Reowolf: Synchronous Multi-Party Communication over the Internet*

Christopher Esterhuyse, Hans-Dieter A. Hiep**

Centrum Wiskunde & Informatica
Amsterdam, the Netherlands

1 Introduction

Programming Internet applications has essentially remained unchanged since the 1980s. The Berkeley Software Distribution (BSD) implementation allows applications to create *sockets* for communication over the Internet, e.g. Transmission Control Protocol over Internet Protocol (TCP/IP), that either listen for incoming connections or are connected outward, resulting in a bi-directional, reliable channel between two peers on the outer edges of the network. Applications consequently control the stream of data into the sending side of a channel, to be received in order by the other side of the channel. Although applications of the 1980s were process-driven, performing send and receive operations by cooperating with an operating system responsible for scheduling application processes. More recently, applications have become event-driven [11], allowing for more fine-grained scheduling decisions by applications themselves. However, the essence of programming Internet applications remains based on controlling a channel between two peers in a network.

In turn, this programming discipline seems to naturally support application architectures that favor centralization. With sockets, each channel between two peers requires a program controlling the stream of data on *both* sides of the socket connection. In a fully connected graph, where each peer maintains a connection to each other peer, one controls a quadratic number of data streams. However, a graph with a single central peer that connects to all other peers, requires controlling only a linear number of data streams at the central peer! In practice, clients use sockets to connect to a single centralized server and that server provides a shared service to its clients. Keeping the client simple solves half of the problem: only the server-side controls most of the data streams.

Recently, a drawback to centralized architecture became apparent by the increasing tension between central service providers (e.g. video streaming, search engines, content delivery) and network operators (providing Internet connectivity). The Workshop on Internet Economy (WIE2017) has discussed that service providers, "deliberately obfuscate both content and signaling information from network operators providing transit for the traffic." This leads to difficulties "to

* Received support from NLnet: NGI Zero PET fund from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 825310.

** Corresponding author: hdh@cwi.nl

improve traffic engineering, police (or secure) usage, and improve their own services. Increasingly, the access and transit network operators have less insight as to the nature of the traffic, and fewer effective traffic management tools." [4]

That service providers "deliberately obfuscate" their data seems reasonable: they namely employ data encryption to increase user privacy, and this is even required by EU regulation [13]. It also seems reasonable that more traffic will be sent encrypted in the future, as this and other similar regulations are implemented over time. Moreover, service providers may use non-standard protocols, for example, to gain a competitive edge against competitors.

Network operators typically deploy *middleware* to increase the quality of their service: to optimize Internet traffic to improve latency and throughput, and to monitor traffic to detect intrusions and abuse such as Denial of Service (DOS). Middleware uses, among other techniques, *deep packet inspection*: scanning further down the packet than the (TCP/)IP headers to take action. On high bandwidth networks, middleware can be implemented close to the metal to keep throughput high, for example, using field programmable gate arrays [5, 14].

A drawback of deep packet inspection is that it is non-standardized: middleware tries to *guess* an application's intent by scanning its traffic. As a simplified example, consider middleware that caches HTTP resources. As more Internet traffic is sent encrypted, deep packet inspection becomes less effective, e.g. HTTPS prohibits caching by a man-in-the-middle. Currently deployed middleware may therefore be wasting effort and may be hard to adapt due to its closeness to the hardware level. Moreover, the opportunity to optimize and monitor may not apply to certain traffic, e.g. obscure or innovative protocols.

In summary, we identify the following three issues:

1. Programming of decentralized Internet applications is more **complicated** than centralized applications (quadratic versus linear).
2. There is a **tension** between requirements of service providers and network operators (privacy versus openness).
3. Traffic monitoring and optimization techniques are **non-standard**, potentially leading to deprecation of existing middleware and impeding innovation.

In this paper we introduce Reowolf: a project that aims to replace the decades-old application programming interface, BSD sockets, for communication on the Internet. A novel programming interface is being implemented at the systems level that is inter-operable with existing Internet applications. It should provide support for middleware to further improve quality of service without having to give up on privacy, and makes programming of decentralized Internet applications simpler: we give arguments as to why we hold this position.

