libmilter implementation

“pool of workers” or “one thread per connection” ?

Based on an idea from Claus Assman

Jose-Marcio.Martins@ensmp.fr

Summary

- Introduction – Why a pool of workers is better ?
- Current libmilter
- Intermediate version
- Pool of workers - 1st version
- Pool of workers - 2nd version
- Current status

...

For example, developers used to believe that a threads stack was an ideal place to store application session state. However, experience has proved that it is generally better to hold session state in well-designed data structures, and deploy a worker thread pool to deliver scalable, predictable performance.

Note : Here, better is used in terms of performance. It is much easier to understand a program in which all state is represented as threads stack state. This also makes tools like pstack(1) very useful for debugging. But such implementations do not scale well as the number of threads increase. For instance, a Web application that provides a dedicated thread for every user session will suffer cache and memory management issues as its mean working set increases with load.

Multithreading in the Solaris Operating Environment - A Technical White-Paper – Sun Microsystems

j-chkmail data

```
martins@paris:~> j-printstats -q -l 6h
...
 First Connection : Mon Jul 21 10:13:47 2003
 Last Connection  : Mon Jul 21 16:13:38 2003
 Connections      : 5453
 Duration (sec)   : 0.005 11.726 3617.055 119.385 (min mean max std-dev)
 Work (sec)       : 0.005  0.043    2.402   0.087 (min mean max std-dev)
```

- **We remark :**
  - ✔ **Most of time, filter threads are idle, blocked in the kernel waiting for commands to run**
  - ✔ **Potential mean number of threads on the filter :**
    - One thread per connection : (5453/21600) * 11.726 = 2.9
    - One thread per sm command : (5453/21600) * 0.043 = 0.01
    - Pool of workers : N (fixed)

    OBS : mean number of threads = mean connection rate * mean stay time

One thread per connection - drawbacks

- **If the server is handling a very high traffic, threads loose time with scheduling issues instead of doing useful work.**

- **Within a milter filter, most of the time, threads are blocked in a system call, waiting for sendmail commands. Different OSs don't behave the same way.**

Current libmilter

- **LISTENER**
  - ✔ **forever**
    - **accept connection**
    - **launch session_handler**
  - ✔ **end forever**

- **SESSION_HANDLER**
  - ✔ **mi_engine**
  - ✔ **dispose_context**

WAITING STATES

A worker shall work, not wait !

- **MI_ENGINE**
  - ✔ **forever**
    - wait for sendmail command
    - read sendmail command
    - handle sendmail command
    - break if error or end of session
  - ✔ **end forever**

Libmilter – intermediate version

1. **MI_ENGINE – potential worker**
2. **Put waiting states out of mi_engine – no waiting states inside session handling**

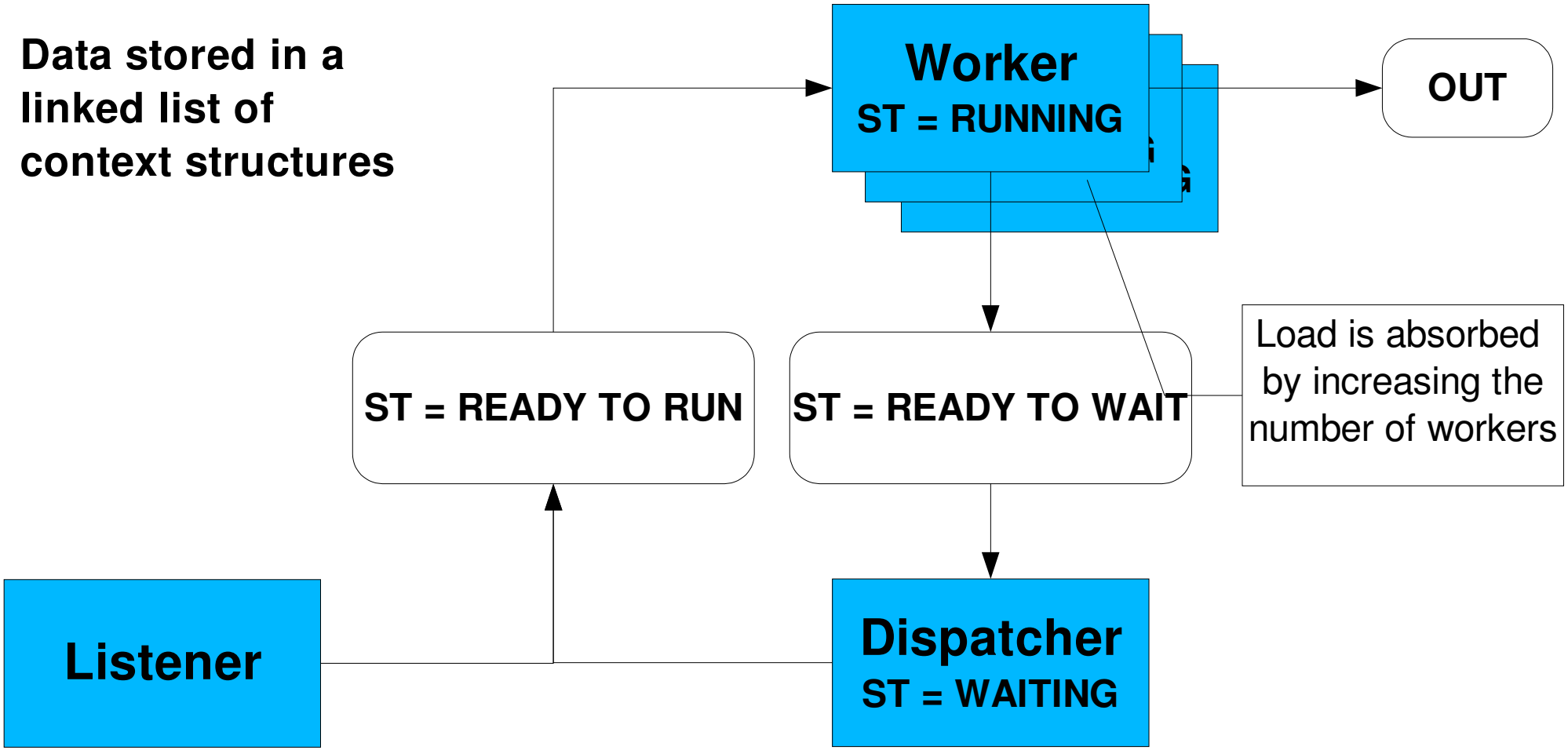
- **SESSION_HANDLER**
  - ✔ **forever**
    - wait for sendmail command
    - mi_engine
    - break If result different MI_CONTINUE
  - ✔ **end forever**
  - ✔ **dispose_context**

