CL-MW

The Development of a Master/Worker Framework in Common Lisp

By:
Peter Keller
(pkeller@sift.net)

March 5th, 2012
TC Lispers

CL-MW

- Rapid prototyping of master/worker style distributed algorithms.
- Scales to ~10K slaves, uses nonblocking network I/O.
- Management API to bound memory consumption while generating tasks.
- Easy integration with well known high throughput batch processing systems such as Condor, PBS, etc.
- Production of a single application binary.
- Robust task execution during slave failure such as unexpected slave death or hang.
- Well documented with manual and example programs.
- SBCL only (for now), but available in Quicklisp.

A CL-MW Application

- N Task Algorithms
- One Master Algorithm.
- One Slave Algorithm (Optional).
- Tasks and results are Common Lisp forms.
- Task flow.
  - Master send tasks to slaves.
  - Slaves convert them to results.
  - Slaves send results back.
- Only available network topology at this time is star.

The Task Algorithm

- (**define-mw-task-algorithm** XXX lambda-list
  &body body-forms)
- Creates:
  - Function: XXX
  - Macro: mw-funcall-XXX
  - Function: mw-set-target-number-for-XXX
  - Function: mw-get-target-number-for-XXX
  - Function: mw-pending-tasks-for-XXX
  - Function: mw-upto-target-number-XXX

Example

```
(define-mw-algorithm hello (str)
  (concatenate 'string “Hello world: “ str))
```

Adding a Task

```
(mw-funcall-hello (str)
  (&key sid tag do-it-anyway (retry t))
```

- Specify a **Task Policy** per task
  - **sid**: Either a SlaveID or NIL.
  - **tag**: User defined form associated with task.
  - **do-it-anyway**: T or NIL. If this was an ordered task, it might be ok to run it in an unordered fashion anyway.
  - **retry**: If the slave failed when it had the task, redo the task.

The Master Algorithm

- What does it do?
  - Process argv.
  - Partition problem space.
  - Create tasks.
  - Acquire/manage ordered slaves.
  - Call CL-MW master event loop function.
  - Process results.
  - Figure out what to do with tasks that can't run.
  - Compute final answer.
- (**define-mw-master** (argv) &body body-forms)

Example

```
(define-mw-master (argv)
  (mw-funcall-hello ("ABC"))
  (mw-master-loop)
  (let* ((result (mw-get-results))
         (answer (mw-result-packet result)))
     (format t "[answer]: ~A~%" answer)))
```

=> [answer]: Hello World: ABC

- Usually one puts a loop around (mw-master-loop)!

The Slave Algorithm

- What does it do?
  - Setup/Tear down of computing environment.
  - Allow arbitrary work to be done in between the processing of tasks.
- (**define-mw-slave** (argv) &body body-forms)
- Example:

```
(define-mw-slave (argv)
  (slave-loop-simple))
```

- **Optional!**

Producing a Binary

- sbcl --disable-debugger
- (**mw-dump-exec**)
- Parses ldconfig -p to find all libraries.
- **sb-sys:*shared-objects***
  - Copies any found libraries from original location to cwd.
  - Changes in-memory paths to point to cwd version.
- **cffi:*foreign-library-directories***
  - Push cwd onto it.
- (**save-lisp-and-die** …)

Running It

- Decouple resource management from execution.
- The Grid is a hostile place.
  - Machine problems.
  - Poor availability of installed lisp versions.
  - Network timeouts.
  - Authentication/Method to execute on machine.
  - Many more!
- Use tools that already exist: Condor, PBS, etc.
- Use a Resource File!

A Resource File

```
(:computation-status :in-progress )

(:timestamp 3488766071)

(:member-id "default-member-id" )

(:update-interval 300)

(:slaves-needed 1000)

(:slave-executable
  ("/home/psilord/bin/a.out"
  ("/home/psilord/bin/libiolib-syscalls.so")))

(:slave-arguments
  ("--mw-slave" "--mw-master-host" "black"
   "--mw-master-port" "47416"))
```