The main idea in Reowolf is to offer a high-level abstraction for communication, called *connectors*. Connectors are complexes of synchronization primitives among multiple data streams, generalizing end-to-end sockets. Programmers create a connector and configure it using a protocol description language (PDL), that allows for the declarative specification of what synchronization and data exchange primitives applications require to communicate on an abstract level.

These connectors are to sockets what high-level programming languages are to machine code: we compile high-level application-defined protocols to low-level operational code that realizes the actual communication.

As an analogy for understanding Reowolf, it is useful to draw parallels between GPU programming and network programming. In GPU programming, one writes a high-level program (i.e., GLSL) that describes graphical manipulations that take place on dedicated hardware (i.e., graphics cards). This alleviates programmers from having to write hardware-specific programs that use low-level, device-specific assembly instructions. Each vendor of graphics hardware supports the high-level programming language and compiles it down to their own target. The novelty of Reowolf is to approach Internet application programming in the same way: one writes a high-level 'program' that *configures* the possible communications between multiple parties on a network, compiled into code that targets peer communication by performing synchronization and exchanging data.

The main benefits of programming Internet applications in this way are:

1. Raising the level of **abstraction** relieves the programmer from controlling a quadratic number of data streams, while still being able to communicate in a decentralized manner. In realizing the abstraction there is ample room for optimization and innovation for the implementer: different optimization techniques apply to different networking circumstances[1], and applications are agnostic to the underlying network protocols.
2. A clean **separation** of application-defined protocols from pure computations results in application programs that are more isolated, making them simpler to reason about, validate, and verify. Moreover, separation of application content from its signaling information allows encryption of content only, while granting middleware the insight in the nature of traffic.
3. Working with application-defined protocols as **explicit** objects allows them to be collected in a standard library, that facilitates protocol reuse. Moreover, application-defined protocols are publicly visible within the network, allowing middleware to perform informed traffic monitoring such as deviation detection without having to guess application's intent.

These main benefits do not fall out of the sky: in the Reowolf project, we turn theoretical and prototypical results of the past two decades [3] into practical, low-level systems work. The protocol description language of Reowolf is largely based on Reo, an exogenous coordination language for synchronous communication [2]. Although Reo has seen recent practical applications, such as the distributed implementation Dreams [12], and compilers for shared-memory synchronization [10], never before was Reo integrated deeper into operating systems.

Reowolf is an ongoing project with the ultimate goal of replacing the *socket* with the *connector* for programming internet applications. In the rest of this paper, we discuss the main ideas underpinning Reowolf, and highlight some of the interesting challenges of implementing Reowolf that remain to be solved.

[1] E.g. local host (virtualized) networking, wireless sensor networks, Beowulf class cluster computing, high-speed local area networks in datacenters, wide-area networks spanning the globe, and satellite networks.

2 Comparing Reo and Reowolf

From the perspective of history of computing, Reowolf is the logical next step for programming Internet applications. Compare this to computing before the Internet age, where early programming languages such as Algol or Fortran lacked the ability to deal with data abstractions. A data structure was treated implicitly, and search and manipulation algorithms were littered all over the program. With abstract data types (ADTs), programs can be written against an abstract interface that describes the possible operations. ADTs can be explicitly specified, for example, by algebraic data structures of a given signature and a (e.g., equational or first-order) specification.

A similar case can be made for today's programs, that lack the ability to deal with concurrency and communication abstractions. A program implicitly deals with protocols, and low-level concurrency code for synchronization is all over the place. Similar to ADTs is the idea of abstract behavior types [1, 8]; programs can be written against an abstract interface that describes all possible behaviors a program could expect. Abstract behavior types are explicitly specified by the protocol description language that is based on Reo.

Reo is a language for specifying exogenous communication protocols by constraining the possible interactions available to participating components: a protocol coordinates its components. Reo structurally separates computation from coordination, unlike other exogenous coordination languages [6]. A *component* is treated as an indivisible black-box with an identity that publicly exposes a number of named *ports*. Components are linked together by complex connectors: connectors are graph-like structures of nodes and primitive channels between nodes. A set of primitive channels is provided: e.g. synchronous channels, lossy channels, and asynchronously buffered channels. Multiple connectors (between possibly the same set or different sets of components) can be composed into larger connectors: Reo's compositional semantics ensures that properties of a composite connector preserve the properties of its constituent connectors.

