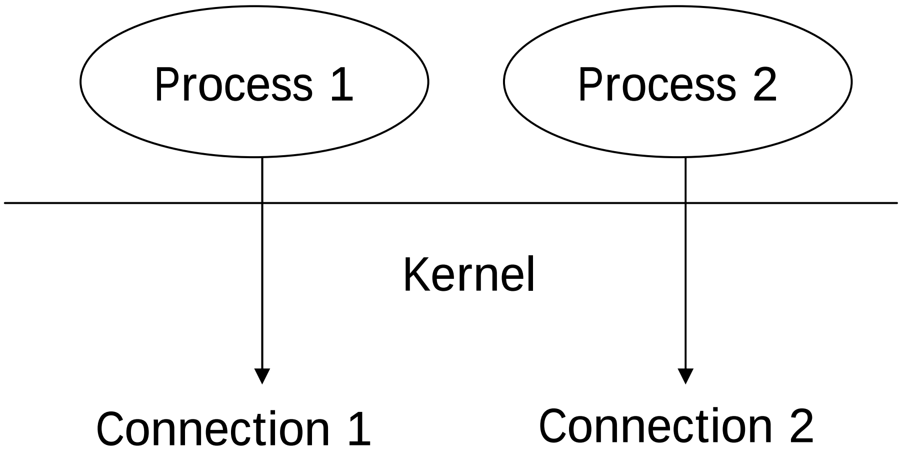
A Scalable Event Dispatching Library for Linux Network Servers

Hao-Ran Liu and Tien-Fu Chen

Dept. of CSIE
National Chung Cheng University

Traditional server: Multiple Process (MP) server

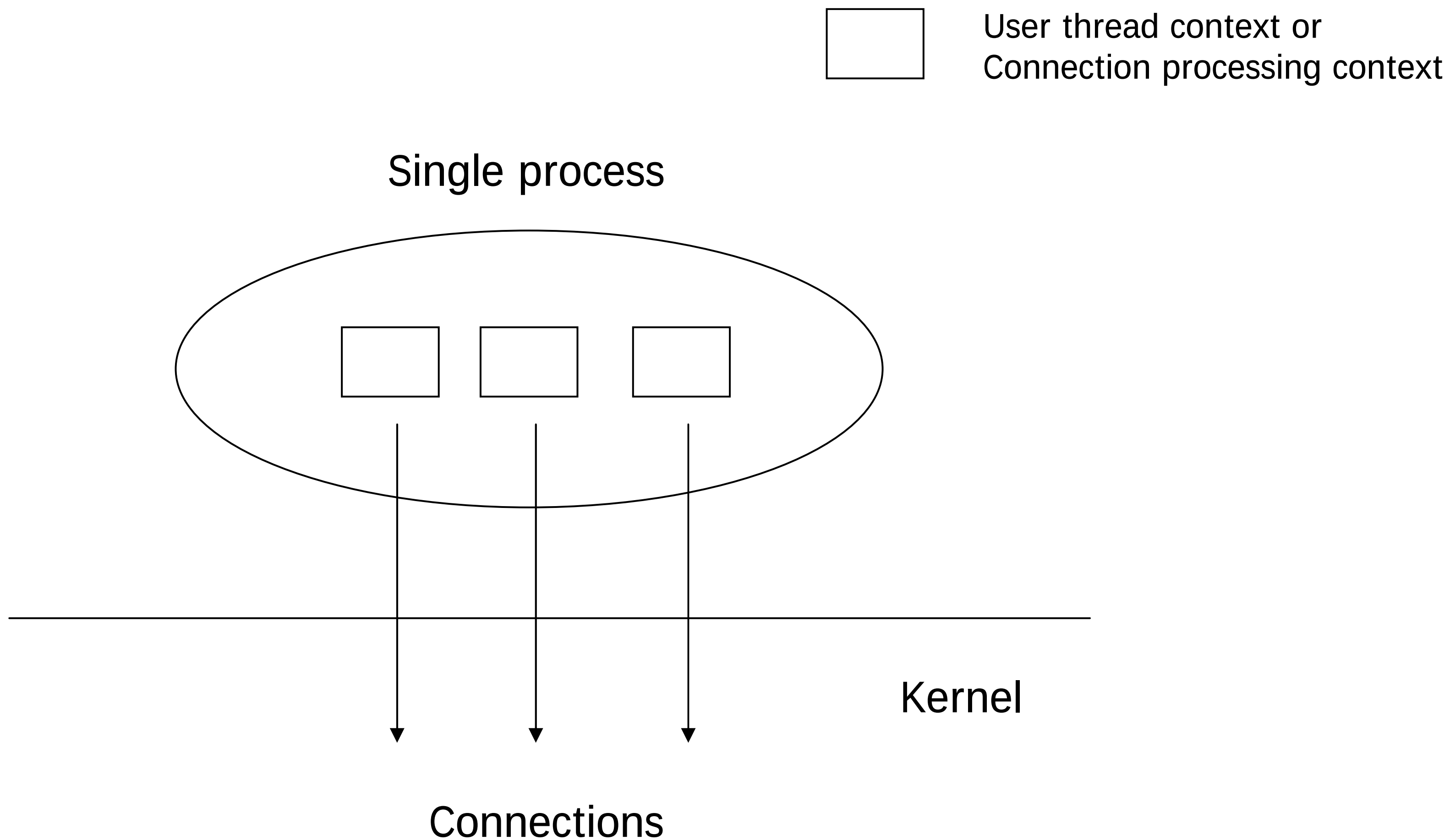
- A dedicated process to every connection.
- Concurrency is provided by OS.
- Disadvantage
  - Context-switching overhead
  - Synchronization overhead
  - TLB miss rate
- Example
  - Apache



Modern server:
Single Process Event Driven (SPED) server

- User level concurrency within a single process.
  - Nonblocking I/O
  - A mechanism to detect I/O events
- Disadvantage
  - Influenced by inefficient design of event dispatching mechanism in OS kernel
  - A single page fault or disk read suspends whole server.
- Example
  - Squid web cache

Event-driven server architecture



Problem statement and our goal

- The problem
  - Inefficiency design of event dispatching mechanism
- Our goal
  - Improve the performance of SPED servers in user-mode.

