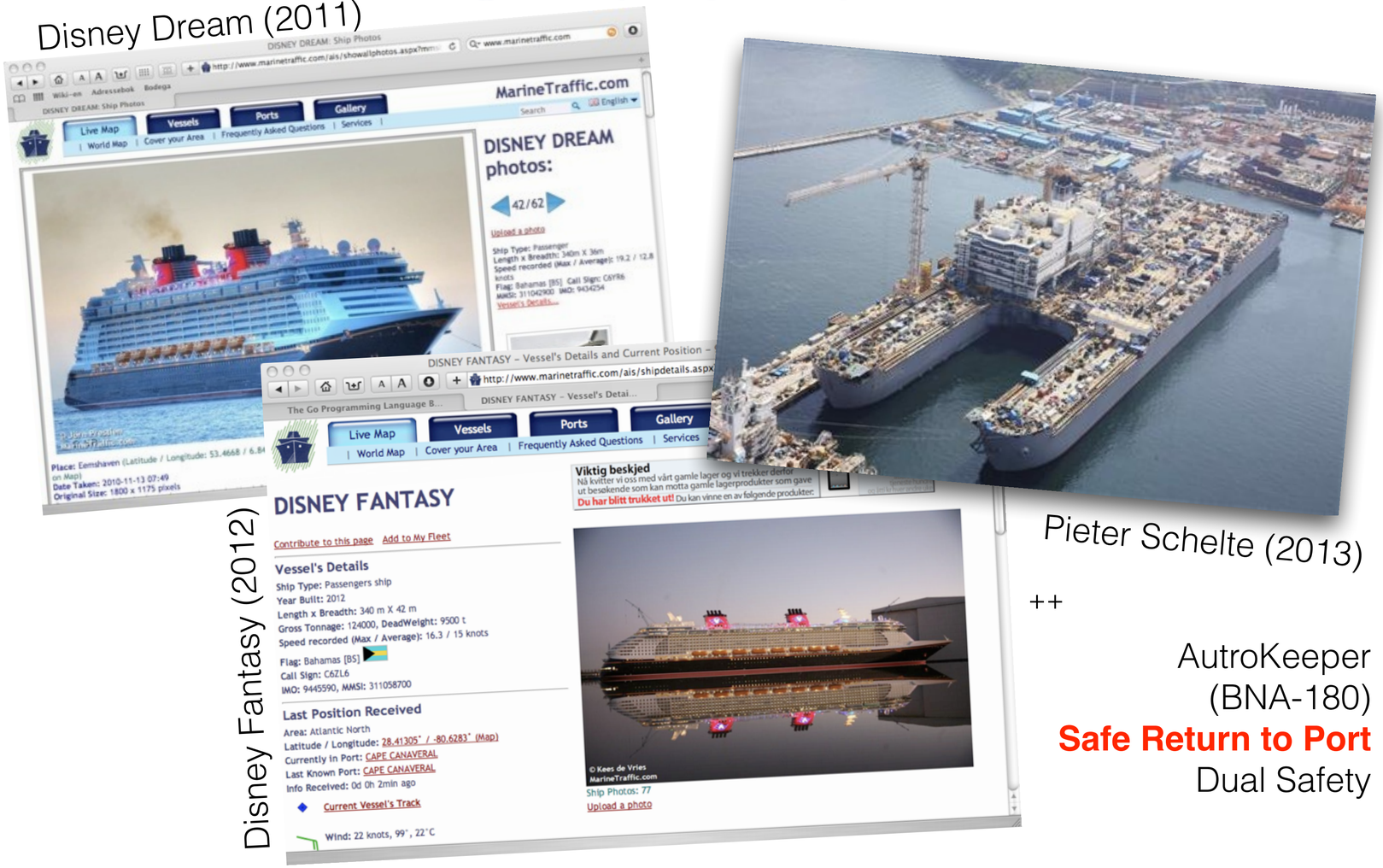
- To make you curious about how CSP-based multi-threaded systems with synchronous blocking communication

can be an alternative to

- single-threaded systems with asynchronous non-blocking calls, registering of callbacks and non-local event-loop to pick up the result of an off-line handling

On the iron!

