Finally, a Use for Componentized Transport Protocols

Tyson Condie, Joseph M. Hellerstein, Petros Maniatis, Sean Rhea, Timothy Roscoe
U.C. Berkeley and Intel Research Berkeley

Abstract

This paper argues a new relevance for an old idea: decomposing transport protocols into a set of resuable building blocks that can be recomposed in different ways depending on application requirements. We conjecture that point-to-point applications may well be adequately served by the existing suite of monolithic protocol implementations, but widely-distributed peer-to-peer systems such as overlays are not: the design space of transport protocols between nodes in a large, highly coordinated system is much larger. We provide several examples of existing systems that have implemented a diverse range of transport protocols, and show how a building-block approach covers these systems well, enabling simple specification of hybrids and variants of the protocols. In particular, we show how all of our examples can be implemented in the networking stack of P2, a multipurpose system for building overlay networks from declarative specifications.

1 Introduction

There has been a steady stream of research over the years into *componentized protocols*: protocol implementations assembled from a variety of building blocks. A promise of such frameworks has generally been flexibility: a protocol stack tailored for a particular application can be easily assembled, usually without writing any new code, by binding protocol objects together.

Despite its conceptual elegance, protocol implementations based on this approach have never caught on, particularly at the transport level. Most applications today use a kernel-provided IP stack, and usually TCP for transport. The consensus is that for both bulk-transfer of data and RPC-like call semantics, TCP appears to be perfectly adequate, and it is not worth inventing something new.

Of course, a few applications have been identified where a radically different transport protocol is appropriate, and in these cases a new, complete, and different protocol has been devised (e.g. RTP [12] for multimedia, or SCTP [25] for PSTN-like signalling traffic) rather than composing a protocol from building blocks. The use of these standard protocols has further diminished the motivation for transport protocols whose functionality can be composed at user level in an application-specific manner.

We suspect that this small set of transport protocols (along with DCCP [10]) covers the needs of almost all point-to-point applications; that is, applications based on the concept of two end-points communicating.

However, in this paper we argue that widely-distributed systems, including structured and unstructured overlays, change this situation and introduce problems that componentized transport protocol mechanisms are uniquely equipped to solve.

In the last few years, considerable research has been devoted to both structured and unstructured *overlay* and *peer-to-peer* applications. As distributed systems, these applications generally include their own techniques for routing messages on an overlay. Many such deployed systems, including Bamboo [24], MIT Chord [26], and P2 [20], use custom transport protocols which provide TCP-friendly congestion control behavior, but over UDP.

In Section 2, we attempt to explain this design shift by examining features of P2P applications and overlays that motivate their designers to adopt custom transport protocols, and the way in which these applications differ from traditional network-based applications. With examples from specific applications, we highlight more generic requirements for transporting data in these settings.

In the end, however, our message is not simply that the features of modern distributed systems require a rethinking of transport protocols. We also argue that these features greatly widen the design space for such protocols and require an ability to easily customize transport protocols for various distributed applications.

To support this argument, we describe the transport protocol portion of P2, a declarative overlay processor we have built [20]. P2 allows custom transport protocols to be assembled from reusable dataflow building blocks. We show how a variety of diverse but important application behaviors can be achieved naturally within P2's framework, in ways that are hard or impossible to achieve with monolithic kernel implementations of transport protocols such as TCP, RTP, SCTP or DCCP.

2 What's different?

We have asserted that new, distributed, UDP-based applications behave in ways that are not well served by traditional transport protocols. In this section we identify several features of such applications that distinguish them from, say, typical web services.

First we note that at a node-to-node level, P2P communication often requires different subsets of the TCP functionality set—in-order delivery, reliability, congestion control. DCCP [10] explicitly addresses this design space, defining an additional kernel datagram protocol providing congestion control and flexible packet acknowledgement.

However, in this paper we argue that a more general approach than DCCP is required. We present use-cases in P2P systems where implementation is not possible using DCCP, and cases which require substantial implementation in addition to DCCP. While most of our examples are from structured P2P overlays, we stress that the principles here are by no means limited to the DHT space. We also note that DCCP is a protocol rather than an implementation per se. The question of whether DCCP itself is naturally implementable within the framework of section 3 is beyond the scope of this paper.

Some of our example distinctions below relate to desired protocol *behavior*, while others concern the design of a suitable protocol *API* for end-systems. However, both affect how the protocol is implemented.

