無

# CL-MUPROC

*Erlang-inspired multi-processing in Common Lisp*

*Klaus Harbo*
*Mu Aps.*

無

# Mu

- Small software development company
  - Copenhagen, Denmark
  - 6 people
- Algorithmic trading platform
  - Financial markets – London, Paris
  - Erlang
- Arbitraging betting platform
  - On-line betting markets – Europe, Asia
  - Common Lisp

無

# Overview

- Motivation
- Muproc goals, non-goals
- Core muproc
  - mumsg, send, receive
- Generic behaviours
  - Generic server
  - Supervisor
- Conclusion

# Erlang

- General-purpose programming language
  - Intended for 'soft real-time' problems
  - Concurrency, distribution, fault tolerance
  - Isolated, message passing processes
  - Functional, single assignment
  - Hot code upgrade, continuous operation
  - Large number of lightweight processes, $>10^5$
- Mature implementation
  - Originally developed by Ericsson
  - Open source, freely available

Joe Armstrong, Ph.D Thesis:
*"Making reliable distributed systems in the presence of software errors"*

無

# Erlang notables

- System is a hierarchy of tasks
- Isolation, shared-nothing
- Unreliable communications
- Pattern-matching on incoming messages
- Inter-process failure detection
  - fail-and-restart, non-local error handling
- Generic behaviours, in libraries
  - Generic server
  - Supervisor
  - Finite state machine
  - Event handler

無

# At Mu we think that...

- MP is here to stay
  - We expect to do a lot of it in the coming years
  - Lock-based approach is too hard
  - We want a sound foundation
- Erlang's MP model works really well
  - Higher productivity, fewer errors
  - Improved reliability
  - Better overall system structure
  - Excellent scalability, performance
  - Distributed operation very easy to achieve

無

# Why Lisp, then?

- Expressive power
- Multi-paradigm
- Small-team productivity
- Embedding special-purpose languages easy

But

We want (some of) Erlang's features in Lisp!

無

# muproc goals

- language, embedded in CL, which has/does:
  - isolated, message-passing processes
  - pattern matching on incoming messages
  - process linking, monitoring
  - generic behaviours
  - enables/supports distributed operation
    - eventually, for scalability

# muproc non-goals

- erlang features we're **NOT** aiming for
  - immutable message objects
    - we're passing Lisp objects around – very efficient
    - treating them as immutable is good policy
  - very large number of processes
    - we rely on the CL implementation's MP system
    - considerably heavier than Erlang processes
  - code upgrades on live systems
  - *...*
- muproc does not try to compete with Erlang

# Core muproc concepts...

# muproc

- muproc – a special Lisp process
  - unique muproc name
  - single input for incoming messages
  - errorstream
  - may be linked to one or more other muprocs
  - may trap exits
  - set of initial special bindings
  - set of registered port names
  - managed by the muproc run-time system

無

# Linking

- linking
  - symmetric relationship between two muprocs
    - if one terminates, the other is notified...
    - ...and vice versa
    - default notification response: termination
- monitoring
  - like linking, but one-way (asymmetric)
- trapping exits
  - enables specialized notification responses
  - special message sent when linked muprocs terminate
  - message: *terminated, reason*

# mumsg

- **mumsg**
  - container abstraction
  - a set of named values
  - basis for pattern matching on messages
  - used with **mumsg-send**, **mumsg-receive**

```
(mumsg :quantity 3 :price 42)
```

# mumsg-send

- **`mumsg-send dest &rest plist`**
  - used to send mumsg
  - **`dest`** is muproc or named port
  - **`plist`** is used to build **`mumsg`**
  - based on lower-level **`muproc-send`**

```
(mumsg-send logger  :msg "Hi there!")
(mumsg-send :logger :msg "Hi there!")
```

# mumsg-receive

- **`mumsg-receive (from) clause*`**
  - performs input pattern matching
  - will look for mumsg matching **`clause*`**
  - binds **`from`** to sender of message
  - each clause consists of
    - a required set of mumsg field names
    - a guard predicate
    - a list of forms which is to be evaluated
  - based on lower-level **`muproc-receive`**