Where Reo remains abstract, Reowolf becomes concrete. We implement Reo components as follows: each component is identified by an IP address, or by a domain name that resolves to an IP address. Components do not correspond 1-to-1 to (physical) machines, as one component can be implemented by multiple machines behind a router with network address translation, and a single machine can host multiple components. Components have several ports, each identified by a number. Later, *port name systems* may be used to resolve ports by name.

Components are responsible for setting up connectors. Each component provides a local view of the connector, by configuring it with a protocol description as it sees fit. The protocol description includes references to the ports of other components to connect to, and it includes which local ports are left open. Multiple components can provide different local views of a connector, and the established connector is the composition of all local views of the connector of its connected components. Components that dynamically allocate new ports over time, and dynamically change their view of the connector by reconfiguring with a new protocol description, are interesting research challenges.

3 Programming with Connectors

Among existing environments for Internet applications are UNIX-like operating systems, where applications are implemented by one or more processes that perform system calls. There are system calls to create sockets, sending and receiving payloads, and handling exceptional network conditions such as timeouts. The implementation of those socket system calls is provided as part of the operating system. Reowolf connectors have a programming interface similar to sockets: creating connectors, configuring the expected behavior of a connector, putting and getting payloads, and handling exceptions such as non-conformance.

Sockets establish a channel between *two* peers and communication is fundamentally *asynchronous*, connectors establish a protocol between *multiple* peers and communication is fundamentally *synchronous*:

Multi-party Communication
Two socket connections are assumed to be unrelated, unless a program controlling their streams intentionally relates them. The expected behavior of such control program is implicit, and unknown to the network. Complex relations between multiple socket connections require complex controlling programs to run at both end-points. Connectors generalize sockets to an arbitrary number of peers, and uses a declarative protocol description language to describe the expected interactions among those peers. The explicit protocol description is provided as part of configuring a connector. A control program that handles the data stream control is generated on-the-fly by the implementation of Reowolf, and is no longer the responsibility of the application. Applications thus interact with a connector, by getting and putting data, abstracting away the communication with multiple peers.

Synchronicity
Synchronous communication means there is not an *a priori* requirement on the order of the events that realize the communication (e.g., sending and receiving happen concurrently). Connectors can still describe asynchronous communication, by imposing a requirement on the order of its realizing events using logical buffers (e.g., send happens first, receive happens later).

Implementation Architecture
From the OSI layering perspective, sockets directly expose applications to (an implementation of) the Transport layer; applications are themselves responsible for session management and representation formats of transmitted data. Reowolf connectors instead expose an abstraction above the Transport layer, including the Session and Representation layers. A Reowolf implementation is free to choose how to realize the communication protocol that the programmer has specified: it can use multiple TCP connections or UDP (or other protocols).

Modular Validation and Verification
A program that makes use of connectors can be validated and verified with respect to the local protocol it configures. All established properties of the program also hold in the case where communication takes place with unknown peers. This ensures that modular development of applications is fruitful. An implementation is correct if all local properties of programs are preserved under all compositions.

4 Leveraging Explicit Protocol Descriptions

We will consider four scenarios by which we exemplify how the explicit protocol descriptions of Reowolf can improve performance and detect abuse.

Shared Memory Optimization
An implementation of Reowolf that recognizes multiple peers on the same physical machine can use a different implementation to realize communication: it is not necessary to send and receive packets over the wire to communicate, as the protocol can be implemented under this circumstance by means of shared memory, possibly even without copying data [7]. Synchronization of peers can then use concurrency primitives such as mutexes and thread barriers. Research has shown that the run-time performance of compiled shared memory protocols is on par with, and sometimes even outperforms, hand-crafted concurrency code [9, 10]. Connectors that involve a mix of local and remote peers can leverage both optimization strategies; local communication takes place via shared memory, while remote communication is performed by exchanging IP packets.

