**Published in DSN-DCCS 2011**

# Coercing Clients into Facilitating Failover for Object Delivery

Wyatt Lloyd, Michael J. Freedman
*Princeton University*

*Abstract*—**Application-level protocols used for object delivery, such as HTTP, are built atop TCP/IP and inherit its host-to-host abstraction. Given that these services are replicated for scalability, this unnecessarily exposes failures of individual servers to their clients. While changes to both client and server applications can be used to mask such failures, this paper explores the feasibility of transparent recovery for *unmodified* object delivery services (TRODS).**

**The key insight in TRODS is cross-layer visibility and control: TRODS carefully derives reliable storage for application-level state from the mechanics of the transport layer. This state is used to reconstruct object delivery sessions, which are then transparently spliced into the client's ongoing connection. TRODS is fully backwards-compatible, requiring no changes to the clients or server applications. Its performance is competitive with unmodified HTTP services, providing nearly identical throughput while enabling timely failover.**

## I. Introduction

Ideally, a client's interaction with a replicated service will fail only when the service fails. Yet most Internet services tie the fate of a client's connection to a single server, because they are built using TCP and inherit its host-to-host bindings. If this single server fails, the client's connection breaks, and it appears to the client that the service has failed. However, if a new server can transparently *failover* the connection—that is, interact with the client exactly as the original server would have—the client's connection can continue uninterrupted and unaware of the failure.

We aim to enable failover for a large class of Internet services, called *object delivery services*, that play an integral role in users' online experiences by giving clients read-only access to content objects, such as webpages, images, and videos. Object delivery services are typically replicated for scalability and fault-tolerance, e.g., there are tens to thousands of servers that all deliver the same set of objects. If one such server fails while delivering an object, another server has the potential to continue delivering it. This paper demonstrates that such recovery can be done transparently, effectively, and practically.

Our system, Transparent Recovery for Object Delivery Services (TRODS), has been designed with the goal of *immediate deployability*, which introduces two challenges. Clients of the service should not be modified: They are often not under the service's control and often run different applications, browsers, and operating systems. Similarly, the server's application code should not be modified: Source code may be unavailable, and application changes would require integration effort for every service that seeks failover. Instead, TRODS is implemented as a server-side kernel module and requires no changes to the client or application.

At a high level, TRODS operates by ensuring that, at failover time, a recovery server has the minimal application-level information necessary to continue a connection. This information is preserved in two ways. First, it can be retransmitted by the client to its recovery server. TRODS does not modify the client to accomplish this, instead, it leverages its on-path position within the server's kernel to manipulate a connection's TCP packets, in order to coerce the client into retransmitting the information to the new server. Second, the information can be saved to a persistent store that will survive the failure of the original server.

We describe two complementary versions of TRODS that use different resources as persistent stores. The first version, TRODS-KV, uses a key-value store for persistence. It improves on previous failover schemes by requiring only a *single* remote operation apart from the original server—a single save to the key-value store—to guarantee any subsequent connection failover. The second version, TRODS-TS, eliminates the need for *any* remote operations by carefully repurposing the TCP timestamp option that accompanies every packet in a connection as the persistent store. These two approaches are complementary: TRODS-KV is more general purpose, handles more abnormal object delivery scenarios, and avoids some additional security concerns. On the other hand, TRODS-TS has very low overhead and requires no additional physical resources for deployment. Together, TRODS-TS can serve the highly-popular objects of a service, while TRODS-KV can handle the unpopular and exceptional cases.

This paper focuses on the use of HTTP as the canonical and ubiquitous protocol for object delivery. However, we believe that TRODS' approach is similarly applicable to other protocols for object delivery.

TRODS has significantly lower overhead than previous transparent failover schemes. Several of these schemes require primary and backup servers to process requests in parallel, e.g., FT-TCP (hot backup) [24] and ST-TCP [14]. This redundant processing reduces the systems' throughput per machine by at least 50%. Other prior schemes that

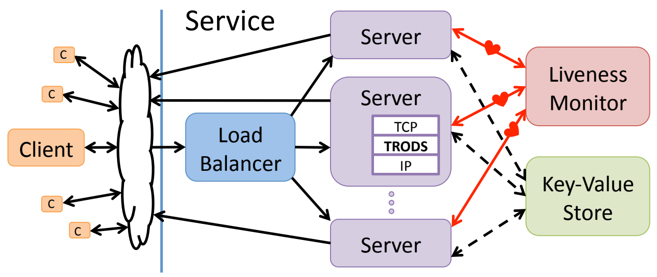


Figure 1. **A typical service architecture that uses TRODS.**

avoid an active backup—e.g., FT-TCP (cold backup) and CoRAL [1]—still require many remote operations to save state so it can be replayed at recovery time. In contrast, TRODS-KV needs only a single remote operation and TRODS-TS eliminates them altogether.

## II. Protocol

This section gives a high-level overview of how TRODS operates. We begin by describing how TRODS fits into the architecture of a typical Internet service. Next, we examine the structure of a connection to an object delivery service. Finally, we detail how TRODS can failover a client's connection during each of its phases.

### A. Architecture

Figure 1 shows the architecture of a typical Internet service using TRODS. Every component is standard and, excluding the key-value store, would likely be found in a TRODS-free version of the service. The clients are unmodified, run their normal networking stack and applications, and connect to the service using TCP. An unmodified load balancer routes all packets in a connection to the same server. A liveness monitor maintains the load balancer's pool of available servers. TRODS does not need a stateful load balancer, so any stateless flow-based hashing suffices, e.g., consistent hashing [9] or standard *mod n* hashing using the flow's 5-tuple for server affinity. The servers terminate the clients' connections and include a TRODS kernel module that sits in their network stacks. This presence allows TRODS to manipulate and control the packets bidirectionally. Figure 1 also includes a key-value store, which TRODS-KV uses as a persistent store, as discussed later.