Application-level routing freedom:

Widely-distributed applications have many choices about where to forward a message. Unlike traditional client-server applications, there may be several equivalent end-points for a message (e.g., to retrieve a replica of some object). Moreover, P2P systems usually incorporate some kind of overlay network, even if it is not explicit in the design (e.g., the structured overlay of a DHT, or the link-state overlay of an enterprise network of Microsoft Exchange servers). This provides options not only for the destination of a message, but also the overlay path taken to get there.

Designers exploit this new-found freedom to achieve high performance (latency, throughput, reliability, etc.) by implementing sophisticated adaptive policies for forwarding data in the system. For example, a node in the Bamboo DHT [24] constantly measures minimum round-trip times to nodes in its routing table, sets aggressive timeouts, and rapidly resends messages to alternate neighbors if these timeouts are exceeded. This performs dramatically better under churn, since Bamboo can rapidly route around failures and transient load spikes [24].

In terms of the implementation, this inverts a traditional ordering of functionality in a transport stack: destination selection (e.g., the lookup in Bamboo's routing table) now takes place downstream of retries, since successive retries for a message can be sent to different destinations.

Congestion control aggregation:

In addition to having flexibility in the choice of destination, some P2P applications have the additional property of choosing among a very large set of such destinations—a set whose size and contents are typically not known in advance. A good example is the iterative routing employed by MIT Chord [9] and the Kademlia [21] variants used in eDonkey [2] and trackerless BitTorrent [1].

A problem thus arises maintaining congestion windows for a large and unpredictable number of destinations, many of which are only needed for a single lookup RPC. To address this problem, DHash++ uses a custom transport protocol called STP [9] that maintains aggregate congestion state for all nodes, rather than the per-node state maintained by TCP, DCCP, etc. Consequently, all outgoing packets traverse a single congestion-control instance before being sent to a variety of destinations.

This technique represents a different change in the transport stack implementation from the Bamboo example above. Here, congestion control is performed independently of the destination of messages. Indeed, the decision of where to send the message may be deferred until the congestion window allows it to be sent.

Application buffer management:

The designers of DCCP point out the benefits to applications of "*late data choice*, where the application commits to sending a particular piece of data very late in the sending process" [19] and suggest using familiar ring-buffer techniques for queueing packets rather than the traditional Unix API. This change allows latency-sensitive applications to revise or replace outgoing packets up until the time when the protocol implementation can send them.

A good motivating example is the use of in-network aggregation techniques for disributed query processors such as PIER [14] and [27]. Data is sent up an aggregation tree to the root, and aggregation computation is performed at any intermediate node holding more than one datum at a time. Ideally, each node would send data up the tree eagerly (whenever congestion control allowed it), but otherwise aggregate it with any new data arriving from below.

In practice, traditional protocol implementations (such as Unix TCP) thwart this, since outgoing data may be held at a node in a buffer (before being sent, or for retry purposes), without being available to the query processor for further aggregation. This limitation can result in situations where stale results are sent even though a fresher one is available.

We therefore embrace DCCP's notion of late data

choice, but extend it further: in addition to being able to revise outgoing packets, widely distributed applications such as distributed query processors benefit from late creation of the packets themselves; an API which provides an upcall to request the next packet to send allows intelligent just-in-time creation of packets.

Furthermore, our approach integrates well with systems that exploit routing freedom to dynamically vary message destinations, as in our first example: a query processor may have several potential "parents" to which it can send partial aggregates [23].

Alternative congestion control algorithms:

Finally, TCP's window-based, sender-driven congestion control algorithm may not be the most appropriate for all applications. Floyd et al. [11] propose TFRC: a rate-based, receiver-driven "TCP-friendly" congestion control algorithm for flows that benefit from slower changes in sending rate, such as some multimedia traffic. DCCP allows for selection of several different congestion control algorithms, of which TFRC is one. Our own experience with overlay network implementations have shown significant advantages to TFRC-like approaches, particularly in latency-sensitive overlays that exhibit high loss or unpredictable message delays.

Selection of particular congestion control algorithms can, of course, be achieved via a parameter to the kernel protocol stack, but when combined with the application routing behavior described above, it becomes hard to build a monolithic protocol implementation that can accomodate different congestion control algorithms, themselves occupying different positions in the data path. A more natural construction factors out congestion control into a replaceable module, a concept we return to below.

Discussion:

Taken as an ensemble, the issues above show that the solution space for overlay networks is much wider than that for client-server applications. This is in part simply because they are distributed, and hence must interact with and adapt to the network as a whole rather than to a single path through it, blurring the boundary between the application and protocol implementation.

