Scalable Network I/O in Linux

Niels Provos, University of Michigan

provos@citi.umich.edu

Chuck Lever, AOL-Netscape, Incorporated

chuckl@netscape.com

Linux Scalability Project

Center for Information Technology Integration

University of Michigan, Ann Arbor

linux-scalability@citi.umich.edu

Overview

1. Introduction
2. POSIX RT Signals
3. Optimizations to poll()
4. Comparison: POSIX RT signals vs. poll()
5. Benchmark
6. Discussion
7. Conclusion

Introduction

- Traditionally server benchmarks focus on improving throughput for clients on a local area network,
- but high latency, low bandwidth connections significantly influence server scalability.
- Experts argue that a combination of I/O completion events and poll() are more scalable than today's servers.

Introduction

- In Linux, POSIX RT signals deliver I/O completion events; the signals can also be queued and delivered to an application one at a time.
- When the signal queue is exhausted, a server needs to switch to polling.
- An efficient poll() is thus still important.
- We found several areas to improve poll() when a large proportion of a server's connections are inactive.

POSIX RT Signals

- `phhttpd` is a static-content caching web server. It uses Apache as its back end.
- `phhttpd` uses Linux's POSIX RT signal mechanism:
  - An application assigns a signal number to a file descriptor: `fcntl(fd, F_SETSIG, signum)`
  - When `read()`, `write()`, etc... complete, the signal is raised.
- To avoid race conditions and complexity, `phhttpd` masks the RT signals and the kernel queues them.
- An application can pick them up with `sigwaitinfo()`, one by one.

POSIX RT Signals

- When the RT signal queue overflows, the kernel sends a `SIGIO`,
- then the application flushes pending signals by setting their handler to `SIG_DFL` and uses `poll()` to recover.
- Events queued before a socket is closed, remain in the RT signal queue,
- an application has to be careful to avoid inappropriate operations on the closed socket.

Optimizations to poll()

- Motivation for optimizing `poll()`:
  - Legacy applications can benefit with little modifications.
  - Even when using POSIX RT signals, `poll()` is still required for handling signal queue overflows.
- We made the following `poll()` enhancements:
  - Interface that maintains state in the kernel: no state has to be passed for every `poll()` invocation.
  - Device driver post completion events to `poll()`.
  - Result copying eliminated by sharing address space between kernel and application.

Optimizations to poll()

- An application needs to call `poll()` with a set of interests, *i.e.* file descriptors that might be ready for I/O.
- This mechanism does not scale well, since the entire set must be copied in and out of theb kernel.
- Banga et al. suggested splitting the interface into two parts:
  - `declare_interest` to incrementally build the kernel state,
  - a `poll` alike to return interests that have I/O ready.

Optimizations to poll()

- `/dev/poll` provides a declare interest interface in Solaris.
- We implemented `/dev/poll` for Linux.
- An application uses `write()` on `/dev/poll`
  - to add interests,
  - to remove them by setting `POLLREMOVE` in the `events` field of `struct pollfd`.
  - or to modify interests by specifying a file descriptor the kernel knows already.

Optimizations to poll()

```
struct dvpoll {
        struct pollfd *dp_fds;
        int dp_nfds;
        int dp_timeout;
}
```

- I/O events are polled with `ioctl` using the `dvpoll` struct.
- This interface scales well, because only the ready set is returned, which normally is small.
- The kernel keeps each interest set in a hash table for fast lookup.

Optimizations to poll()

- Using `/dev/poll`, a process is put to sleep until a relevant event occurs.
- The kernel has to scan all file descriptors in the interest set to check for status changes.
- We extended the `/dev/poll` implementation, so that device drivers can mark file descriptors that changed status with a hint.
- When there are a large number of high latency connections, the hints allow us to avoid unnecessary calls into the `poll()` functions of device drivers.

Optimizations to poll()

- There are no hints for a change from ready to not-ready.
- A cached result indicating readiness has to be reevaluated every time.
- To prevent modification of every device driver, the drivers indicate whether they support hinting.

Optimizations to poll()

- With increasing size of interests, the overhead to copy results back to the application increases.
- We created a memory map shared between application and kernel, that the kernel uses to report results from `poll()`.
- An extra `ioctl()` allocates room for a specific number of results.
- The mapping is shared by an `mmap()` call on `/dev/poll`

Comparison: POSIX RT signals vs. poll()

- How big a server workload is required before polling is more efficient than RT signals? If ever.
- How big is the performance difference between POSIX RT signals and `poll()`?
- We study these questions with the following benchmark.

Benchmark

- Our test harness consists of two Linux machines connected via 100 MBit/s Ethernet.
- The workload is driven by an Intel SC450NX with four 500 MHz Xeon Pentium III processors with 512 MByte memory.
- The web server runs on a 400 MHz AMD K6-2 processor with 64 MByte memory and a single 8 GByte 7.2 KRPM IDE drive.
- The server hardware is small, so that it is easy to drive the server into overload.

Benchmark

- The web servers are:
  - **thttpd**, a simple single-process event-driven web server,
  - **phhttpd**, an experimental server using POSIX RT signals.
- The benchmark is driven by **httperf** providing repeatable server workloads.
- We use only static content as it exercises the system components we are interested in.