This figure illustrates one concrete example of a service architecture that uses TRODS. TRODS can work for any general object delivery service that meets three requirements: it is comprised of replicated servers that serve static objects, its servers can all use the same IP address(es), and it has an updatable load balancer.

In this paper, we do not consider the failure of the load-balancer or the liveness monitor. Both can be supported by standard replication and failover techniques, and their state need be linear only in the number of servers, not the number of flows. Further, unlike typical deployments, TRODS can tolerate inconsistent state between load-balancers, e.g., in their known set of live servers. If one sends the subsequent packets of an existing connection to a new server, the connection is dealt with as a normal case of failover. We do consider the failure of the key-value store in §IV-A.

### B. The Anatomy of an ODS Connection

To clarify how TRODS enables failover, we first identify the two stages of a connection to an object delivery service. The first phase is connection setup, where the client and server negotiate what object the client is going to download. In HTTP, for example, this constitutes the HTTP GET request and the server's response header. The second phase of the connection is the object download. In HTTP, this corresponds to the transmission of the response body.

Transparent failover requires a new server to continue a client's interaction with the service over its pre-existing TCP connection. If the client is in the setup phase, the new server should continue negotiating with the client and then start the object delivery. If the client is in the download phase, the new server should continue the client's download exactly where the old server left off.

The two phases of a connection are quite different. The setup phase is typically short, in terms of both bytes and packets, and application-layer data flows in both directions. The download phase can be long, and application-layer data only flows from the server to the client. Accordingly, TRODS handles failover for each phase quite differently.

### C. Failover

TRODS takes the following steps to failover a client's connection on a new server:

1) Detect a server failure.
2) Redirect the client's connection to a new server.
3) Initiate failover on the new server.
4) Determine the connection's current phase.

If in the setup phase:

5) Continue negotiating with the client.

If in the download phase:

5) Determine what object the client is downloading.
6) Determine the client's current offset into the object.
7) Resume sending the object from that point.

**Failure Detection.** To detect server failures, we apply standard unreliable failure detection [3]. Periodically, a liveness monitor sends a heartbeat packet to each server and each server responds with their own packet. The server is determined to have failed if the liveness monitor does not hear from a server for longer than a threshold amount of time (25 ms in our implementation). This scheme detects hardware failures, but not necessarily application failures. Our implementation uses its position in the kernel of each host to locally detect application failures (e.g., process crashes). It then prevents the machine from exposing those failures to the client (e.g., the TRODS module drops RST packets arising in such scenarios), while also triggering failover by the liveness monitor.

**Connection Redirection.** Connections to the failed server must be rerouted to new servers so that failover can begin. Once the liveness monitor detects a server has failed or a new one has started, it updates the load balancer's state about the pool of active servers and their corresponding MAC addresses. The load balancer will then start routing packets to the new set of servers. Now, all new connections will be handled by a live server.[1]

The choice of load-balancing scheme affects how ongoing connections are remapped to servers. If consistent hashing [9] is used, only connections to the failed server will be reassigned elsewhere. By contrast, if a less smooth hashing function is used—such as selecting a server by randomly hashing *mod* the server-pool size—then almost all ongoing connections will be reassigned to new servers. While more disruptive, TRODS still handles this scenario, treating such reassignments as normal cases of failover.

**Failover Initiation.** After a load-balancer redirects a connection, the new server will receive any packets the client sends. The new server will recognize that these packets are in the middle of a TCP connection that does not exist on this server and thus must be failed over. While there will often be outstanding packets in the network when a server fails—especially given the large TCP window size of an ongoing download—TRODS cannot rely on these packets either to exist or to arrive at the new server in order to initiate failover. Instead, TRODS ensures the client will send a packet that reaches the new server by leaving at least one packet from the client unacknowledged at all times, coercing its TCP stack to continue to retransmit it. Fortunately, this does not affect the application-layer connection, as the TCP specification allows the client to receive the server's application-layer response, even when its request has not been acknowledged at the transport layer.

**Determining the Current Phase.** TRODS requires some state to be shared between a connection's original and recovery servers, in order to accurately determine the current phase of the connection. TRODS accomplishes this by blocking a connection from entering the download phase until it has saved some information to a *persistent store* that will survive the failure of the original server. When a new server starts to failover a connection, it first looks up the connection in the persistent store. If the connection is not found, the new server knows the connection is still in the setup phase; otherwise, it is in the download phase. We discuss the corner cases of phase determination in §III.

**Continuing Negotiations.** If the connection is in the setup phase, the new server must continue the negotiation with the client. Negotiation is stateful, which might suggest that TRODS needs to save already-negotiated state to the persistent store, in order to continue negotiation after failover. However, TRODS exploits the short length of the setup phase to avoid this.

Because setup differs between protocols, TRODS deals with each uniquely. The common theme is that TRODS uses control of the TCP layer to effectively coerce the client into providing storage unbeknownst to it. In HTTP, for example, TRODS does not acknowledge the client's request until after the client has entered the download phase. Thus, if a server failure occurs during the setup phase, the client's TCP stack will timeout and retransmit the request so a new server can handle it. Here, TRODS again exploits the separation between application-layer data and TCP-layer acknowledgments, which allows a client's application to operate normally while its transport layer attempts to retransmit packets. We discuss further details in §III.

**Determining the Object.** To continue a connection in the download phase, a new server needs to determine both the object being downloaded and the client's offset into that object. We assume that each service object will have an *objectID*, a unique, concise identifier of the object, such as a filename or URL. We further assume that all objects are immutable, we have omitted the discussion of TRODS' use with versioned and dynamic objects due to space constraints. Thus, if a new server knows the objectID associated with a connection, it knows the object the client is downloading. TRODS makes this objectID available to the new server by persistently storing it.

**Determining the Client's Offset.** Once TRODS has determined which object a client is downloading, it still needs to determine how far into the download the failure occur. TRODS derives this offset by again leveraging cross-layer information. TRODS compares the *objectISN*—the TCP sequence number for the first byte of the object download, which had been saved earlier to the persistent store—and the most recent TCP sequence number the client has acknowledged. The difference between these two values gives the client's current offset into the object; all preceding bytes have been successfully received at the client.