Batch Systems

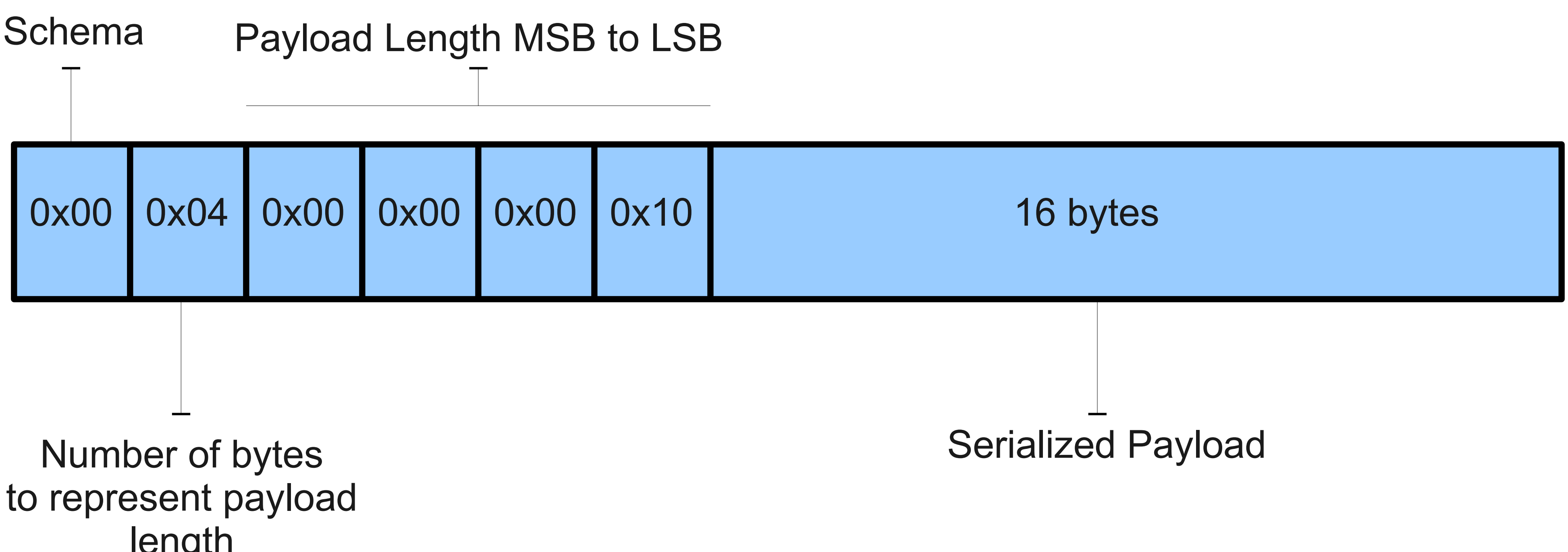
- “Glue Scripts” read the resource file.
- Knows where to find slaves, how many the master needs, how to invoke the slaves.
- Get features for free:
  - Starting executables, Managing user identity.
  - File movement, Secure credential management.
  - Detecting and killing runaway processes.
  - Managed Slave resource consumption in pools.
  - Access to other resources through other batch systems like Amazon EC2, Globus, etc.
  - Many more!

Supplied Examples

- Hello World
- Ping
- Monte-Carlo Computation of PI
- Argument Processing
- Higher Order