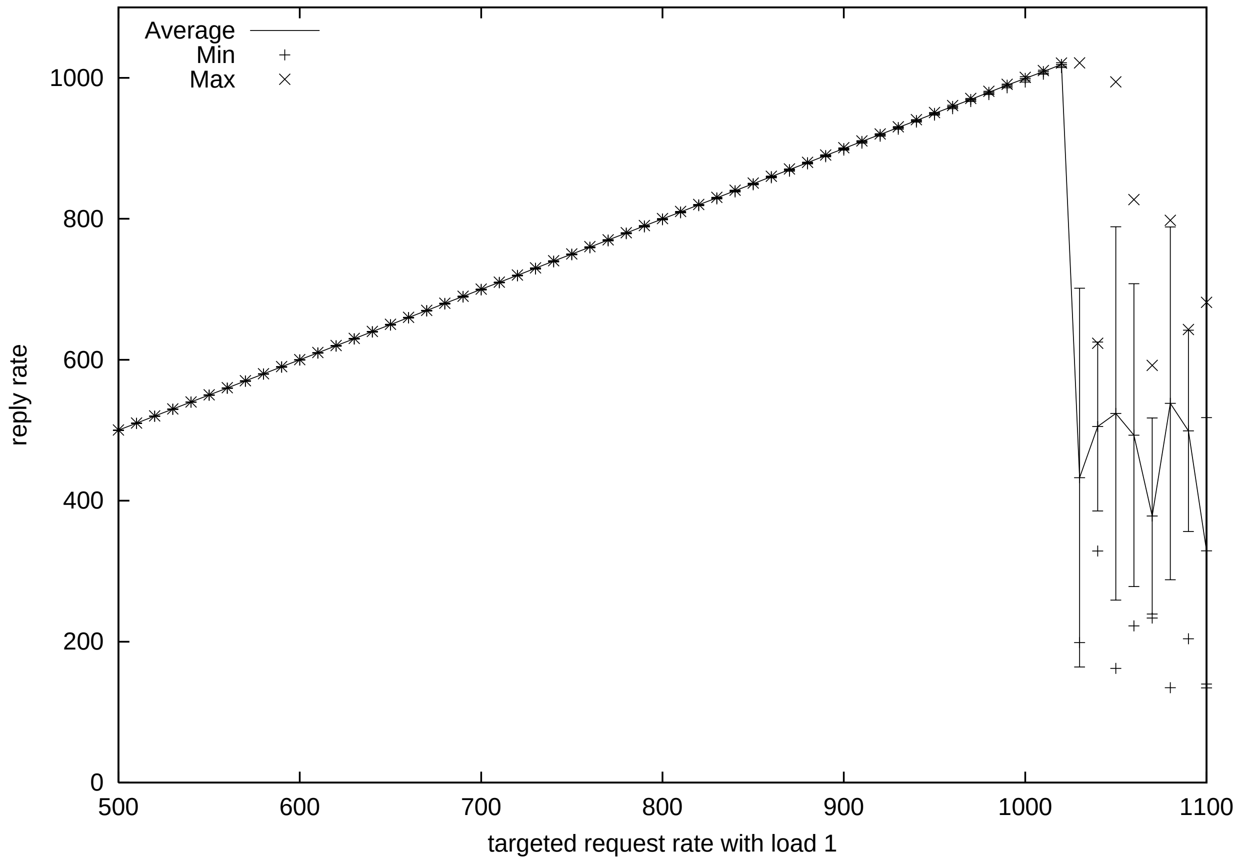
Benchmark

- To simulate high-latency connections, we add client programs that do not complete an http request.
- To keep the number of high-latency connections constant, they are reopened when a server times them out.
- There are several limitations influencing the benchmark:
  - We had to modify `httperf` to deal with more than 1024 file descriptors.
  - We can only have 60K open sockets at any time, so each benchmark is limited to 35K connections.
  - Before the next benchmark starts, we wait for all sockets to leave `TIMEWAIT`.

Benchmark

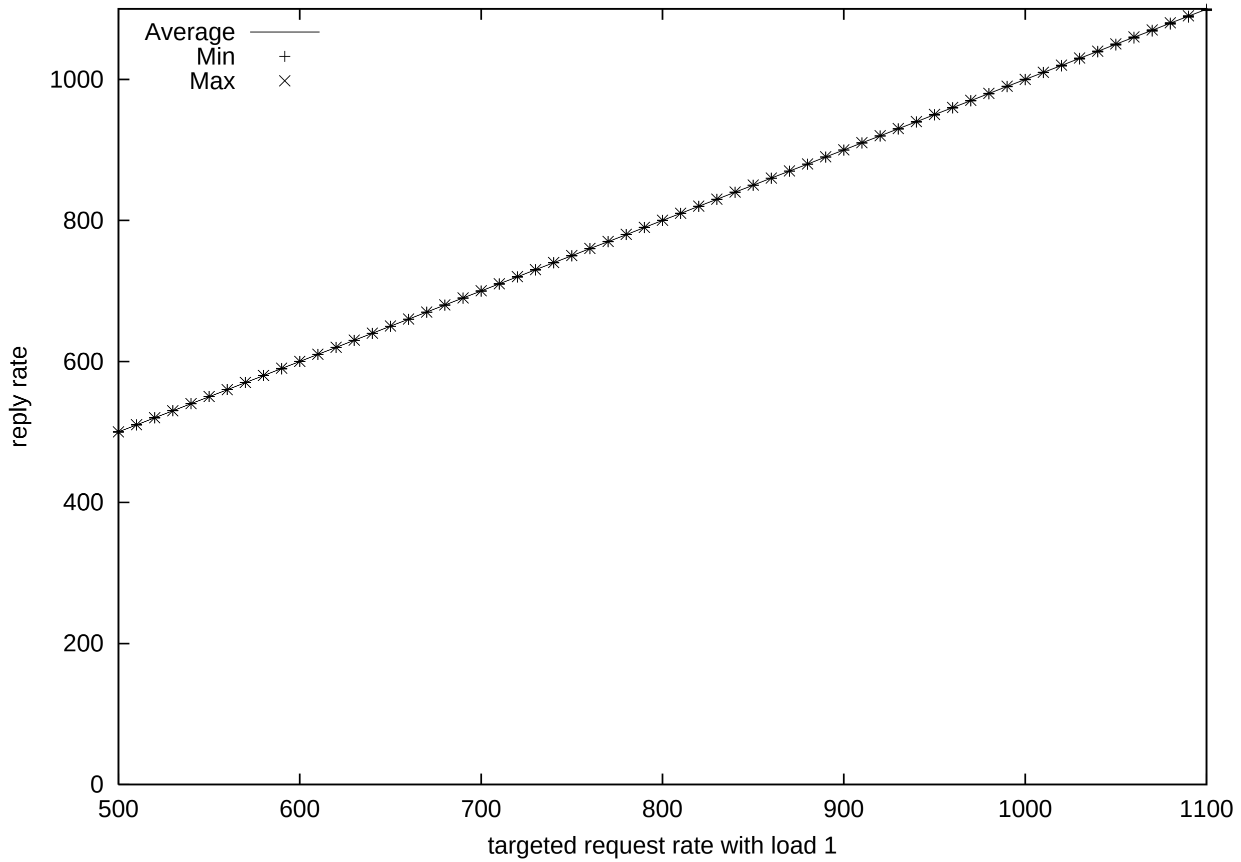
- To compare normal `poll()` with `/dev/poll`, we run two series of tests:
  - stock `thttpd` on Linux 2.2.14,
  - `thttpd` modified to use `/dev/poll`.
- We measure with 1, 251 and 501 inactive connections while constantly increasing the request rate of `httperf`.