A list of event dispatching mechanisms

- Scalable
  - Devpoll (Solaris 8)
  - RT signals (Linux)
  - I/O completion port (Windows 2000)
- Non-scalable
  - select() (POSIX)
  - poll() (POSIX)

poll() and select()
Event dispatching mechanisms in Linux

- select() or poll() scales poorly with large set of file descriptors.

- 30% of CPU time are spent on select() in a overloaded proxy server (Banga 99)

Interface of select() and poll()

```
int select(int nfds,
           fd_set *readfds,
           fd_set *writefds,
           fd_set *exceptfds,
           struct timeval *timeout);

struct pollfd {
    int fd;
    short events;
    short revents;
}

int poll(struct pollfd *ufds,
         unsigned int nfds;
         int timeout);
```

Source of overhead on poll()

Step by step:

Poll() is called , array copy

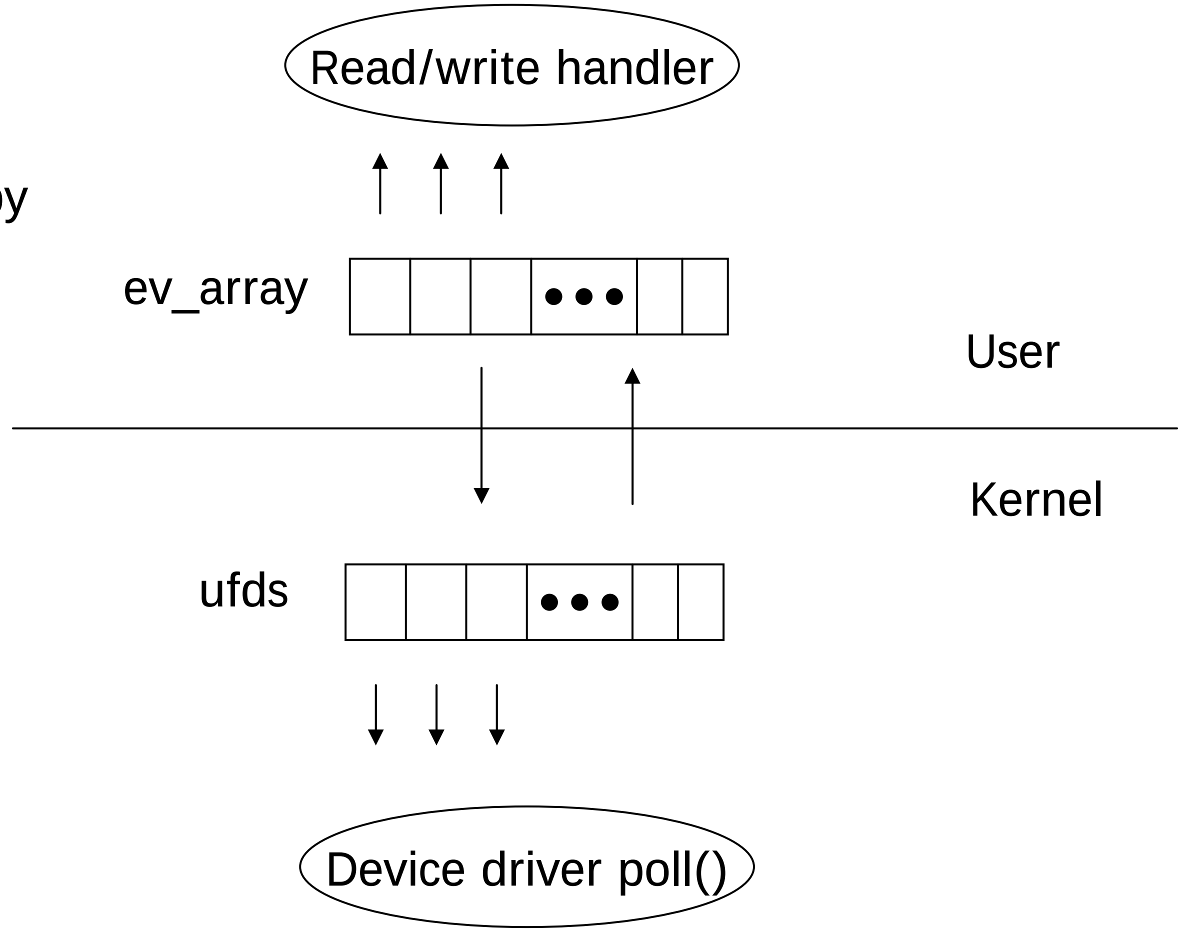
Linear scan in kernel

Driver poll callback

Poll() return , array copy

Linear scan in app.

Read or write handler



Source of overhead on poll()

