University of Glasgow

School of Computing Science

60 YEARS OF COMPUTING AT GLASGOW

Networking

Advanced Operating Systems

Tutorial 7

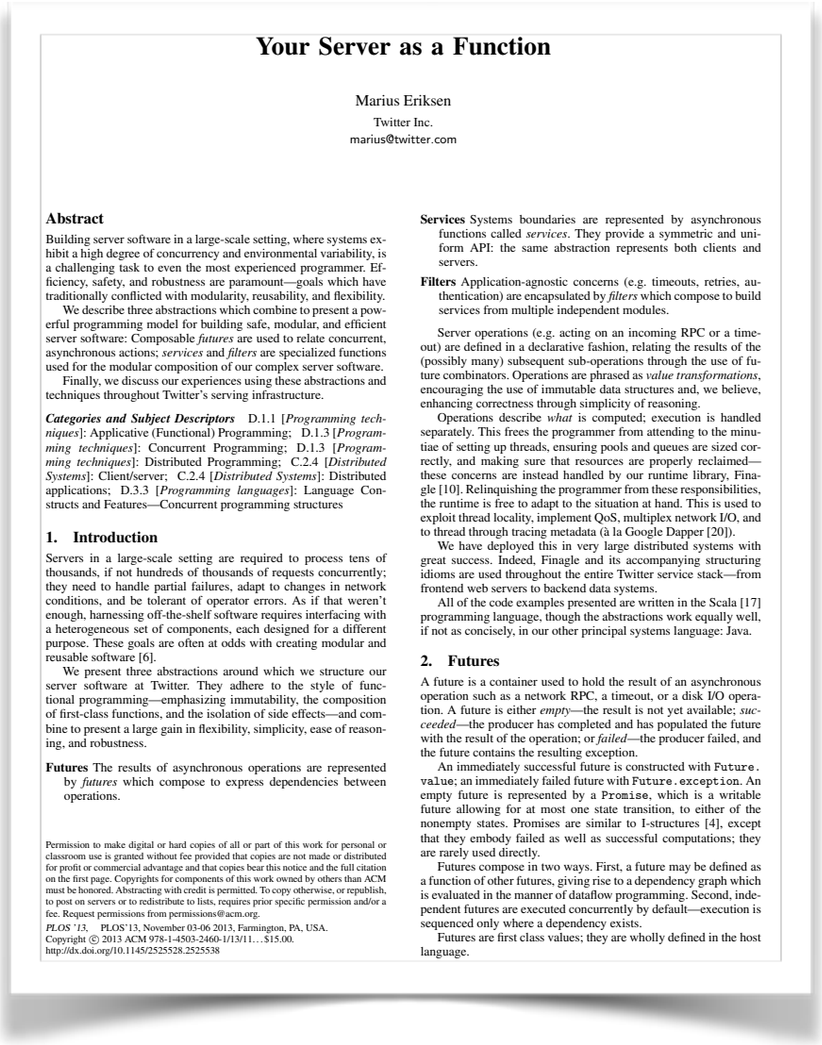
Message Passing and Networks

- Actors for network programming
  - Integrating networking with actors
  - Message parsing and serialisation
  - Applicability
- Asynchronous I/O
  - Event loops
  - libev/Rust MIO (Berkeley Socket `select()`?)
  - Higher-level approaches: Futures, Your Server as a Function

Discussion: Your Server as a Function

- M. Eriksen, “Your server as a function”, Proc. Workshop on Programming Languages and Operating Systems, Farmington, PA, USA, November 2013. ACM.

- Discusion:
  - Higher-level abstraction for asynchronous I/O
  - Futures to abstract asynchronous operations – in progress, succeeded, or failed; action runs concurrently until attempt made to read the result, then rendezvous
  - Services are functions that return a Future; encapsulate an operation to be performed by a server
  - Filters allow composition of services
  - Is this a good way of building network services?
  - Does it scale?
  - Too much magic? Or appropriate level of abstraction?