The situation is further complicated by heterogeneous requirements *within* an application: different parts of an application may require different transport characteristics. For example, a storage service may have one set of requirements for retrieving blocks or fragments, and another for performing lookups.

The challenge is to provide such flexible functionality in a reusable form. We identify two patterns of reconfiguration in the examples above: (1) *re-ordering* of functionality in the receive and transmit data paths of the implementation, and (2) *substitution* of one of a family of

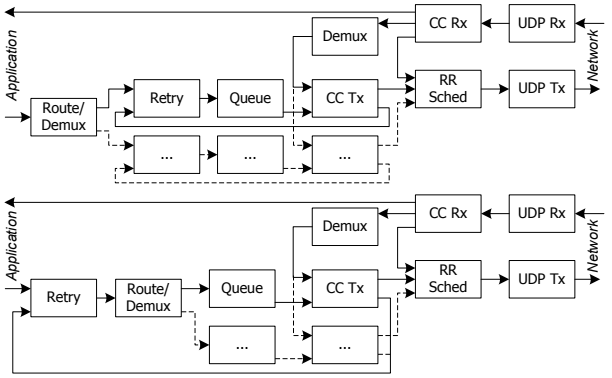

Figure 1: Persistent retries (top) vs. rerouting.

algorithms (e.g., for congestion control).

These are precisely the facilities promised by componentized protocol implementations, suggesting they may have a compelling area of applicability after all.

3 P2's transport stack

P2 [20] is an overlay construction and maintenance facility that uses a high-level declarative query language to specify properties of overlay networks. P2 dynamically translates overlay specifications into Click-like [18] dataflow networks, which are executed to maintain the overlay network. A P2 dataflow network on a particular node consists of C++ objects representing dataflow elements, with bindings between such objects corresponding to arcs on the dataflow graph.

Like Click, dataflow arcs between elements pass data items via "push" or "pull" function calls. However, P2 is primarily a query processor rather than an IP router. Unlike Click, P2 passes relational tuples rather than IP packets. Furthermore, since P2 elements often produce and consume tuples via computation rather than passing them through, P2 dataflows stop and start more frequently and thus have more complex inter-element synchronization and scheduling mechanims.

P2 has a wide repertoire of dataflow element classes, including operators necessary for performing relational query plan operations (joins, aggregations, etc.) over tuples in both streams (e.g., from the network) and local soft-state tables.

P2 extends the dataflow model into the network stack, which is responsible for inter-node tuple transfer. Dataflow elements handle congestion control, marshaling, packet scheduling, and demultiplexing. Our initial motivations for this novel design were ease of implementation and consistency with the rest of the system, but we have come to recognize its value as an abstraction for configuring transport protocols. In the following sections we show

examples of the use of P2 dataflow elements to address the issues identified in Section 2 by selective reordering and substitution.

Sending Retries Down Alternate Paths:

Our first example allows an application to retry transmission of a message to a potentially different destination. Fig. 1 shows two dataflow graphs, each of which represents a possible configuration of transport protocol elements in P2. Broadly, packets to be sent move from left to right, and received packets move from right to left. For simplicity we do not show whether bindings between elements are "push" or "pull," and in some cases we have collapsed chains of elements into a single box where keeping them separate would not have aided comprehension.

The upper diagram in Fig. 1 shows conventional TCP-like behavior for a P2P system. A packet to be sent first passes through a "route/demux" element which uses the overlay's routing table to pick a next-hop destination, and hands the packet to a per-neighbor retry element which enqueues it. A packet is removed from the head of this queue by a per-neighbor transmit-side congestion control element ("CC Tx"), which in turn is scheduled by a global round-robin element that pulls packets from the "CC Tx" elements and sends them to the socket.

Conversely, incoming packets from the network are pushed to a receive-side congestion control element ("CC Rx"). Incoming application data is acknowledged via the element's path back to the round-robin scheduler, and pushed to the application. Incoming acknowledgements from peers are demultiplexed and sent to the appropriate "CC Tx" elements (to maintain congestion state), which subsequently signal the retry elements (to remove data from the retry queue).

The lower diagram in Fig. 1 shows how to achieve behavior more like Bamboo, where successive retries may be sent to different destinations. We simply move the retry element to the head of the dataflow ahead of the route/demux element, which causes each retransmission to perform a new route lookup. Not shown is the logic by which the route lookup has access to congestion and loss statistics about neighbors – this information is maintained as a P2 table, and the policy by which the route/demux element uses this information to select a destination is likely to be application-specific. Note, however, that apart from expressing such preferences, no new code has to be written to achieve the desired behavior.