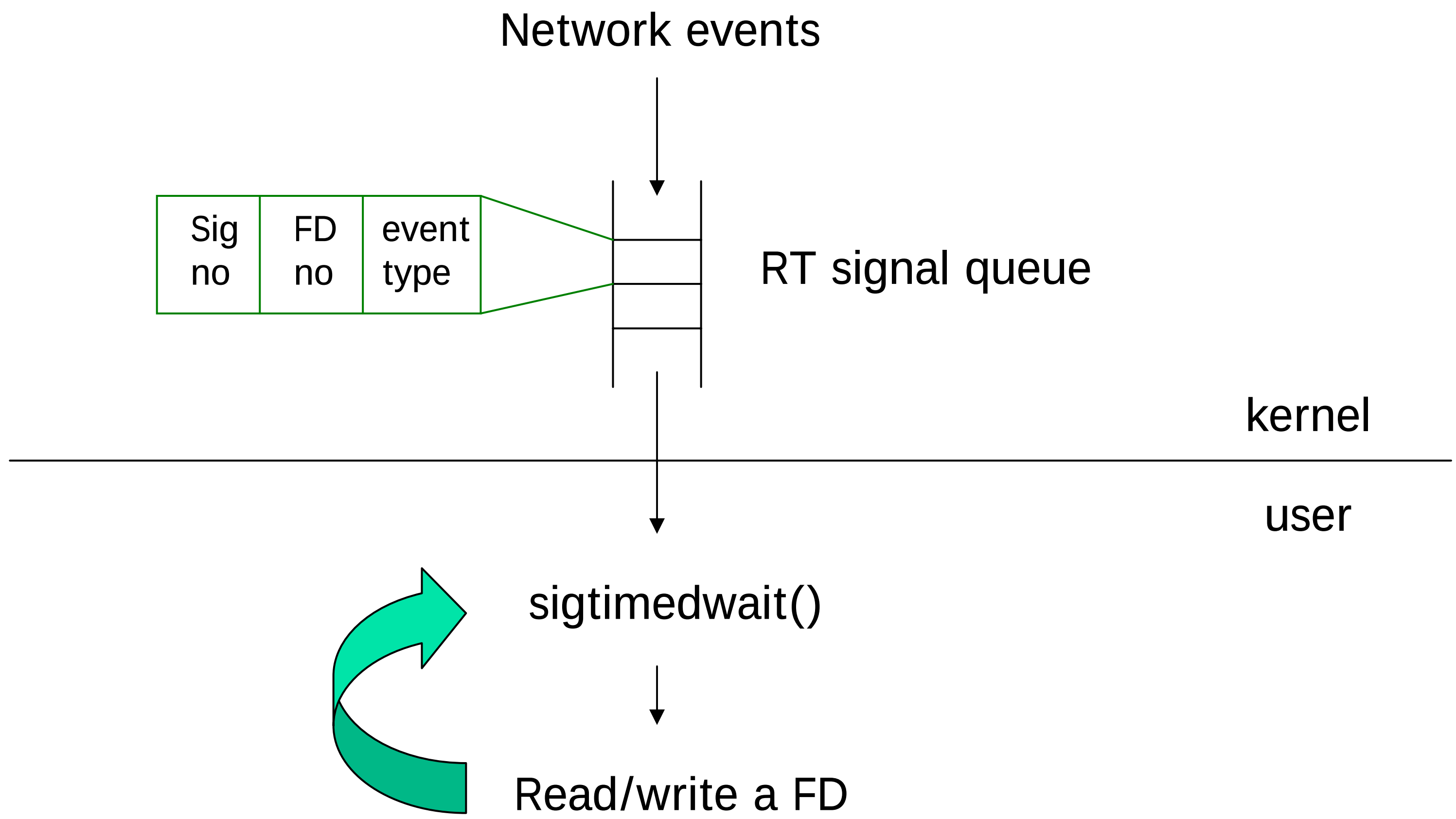
- Linear scan of ev_array and call device driver's poll callback function.
- Linear scan in server application to detect network events.
- Most work are wasted for idle connections.

POSIX.4 Real Time Signal

Event dispatching mechanisms in Linux

- POSIX.4 RT extension allows signal delivery with a payload.
  - sigwaitinfo(), sigtimedwait()
  - Payload can carry sender PID, UID, etc.
- Linux extension of POSIX.4 RT signal
  - Allow delivery of socket readiness via a particular real time signal.
- *Pro*
  - Official support in Linux kernel since version 2.4
- *Con*
  - Linux specific

Flowchart of POSIX.4 Real Time Signal



Problems in RT signals

- Edge-triggered readiness notification
  - Signal queue may contain multiple events of a FD
  - Stale event
- Kernel signal queue size limit.
  - 2048 entries
- RT signal queue overflow
  - Some connections will fall into deadlock state.

Solution to RT signal queue overflow

- Application solution
  - Server falls back to traditional select() or poll()
- Kernel solution
  - Signal-per-fd [Chandra 01]
    - Collapse events of the same file descriptor
    - (signal queue size == max no. of files a process can open) => no signal queue overflow

Our solution to reduce poll() overhead: Scalable event dispatching library

- Motivation
  - Web connections are idle most of the time.
  - Network events in a single HTTP transaction are bursty.
- Our strategy
  - Save the time wasted on idle connections for more useful works.
    - The frequency of calling poll() on a idle connection is decreased.
    - Average number of file descriptors at every poll() is decreased.
  - Exploit temporal locality among events in a connection to reduce the frequency of calling poll() on a FD.