Informed Route Optimization
The protocol description and state of connectors are publicly visible by the network. Intermediary nodes along the routing path can use this information for route optimization. Consider three participants in a protocol: a node in Helsinki repeatedly sends the same payload to two nodes in Tokyo. With sockets there are two channels, both from Helsinki to Tokyo. Each payload has to travel twice the distance. In Reowolf, a single connector is established between the three parties. Implementations can recognize by the protocol description that data is duplicated. Instead of sending the same payload twice over the long distance, it could be sent once and forwarded in Tokyo to the other recipient. The traffic on the long distance path of the network is then halved.

Local Deviation Detection
Reowolf will inspect traffic and checks it with protocol conformity. Within a peer, conformity to the local protocol is checked before passing data to an application. Since each connector is configured with an explicit protocol description, implementations of Reowolf can locally intercept incoming network traffic that violates the configured protocol, possibly raising an exceptional network condition for the application to handle. Outgoing traffic that violates the local protocol can be dropped immediately.

From Eavesdropping to Eager Dropping
During the configuration phase of connectors, peers distribute their local protocols over the network to reach a consensus on the composed protocol. Consequently, outgoing traffic that violates the composed protocol can be dropped immediately: otherwise, it would be dropped locally at the receiving peer. Moreover, intermediate network nodes are able to snoop on the composed protocol, and use it to check packets it forwards with protocol conformity, too. Thus, all nodes cooperate in checking conformity to the protocol. Since dropping reduces superfluous network traffic, it is in the best interest of the intermediary nodes to do so. Similar to TCP session hijacking, it is a challenge when considering malicious peers too: a naïve implementation may unintentionally allow a DOS.

References

1. Farhad Arbab. Abstract behavior types: A foundation model for components and their composition. In *International Symposium on Formal Methods for Components and Objects*, pages 33–70. Springer, 2002.
2. Farhad Arbab. Proper protocol. In *Theory and Practice of Formal Methods*, pages 65–87. Springer, 2016.
3. Giovanni Ciatto, Stefano Mariani, Maxime Louvel, Andrea Omicini, and Franco Zambonelli. Twenty years of coordination technologies: state-of-the-art and perspectives. In *Proceedings of the 20th International Conference COORDINATION*, pages 51–80. IFIP, 2018.
4. Kimberly C. Claffy, Geoff Huston, and David Clark. Workshop on Internet Economics (WIE2017) Final Report. *SIGCOMM Computer Communication Review*, 48(3):42–45, September 2018.
5. Sarang Dharmapurikar, Praveen Krishnamurthy, Todd Sproull, and John Lockwood. Deep packet inspection using parallel bloom filters. In *Proceedings of the 11th Symposium on High Performance Interconnects*, pages 44–51. IEEE, 2003.
6. Kasper Dokter, Sung-Shik Jongmans, Farhad Arbab, and Simon Bliudze. Relating BIP and Reo. *arXiv preprint arXiv:1508.04848*, 2015.
7. Micha Hergarden and Sung-Shik Jongmans. Shared memory implementations of protocol programming languages, data-race-free. In *Proceedings of the 13th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems*, pages 36–40. ACM, 2018.
8. Marián Jenčik and Daniel Mihályi. Program components & abstract behavioral types. *Acta Electrotechnica et Informatica*, 12(1):38–43, 2012.
9. Sung-Shik Jongmans. *Automata-theoretic protocol programming*. PhD thesis, Centrum Wiskunde & Informatica (CWI), Leiden University, 2016.
10. Sung-Shik Jongmans and Farhad Arbab. Can high throughput atone for high latency in compiler-generated protocol code? In *Fundamentals of Software Engineering*, pages 238–258. Springer International Publishing, 2015.
11. Gero Mühl, Ludger Fiege, and Peter Pietzuch. *Distributed event-based systems*. Springer Science & Business Media, 2006.
12. José Proença, Dave Clarke, Erik de Vink, and Farhad Arbab. Dreams: a framework for distributed synchronous coordination. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, pages 1510–1515. ACM, 2012.
13. Colin Tankard. What the GDPR [General Data Protection Regulation] means for businesses. *Network Security*, 2016(6):5–8, 2016.
14. Fang Yu, Zhifeng Chen, Yanlei Diao, Tamil V. Lakshman, and Randy H. Katz. Fast and memory-efficient regular expression matching for deep packet inspection. In *Proceedings of the 12th Symposium on Architecture for Networking and Communications Systems*, pages 93–102. IEEE, 2006.