**Resuming Object Downloads.** Once TRODS knows the objectID and offset for a connection, it must transfer the object, starting at this offset, from an application running on the new server. TRODS accomplishes this by initiating a new local connection to the application, and using the objectID to synthesize an application-level request for the client's object. It quickly acknowledges and discards the downloading object until the client's current offset is reached, at which point it begins transmitting the data from the server application to the client. In many applications, this initial "discard" phase can be avoided by requesting the client's offset directly, e.g., through `Range-Request` headers in HTTP.

[1]The handling of new connections is what load-balancer products and high-availability software packages refer to as failover. In these systems, unlike in TRODS, ongoing TCP connections remapped to a different server will be unable to continue.

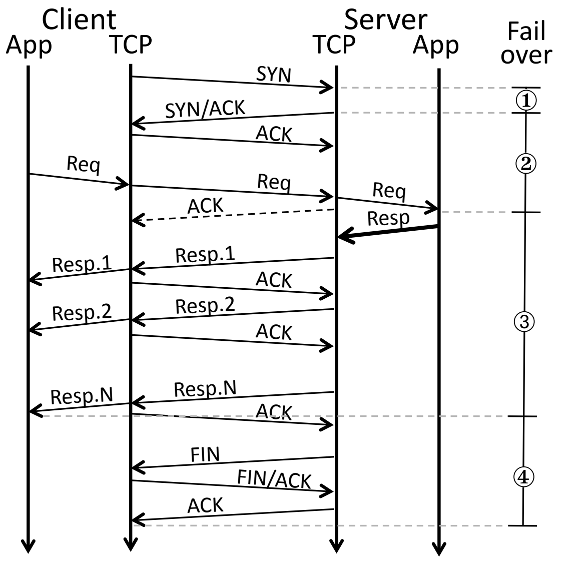


Figure 2. **A typical client-server HTTP connection at both the application and TCP layers. The dashed acknowledgment for the client's request is sent by the server's TCP stack, but dropped by TRODS. The right-most "failover" label indicates a stage of the connection; we detail how TRODS handles failover for each in §III-B.**

## III. TRODS For HTTP

While TRODS provides a general framework for performing failover, it does require a mechanism for extracting a connection's objectID and objectISN, which typically requires application-specific parsing. In HTTP, for example, this objectID is commonly the request URL, while the objectISN is the first byte of the HTTP response body. In our TRODS prototype, this application-specific HTTP knowledge constitutes about 100 lines of code. For concreteness, this section details TRODS' handling of HTTP connections.

We start by exploring how TRODS handles a normal connection at a packet-by-packet level. We then show how this behavior allows TRODS to failover that connection to a new server for all possible connection states.

We make the these assumptions for a typical connection:

1) The request fits in a single packet.
2) The response header fits in a single packet.
3) The response body is less than 4 GB in size.
4) The object download takes less than 13 minutes.
5) Neither persistent nor pipelined connections are used.
6) HTTP chunked transfer encoding is not used.

The first four assumptions hold true for the majority of HTTP connections, and TRODS takes advantage of them to improve performance. The next two assumptions simplify the basic description of TRODS. We complete our specification by relaxing each assumption in §III-C.

### A. Normal Operation

Figure 2 shows a HTTP connection at both the application and transport layers, and Table I briefly summarizes how TRODS interacts with this connection from its position underneath the server's TCP layer.

| Cli | Srv | TRODS Operation |
|---|---|---|
| Syn | | |
| | Syn | Locally store knowledge of this connection |
| Ack | | |
| Req | | Extract and locally save objID |
| | Ack | Drop |
| | $\text{Resp}_1$ | Extract objISN<br>Block until objID/objISN are persistently stored<br>Do not ack client's request |
| Ack | | |
| | $\text{Resp}_2$ | Do not ack client's request |
| Ack | | |
| ... | ... | |
| | Fin | Locally store sequence number of FIN |
| Fin/<br>Ack | | Delete objID/objISN from persistent storage<br>Delete connection from local storage |
| | Ack | Ack client's request and Fin |

Table I
**Normal operation during a typical HTTP connection.**

The connection begins with TCP's three-way handshake. During the handshake when the server sends a response SYN packet, TRODS locally stores knowledge of this connection by saving the client's IP address and port into an in-memory hashtable. This allows TRODS to distinguish between normal packets to the server, whose connections will be in the hashtable, and packets that should initiate failover, whose connections will not be in the hashtable because they originated at another server.

The connection continues with the client sending a HTTP request that fits in a single packet. From this request, TRODS extracts the objectID from the packet, which normally consists of the URI.[2] Under normal processing, the server's transport layer immediately responds to receiving this request with an ACK; TRODS instead drops this packet. If TRODS did not drop this ACK and the server failed after acknowledging the request, but before persistently storing anything, the client's requested objectID would be lost.

The application server then attempts to send the client a response. This response is often too large to fit in a single packet, so the TCP stack on the server distributes it over many TCP segments. The first segment (and packet) will include the response header and the beginning of the response body. TRODS determines the objectISN—the sequence number of the first byte of the response body—by searching through the HTTP payload for the double CRLF that delineates the end of the HTTP response header. TRODS saves the objectISN and objectID to the persistent store, before releasing the TCP stack to transmit the packets back to the client. TRODS also modifies all packets that carry the response to not acknowledge the client's request. This ensures that if the server fails, the client's TCP stack will eventually retransmit a failover-initiating packet.

[2]The objectID may also include some HTTP request headers, such as cookies. If only the URI is used when other headers affect the server's response, the interaction can appear to be non-deterministic, which we omit discussion of due to space constraints.