normal poll(), one inactive connection



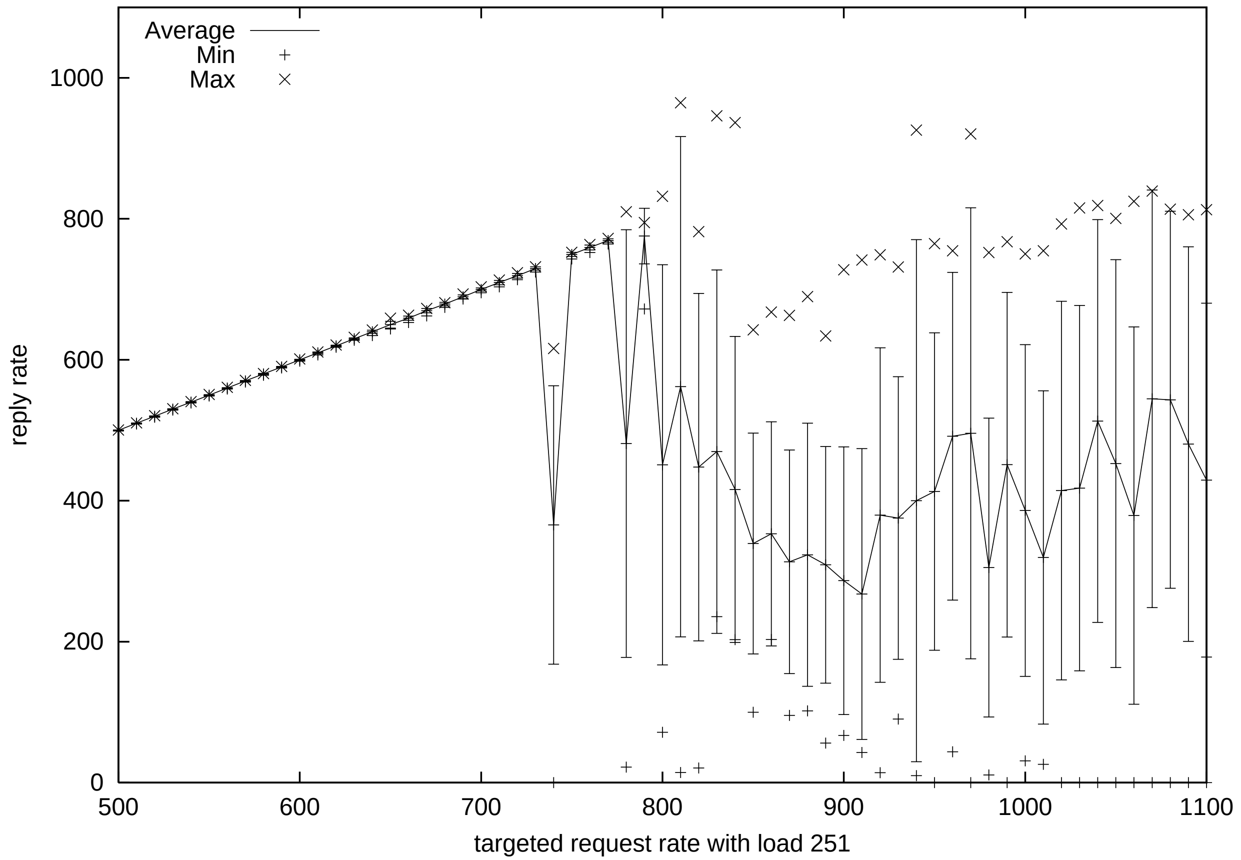
- At high request rate, server performance breaks down.