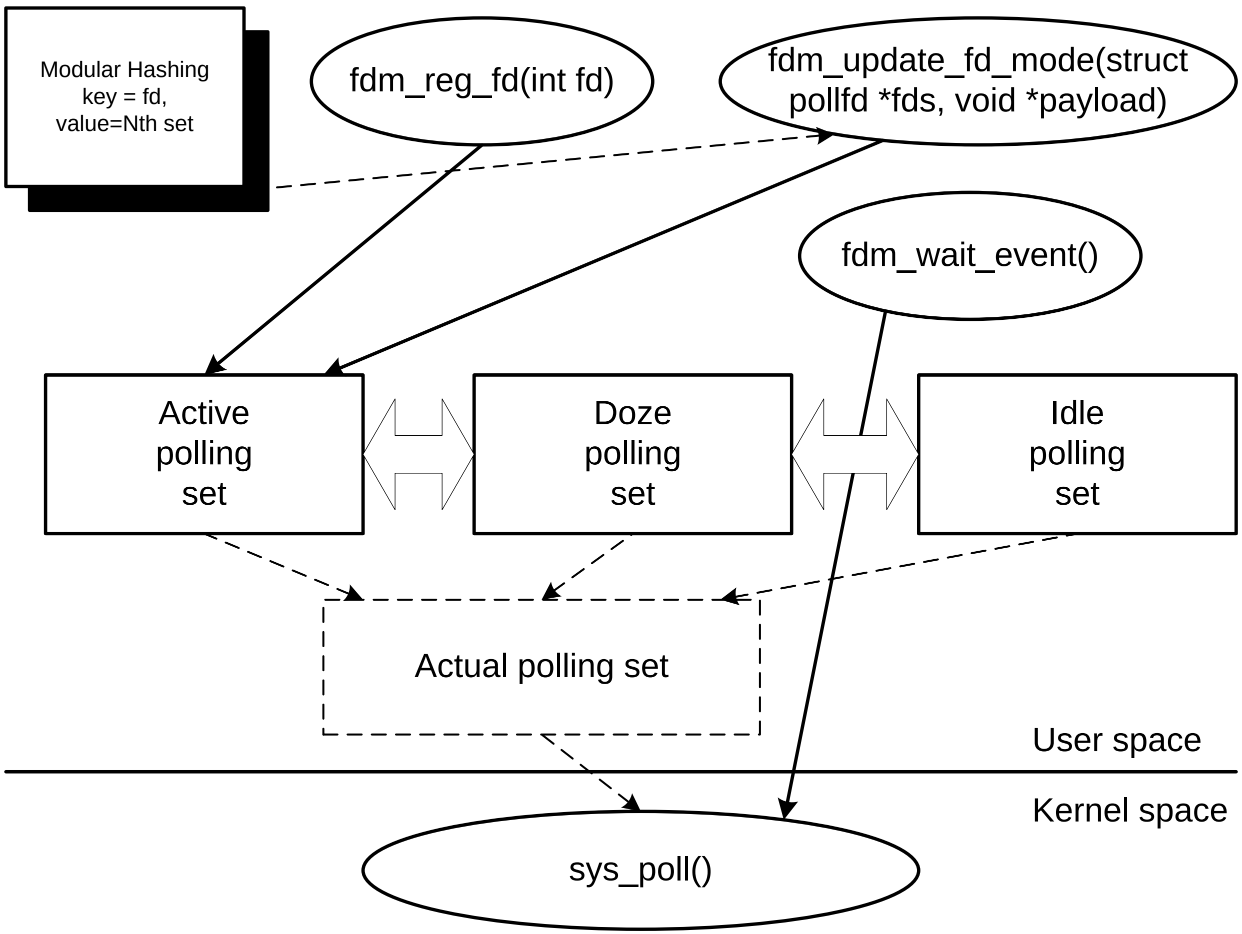
Live counter and polling sets: Keys to implement locality poll()

- A live counter is associated with a file descriptor. On every poll() to the FD:
  - If event is detected, counter increases.
  - If no event is detected, counter decreases.

- All FDs in a server are divided into three sets according to their live counter.

- The frequency of calling poll() on each set is different.

Architecture view of event dispatching library (FDM)



Library interface design logic

- Reduce copy of FD array
  - Interesting set are built gradually
  - Separate routine to fetch event
- Reduce linear scan of FD array in server code
  - Timeout timestamp and payload associated with a file descriptor is maintained in this library.
- Listening socket descriptor can be locked in the active polling set.

Proposed library interface

```
typedef struct {
   int fd;
   short events;
   void *payload;
} fdm_event_t;    // data structure return by wait event

int fdm_start();  // library init
int fdm_stop();   // library shutdown
int fdm_reg_fd(const struct pollfd *fds, int lock);
int fdm_unreg_fd(int fd);

// tell library a fd is read interest or write interest
int fdm_update_fd_mode(const struct pollfd *fds,
                       void *payload,
                       struct timeval *tv);
int fdm_wait_event(fdm_event_t *ev_array,
                   unsigned int array_size,
                   int timeout);
```