Figure 2 shows the results of a highly artificial experiment comparing the performance of the two dataflow graphs. We build a very small (32-node) Chord network. One particular node in the network performs 1 lookup per second to a random key. A lossy link (50% drop rate) is placed between that lookup node and its furthest finger table entry. This experiment is, of course, simplistic in the extreme, but it at least serves to isolate the effect of this change to the transport protocol.

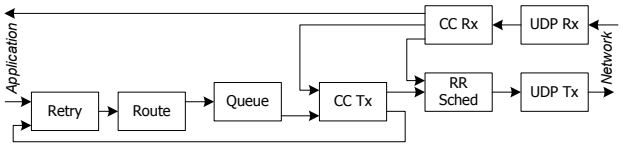

Figure 3: Shared congestion-control state for all destinations.

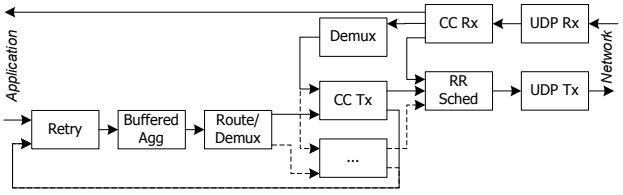

Figure 4: Late data choice.

The results are unsurprising, and in line with much more rigorous studies [9, 24]: moving retries ahead of routing causes more hops to be traversed (since the path may not be the best one available in the routing table), but the latency is reduced (since we fail over links quickly). Figure 2 (c) clearly shows that this is achieved by reducing the number of retries, since the lossy link is quickly avoided.

Shared Congestion Control State:

An overlay STP-like behavior [9], where a single congestion window is maintained for all destinations can also be generated without writing any additional code. Figure 3 illustrates how the same elements used above can be rearranged to provide the functionality of STP using a single queue and congestion control strand for all outgoing packets regardless of destination.

Note that the dataflow model makes it easy to combine orthogonal functionality: we could have elected to perform retries ahead of routing as above without requiring any new code.

Aggregation and late data choice:

Figure 4 shows an example of late data choice for a distributed aggregation scenario, such as computing the maximum or mean of a distributed set of values. We move all buffering in the data flow graph "upstream," next to the application.

This simple change in the dataflow means that the number of aggregate values sent by a node over time is equal to the load the network is willing to handle. When bandwidth is available, values are sent out of the buffer as soon as they are produced, minizing the latency with which the partial aggregates arrive at the next level of the aggrega-

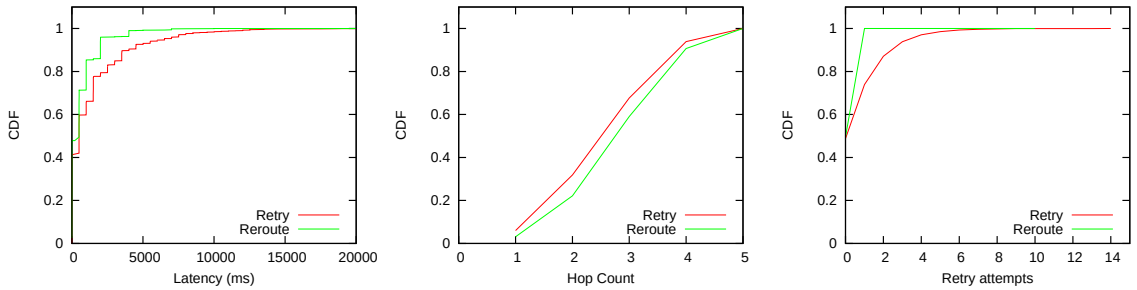

Figure 2: Comparing retry policies for Chord: (a) latency, (b) hop count, (c) retries.

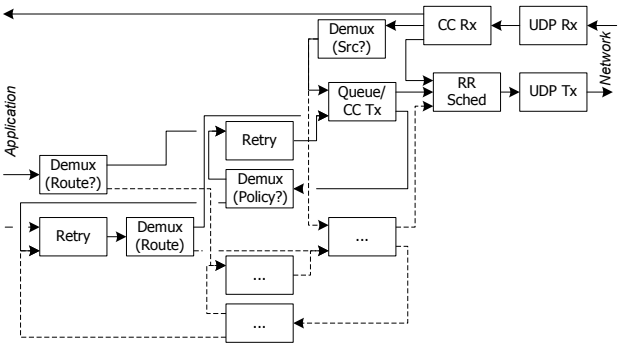

Figure 5: Different, concurrent transport policies.

tion tree. In the event of congestion, values remain in the application buffers and are aggregating with any new values that arrive, reducing bandwidth usage.