/dev/poll, one inactive connection



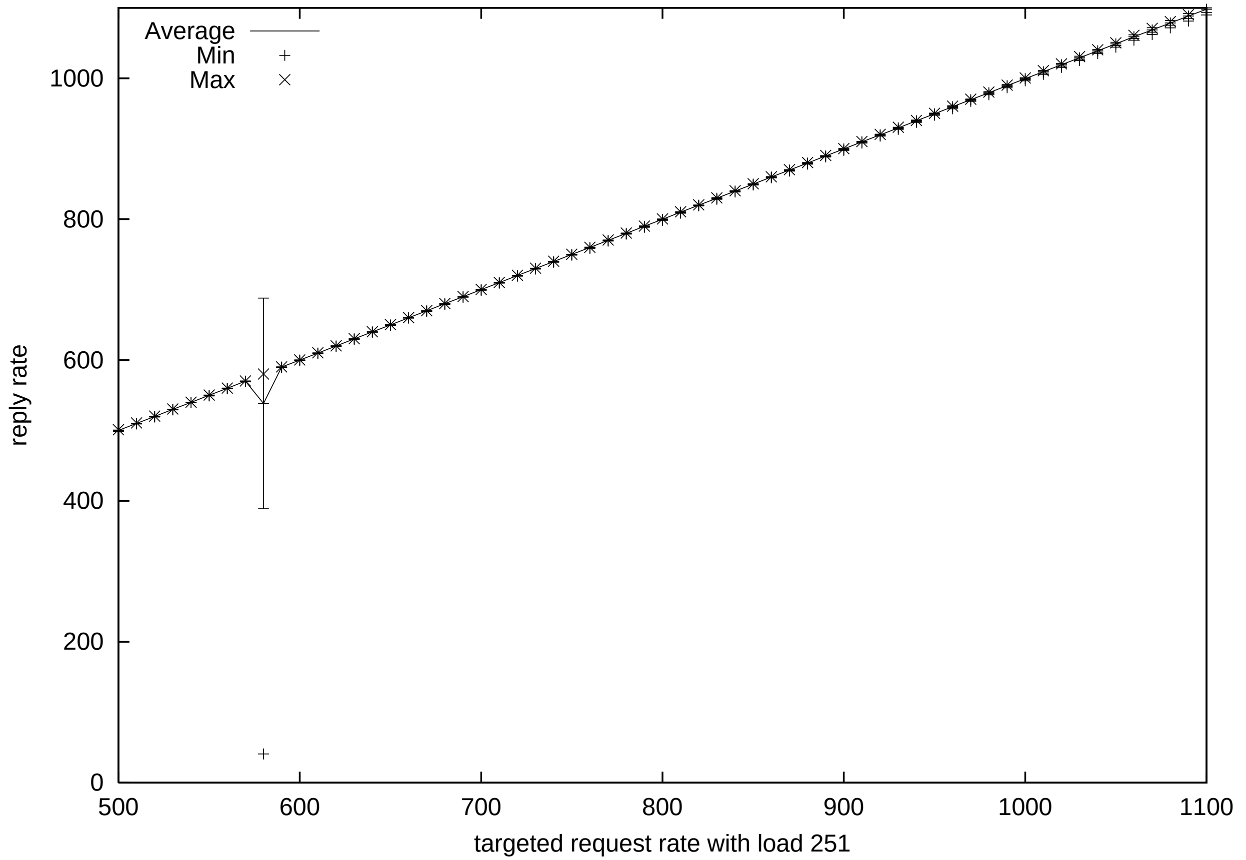
- Server performs well for all request rates.

normal poll(), 251 inactive connection



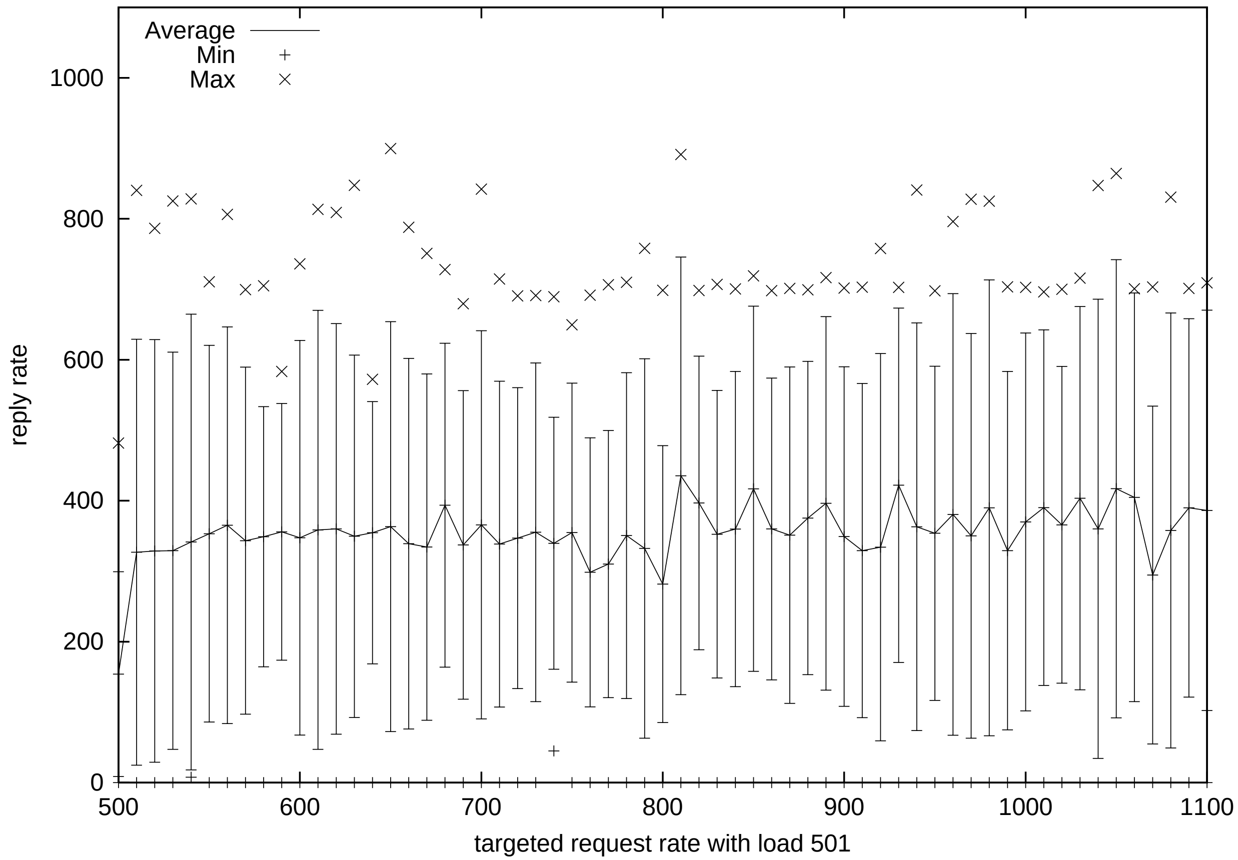
- As inactive connections increase, server performance suffers.