# mumsg-receive

- **mumsg-receive (cont'd)**
  - messages are matched in order
  - clauses are matched in order
  - blocks if there are no matching messages
  - consumes matched message
  - you don't want side-effects in guards!

```
(mumsg-receive (from)
  ((counter) (and (integerp counter)
                  (oddp counter))
   (out "Counter is an odd integer."))
  ((counter) t
   (out "Counter is not an odd integer."))
  (() t
   (out "Unexpected mumsg: ~a." *muproc-packet*)))
```

# Areaserver

```
(defun areaserver ()
  (flet ((server ()
           (muproc-with-registered-port (:areaserver)
             (let ((total 0))
               (loop
                (mumsg-receive (from)
                  ((cmd x) (eq cmd 'square)
                   (mumsg-send from :answer (let ((area (* x x)))
                                              (incf total area)
                                              area)))
                  ((cmd) (eq cmd 'sum)
                   (mumsg-send from :answer total))
                  ((cmd) (eq cmd 'quit)
                   (out "Terminating.")
                   (muproc-exit :received-quit-cmd)))))))))
    (muproc-spawn 'areaserver #'server ()
                  :errorstream *trace-output*)))
```

# Areaclient

```
(defun areaclient (count)
  (flet ((client (max)
           (loop repeat max
                 do (let ((random (random 100)))
                      (mumsg-send :areaserver
                                  :cmd 'square
                                  :x random)
                      (mumsg-receive (from)
                        ((answer) t
                         (out "The square of ~a is ~a."
                              random answer))))))))
    (muproc-spawn 'areaclient #'client (list count)
                  :errorstream *trace-output*)))
```

# Detecting remote failure

```
(defun supervision-example (max)
  (labels ((child ()
             (sleep 1) (muproc-exit :done))
           (parent (count)
             (flet ((spawn-one ()
                      (muproc-spawn (gensym "CH-") #'child () :link t)))
               (spawn-one)
               (loop
                (mumsg-receive (from)
                  ((terminated reason) (plusp count)
                   (decf count)
                   (out "~a terminated - restarting..."
                        (muproc-name terminated))
                   (spawn-one))
                  ((terminated reason) t
                   (out "Too many restarts -- terminating.")
                   (muproc-exit :too-many-restarts))))))))
    (muproc-spawn 'parent #'parent (list max)
                  :trap-exits t :errorstream *trace-output*)))
```

無

# Other operators

- other major muproc operators
  - **muproc-link, muproc-monitor**
  - **muproc-set-trap-exits**
  - **muproc-schedule**
  - **muproc-with-timeout**
  - **muproc-with-message-tag**
  - **muproc-exit, muproc-kill**
  - **muprocn**