Heterogeneous transports:

Our final example briefly illustrates the ease with which different combinations of transport protocol behaviors can be combined in an application in a straightforward manner. Figure 5 shows how different policies can be combined at runtime, and choices made on a per-packet basis.

4 Related work

An early contribution in the decomposition of network protocols came from the x-Kernel operating system [15], in which network services are handled by composable, user-level protocol objects. To specify such objects, the project team developed the Morpheus [3] programming language, which codifies valid protocol object compositions and enables overhead-reducing optimizations. Unlike our lightweight elements, protocol objects in x-Kernel are complete protocol implementations (i.e., TCP, Psync, BLAST), restricting their flexibility to stacks that layer objects on top of one another. With the slightly different motivation of making network protocol specifications more *readable*, Kohler et al. developed the Prolac Protocol Language [17]. Prolac is an expression language, intended as for developing complete protocols (e.g., TCP).

Sharing our high-level goals but taking a completely different approach, Braden et al. [6] propose a heap based protocol abstraction in their Role-Based Architecture (RBA). In RBA, messages are addressed to role actors, instead of end-hosts. This allows middle boxes (e.g., firewalls, NAT boxes, caches, etc.), running as roles, to be addressed in message headers. Conceivably, RBA is an alternative to our dataflow architecture for transport protocols although it seems ultimately intended for larger-granularity actors, compared to our finer-grained plumbing of protocol components before or after a multiplexer.

Instead of offering flexibility via componentization, a number of approaches expose the internals of monolithic protocol specifications, making it easier to pick and choose which flexibility to enable and which to suppress. Mogul et al. [22] describe a put/get interface to the transport layer that allows for the setting and querying of network protocol state. The Datagram Congestion Control Protocol (DCCP) [16] builds a transport "suite" that can flexibly alter its behavior while remaining TCP-friendly. In both cases, the resulting protocol instantiations are geared towards point-to-point communication, and keep all pieces of the transport functionality within the same "shell," making it harder to effect the flexible reordering of components across traditional layer boundaries that we propose in our work.

The software dataflow abstraction has long been used to model data movement in the database literature, in particular in networked and stream query engines [5, 8], which have an intimate relation to data-intensive networking. More recently it has been embraced for network routers [13, 18]. Many of the optimizations found in these systems are a direct result of using a dataflow model, and the flexibility thereof. Network transport protocols can also benefit from a dataflow model, and in this paper we have presented a few such examples. An interesting synergy may also exist between the flexibility we have

seen in this paper and database query optimization techniques, particularly *adaptive* approaches to reoptimizing live dataflows [4, 8].

5 Conclusion

Overlay networks offer a new network model that applications are beginning to demand. This new communication medium brings with it a number of design decisions that go beyond the scope of a small number of monolithic transport protocol services. Component based transport protocols provide a natural replacement of black box protocol implementations, with small processing units that can be arranged to form the desired semantics. Besides flexibility, designing a system around small components promotes good code reuse.

We embrace a dataflow architecture for our component-based transport protocols. Dataflows have been shown to provide good data independence properties in many other systems, and are certainly capable of supporting high-performance operation [7]. The flexibility of a dataflow abstraction makes the rich set of optimizations afforded by overlay networks easier to attain. Moreover, it provides an ideal glue layer between the application and network, one which we have shown to support fresh results and good synchronization properties.

In the future, it may become important to ensure that component-based transport protocols mimic the wire formats of existing transports, particularly TCP. This would ease interoperation with existing middleboxes, allowing for example firewalls and NATs to maintain per-flow state obliviously. Mimicking the particular timing characteristics of specific TCP implementations, however, may prove more challenging.

Finally, a particularly challenging and ambitious next step for our exploration of this space is the specification in a high-level language of *desired properties* for a particular instantiation of protocol components. In P2, we define and compile entire application-level overlay dataflows from such declarative language specifications. Consequently, this approach would allow us to redraw the boundary between the overlay and the transport.

References

[1] Bittorrent goes trackerless: Publishing with bittorrent gets easier! `http://www.bittorrent.com/trackerless.html`.

[2] eDonkey2000 – Overnet. `http://www.edonkey2000.com/`.