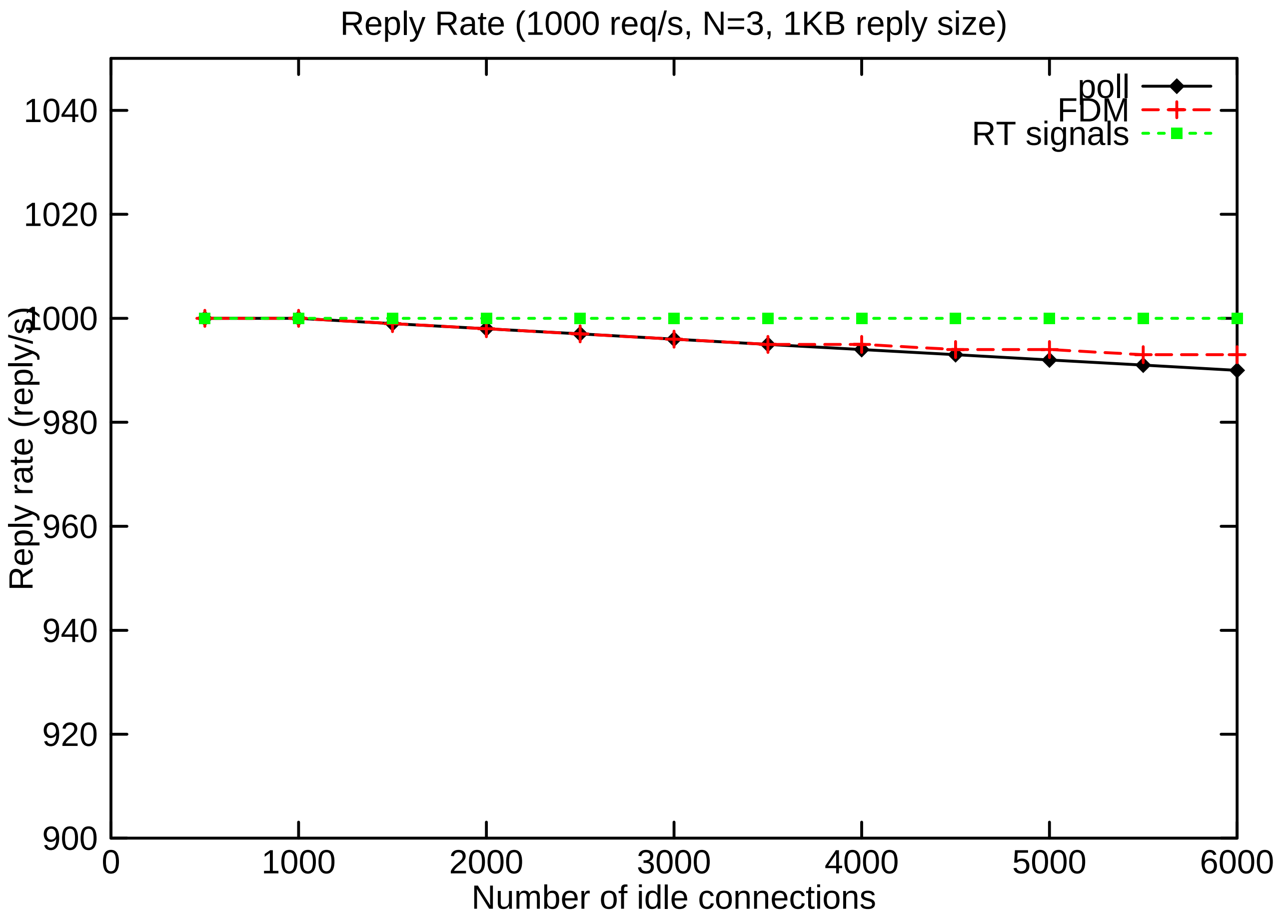
Performance evaluation

- Compare event dispatching mechanisms available in Linux 2.4, including our FDM library
- Test server
  - dphttpd + event-threading + FDM
  - All tests are under *Multi-accept* implementation
  - Pentium III 600MHZ, 128MB RAM
- Test Client
  - Httperf, HTTP workload generator
  - three Pentium III 1GHZ, 512MB RAM

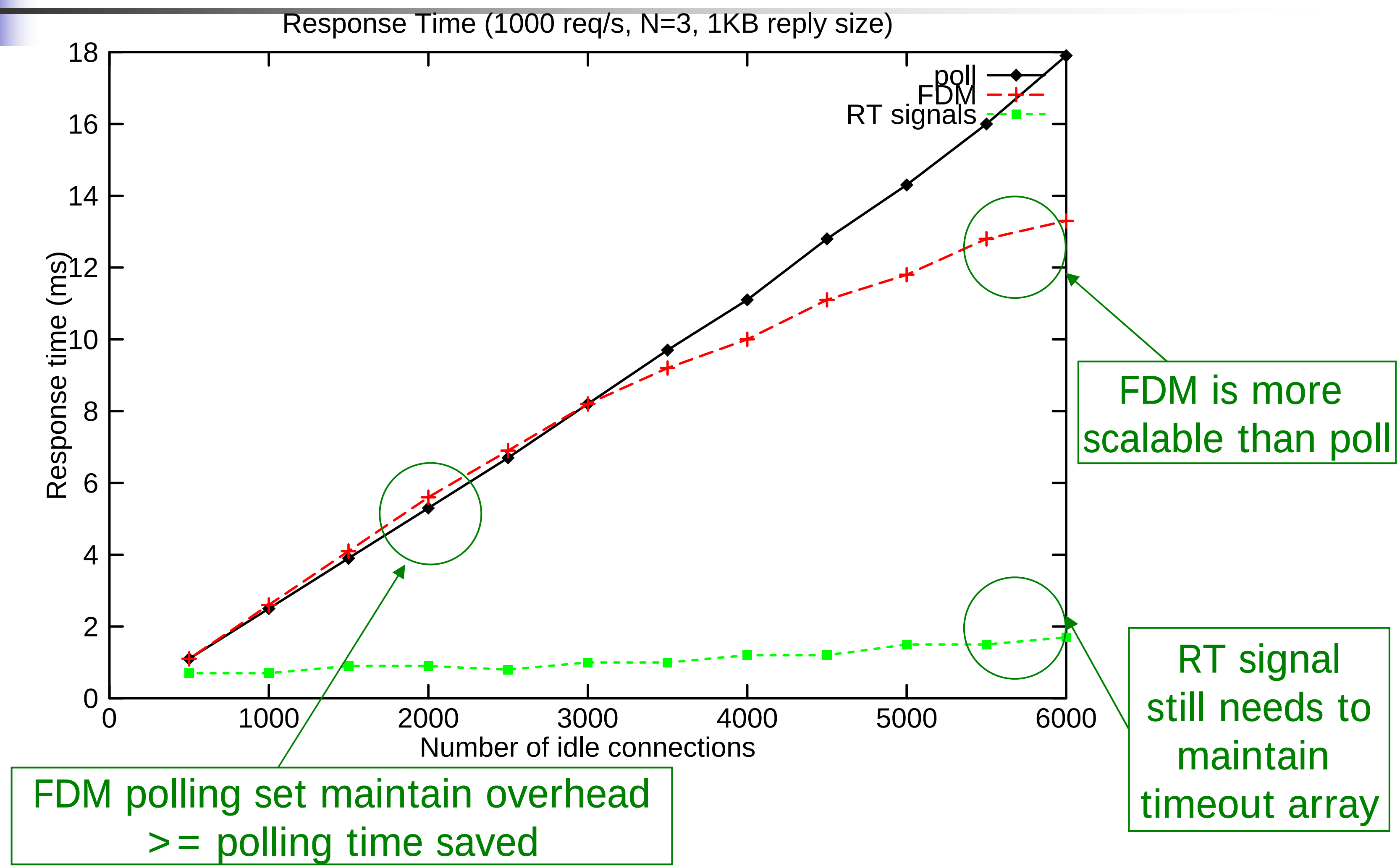
Dispatching overhead

- Goal
  - see the overhead under
    - Large number of idle connections
    - Request rate is fixed at a pretty light load.