/dev/poll, 251 inactive connection



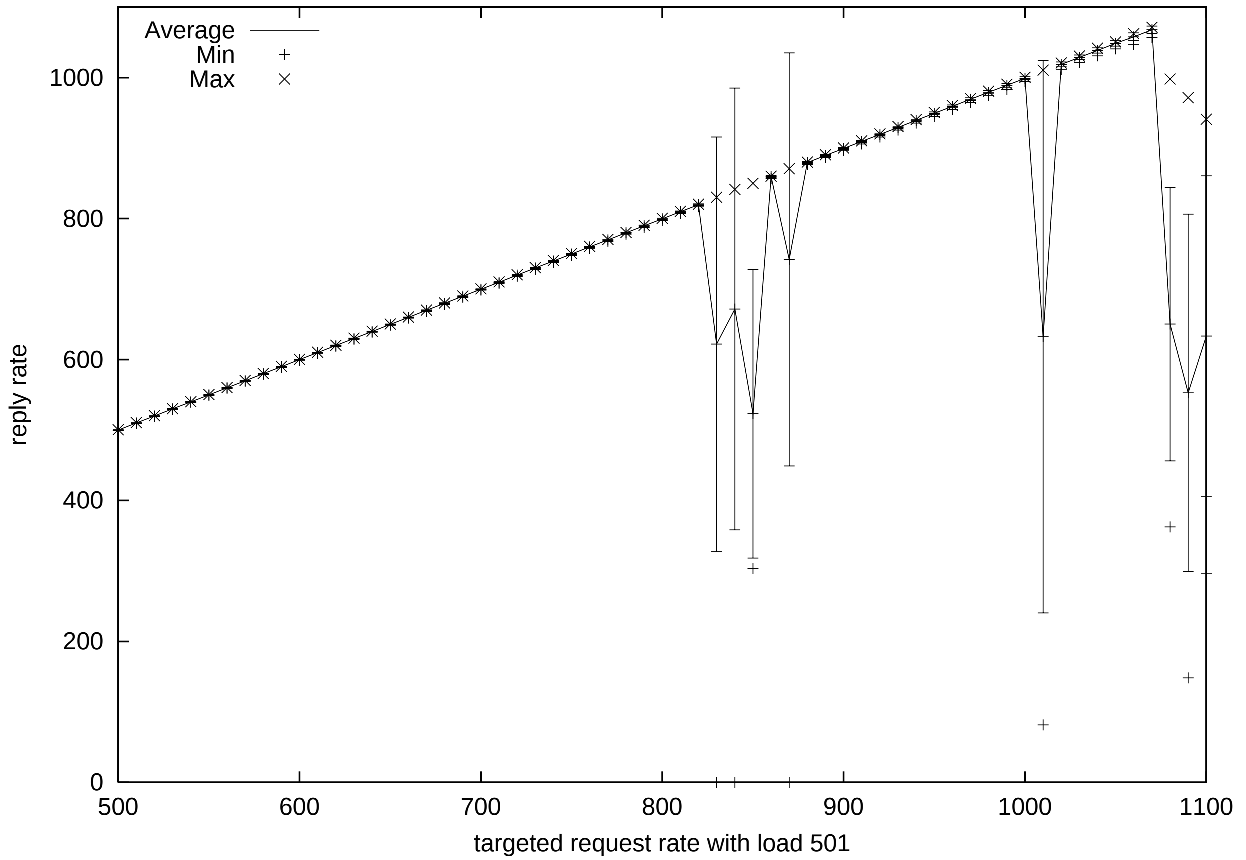
- Server performs well even with some inactive connections.

normal poll(), 501 inactive connections



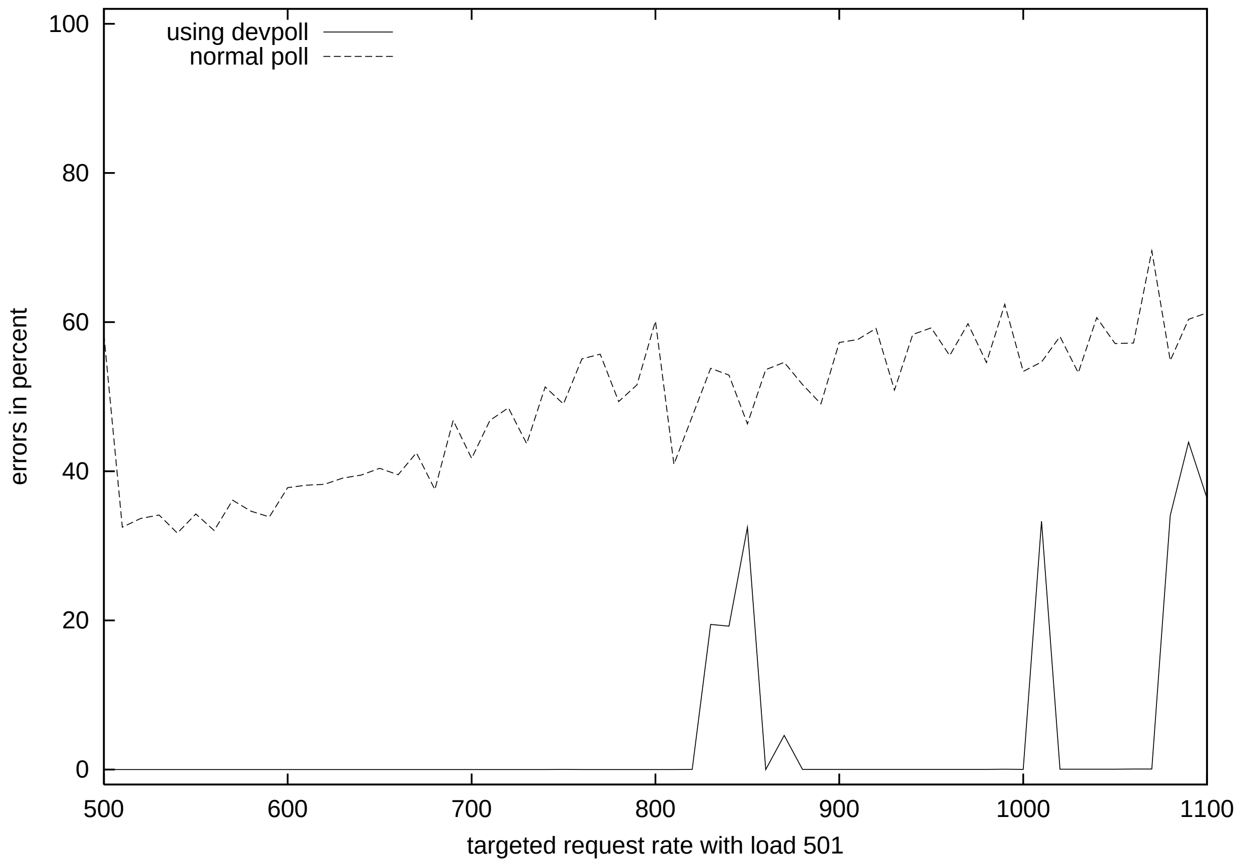
- Poor performance for all request rates.