Disney Dream (2011)



Disney Fantasy (2012)

Pieter Schelte (2013)

++

AutroKeeper
(BNA-180)
Safe Return to Port
Dual Safety

AutroKeeper: patented 329859 in Norge, PCT/NO2009/000319 EU (granted as #2353255)

Mind map?

- This lecture is a Mind Map (like brainstorming?)
- of what I have learnt lately (the odd matter)
- in the light of my personal experience (biased)
- and the background I read about you (rather mixed?)

Go-style concurrency, based on

- CSP = Communicating Sequential Processes
- CSP is a process algebra (Hoare, 1978, 1985)
- Influential (3rd most cited paper at some time)
- Model with CSPm and formally verify with FDR2 / FDR3
- Also PAT (Process Analysis Toolkit) from Singapore

#1

Single-threading sometimes is more multi-threaded than one might think

Multi-threading shown

with an (almost) single-threaded language

node-csp
«Communicating sequential processes for Node.js»
«Go style concurrency with channels»
(an experiment)

node-csp

- Uses the asynchronous event system of Node.js
- and «generators» of EcmaScript6 (ES6)
- Written in JavaScript, just to explore any potential

Blog:
http://www.teigfam.net/oyvind/home/technology/084-csp-on-node-js-and-clojurescript-by-javascript/

node-csp (example interleave.js)

```
1  var csp = require("..");
2
3  var chan1 = new csp.Chan(); // Create an unbuffered channel.
4
5  csp.spawn(function* () {
6    for (var i = 0; i < 10; i++) {
7      console.log("put", i);
8      yield chan1.put(i); // Send 'i' on channel 'chan1'.
9    }
10   yield chan1.put(null);
11 });
12
13 csp.spawn(function* () {
14   for (;;) {
15     var i = yield chan1.take(); // Take a value of 'chan1'.
16     if (i === null) break; // Quit if we get 'null'.
17     console.log("take", i);
18   }
19 });
```

By:
Ola Holmström & David Nolen

node-csp (example interleave.js)

```
 1 var csp = require("..");
 2
 3 var chan1 = new csp.Chan(); // Create an unbuffered channel.
 4
 5 csp.spawn(function* () {
 6   for (var i = 0; i < 10; i++) {
 7     console.log("put", i);
 8     yield chan1.put(i); // Send 'i' on channel 'chan1'.
 9   }
10   yield chan1.put(null);
11 });
12
13 csp.spawn(function* () {
14   for (;;) {
15     var i = yield chan1.take(); // Take a value of 'chan1'.
16     if (i === null) break; // Quit if we get 'null'.
17     console.log("take", i);
18   }
19 });
```

Src: https://github.com/olahol/node-csp/blob/master/examples/interleave.js#L1

Concurrency: doing more than one thing

- «Single-threaded»
- Multi-threaded
- Concurrent
- Parallel

Concurrency is all over

- Web pages (in browsers)
- Applications (on Linux, Windows, OS X)
- Embedded systems
  - Integrity, VxWorks, Linux, some run-time system and scheduler, even main+interrupts
  - Languages: occam (was), Go (at least not yet), Ada, Erlang, ..?
  - «Synchronous system» with Rate Monotonic Scheduling etc. not in this lecture

#2

«A person’s mental model is probably multithreaded,
although this is rarely conscious»

Trygve Reenskaug (in a mail)

Unit of simultaneousness

- “Process model”
- By a ready tool (language, pattern) or
  built from a kit (library, pattern, stereotype)?
- Traditional objects in OO are only half the way:
  - Java: «**class MyThread implements runnable**» (interface)

Unit of locality

- Concurrency + multi-core = parallel
- Shared or distributed memory
- Go has a shared memory model,
  occam has shared/distributed; both use channels
  Ada?
  Erlang?

Single-threaded

- How can you do simultaneous things non-spaghetti'ish?
- Single-threaded + non-blocking =
  register callback function and treat it in a (global, local) event loop =
  asynchronous
- Single-threaded + many independent jobs =
  communicating callbacked functions =
  probably not..

Single-threaded?