After the server's TCP stack has transmitted the entire response to the client, it sends a FIN packet to start tearing down the connection. TRODS stores the TCP sequence number for the FIN in its local hashtable, to help it later determine if the client has received the entire response. The client will respond to the server's FIN with a FIN/ACK of its own; TRODS checks that this acknowledges the server's FIN, and then knowing the client has received the entire response, deletes the connection from the persistent store and local hashtable. The connection terminates when the server sends the client an ACK that cumulatively acknowledges the client's request and FIN.

Deleting connection information from the persistent store is performed to reduce saved state, not to maintain correctness. Thus, it can be done in the background or during periods of low-server load; it does not delay the connection.

### B. Failure Recovery

Figure 2 groups the different stages of a connection into failover cases. We now enumerate these stages, showing how TRODS provides failover in each case.

**Before Setup ①.** A server can fail after receiving a client's SYN but before responding with a SYN/ACK. If this happens, the client's TCP stack times out and retransmits its SYN. This SYN will be routed to a new server and the connection will proceed normally. If a server fails after issuing a SYN/ACK but the network drops the packet, the system's behavior is identical. In later cases, we do not discuss drops that are equivalent to scenarios without them.

**During Setup ②.** A server can fail after the client receives the SYN/ACK but before the server sends the response. Because the client's request remains unacknowledged, the client's TCP stack will eventually timeout and retransmit the request. The load balancer will direct this request to a new server, which will initiate failover.

On the new server, TRODS will lookup the client in the persistent store. If the lookup succeeds, the connection is currently in the download phase and is recovered as described in ③. If the lookup fails, the client is still in the setup phase and has not received any part of the response yet. TRODS will then open a TCP connection to the new application server on the localhost and proceed with TCP's three-way handshake. Once the connection is established, TRODS will *splice* together this new connection and the client's connection.[3] The request will then be forwarded to the server and the connection will proceed normally.

**During Download ③.** If a server fails during the download phase of a connection, the client's TCP stack will eventually timeout and retransmit the packet TRODS purposefully did not acknowledge. This packet, or one that was in the network when the server failed, can be combined with the information in the persistent store to find the objectID and objectISN for this connection.

The new TRODS instance that receives this packet will start a connection with the local application instance, sending a request for the object constructed from the objectID. If supported, this request includes a `Range-Request` header, indicating that the application server should start transmitting the object at the client's current offset. The server will respond with a new response header and the object starting at the specified offset. TRODS drops the response header, and it splices together this connection with the client's original one.

**After Download ④.** If the server fails after the client finishes downloading the object, but before TRODS deletes history of the connection from the persistent store, TRODS might attempt failover as in ③. However, the new server's HTTP response will be an error (status code 416), as the range request specified an offset that is one byte past the end of the object. TRODS will recognize that the client has completed the download, drop the server's response, close the connection to the server, delete the client's connection from the persistent store, and, if the client's packet was a FIN, respond to the client with an ACK.

Some packets from a client may be delayed by the network and not arrive until after the connection has completed and TRODS has removed it from its local hashtable. If this occurs, TRODS will attempt to failover the connection. However, as the client has already completed its download and closed the connection, it will respond to any new packets from the server with a RST packet. TRODS forwards this RST to the server, closing the newly-established connection. While this does not affect the correctness of TRODS, it does waste server resources. We describe how to restrict these wasted failover attempts in §V-A, so that they only occur when it is plausible that their original server has failed.

### C. Extensions

For brevity, we omit the detailed explanations of how TRODS handles HTTP connections that violate our assumptions described earlier. Instead, we briefly sketch the main ideas for dealing with any violations. If the request is spread across multiple packets—a rare event for GET requests—TRODS persistently stores each packet before allowing its corresponding acknowledgment to flow back to the client. TRODS handles multi-packet response headers similarly, by saving them in their entirety to the persistent store before allowing them to flow to the client. If an object is over 4GB the TCP sequence numbers will wrap around so TRODS uses separate objectIDs for different sections of the object. If an object download takes more than 13 minutes, the client's connection must be acknowledged to prevent its TCP stack from resetting the connection. TRODS does acknowledge these rare connections, and then saves them to a list in the key-value store for special handling. TRODS handles persis-

[3]TCP splicing joins two separate connections together so that they act as one; it is accomplished by translating the IP addresses, port numbers, and sequence numbers in every packet.

tent and pipelined connections by splitting apart any packets that include data for multiple objects. Chunked-encoding may only be used if it is deterministic across replicas, as its in-line metadata of chunk lengths prohibit TRODS' simple determination of the client's application-level offset into the response object. We have verified that lighttpd's static file and flash video modules are deterministic by examining their source code and expect that most other chunking schemes are as well.

## IV. Persistent Storage

The TRODS protocol refers opaquely to a "persistent store" that assists with saving connection state necessary for failover. This store is *persistent* in that it survives the failure of the original server. In this section, we describe the two persistent stores we implemented.

### A. Key-Value Store

The first persistent store is a key-value storage system (e.g., memcached [5]). The storage key that TRODS uses for each connection is comprised of the client's IP address and port number. The key-value store can be used for arbitrarily-sized objects, which is not true for TCP Timestamps. Thus, if a large store is needed—e.g., when multiple response header packets need to be stored before being sent to the client—TRODS uses the key-value store.

The configuration and deployment of the key-value store trades off efficiency and availability. Key-value storage servers can be colocated in the same rack, cluster, or data-center as application servers. As the key-value store moves closer to its application servers, latency decreases but the probability of correlated failure increases. Data in the key-value store can be replicated for additional fault-tolerance, but even unreplicated storage provides resilience to a single failure: A connection fails only when its application and key-value server fail simultaneously. For this reason, many deployments may choose an in-memory key-value store (e.g., memcached [5]) for low latency and high throughput.

### B. TCP Timestamps

The second persistent store is the TCP timestamp option [8] that accompanies every packet in a connection. Failover in TRODS is always initiated by a packet from the client, which is what makes this store *persistent*. The TCP timestamp option is negotiated during connection setup: Each host attaches the TCP timestamp option to its SYN packet. Once negotiated, each host will attach its own 4-byte timestamp value and a 4-byte timestamp echo reply to every packet. The timestamp echo reply effectively repeats the last timestamp value that a host received. The use of the TCP timestamp option is widespread: It is used by default in modern versions of Linux, FreeBSD, OS X, and Windows. In the rare event that a host does not use the option, TRODS can fall back to its key-value store for persistent storage.