# Generic Behaviour

## Generic Server

# Early muproc idiom

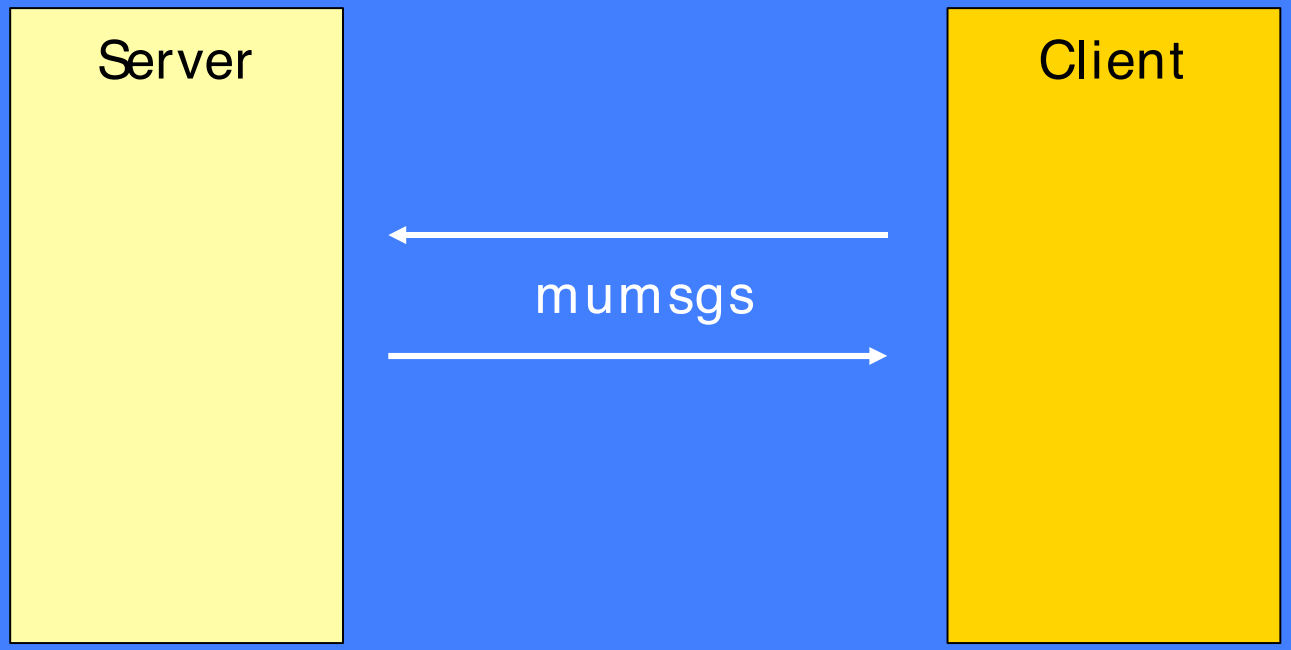
```
(defun %handle-msg-type1 (from arg1 arg2 arg3 tag)
  ...
  (mumsg-send from ... :tag tag))

(defun %handle-msg-type2 (from arg1 arg2 tag)
  ...
  (mumsg-send from ... :tag tag))

(defun some-server (...)
  ...
  (loop
   ...
   (muproc-receive (from)
     ((msgtype arg1 arg2 arg3 tag) (and (eq msgtype :type1) ...)
      (%handle-msg-type1 from arg1 arg2 arg3 tag))
     ((msgtype arg1 arg2 tag) (and (eq msgtype :type2) ...)
      (%handle-msg-type2 from arg1 arg2 tag))
     (...))
    ...))
```