/dev/poll, 501 inactive connection



- Performance breaks down for high request rates.

Error rate reported by httperf

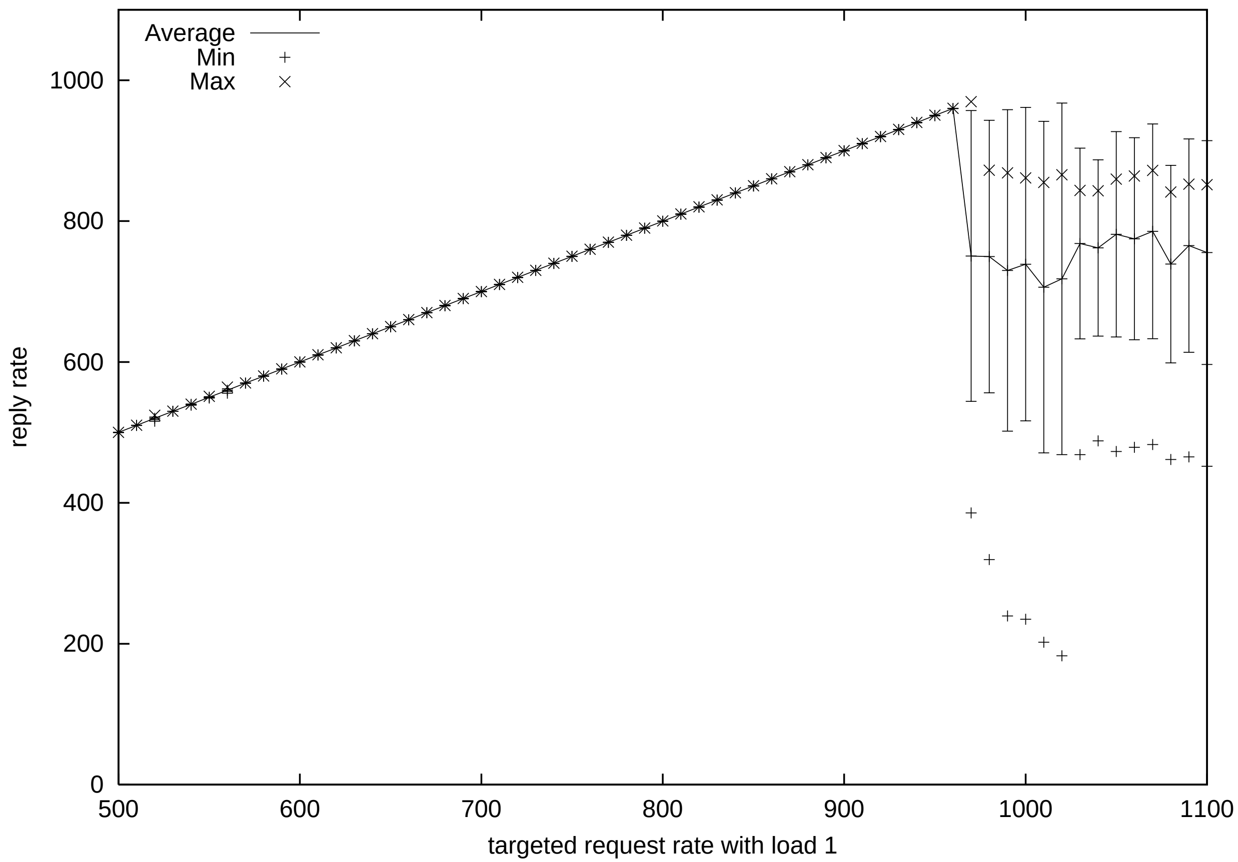


- stock `thttpd` errors increase to 60%, `/dev/poll` sporadic.