TCP timestamps were intended for two purposes. First, they help improve the accuracy of RTT estimation. A host will subtract the timestamp echo reply in an ACK packet from the current time to obtain a new RTT. This allows the host to accurately sample the RTT at a high rate and is "vitally important" for large TCP window sizes [8]. Thus, when co-opting the TCP timestamp option as persistent storage, TRODS must ensure that it does not interfere with accurate RTT measurement.

Second, the TCP timestamp option helps protect against wrapping sequence numbers (PAWS). PAWS is used to prevent old duplicate segments from a previous connection from corrupting a current connection between the same hosts using exactly the same ports. This will only happen if (1) a client reconnects to the same server in a short window of time (less than 2 maximum segment lifetimes, or about 4 minutes); and (2) in between these connections, the client makes some number of other connections that is an exact multiple of its ephemeral port range.[4] This is sufficiently unlikely that TRODS does not handle this possibility. However, because the client cannot be changed, TRODS' use of the timestamp must not interfere with the client's PAWS processing. To enforce PAWS, the client will drop all packets with a server timestamp that is deemed too "old". TRODS ensures timestamps are non-decreasing in modular 32-bit space,[5] so they will be accepted.

To summarize, the TCP timestamp option provides 32 bits that the client will echo back with two constraints: The timestamps must be non-decreasing in modular 32-bit space and they still must provide accurate RTT measurement. These 32 bits cannot naively hold the objectID and objectISN: The objectISN alone is 32 bits and the objectID has been unconstrained until now. Thus, TRODS must reduce the number of bits needed for the objectID and objectISN to fit in the TCP timestamp, while obeying these constraints.

**5 Bits for the ObjectISN.** The objectISN can be derived by summing two values: the TCP connection's initial sequence number (ISN) and the length of the response header. TRODS uses this property, as well as small changes at the TCP and HTTP levels, to store the objectISN in 5 bits rather than 32.

At the TCP level, we fix the connection's ISN to a value derived from the client's IP and port. This avoids needing any bits to store the connection's ISN, but raises some security concerns that we address in §V-C.

If the response header is longer than a TCP segment size (typically 1448 bytes with the TCP Timestamp option), then the entire response needs to be stored in the key-value store. Consequently, we only consider response headers that are less than 1448 bytes. Storing its length still requires $\lceil \lg 1448 \rceil = 11$ bits. However, TRODS uses an HTTP-level optimization to reduce this further: It pads the response header to a multiple of 64 bytes, which reduces the number of bits needed to $\lceil \lg \lceil 1448/64 \rceil \rceil = 5$. TRODS pads the

[4]The smallest ephemeral port range we could find was 3975 [23].

[5]That is, $ts_a \geq ts_b$ when $0 \leq (ts_a - ts_b) < 2^{31}$ in unsigned 32-bit math.

header by adding linear white space to the last header field, which HTTP clients ignore [4]. Our choice to pad to 64-byte multiples is arbitrary; we could pad to 128 bytes and then only need 4 bits for the response length.

When TRODS pads the header, it misaligns the TCP sequence number space between the client and server: The client has now received more bytes that the server has sent. TRODS modifies the sequence numbers in all subsequent packets to correct for this difference.

**7 Bits for the Timestamp.** TRODS ensures accurate RTT measurement by passing packets to the server's TCP layer with the appropriate timestamps replaced. When the TCP layer passes TRODS a packet for transmission, TRODS saves the timestamp in a per-connection 128-entry array. It then overwrites the packet's original timestamp with its own value that includes a 7-bit index into that connection's array. When TRODS receives a packet to pass up to the local TCP stack, it uses the 7-bit index embedded in its own timestamp to look up the origin timestamp, which it swaps in before sending the packet up the stack.

The use of a 7-bit index limits the number of outstanding timestamps to 128, and TRODS blocks packets to stay under this limit. With a normal MSS size of 1448 bytes, this means at most 185 KB can be in flight from the server at any point (e.g., a connection with a 50 ms RTT could download at 30 Mbps). This behavior seems reasonable for most web services, but if this limit is too low for a particular service, it can increase the size of the array and bit-length of the index, at the cost of requiring either further response header padding or supporting fewer objectIDs.

**20 Bits for the ObjectID.** The objectID is represented by a long, unique string, such as a file path or full URL. This objectID cannot be embedded in a timestamp, so TRODS instead embeds a shorter index that selects from an array of objectIDs. This array is normally static and replicated on each server. With 20 bits, TRODS can uniquely identify over one million objects. If a service has more objects than can fit in the array, it can use the timestamp option as the persistent store for its most popular million objects and a key-value store for less popular objects. Given the Zipfian nature of Web traffic [2], the million popular objects that can use the TCP timestamp option should cover the majority of a service's traffic. TRODS can also consistently update this array to account for new or newly popular objects, but we omit a description of this behavior for brevity.

**Ordering the Fields.** Finally, TRODS orders its fields in the timestamp option carefully, as shown in Figure 3, to ensure they pass the client's PAWS check by being non-decreasing in modular 32-bit space. The timestamp index resides in the highest-order bits, followed by the objectISN, while the objectID resides in the lowest-order bits. The objectID and objectISN field do not change once set, but the timestamp index does: it increases and eventually wraps around. By placing it in the high-order bits, TRODS ensures that when it wraps around, the numerical representation of the timestamp itself wraps around and thus remains non-decreasing.