- **MI_ENGINE**
  - ✔ **forever**
    - read sendmail command
    - handle sendmail command
    - break if error or end of session
    - break if next command not ready
  - ✔ **end forever**

Waiting States

Libmilter – pool of workers - 1st version

Data stored in a linked list of context structures



Libmilter – pool of workers - 1st version

1. **Store session context in a global linked list, e.g., not at some thread stack**
2. **Create task dispatcher thread**
3. **Create worker thread**
4. **Assign and manage session states : waiting, ready to run, running, ready to wait**

- **SESSION_HANDLER**
  - add context to linked list
  - context state <- “ready to run”
  - Launch new worker if no idle worker

- **WORKER**
  - **forever**
    - wait for task (pthread_cond_wait)
    - context state <- “running”
    - call mi_engine
    - If result == MI_CONTINUE
      - context state <- “ready to wait”
    - else
      - Context state <- “end”
      - Cleanup context
    - endif
    - break if idle workers > 1
  - **end forever**

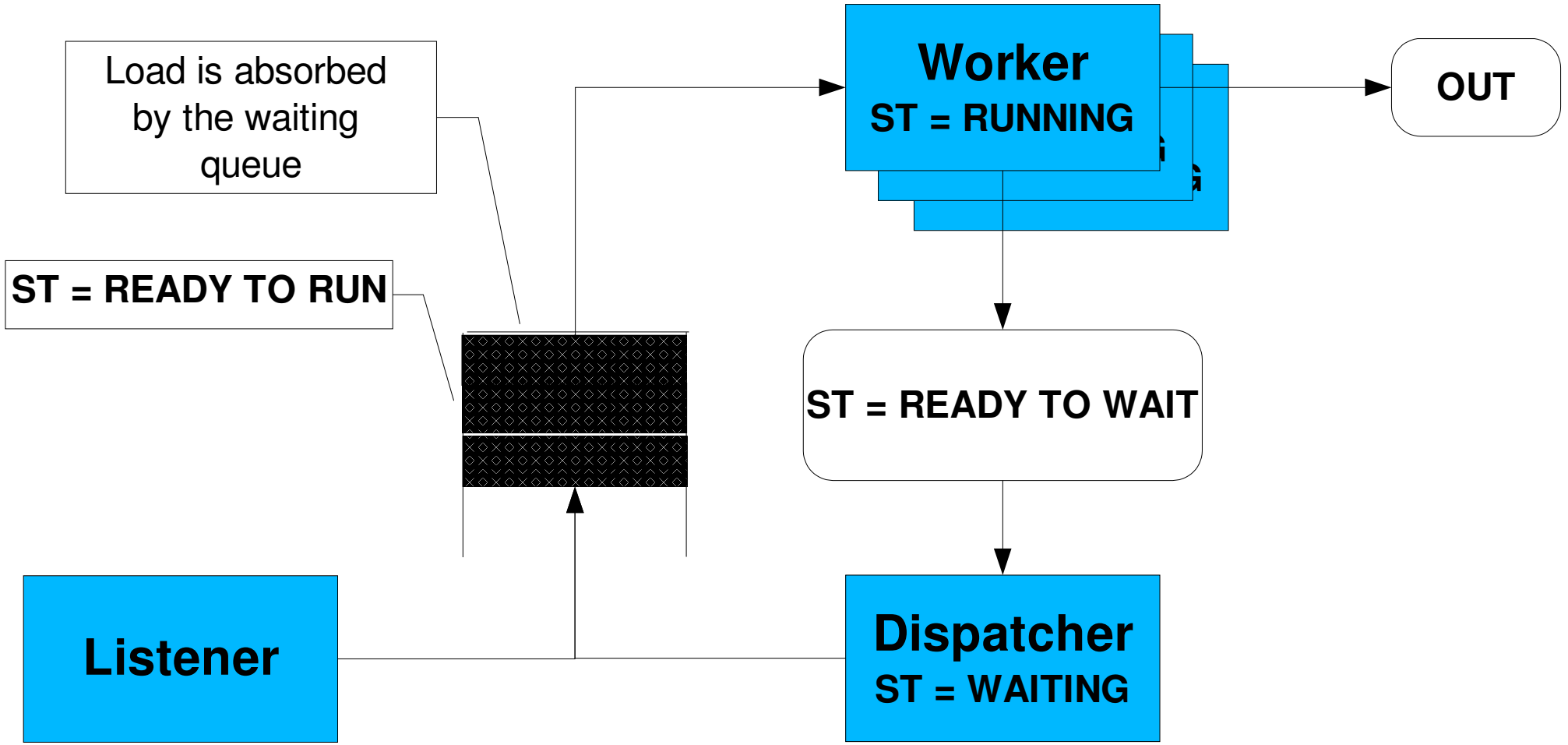
Libmilter – pool of workers - 1st version