Benchmark

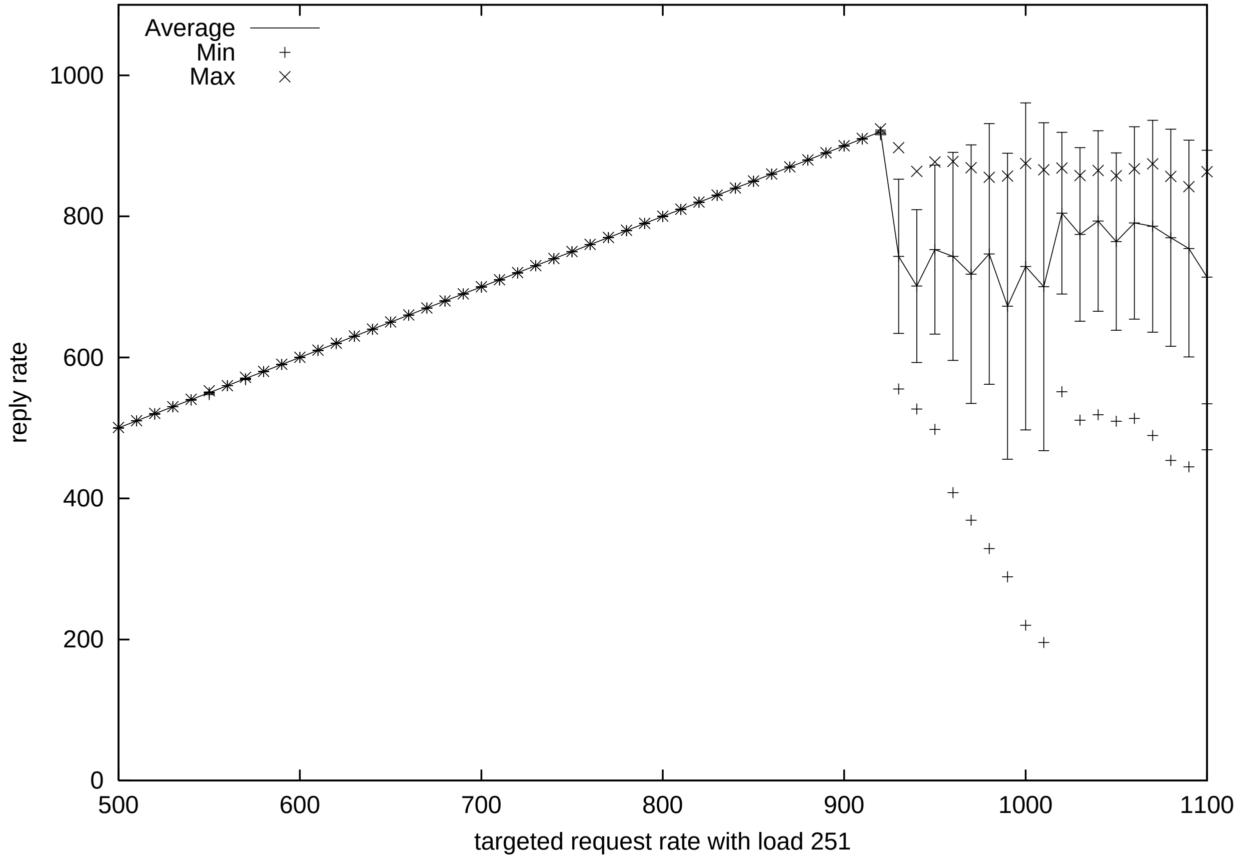
- Second series of benchmark measure a POSIX RT signal based server.
- `phttpd` runs on Linux 2.2.14 with complete support for RT signals back-ported from 2.3.x kernels.
- We compare a single-threaded `phhttpd` configuration against `thttpd`, a single process server.

phhttpd, one inactive connection



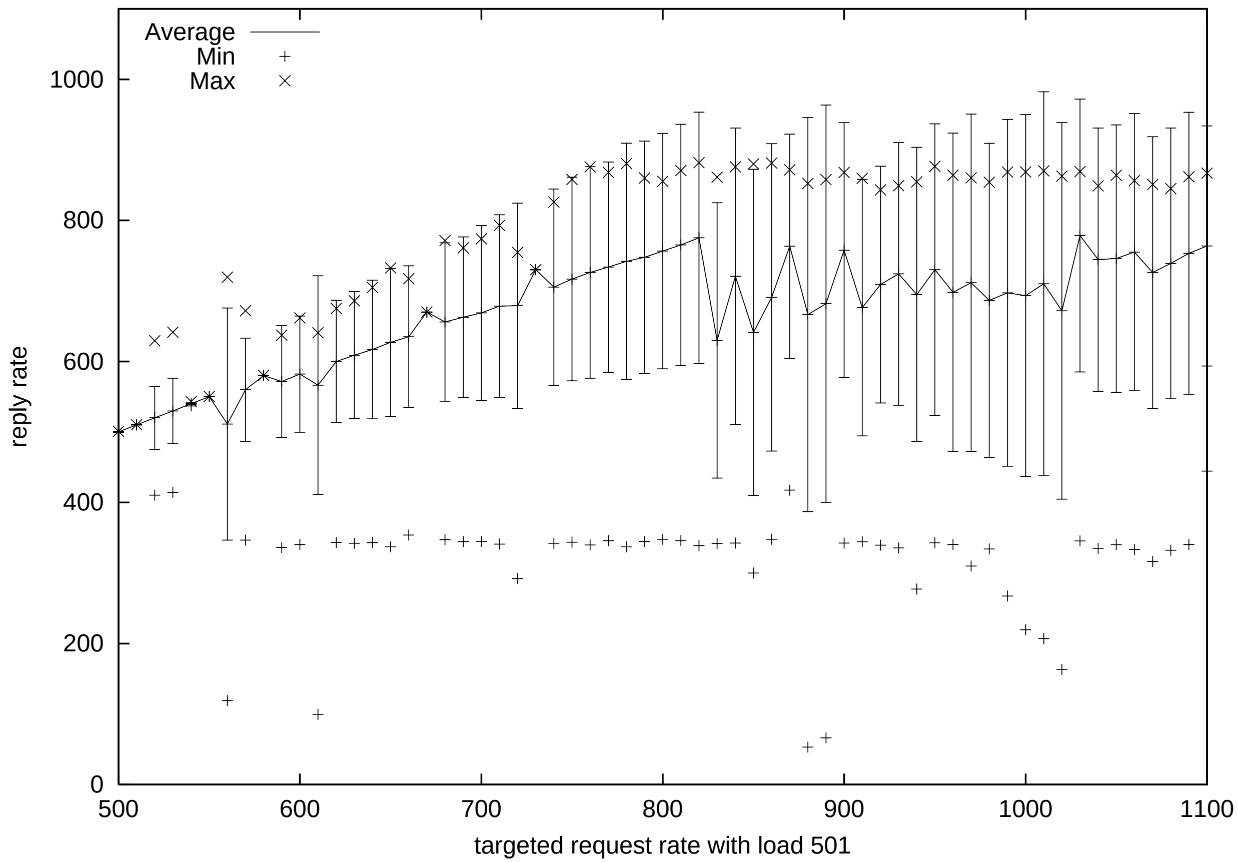
- Overhead of processing RT signals slows server for high rates.