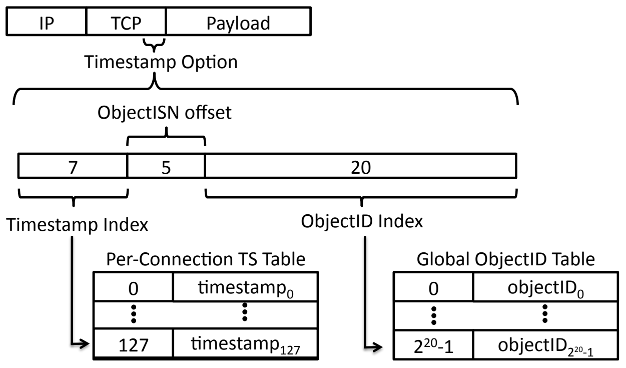


Figure 3. **The relationship between a packet, its TCP timestamp option, the fields TRODS shoehorns into that option, the per-connection timestamp table, and the global objectID table.**

## C. *Combining KV and TS Storage*

The key-value store and timestamp storage play complementary roles. TRODS-KV is more general purpose and scalable, yet introduces higher overhead than TRODS-TS. Thus, one could use these two variants together, and gain the benefits of both. In fact, we evaluate such a dual deployment in §VI. When both variants of TRODS are used together, however, a recovery server needs to know which durable store to access in order to find the connection's state. If the TCP timestamp option is not present, the TRODS module can immediately conclude that a key-value store lookup is required. If the option is present, TRODS stores a hint indicating which store to access in the high-order 7 bits of the timestamp. When the key-value store is used these bits are set to a special reserved value that TRODS-TS does not use as a timestamp index. For these connections, TRODS still needs to perform translation on the timestamps.

## V. SECURITY CONCERNS

The use of TRODS introduces some security concerns: attackers can spoof packets to try to initiate TRODS failover, they can modify TCP timestamps to attempt to gain access to unauthorized content, and they can more readily guess TCP sequence numbers to spoof or hijack a TCP connection. This section describes how TRODS mitigates these concerns.

## A. *Denial-of-Service Attacks*

**Bogus ACKs and Requests.** An attacker can send requests or ACK packets to a TRODS-enabled service with spoofed, random client addresses, attempting to cause TRODS to failover non-existent connections. After all, TRODS' normal response to an unknown request or ACK packet is to initiate failover to its local application instance, wasting both application and persistent store resources.

TRODS can limit its vulnerability to such DoS attacks by initiating failover only when it can verify that it received this packet because another server recently failed. To support this, we replicate the load balancing information to the TRODS instance on each server, i.e., its key range in the case of consistent hashing, or the server pool size ($n$) and its assigned number in the case of *mod n* hashing. If a failure-initiating packet arriving at a server is outside its known range—i.e., it should not be selected given the packet's 5-tuple and its knowledge of the load balancer's hashing scheme and state—then the server would only have received this packet if the load balancer's server pool recently changed. In this case, the server initiates failover. Otherwise, when the packet is in the server's known range, it is dropped as illegitimate, as it should have been in the server's local hashtable.

Therefore, TRODS mitigates this failure-initiation DoS attack, as it can be performed only temporarily on the servers directly affected by another's failure. TRODS can weaken this attack further by giving normal packet processing higher priority than failover processing. This reduces the attack from a denial-of-service to a denial-of-failover.

**Clients Forcing the Slow Path.** A client can force TRODS onto the slow path by sending requests that are longer than two packets and thus need to be saved to the key-value store. It can also force the slow path by sending a request that results in a multi-packet response header, which also needs to be saved to the key-value store. In either case, TRODS has no way to distinguish legitimate slow-path connections from malicious ones, so it must serve them all. However, TRODS can limit the attacker's damage by lowering the processing priority of slow-path connections, as it does with failover. Thus, slow-path attacks can still degrade the service of other slow-path connections, but they have difficulty in degrading the service of normal connections.

### *B. Accessing Unauthorized Content*

When TCP timestamps are used for persistent storage, a client can potentially download an object they do not have permission to access, by sending an ACK packet to trigger failover with a timestamp that indexes an unauthorized objectID. This is partially unavoidable when timestamps are used, but given the enhancements to TRODS in §V-A, clients can only trigger failover after an actual failure has occurred. Thus, this attack will only work when a server has failed recently, and the attacker can guess the objectID index for the object it desires. If these security measures are not sufficient, a service should use TRODS' key-value store for all protected content. With TRODS-KV, the objectID of the client's download cannot be modified by the client.

### *C. TCP Sequence Number Guessing*

When TRODS-TS is used, the server uses an ISN that is generated deterministically from the client's IP and port. This will raise security concerns for anyone familiar with TCP sequence number guessing attacks [15]. In these attacks, an attacker spoofs a SYN packet from a client, and then spoofs an ACK packet that acknowledges a guess of the server's ISN. If this guess is correct, it completes the TCP three-way handshake and the attacker can send a data packet that appears to be from the client.

TRODS is not vulnerable to traditional sequence number hijacking, but its approach allows malicious clients, once having completed a successful connection, to initiate new downloads before fully establishing new connections.[6] This vector may be used as a denial-of-service attack. In particular, rather than randomly, TRODS generates its ISN from a cryptographic hash of the client IP, port, time epoch, and a private key that is known to all servers in a TRODS cluster. Thus, an attacker learns the ISN for a given IP and port only if it can receive traffic at that network location. This is akin to the protections offered by normal randomized ISNs, except that this ISN is constant across the entire epoch. After learning the ISN, a client then can spoof connections from other network locations during the same epoch. That said, TRODS is used for services that are inherently read-only and so the attack can only be used to start illegitimate downloads and cannot modify state. If limiting the sequence number guessing attack to a DoS attack from a limited range of IPs and ports is unacceptable for a service, it should use TRODS-KV instead.