Server reply rate with fixed light load



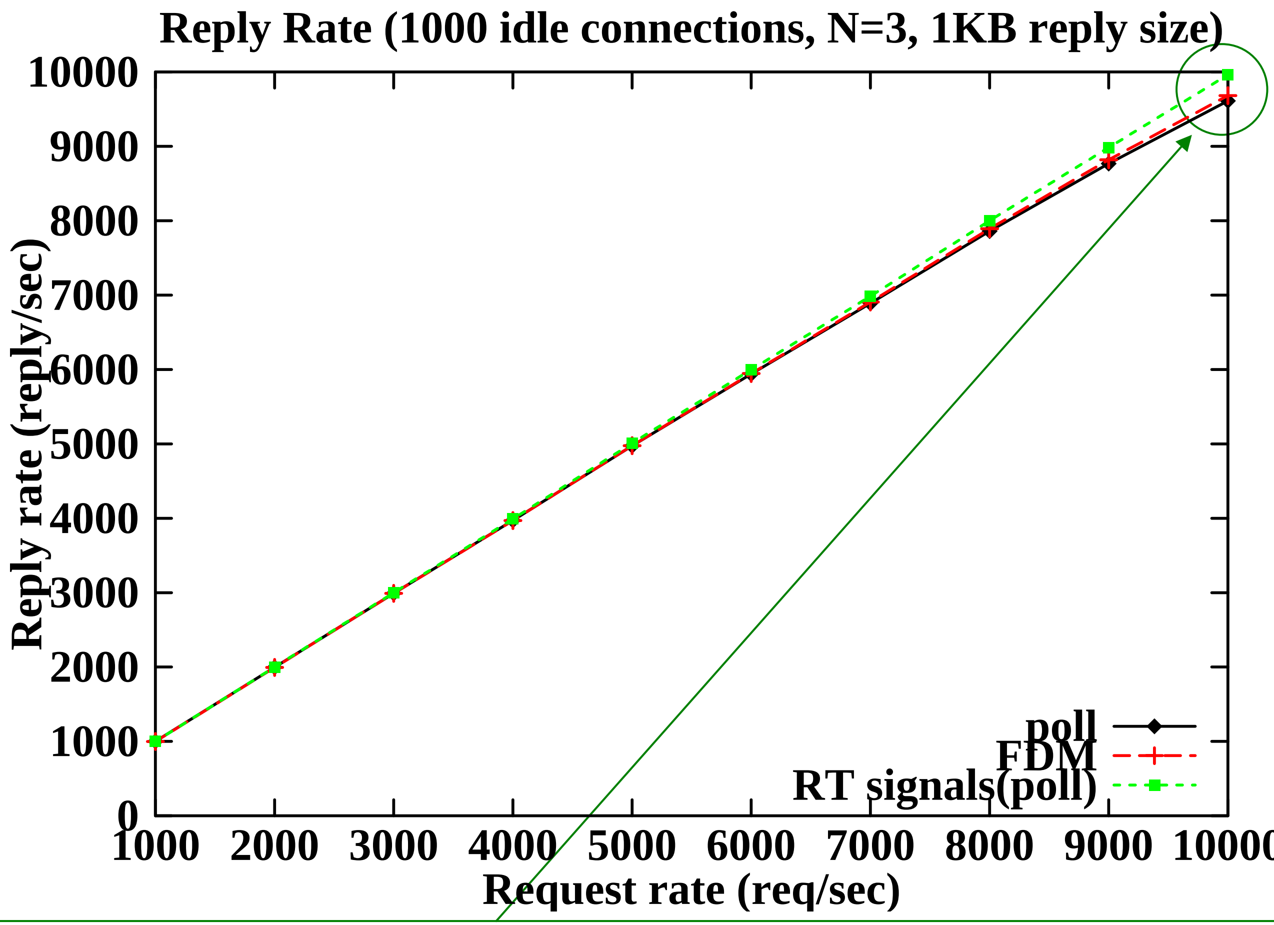
Server response time with fixed light load



Dispatching Throughput

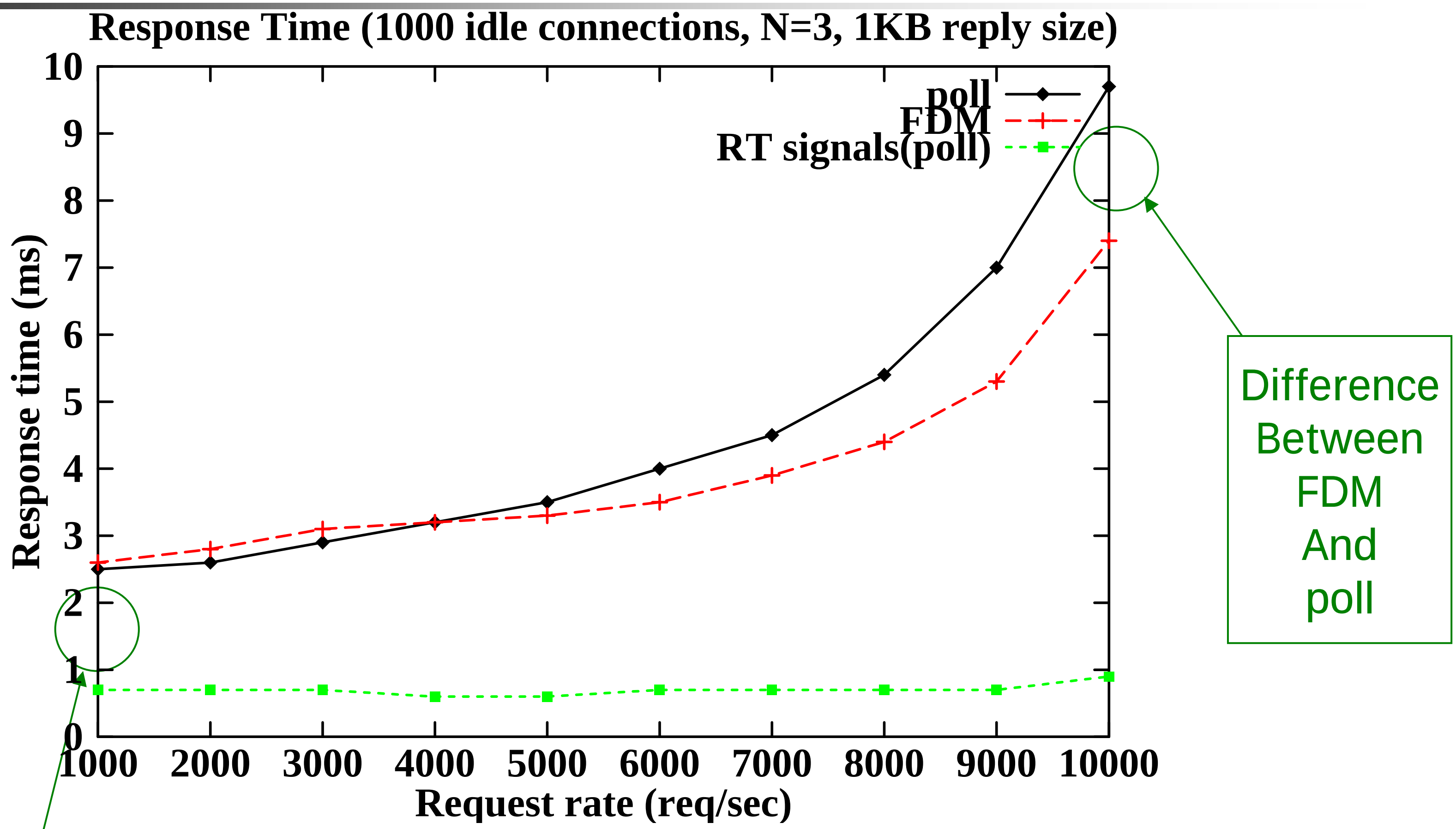
- Goal
  - See the throughput under
    - Fixed number of idle connections
    - Overloaded request rate