phhttpd, 251 inactive connections



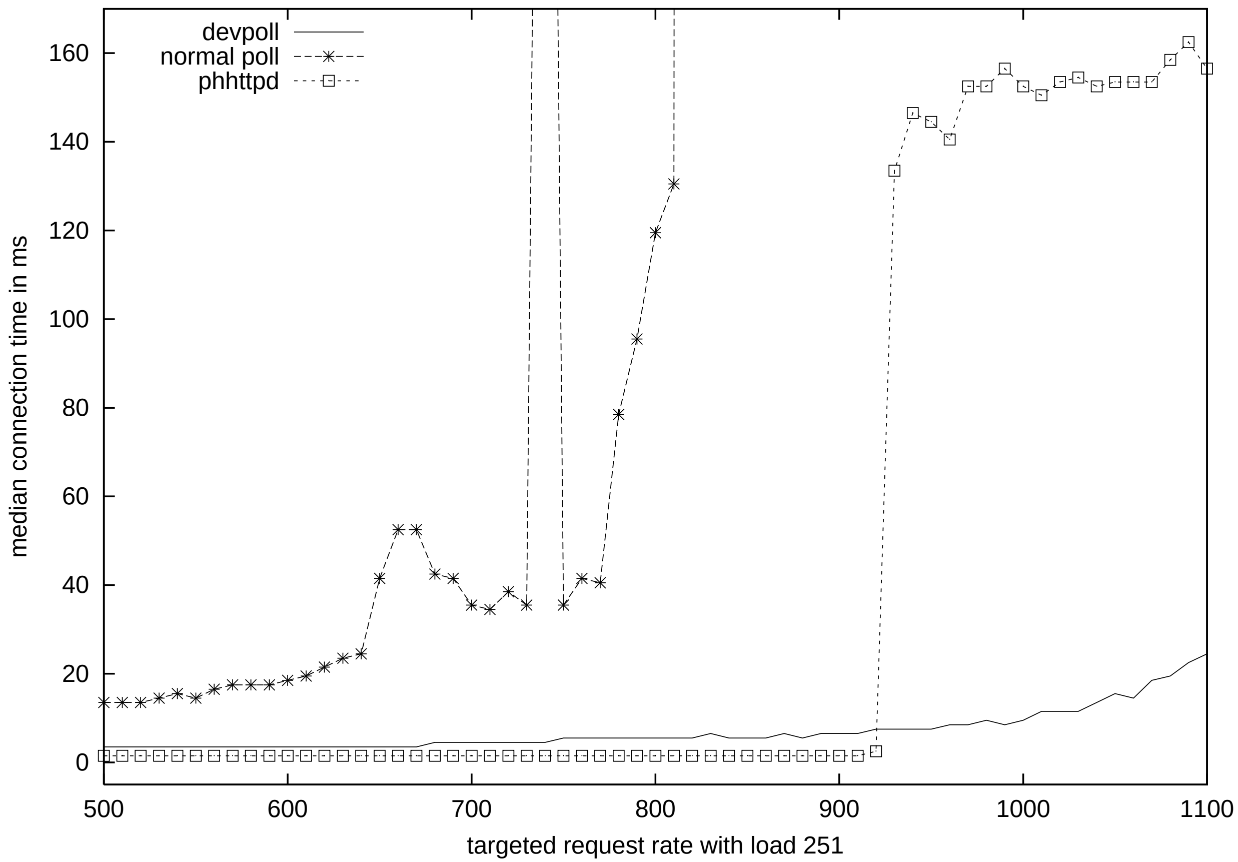
- Server reaches performance knee sooner.

phhttpd, 501 inactive connection



- Compared to `/dev/poll`, server scales less well.

Median latency,251 inactive connections



- For request rates up to 900, `phhttpd` has better latency.

Discussion

- Masked POSIX RT signals behave much like `poll()`.
- `poll()` information is bounded, current activity on a file descriptor replaces previous information.
- In contrast, with the RT signal queue, new events are simply added to the tail.
  - Special recovery mechanism is required for queue overflow.
  - Events picked from the queue can be treated only as hints.
- To optimize signal handling, the kernel could dequeue signals in groups.
- We plan to implement `sigtimedwait4()` so that the kernel can return more than one event.

Conclusion

- In our benchmark, `thttpd` using `/dev/poll` scales better than singled-threaded `phhttpd` using RT signals.
- Latency results at lower loads favor `phhttpd`, with higher load `/dev/poll` maintains a stabler and lower median response time.
- Software enhancements are freely available at our webpage: `http://www.citi.umich.edu/projects/linux-scalability/`.
- Acknowledgments:
  - Peter Honeyman and Stephen Tweedie for their guidance,
  - the FREENIX reviewers for their comments,
  - Zach Brown and Dan Kegel for their insights,
  - Intel Corporation for equipment loans.