## VI. Evaluation

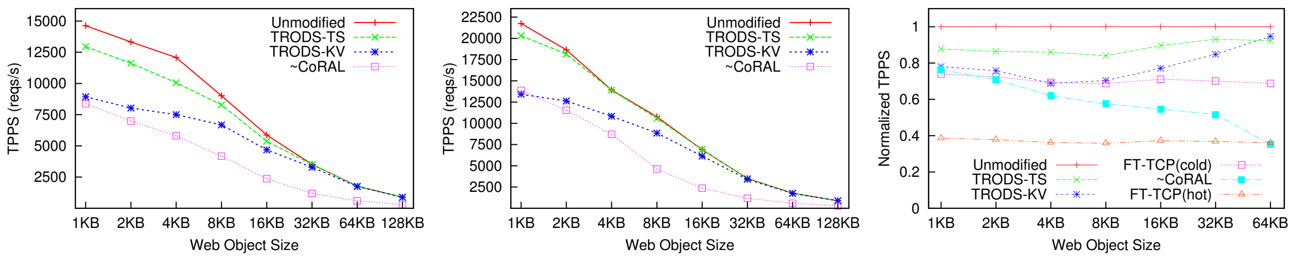
This section demonstrates the practicality and effectiveness of TRODS. We first quantify TRODS' cost in terms of decreased throughput and increased latency. We then evaluate how TRODS handles failure in a cluster setting and how much excess latency it incurs due to failover.

**Implementation.** The TRODS implementation is approximately 3,000 lines of C code. It is a loadable kernel module for Linux 2.6.32.3 and using it does not require recompiling the base kernel or rebooting the machine. The current TRODS implementation handles the normal cases, where none of our assumptions from Section §III are violated.

We also implemented ~CoRAL, a partial implementation of CoRAL [1] in a kernel module for Linux 2.6.32.3. CoRAL routes requests to a primary server through a backup server and saves the entire response on the backup before sending any of it to the client. Our implementation only saves the response on a backup machine and thus gives a rough upper bound on the true performance of CoRAL.

**Experimental Setup.** Our lighttpd throughput, hybrid throughput, and latency experiments use a total of three machines: one to run clients, one for a TRODS server, and one for a key-value store. The excess-latency-due-to-failover experiment uses those machines and an additional one for load-balancing. The failure recovery experiment uses three server machines, as well as one machine each for clients, a load balancer, and the key-value store. Each machine used in these experiments has eight 2.3GHz cores and 8 GB

[6]Note that clients of a TRODS service are not vulnerable to this attack, as they use standard TCP ISNs.



(a) Lighttpd web server, 1 core (b) Lighttpd web server, 8 cores (c) Apache web server

Figure 4. **HTTP throughput experiments. The first experiment (a) uses lighttpd with a single server core, which becomes CPU bound. The second (b) uses lighttpd with 8 cores and is not CPU bound. The final (c) uses Apache in order to compare TRODS to FT-TCP. For all, the median value over 25 trials is shown; min and max values are within 5% of the median and omitted.**

of memory, and is connected to a 1 Gbps switch. Our Apache throughput experiment was run on Emulab [22] using pc3000 nodes connected via 1 Gbps links.

We use memcached 1.4.4 [5] without expiration or eviction as our key-value store and we use lighttpd 1.4.23 [12] as our regular web server. We use a simple Click [11] configuration run in the kernel as our load balancer.

All throughput experiments show the median value of 25 trials; the min and max values are always within 5% of the median and are omitted. Each throughput trial consisted of enough client processes to saturate server throughput continuously fetching a web object. We ran the tests for 40s and exclude the first and last 5s of each trial to avoid including experimental artifacts or non-steady-state rates.

**Throughput Per Server (TPPS).** We evaluate the throughput per server of each system. This throughput metric accounts for *all* the machines needed to provide failover. For instance, in TRODS-KV, it accounts for the use of the key-value store, while in FT-TCP, it accounts for the use of the backup server. These "complete system costs" allow us to more accurately evaluate each approaches' overhead compared to an unmodified system lacking failover capabilities; prior work largely avoided such comparisons.

We calculate TPPS by dividing the throughput of primary servers by $1+TP_P/TP_S$, where $TP_P$ and $TP_S$ are the primary and secondary servers' throughputs, respectively. So, in a "hot backup" scheme where each primary has its own backup ($TP_P = TP_S$), the TPPS is $\frac{TP_P}{1+(TP_P/TP_S)} = \frac{TP_P}{2}$. To calculate the TPPS for TRODS-KV, we benchmarked its fixed size saves to our memcached key-value store at 123,468 saves/s. Then, in Figure 4, when the primary server for TRODS-KV achieved a throughput of $TP_P$, we report a TPPS of $\frac{TP_P}{1+TP_P/123468}$. To calculate the TPPS for ~CoRAL, we benchmarked its object-sized saves to our memcached backup. For 1K objects, memcached can handle 95,037 saves/sec and the CoRAL primary can handle 9,209 requests/sec, so the 1K TPPS is $\frac{9209}{(1+(9209/95037))} = 8395$.

## A. Throughput

We ran three sets of experiments to examine how TRODS affects the maximum throughput of a web server.

In our first experiment, shown in Figure 4(a), we turned off all but one of the cores on the server machine, and ran the web server as a single process, which ended up consuming 100% of the CPU. TRODS operations in the kernel steal cycles from the web server, leading to a larger performance degradation that in the non-CPU-bound experiment. Using only a single core, TRODS-TS experiences an 11% decrease in TPPS for 1 KB web objects (from 14,608 requests/s to 12,980 reqs/s). TRODS-KV sees a 39% reduction in TPPS (to 8,940 reqs/s). TRODS-KV's higher overhead arose from both its reduced throughput on the web server machine, as well as its consumed resources on the key-value store. However, as object sizes increase, the web server becomes less CPU bound and, as a result, the throughput of both TRODS variants become competitive with the unmodified service. For example, when objects are 32 KB, TRODS-TS's TPPS is less than 0.01% lower than the unmodified system, and TRODS-KV's TPPS is only 20% lower. In contrast, ~CoRAL sees a 43% lower throughput for 1 KB web objects than an unmodified service, and this grows to a 66% lower TPPS for 32 KB objects: ~CoRAL's overhead grows as object sizes increase, as saving the entire object on a backup machine becomes more costly.