- **DISPATCHER**
  - **forever**
    - Select all context sessions with state = “waiting” or “ready to wait”, and add their FD to to wait set. Set state to “waiting”
    - wait for FD (poll/select) event or timeout
    - If timeout continue
    - for each FD ready in wait set
      - State <- “ready to run”
      - If no idle worker
        - launch new worker to handle this task
      - else
        - signal worker (pthread_cond_signal)
      - endif
    - end for each
  - **end forever**

Libmilter – pool of workers - 1st version

- **At some moment, the number of workers equals the number of tasks ready to run. The number of threads isn't bounded.**
- **Question : what's the best way to interrupt and add a "ready to wait" session context to the dispatcher when it's in waiting for an event ?**
  - ✔ **Asynchronously – interrupt the dispatcher**
    - Interrupt poll/select by sending SIGINT to the thread (probably not a good idea)
    - Dummy file descriptor – write to it – shall manage concurrent access to file descriptor
  - ✔ **Synchronously – do nothing - wait poll/select timeout or fd ready**
    - Select a good timeout – maybe 200 ms – 500 ms ?
    - If traffic is low, waiting 200 ms isn't a real problem
    - If traffic high enough, dispatcher will probably be interrupted by a FD event before 200 ms

Libmilter – pool of workers - 2nd version



Libmilter – pool of workers - 2nd version

Goal : to bound number of threads

1. **Create a waiting queue with "tasks ready to run"**
2. **Probably better using semaphores instead of pthread_cond_wait (producer/consummer)**

- **SESSION_HANDLER**
  - add task to ready tasks queue
  - context state <- "ready to run"
  - release task (sem_post)

- **WORKER**
  - **forever**
    - wait for task (sem_wait)
    - get task from waiting queue
    - context state <- "running"
    - call mi_engine
    - If result == MI_CONTINUE
      - context state <- "ready to wait"
    - else
      - context state <- "end"
      - cleanup context and close
    - endif
  - **end forever**

Libmilter – pool of workers - 2nd version

- **DISPATCHER**
  - **forever**
    - Select all sessions with state = “waiting” or “ready to wait”, and add their FD to wait set
    - wait for FD (poll/select) event or timeout
    - If timeout continue
    - for each FD ready in wait set
      - add task to ready tasks queue
      - context state <- “ready to run”
      - release task (sem_post)
    - end for each
  - **end forever**

Open questions

- **Open issues :**
  - ✔ **Management of many file descriptors (one per session) by the dispatcher seems too complicated**
  - ✔ **Maybe it could be better to multiplex sm <-> libmilter communications over the same file descriptor**
  - ✔ **Secondary effect : interrupting poll/select is no more a problem**
  - ✔ **Sendmail 8 issues**
    - One process per connection
    - Concurrent access management on a single file descriptor shared by many processes

Current status

- Current implementation :
  - ✔ One thread per sendmail connection
- Patch available for 1st version – sendmail 8.12.7
  - ✔ http://www.sendmail.org/~ca/libmilter.worker.html
  - ✔ Used on our mail server for two months without problems
  - ✔ Not really clean programming
- 2nd version
  - ✔ Half done... 8-(