Server reply rate with 1000 idle connections



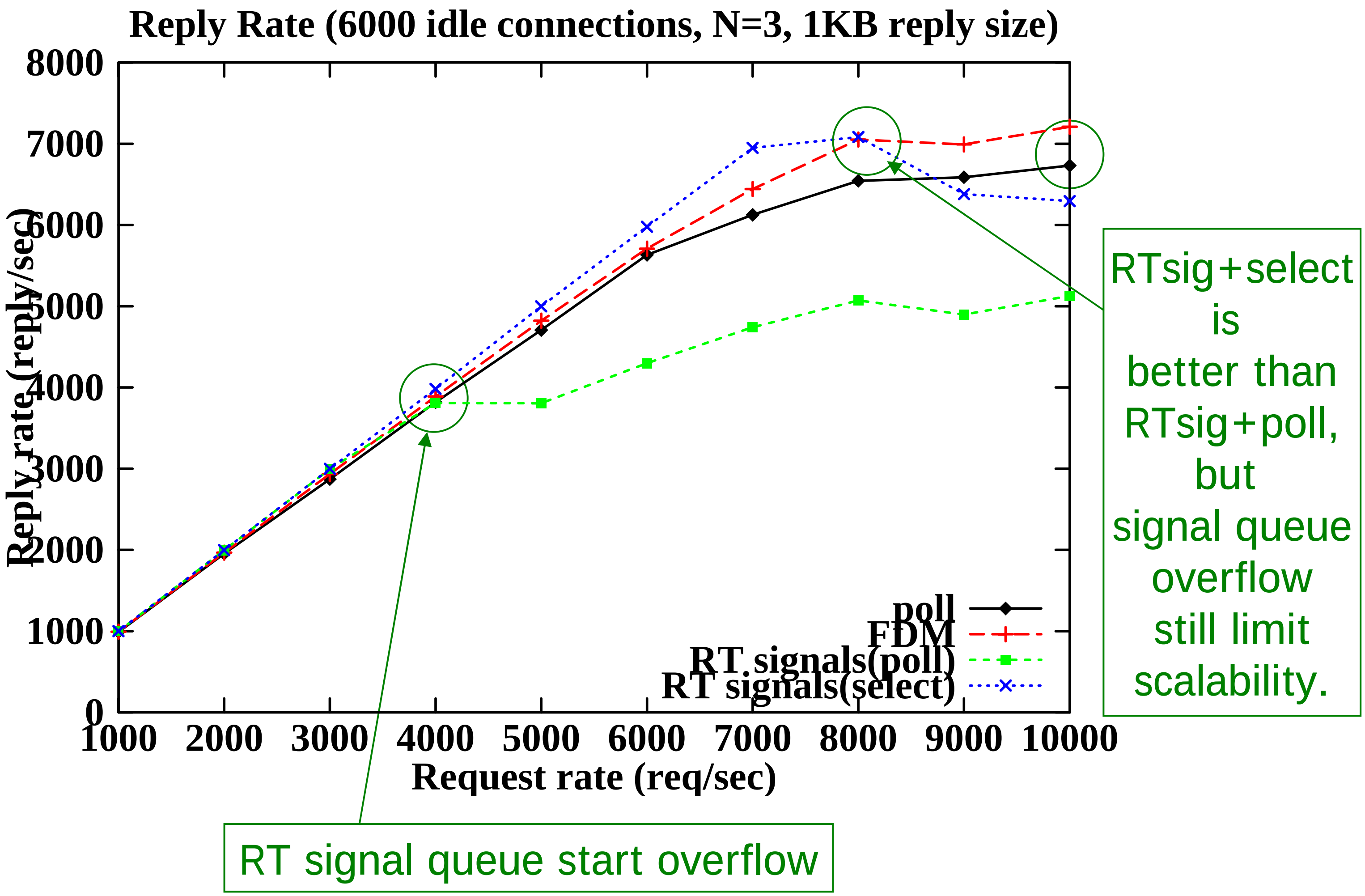
100Mb Ethernet saturated, poll performs well because of *multi-accept*