- This not as “single-threaded” as you’d like to think
- “Callback hell”(?)
- Even W3C’s DOM has mutexes (DOM = Document Object Model)
  It’s a «storage mutex»

Process model

- Many languages add concurrency or parallelism as an afterthought
  - First real language was Concurrent Pascal, 1975 (Per Brinch Hansen)
- C11: thread C++11: thread, futures etc.
- Process: not only for concurrent activities
  - Also as abstraction of a different type of “object” (=”process”)
  - Encapsulate state

Process-oriented

occam

```
PAR
  P(c)
  C(c)
```

Go

```
go P(c)
go C(c)
// Some join
```

pyCSP

```
import sys
from pycsp.parallel import *

@process
def source(chan_out):
    for i in range(10):
        chan_out("Hello world (%d)\n" % (i))
    retire(chan_out)

@process
def sink(chan_in):
    while True:
        sys.stdout.write(chan_in())

chan = Channel()
Parallel(
    5 * source(-chan),
    5 * sink(+chan)
)

shutdown()
```

«Brings CSP to Python»

https://code.google.com/p/pycsp/

Communication

- Often the concurrent units need to communicate
- Shared memory and/or messages
- Contracts: protocols, typed (with language support?)
- This is not a pipe
  - It’s a channel (or a rendezvous/Ada)
- “First class” = like sending a channel over a channel

Messages

- Channels carry typed messages (as said, some even send channels), zero buffered (synchronous) or buffered (asynchronous).
  - When blocking, synch and comm is the same event
- SDL (Spec & Design Lang) (and UML(?)) do send-and-forget of messages into infinite buffers: asynchronous
  - I have blogged a lot about the pros and cons, no repeat here
- Both may be used to build communication FSM (Finite State Machine)s

`try` (to say in 1990)

«Processes and synchronous, blocking channels»

`catch` (the late response in 2000)

- “nobody else does it”
- “the problem you say it solves isn’t ours”
- «use send-and-forget»

(digest) `until` (2009!)

- Google delivers Go
- It enforces “occam thinking”
- in a wonderfully different, old & new way
- “Go-style concurrency”
- Bell Labs had now *also* been doing it for 30 years
  - (Now they dared say it)

“**Bell Labs and CSP Threads**” by Russ Cox
see http://swtch.com/~rsc/thread/

«Why build concurrency on the ideas of CSP?»

- From Google Go language's authors:
- Mutexes :-(
- Condition variables :-(
- Memory barriers :-(
- Higher-level interfaces enable much simpler code :-)

http://golang.org/doc/faq#csp

«Why build concurrency on the ideas of CSP?»

- Successful: CSP :-)
- Occam and Erlang :-)
- Go’s channels as first class objects :-)
- Fits well into a procedural language framework :-)

Nothing really «blocks»

- Processor cycles seldom used to spin around waiting for a single resource
- So “blocking” entails being able to do something else in the meantime
- Like sleep, or callback, directly or hidden in a language mechanism
- This is the basic problem
- If there is no good mechanism to handle this, then “blocking is evil” is a valid fear of spaghetti concurrency!
  - And at some (but not all!) levels timeout must catch it!
- Then even the “I program single-threaded” statement is a valid response

“Blocking is evil”

- Asynchronous and non-blocking
- Synchronous and blocking
- Neither is «evil» if both mechanisms are easily available
- But in a safety-critical system buffered must have control of max buffer need!

Buffer where?

- Buffer in a pipe-type untyped buffer, «max 1500 bytes"?
- Or in a typed buffered channel?, «message»
- Or inside a process?, «flush, prioritise»
  - Why buffer in the tube and not in a tank?
- Worse: a tube is not an expansion tank! (overflow?)

Rich interface component modelling

- A component also has dynamic properties
  not only a standard API
- Can be modelled in several languages
  CSP, UML..
- To make verified connectable sw components:
  Less testing(?)
- I have tried to understand this:
  http://www.teigfam.net/oyvind/home/technology/081-rich-interface-component-modeling/
- I am looking forwards to hearing BitReactive (here now?) talk about their solution!

Autronica (C.V.)