無

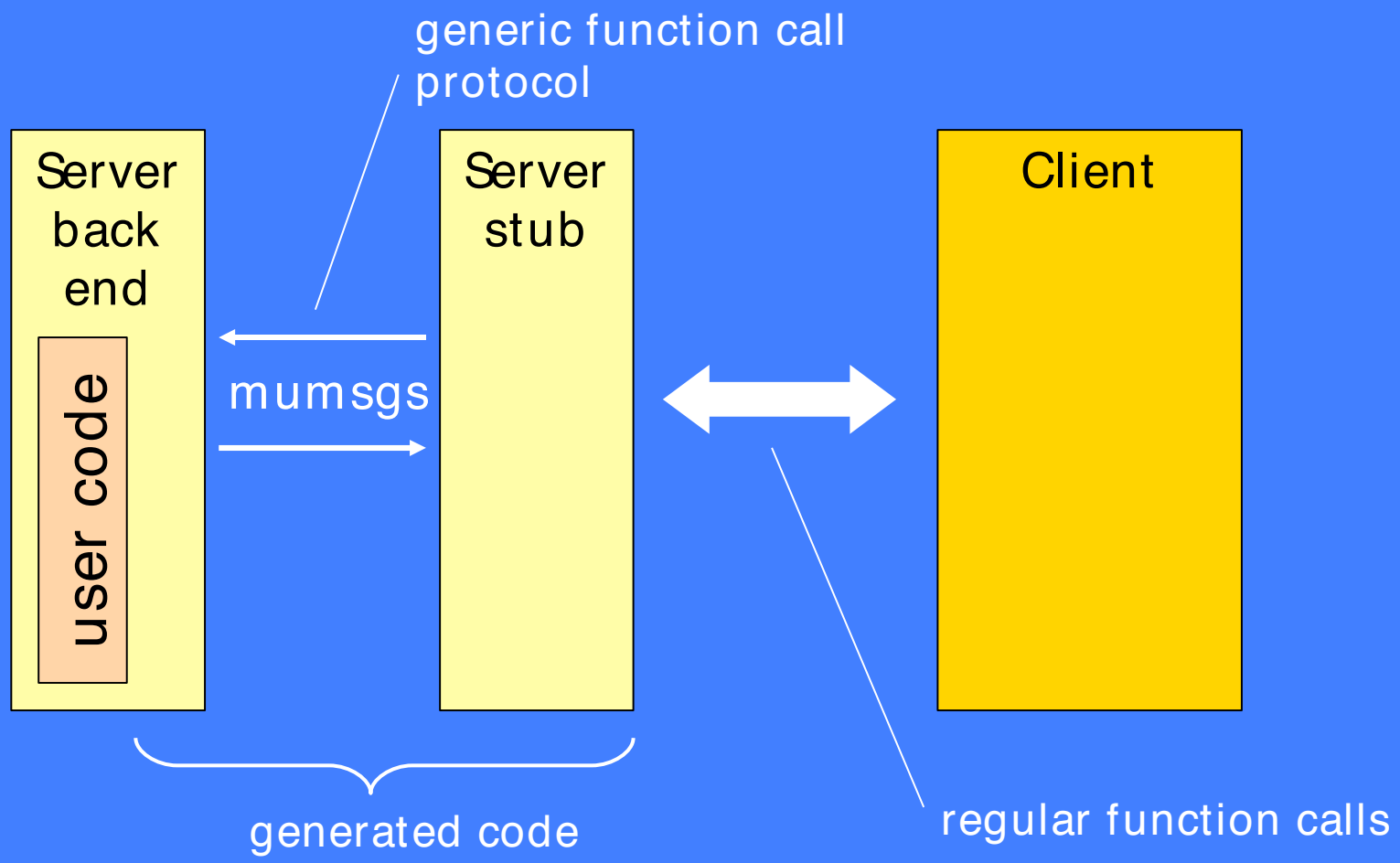
# Generic server

- generate message handling code from spec
  - concept / idea from Erlang/OTP
- interface defined by function
- combines generic and user code into a server
- synchronous, asynchronous
  - **`muproc-define-CALL-handler`**
  - **`muproc-define-CAST-handler`**
- managed server life cycle
  - **`initialize`**, **`terminate`** call-backs
  - **`generic-server-start`**
- multiple server instances possible

# Protocol interface



# Generic Server



# Generic Behaviour

## Supervisor

無

# Supervisor

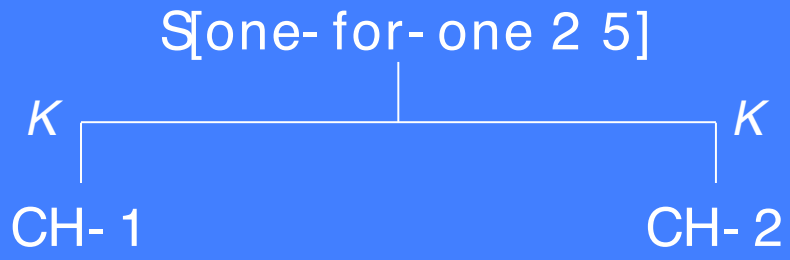
- generic supervision behavior
- implements 'process management policy'
  - watch a set of muprocs, restart them if the fail
  - according to 'supervision spec'
  - concept / idea from Erlang/OTP
- policies: *all-for-one, one-for-one*
- terminating children – *kill, <period>*
  - using *muproc-kill* or *muproc-exit*
- supervisors can watch other supervisors
  - nest to build any 'supervision tree'

無

# Supervisor child

```
(defun super-child (period sym)
  (out "Starting.")
  (unwind-protect
      (loop do (sleep period)
            do (out "~a" sym))
    (out "Terminating.")))
```