Your Server as a Function

Marius Eriksen
Twitter Inc.
marius@twitter.com

Abstract

Building server software in a large-scale setting, where systems exhibit a high degree of concurrency and environmental variability, is a challenging task to even the most experienced programmer. Efficiency, safety, and robustness are paramount—goals which have traditionally conflicted with modularity, reusability, and flexibility.

We describe three abstractions which combine to present a powerful programming model for building safe, modular, and efficient server software: Composable *futures* are used to relate concurrent, asynchronous actions; *services* and *filters* are specialized functions used for the modular composition of our complex server software.

Finally, we discuss our experiences using these abstractions and techniques throughout Twitter's serving infrastructure.

Categories and Subject Descriptors D.1.1 [*Programming techniques*]: Applicative (Functional) Programming; D.1.3 [*Programming techniques*]: Concurrent Programming; D.1.3 [*Programming techniques*]: Distributed Programming; C.2.4 [*Distributed Systems*]: Client/server; C.2.4 [*Distributed Systems*]: Distributed applications; D.3.3 [*Programming languages*]: Language Constructs and Features—Concurrent programming structures

1. Introduction

Servers in a large-scale setting are required to process tens of thousands, if not hundreds of thousands of requests concurrently; they need to handle partial failures, adapt to changes in network conditions, and be tolerant of operator errors. As if that weren't enough, harnessing off-the-shelf software requires interfacing with a heterogeneous set of components, each designed for a different purpose. These goals are often at odds with creating modular and reusable software [6].

We present three abstractions around which we structure our server software at Twitter. They adhere to the style of functional programming—emphasizing immutability, the composition of first-class functions, and the isolation of side effects—and combine to present a large gain in flexibility, simplicity, ease of reasoning, and robustness.

Futures The results of asynchronous operations are represented by *futures* which compose to express dependencies between operations.

Services Systems boundaries are represented by asynchronous functions called *services*. They provide a symmetric and uniform API: the same abstraction represents both clients and servers.

Filters Application-agnostic concerns (e.g. timeouts, retries, authentication) are encapsulated by *filters* which compose to build services from multiple independent modules.

Server operations (e.g. acting on an incoming RPC or a timeout) are defined in a declarative fashion, relating the results of the (possibly many) subsequent sub-operations through the use of future combinators. Operations are phrased as *value transformations*, encouraging the use of immutable data structures and, we believe, enhancing correctness through simplicity of reasoning.

Operations describe *what* is computed; execution is handled separately. This frees the programmer from attending to the minutiae of setting up threads, ensuring pools and queues are sized correctly, and making sure that resources are properly reclaimed—these concerns are instead handled by our runtime library, Finagle [10]. Relinquishing the programmer from these responsibilities, the runtime is free to adapt to the situation at hand. This is used to exploit thread locality, implement QoS, multiplex network I/O, and to thread through tracing metadata (à la Google Dapper [20]).

We have deployed this in very large distributed systems with great success. Indeed, Finagle and its accompanying structuring idioms are used throughout the entire Twitter service stack—from frontend web servers to backend data systems.

All of the code examples presented are written in the Scala [17] programming language, though the abstractions work equally well, if not as concisely, in our other principal systems language: Java.

2. Futures

A future is a container used to hold the result of an asynchronous operation such as a network RPC, a timeout, or a disk I/O operation. A future is either *empty*—the result is not yet available; *succeeded*—the producer has completed and has populated the future with the result of the operation; or *failed*—the producer failed, and the future contains the resulting exception.

An immediately successful future is constructed with `Future.value`; an immediately failed future with `Future.exception`. An empty future is represented by a `Promise`, which is a writable future allowing for at most one state transition, to either of the nonempty states. Promises are similar to I-structures [4], except that they embody failed as well as successful computations; they are rarely used directly.

Futures compose in two ways. First, a future may be defined as a function of other futures, giving rise to a dependency graph which is evaluated in the manner of dataflow programming. Second, independent futures are executed concurrently by default—execution is sequenced only where a dependency exists.

Futures are first class values; they are wholly defined in the host language.