- Autronica shipped several products programmed in occam in the nineties
  - …
  - Diesel engine data acquisition and computation unit (NK-200).
    With transputers
  - Radar-based fluid level gauging (GL-100, now Kongsberg product).
    With signal processor and occam to C-with-scheduler translator
- Transputers and occam were “CSP engines”

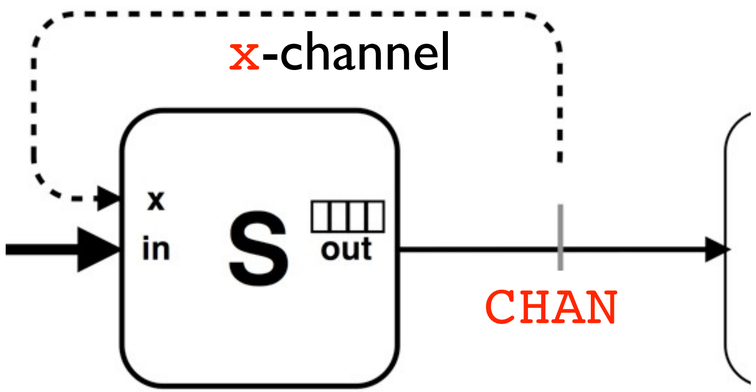
Autronica (C.V.)

- Some CSP-based run-time schedulers used in ARM, Atmel ATmega, XMEGA
  - One built on top of an asynchronous SDL-based kernel (channel ready then run process converted run-«messages»)
  - A «naked», with synchronisation points generated by a tool in ANSI C (much like the yield we started with) (Channel ready is synchronisation only)
- Published papers and discussion of the pros and cons of each of these
- (But we mostly seem to use the SDL-based systems for fire loop units)

XCHAN

Higher order channel and pattern

Trying to merge synchronous and asynchronous traditions



«Safe» send-and-forget

Communicating Process Architectures 2012
P.H. Welch et al. (Eds.)
Open Channel Publishing Ltd., 2012

© 2012 The authors and Open Channel Publishing Ltd. All rights reserved.

XCHANs: Notes on a New Channel Type

Øyvind TEIG [1]
Autronica Fire and Security AS [2], Trondheim, Norway

(3 typos fixed, 31.Aug.2012)

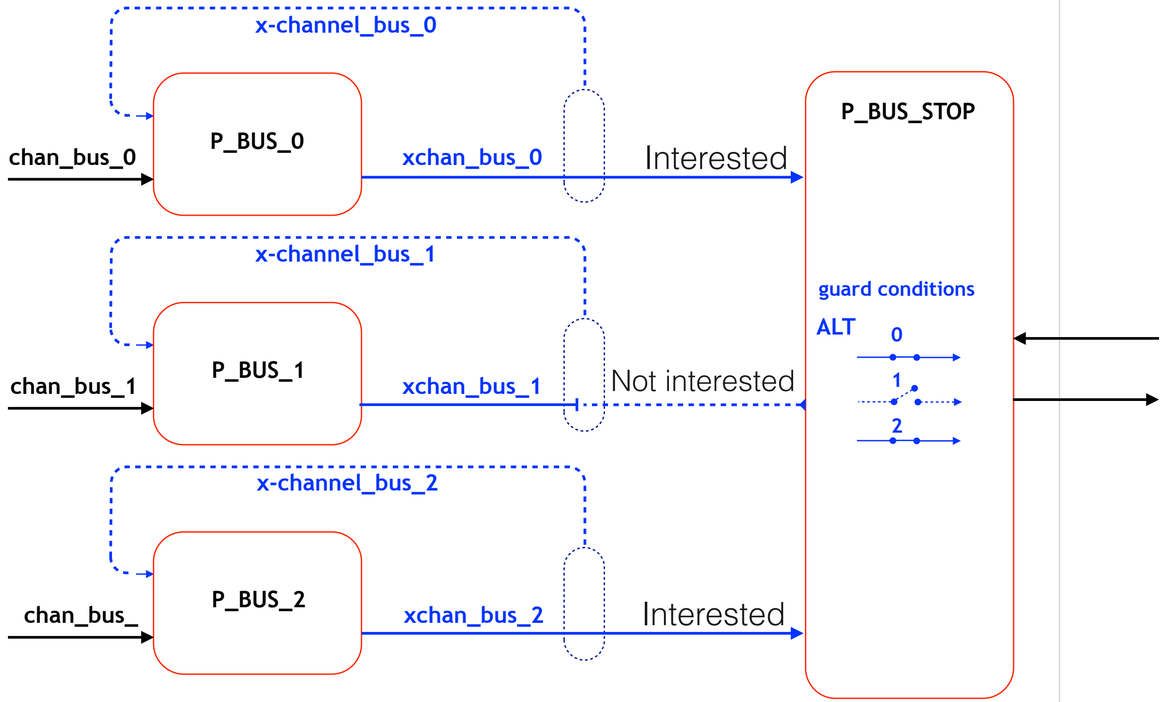
Abstract. This paper proposes a new channel type, **XCHAN,** for communicating messages between a sender and receiver. Sending on an **XCHAN** is asynchronous, with the sending process informed as to its *success*. **XCHAN**s may be *buffered*, in which case a successful send means the message has got into the buffer. A successful send to an unbuffered **XCHAN** means the receiving process has the message. In either case, a failed send means the message has been discarded. If sending on an **XCHAN** fails, a built-in feedback channel (the ***x****-channel*, which has conventional channel semantics) will signal to the sender when the channel is ready for input (i.e., the next send will succeed). This **x**-channel may be used in a **select** or **ALT** by the sender side (only input guards are needed), so that the sender may passively wait for this notification whilst servicing other events. When the x-channel signal is taken, the sender should send as soon as possible – but it is free to send something other than the message originally attempted (e.g. some freshly arrived data). The paper compares the use of **XCHAN** with the use of output guards in **select**/**ALT** statements. **XCHAN** usage should follow a design pattern, which is also described. Since the **XCHAN** never blocks, its use contributes towards deadlock-avoidance. The **XCHAN** offers one solution to the problem of overflow handling associated with a fast producer and slow consumer in message passing systems. The claim is that availability of **XCHANs** for channel based systems gives the designer and programmer another means to simplify and increase quality.