Server response time with 1000 idle connections

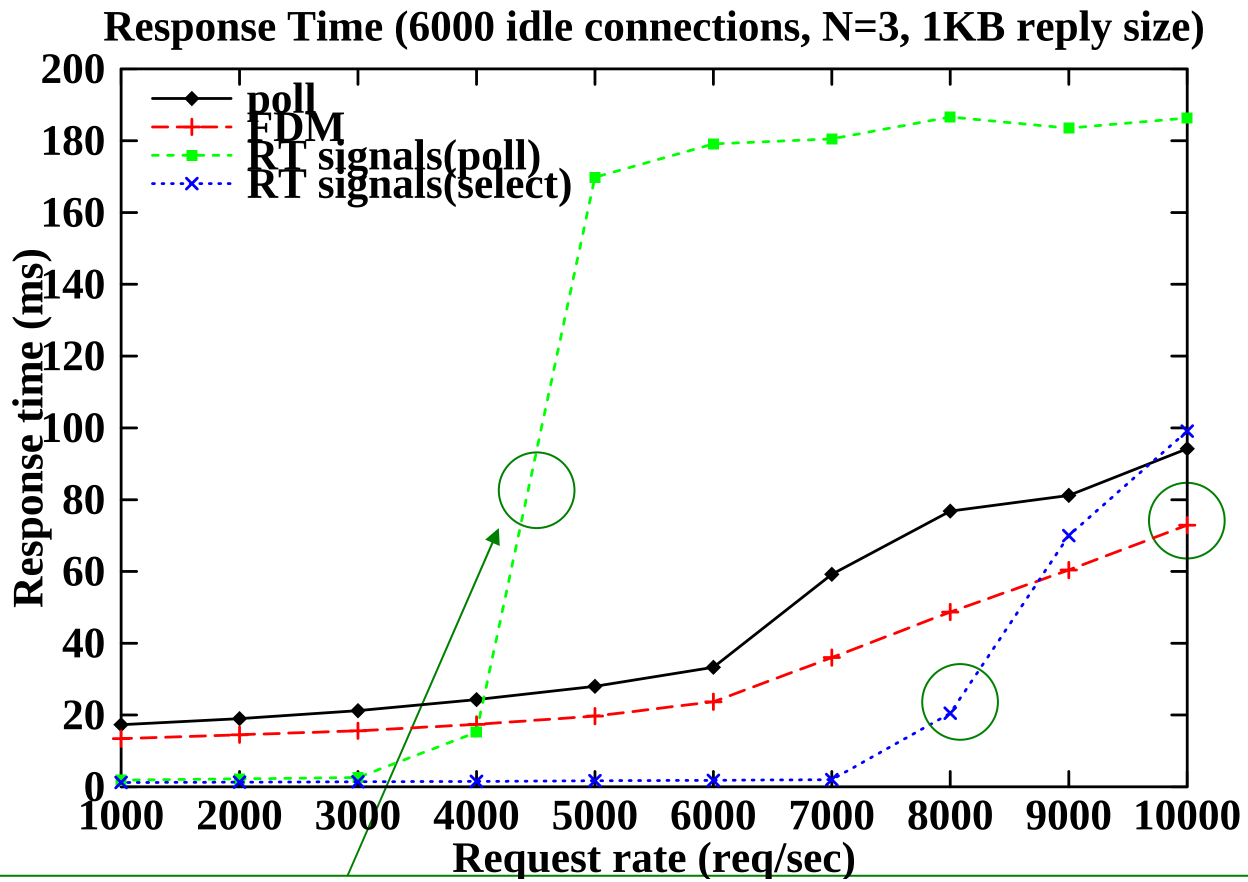


Even at light load, we can observe overhead of 1000 idle connection. FDM suffers too since it depends on poll()

Server reply rate with 6000 idle connections



Server response time with 6000 idle connections



Signal queue overflow, server fall back to poll(), poll overhead at 6000 idle connections is a lot larger such that this behavior can't be observed at 1000 idle connections.

Summary of performance evaluation

- At light load
  - FDM incurs low overhead
- At heavy load
  - FDM improves poll()
- RT signal queue overflow should be protected by select(), not poll()

Conclusion

- FDM library improves poll()
  - exploiting temporal locality property of events in a file descriptor.
  - Better performance than RT signals if RT signal queue overflowed.
- RT signal queue overflow recover
  - select() is a better choice.
- Availability of FDM and test web server
  - http://arch1.cs.ccu.edu.tw/~lhr89/fdm/

Backup slides follow

Keys to scalable event dispatching mechanisms

- Interesting set of file descriptors is built gradually inside kernel
  - Separate building of interesting set from event retrieval
  - Only return a file descriptor if there is a event.
- Collect event efficiently
  - Cache device driver poll result
    - Don't run driver's poll() at every poll()
  - Or, maintain a event queue and collect events gradually at every call to TCP/IP event handler.

Summary of event dispatching mechanisms

features / methods	Scalable to large set of file descriptors	Event collapsing	Dequeue multiple event per system call	Event queue overflow	When queue overflow, kernel fallback to traditional poll()	Return initial state when declare interest fd	Multiple interest sets maintained in kernel for per process
Select()	No	NA	Yes	NA	NA	NA	NA
Poll()	No	NA	Yes	NA	NA	NA	NA
/dev/poll	Yes	NA	Yes	NA	NA	NA	Yes
RT signals	Yes	No	No	Yes	No	No	Yes
RT sig-per-fd	Yes	Yes	No	No	NA	No	Yes
Declare_interest	Yes	Yes	Yes	Yes	Yes	Yes	No

Polling frequency of a set

- Depends on the default value of live counter
- Assume default value is *N*
  - On every fdm_wait_event()
    - Active polling set is polled
  - On every *N* fdm_wait_event()
    - Doze polling set is polled
  - On every *N^2* fdm_wait_event()
    - Idle polling set is polled

Server performance with different default value of live counter