PLOS '13, PLOS'13, November 03-06 2013, Farmington, PA, USA.

http://dx.doi.org/10.1145/2525528.2525538

High Performance Networking

- Growth in network performance relative to CPU performance → implications for network stack
- Alternative APIs:
  - netmap – shared ring buffers between application and kernel; dedicated network interface; high performance, but no abstraction
  - StackMap – new API for a TCP/IP stack, combining netmap API for data plane with Sockets API for control plane

Discussion: netmap and StackMap

- L. Rizzo, “netmap: a novel framework for fast packet I/O”, Proc. USENIX Annual Technical Conference, Boston, MA, USA, June 2012.
  - Background: NIC operation/data structures; kernel APIs; overheads
  - Pre-allocated buffers, shared between kernel and application; reduced numbers of system calls; dedicated network interfaces – is this a good API? How general purpose is it?
  - Performance – does it improve performance compared to the regular stack? For what applications?
  - Is this a good building block going forward? A general purpose API?

netmap: a novel framework for fast packet I/O

Luigi Rizzo, *Università di Pisa, Italy**
Proceedings of the 2012 USENIX Annual Technical Conference, June 2012.
https://www.usenix.org/conference/atc12/ †

Abstract

Many applications (routers, traffic monitors, firewalls, etc.) need to send and receive packets at line rate even on very fast links. In this paper we present *netmap*, a novel framework that enables commodity operating systems to handle the millions of packets per seconds traversing 1..10 Gbit/s links, without requiring custom hardware or changes to applications.

In building *netmap*, we identified and successfully reduced or removed three main packet processing costs: per-packet dynamic memory allocations, removed by preallocating resources; system call overheads, amortized over large batches; and memory copies, eliminated by sharing buffers and metadata between kernel and userspace, while still protecting access to device registers and other kernel memory areas. Separately, some of these techniques have been used in the past. The novelty in our proposal is not only that we exceed the performance of most of previous work, but also that we provide an architecture that is tightly integrated with existing operating system primitives, not tied to specific hardware, and easy to use and maintain.

netmap has been implemented in FreeBSD and Linux for several 1 and 10 Gbit/s network adapters. In our prototype, a single core running at 900 MHz can send or receive 14.88 Mpps (the peak packet rate on 10 Gbit/s links). This is more than 20 times faster than conventional APIs. Large speedups (5x and more) are also achieved on user-space Click and other packet forwarding applications using a libpcap emulation library running on top of *netmap*.

*This work was funded by the EU FP7 project CHANGE (257422).
†USENIX plans to publish this paper on the *Proceedings of the 2012 USENIX Annual Technical Conference*, which will be available at this URL after June 13, 2012. You may also find this paper, with related material, on the author's site, http://info.iet.unipi.it/~luigi/netmap/

1 Introduction

General purpose OSes provide a rich and flexible environment for running, among others, many packet processing and network monitoring and testing tasks. The high rate raw packet I/O required by these applications is not the intended target of general purpose OSes. Raw sockets, the Berkeley Packet Filter [14] (BPF), the AF_SOCKET family, and equivalent APIs have been used to build all sorts of network monitors, traffic generators, and generic routing systems. Performance, however, is inadequate for the millions of packets per second (*pps*) that can be present on 1..10 Gbit/s links. In search of better performance, some systems (see Section 3) either run completely in the kernel, or bypass the device driver and the entire network stack by exposing the NIC's data structures to user space applications. Efficient as they may be, many of these approaches depend on specific hardware features, give unprotected access to hardware, or are poorly integrated with the existing OS primitives.

The *netmap* framework presented in this paper combines and extends some of the ideas presented in the past trying to address their shortcomings. Besides giving huge speed improvements, *netmap* does not depend on specific hardware[1], has been fully integrated in FreeBSD and Linux with minimal modifications, and supports unmodified libpcap clients through a compatibility library.

One metric to evaluate our framework is performance: in our implementation, moving one packet between the wire and the userspace application has an amortized cost of less than 70 CPU clock cycles, which is at least one order of magnitude faster than standard APIs. In other words, a single core running at 900 MHz can source or sink the 14.88 Mpps achievable on a 10 Gbit/s link. The same core running at 150 MHz is well above the capacity

[1] *netmap* can give isolation even without hardware mechanisms such as IOMMU or VMDq, and is orthogonal to hardware offloading and virtualization mechanisms (checksum, TSO, LRO, VMDc, etc.)