Keywords. Channels, synchronous, asynchronous, buffers, overflow, flow control, CSP.

Introduction

Feathering

An implicit subscriber pattern built on XCHAN



Communicating Process Architectures 2013
P.H. Welch et al. (Eds.)
Open Channel Publishing Ltd., 2013
© 2013 The authors and Open Channel Publishing Ltd. All rights reserved.

Selective Choice 'Feathering' with XCHANs

Øyvind TEIG[1]
Autronica Fire and Security AS[2], Trondheim, Norway

Abstract. This paper suggests an additional semantics to **XCHAN**s, where a sender to a synchronous channel that ends up as a component in a receiver's selective choice (like **ALT**) may (if wanted) become signaled whenever the **ALT** has been (or is being) set up with the actual channel *not* in the active channel set. Information about this is either received as the standard return on **XCHAN**'s attempted sending or on the built-in feedback channel (called **x-**channel) if initial sending failed. This semantics may be used to avoid having to send (and receive) messages that have been seen as *uninteresting*. We call this scheme *feathering*, a kind of low level *implicit* subscriber mechanism. The mechanism may be useful for systems where channels that were not listened to while listening on some *other* set of channels, will not cause a later including of those channels to carry already declared uninteresting messages. It is like not having to treat earlier bus-stop arrival messages for the wrong direction after you sit on the first arrived bus for the correct direction. The paper discusses the idea as far as possible, since modeling or implementation has not been possible. This paper's main purpose is to present the idea.

Keywords. channels, synchronous, asynchronous, buffers, overflow, flow control, CSP, modeling, semantics, feathering

Introduction

The idea of the suggested semantics appeared after reading Tony Hoare's lecture "Concurrent programs wait faster" from 2003 [1]. After some pondering of a situation described