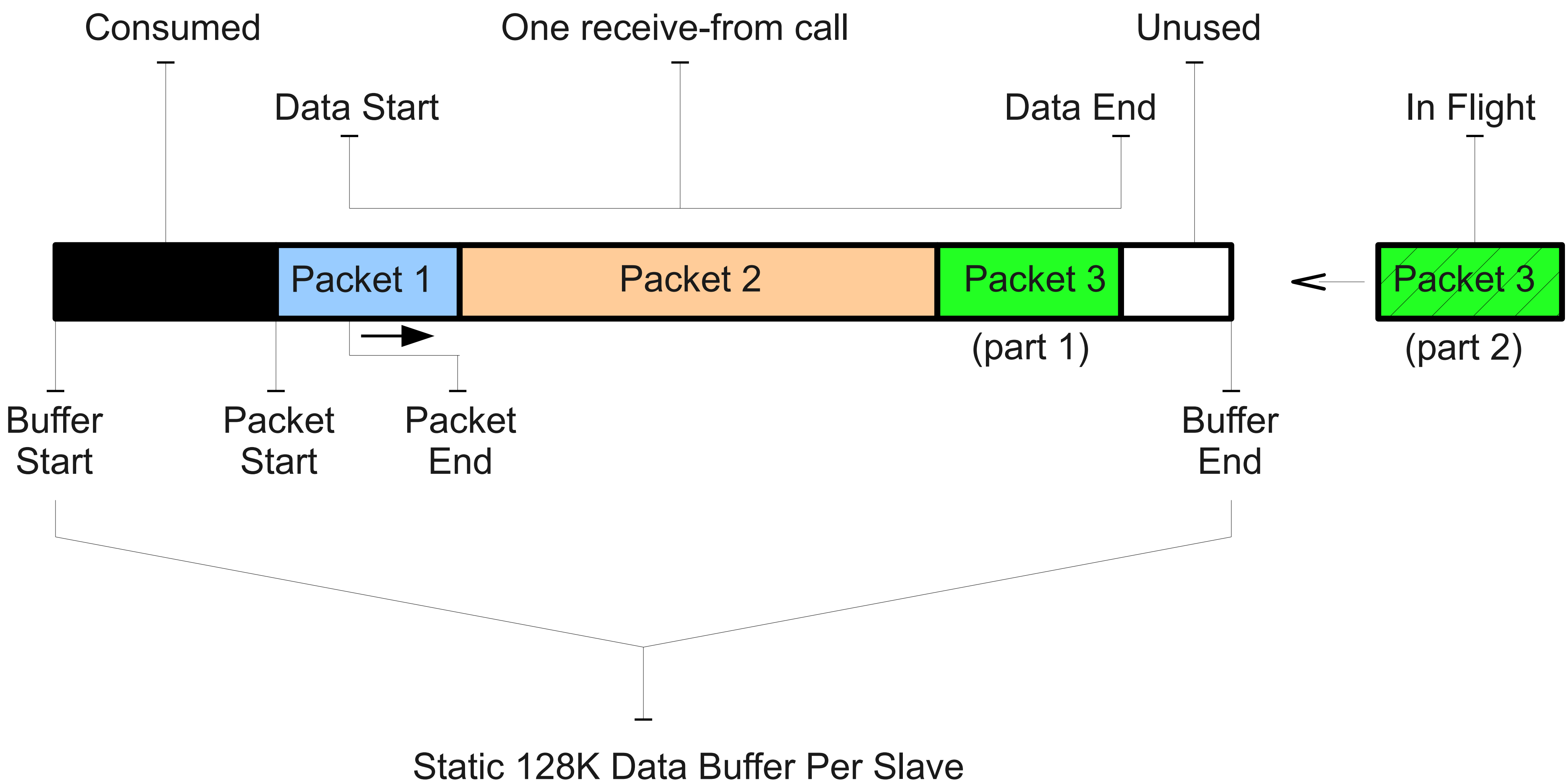
Internals: Network I/O

- Built on top of IOLib's multiplexing I/O.
- Layered read/write packet buffers on top.
- Example packet:



Internals: Network I/O

- Efficient reading and data buffer reuse:



Internals: Scaling

- Asynchronous network protocols on top of nonblocking calls.
- IDs are strings instead of interned keywords

```
(loop :repeat (expt 10 10) :do
   (intern (symbol-name (gensym)) :keyword))
;; BOOM!
```

- **One** object per task or slave, **many** references to them.
- Hash tables everywhere, so far it has been sufficient.
- Graham's queue implementation is space efficient enough.
- Hard limits on read buffers.
- No complex statistics
- Task target numbers.

Limitations

- No security mechanisms.
- No check-pointing.
- Hard network port limits.
- No proxy masters.
- Slaves can't find a new Master.
- Task/Result batching not dependent on task algorithm needs, but hard coded.

Future Work

- Check-pointing (user API?)
- Task Speculation (almost done).
- Control connection to Master Process.
- Audit log in lisp format.
- Libraries of useful parallel algorithms.
- Functional form of task submission.
- Higher order task flow descriptions:
  - Asynchronous/nonblocking large data/file transfer.
  - Formalization of task/result data flow across multiple task algorithms.

Thank you!

- You can find CL-MW at:
  http://pages.cs.wisc.edu/~psilord/lisp-public/cl-mw.html
- Questions?