1

StackMap: Low-Latency Networking with the OS Stack and Dedicated NICs

Kenichi Yasukata[†‡], Michio Honda[2], Douglas Santry[2], and Lars Eggert[2]
[1]*Keio University*
[2]*NetApp*

Abstract

StackMap leverages the best aspects of kernel-bypass networking into a new low-latency Linux network service based on the full-featured TCP kernel implementation, by dedicating network interfaces to applications and offering an extended version of the netmap API as a zero-copy, low-overhead *data path* while retaining the socket API for the *control path*. For small-message, transactional workloads, StackMap outperforms baseline Linux by 4 to 80 % in latency and 4 to 391 % in throughput. It also achieves comparable performance with Seastar, a highly-optimized user-level TCP/IP stack for DPDK.

1 Introduction

The TCP/IP protocols are typically implemented as part of an operating system (OS) kernel and exposed to applications through an application programming interface (API) such as the socket API [61] standard. This protects and isolates applications from one another and allows the OS to arbitrate access to network resources. Applications can focus on implementing their specific higher-level functionality and need not deal with the details of network communication.

A shared kernel implementation of TCP/IP has other advantages. The commercialization of the Internet has required continuous improvements to end-to-end data transfers. A collaboration between commercial and open source developers, researchers and IETF participants over at least the last 25 years has been improving TCP/IP to scale to increasingly diverse network characteristics [11, 39, 58], growing traffic volumes [13, 32], and improved tolerance to throughput fluctuations and reduced transmission latencies [1, 10, 49].

A modern TCP/IP stack is consequently a complex, highly optimized and analyzed piece of software. Due to these complexities, only a small number of stacks (*e.g.*, Linux, Windows, Apple, BSD) have a competitive feature set and performance, and therefore push the vast majority of traffic. Because of this relatively small number of OS stacks (compared to the number of applications), TCP/IP improvements have a well-understood and relatively easy deployment path via kernel updates, without the need to change applications.

However, implementing TCP/IP in the kernel also has downsides, which are becoming more pronounced with larger network capacities and applications that are more sensitive to latency and jitter. Kernel data processing and queueing delays now dominate end-to-end latencies, particularly over uncongested network paths. For example, the fabric latency across a datacenter network is typically only a few µs. But a minimal HTTP transaction over the same fabric, consisting of a short "GET" request and an "OK" reply, takes tens to hundreds of µs (see Section 3).

Several recent proposals attempt to avoid these overheads in a radical fashion: they bypass the kernel stack and instead implement all TCP/IP processing inside the application in user space [24, 29, 37] or in a virtual machine context [4]. Although successful in avoiding overheads, these kernel-bypass proposals also do away with many of the benefits of a shared TCP/IP implementation: They usually implement a simplistic flavor of TCP/IP that does not include many of the performance optimizations of the OS stacks, it is unclear if and by whom future protocol improvements would be implemented and deployed, and the different TCP/IP versions used by different applications may negatively impact one another in the network.

It is questionable whether kernel-bypass approaches are suitable even for highly specialized network environments such as datacenters. Due to economic reasons [17], they are assembled from commodity switches and do not feature a centralized flow scheduler [2, 45]. Therefore, path characteristics in such datacenters vary, and more advanced TCP protocol features may be useful in order to guarantee sub-millisecond flow completion times.

[‡] Most of the research was done during an internship at NetApp.

- K. Yasukata, M. Honda, D. Santry, and L. Eggert, “StackMap: Low-latency networking with the OS stack and dedicated NICs”, Proc. USENIX Annual Technical Conference, Denver, CO, USA, June 2016.
  - Combines netmap with kernel TCP/IP stack; new API for data path
  - Why are retransmissions complex? How are they handled?
  - Does this offer sufficient performance benefit to be worthwhile?