# super-1

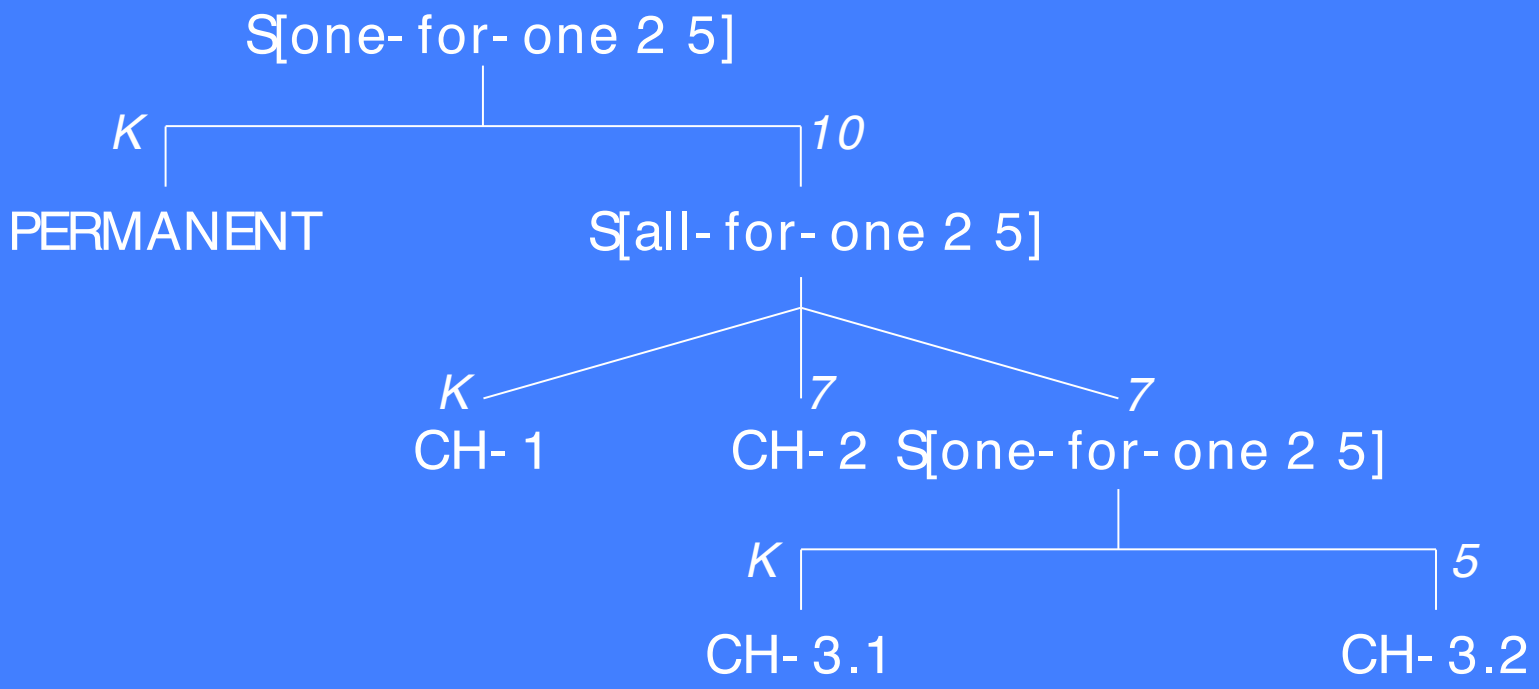



```
(supervisor
 :one-for-one 2 5
 (:permanent :kill (supervised-muproc
                    ch-1 #'super-child (list 3 'ch-1)))
 (:permanent :kill (supervised-muproc
                    ch-2 #'super-child (list 2 'ch-2))))
```

# super-2


```
(supervisor
 :all-for-one 2 5
 (:permanent :kill (supervised-muproc
                    ch-1 #'super-child (list 3 'ch-1)))
 (:permanent 10 (supervised-muproc
                 ch-2 #'super-child (list 2 'ch-2))))
```

# super-3


S[one-for-one 2 5]
K
10
PERMANENT
S[all-for-one 2 5]
K
7
7
CH-1
CH-2
S[one-for-one 2 5]
K
5
CH-3.1
CH-3.2

無

# Experiences so far

- cl-muproc used in on-line betting project at Mu
  - 14 muprocs total
    - 9 are derived from Generic Server
    - 4 are Supervisors
      - supervision tree 3 levels deep
      - 3 'all-for-one', 1 'one-for-one'
    - 1 'basic' muproc
  - generic behavours are a huge win
- Muproc has performed well in project
  - offers a clear way to organize system
  - muproc isolation provides strong separation of concerns
  - adequate performance
- cl-muproc is not Erlang
  - obviously
  - processes heavier, ultimately fewer
  - message handling response time not as good
    - but seems on par with Java, C#

無

# Muproc status, plans

- Current muproc
  - Lispworks only – Linux, Windows (Mac?)
  - Performance adequate for current use
  - ~2500 LOC
- Wish list
  - Distribution!
  - Support for other implementations
  - Generic Finite State Machine
  - Documentation (in progress)
- Now BSD licensed

# Questions?