[3] M. B. Abbott and L. L. Peterson. A language-based approach to protocol implementation. *IEEE/ACM Transactions on Networking*, 1(1), Feb. 1993.

[4] S. Babu and P. Bizarro. Adaptive query processing in the looking glass. In *Proc. Conference on Innovative Data Systems Research (CIDR)*, 2005.

[5] H. Balakrishnan, M. Balazinska, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, E. Galvez, J. Salz, M. Stonebraker, N. Tatbul, R. Tibbetts, and S. Zdonik. Retrospective on Aurora. *VLDB Journal*, 13(4), Dec. 2004.

[6] R. Braden, T. Faber, and M. Handley. From protocol stack to protocol heap: role-based architecture. *SIGCOMM Comput. Commun. Rev.*, 33(1):17–22, 2003.

[7] E. A. Brewer. Combining Systems and Databases: A Search Engine Retrospective. *Readings in Database Systems, J. M. Hellerstein and M. Stonebraker eds.*, pages 711–724, 2005.

[8] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. A. Shah. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *CIDR*, 2003.

[9] F. Dabek, J. Li, E. Sit, F. Kaashoek, R. Morris, and C. Blake. Designing a DHT for low latency and high throughput. In *Proc. NSDI*, 2004.

[10] S. Floyd, M. Handley, and E. Kohler. Problem Statement for DCCP. `http://www.icir.org/kohler/dccp/draft-ietf-dccp-problem-01.txt`, June 2005.

[11] S. Floyd, M. Handley, J. Padhye, and J. Widmer. Equation-based congestion control for unicast applications. In *Proc. SIGCOMM*, pages 43–56, Stockholm, Sweden, August 2000.

[12] A.-V. T. W. Group, H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RTP: A Transport Protocol for Real-Time Applications. RFC 1889, IETF, Jan. 1996.

[13] M. Handley, A. Ghosh, P. Radoslavov, O. Hodson, and E. Kohler. Designing extensible IP router software. In *Proc. NSDI*, May 2005.

[14] R. Huebsch, B. N. Chun, J. M. Hellerstein, B. T. Loo, P. Maniatis, T. Roscoe, S. Shenker, I. Stoica, and A. R. Yumerefendi. The architecture of PIER: an Internet-scale query processor. In *CIDR*, pages 28–43, 2005.

[15] N. C. Hutchinson and L. L. Peterson. The x-kernel: An architecture for implementing network protocols. *IEEE Trans. Softw. Eng.*, 17(1):64–76, 1991.

[16] E. Kohler, M. Handley, and S. Floyd. Datagram congestion control protocol (dccp). `http://www.icir.org/kohler/dcp/draft-ietf-dccp-spec-11.txt`, Mar 2005. Work in progress.

[17] E. Kohler, M. F. Kaashoek, and D. R. Montgomery. A readable TCP in the Prolac protocol language. In *Proc. SIGCOMM*, 1999.

[18] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click modular router. *ACM Trans. Comput. Syst.*, 18(3):263–297, 2000.

[19] J. Lai and E. Kohler. A Congestion-Controlled Unreliable Datagram API. `http://www.icir.org/kohler/dccp/nsdiabstract.pdf`, March 2005.

[20] B. T. Loo, T. Condie, J. M. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica. Implementing declarative overlays. In *Proc. ACM SOSP*, October 2005.

[21] P. Maymounkov and D. Mazieres. Kademlia: A peer-to-peer information system based on the XOR metric. In *Proc. IPTPS*, 2002.

[22] J. Mogul, L. Brakmo, D. E. Lowell, D. Subhraveti, and J. Moore. Unveiling the transport. *SIGCOMM Comput. Commun. Rev.*, 34(1):99–106, 2004.

[23] S. Nath, P. B. Gibbons, S. Seshan, and Z. Anderson. Synopsis diffusion for robust aggregation in sensor network streams. In *ACM SenSys*. 2004.

[24] S. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz. Handling Churn in a DHT. In *Proc. of the 2004 USENIX Technical Conference*, Boston, MA, USA, 2004.

[25] R. Stewart, Q. Xie, K. Morneault, C. Sharp, H. Schwarzbauer, T. Taylor, I. Rytina, M. Kalla, L. Zhang, and V. Paxson. Stream Control Transmission Protocol. RFC 2960, IETF, Oct. 2000.

[26] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Trans. Netw.*, 11(1):17–32, 2003.

[27] P. Yalagandula and M. Dahlin. A Scalable Distributed Information Management System. In *Proc. SIGCOMM*, Portland, OR, USA, 2004.