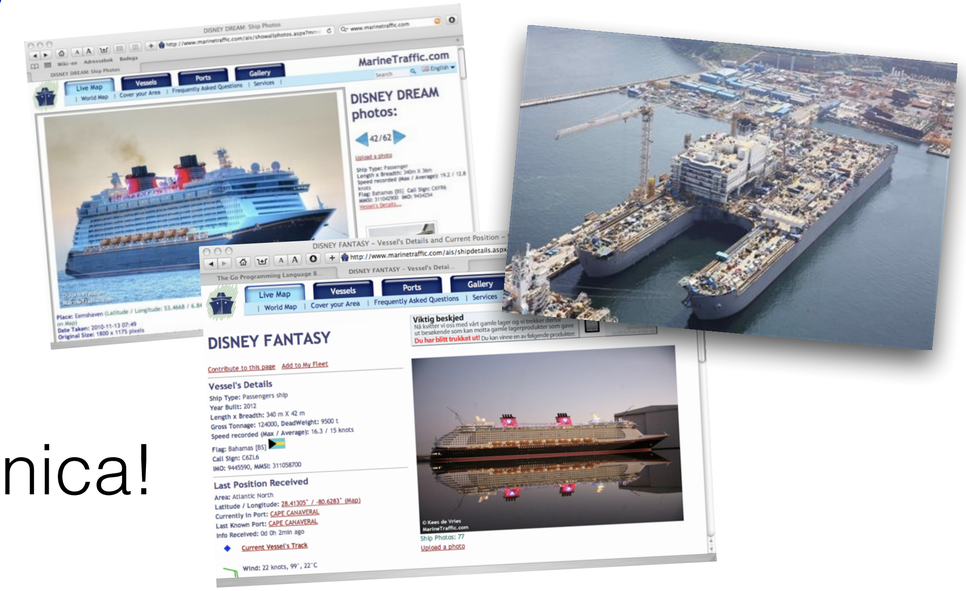
«CSP: arriving at the CHANnel island»

- My CPA-2000 paper
  www.teigfam.net/oyvind/pub/pub_details.html#CSP:arriving_at_the_CHANnel_island
- This was 9 years before Go
- But 10 years of working with occam
- What influence will Go have?
  - Students at NTNU now «love Go»!
- A small step..or?

Haberdashery

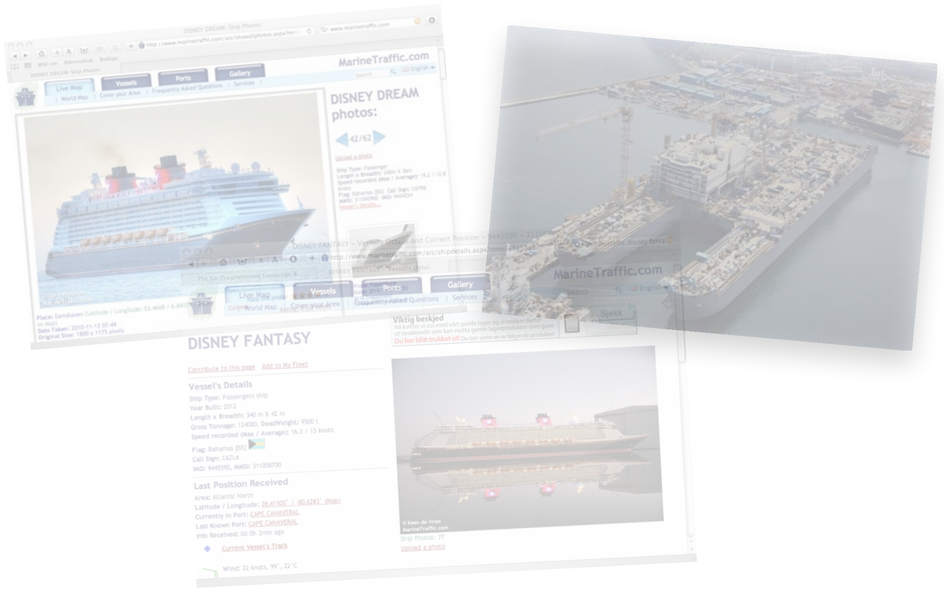
- Channels have no busy-poll: channel (msg, tim, int) *drives* scheduling
  - -> Low power?
- Present CSP processor: www.xmos.com
  - «xCore MULTICORE», xC with channels, guaranteed to meet timing requirements

Summary



- { AUTRONICA United Technologies ?
- There are *other* stories from Autronica!
- To learn more Go, register on: golang-nuts
- This lecture will be on my home page / blog: http://www.teigfam.net/oyvind/home/

Summary



- { AUTRONICA United Technologies
- There are
- To learn more Go, register on: golang-nuts
- This lecture will be on my home page / blog: http://www.teigfam.net/oyvind/home/

Thank you!