In our second experiment, shown in Figure 4(b), all 8 cores on the server are used by 8 server processes. The web server is no longer CPU bound and TRODS processing has a smaller effect on throughput. For 1 KB web objects, TRODS-TS decreases TPPS by 7% (from 21,745 reqs/s to 20,315 reqs/s), while TRODS-KV decreases TPPS by 38%. As with a single core, as object size increases, the TRODS variants become more and more competitive with the unmodified service: TRODS-TS is within 3% of the unmodified for all objects 2 KB and larger, while TRODS-KV is within 11% for all objects 16 KB and larger. On the other hand, ~CoRAL again experiences a 40% decrease in TPPS for 1 KB web object, with its relative TPPS continuing to worsen as object size increases.

In our third experiment, shown in Figure 4(c), we evaluated throughput with the Apache web server included with the FT-TCP codebase. As FT-TCP requires a Linux 2.4.20 kernel, we ran these experiments on Emulab. Figure 4(c) shows the TPPS for FT-TCP's hot and cold backup approaches,[7] normalized against the TPPS for unmodified Apache on a 2.4 kernel. The figure also includes the TPPS for TRODS and ~CoRAL, normalized against unmodified Apache on a 2.6 kernel, which is slightly more efficient than on the older kernel. Both TRODS and ~CoRAL exhibit behavior similar to the previous experiments. FT-TCP's variants exhibit relatively stable normalized TPPS, as the amount of additional work the scheme uses is a fixed percentage of the total work that a given response requires.

In summary, when objects are 16 KB or larger, TRODS is competitive with an unmodified service and achieves much lower overhead than either ~CoRAL or FT-TCP. When objects are small and the server is CPU limited, TRODS suffers moderately decreased throughput, but it still has lower overhead than ~CoRAL or FT-TCP.

### B. Hybrid Throughput

Figure 5(a) characterize TRODS' performance when using our two persistent stores in varying proportions. We measure the median throughput of 25 trials, run with the same parameters as our previous single-core throughput experiment. We normalize these throughputs against that achieved when only using the key-value store (i.e., 100% KV), to demonstrate the relative performance gains from a hybrid deployment. For reference, we also plot an "ideal 50/50" line that shows the average of TRODS-KV and TRODS-TS.

The experiment demonstrates that the hybrid version of TRODS performs well. When 50% of connections use the KV store and the other 50% use timestamps, TRODS throughput is within 4% of the ideal. This alleviates our concern that requests that use the slower KV store will unduly decrease the throughput of the hybrid system.

### C. Latency

Table 5(b) shows the median and 99th percentile latencies for different sections of 10,000 sequential fetches of a 1 byte web object. The latencies are measured by analyzing `tcpdump` logs of the client's connections.

The period between the client sending a SYN packet and receiving a SYN/ACK experienced no additional latency for either variant of TRODS: storing local knowledge of a connection has no discernible overhead. Between sending an HTTP request (Req) and receiving the first data packet in the response (Resp.1), TRODS-KV has notably higher latency than an unmodified service. This comes from TRODS-KV blocking the connection until its save to the key-value store completes. In contrast, TRODS-TS does not block the connection and avoids any latency penalty.

Examining the latency of the entire connection (the SYN-FIN section) reveals that TRODS-TS imposes less than 10 $\mu s$ of additional latency, while TRODS-KV imposes less than 150 $\mu s$ of additional latency. Both of these increases are miniscule compared to the tens to hundreds of milliseconds of delay between clients and servers across the wide-area.

Having shown that TRODS adds little to no overhead to server throughput, we now demonstrate that it successfully recovers client connections from server failures. Figure 5(c) shows the per-second throughput of a 3-server cluster over a time period with individual server failures. Each server runs TRODS-TS; the resulting graph for TRODS-KV (not shown) is similar. Web requests for 8 KB objects are concurrently issued by 200 HTTP clients (running on the same physical machine), who access the cluster through a single load-balancer. The load-balancer also delays packets to create a synthetic 20 ms RTT from clients to servers, emulating wide-area connection latencies to nearby datacenters.

We fail server 1 during the 7th second of the experiment by taking down its network interface. Upon detecting this failure, the load balancer updates its server pool. Previously established connections to server 1 are reassigned to the remaining servers; the TRODS components of these servers recover the reassigned connections. We further fail server 2 during the 20th second of the experiment, at which time the load balancer directs all connections to the remaining server. We find that the cluster's total throughput, as shown in the solid line at the top of Figure 5(c), remains constant throughout the experiment; the overhead of recovering failed connections does not have a noticeable effect.[8]

### D. Failover Latency

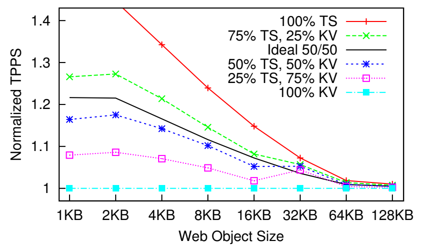
While we have demonstrated that TRODS successfully fails over connections between servers, this does not quantify the delay experienced by clients during this process. Figure 6 shows the *excess* latency due to failover for a client requesting various object sizes. The figure shows the results for TRODS-KV using a single server; the results for TRODS-TS are similar and omitted. A load-balancer sits on path and delays packets to create a 20 ms RTT. For each run, a client requests an object of a given size for 5 minutes, while we synthesize a failure in the every 4 seconds at the server.

We synthesize the failure in the kernel module by dropping all packets during failure period and sending RST packets to the server application for all existing connections (we can synthesize such failures at a much higher rate than we can induce actual ones). We simulate the use of a 25 ms heartbeat timer by setting the failure period to a random amount of time less than 25 ms, effectively mimicking the behavior we observed in our previous experiment.

We measure latency as the time between the client sending its first SYN packet and receiving the final byte of the response. A transfer's excess latency is defined as its increase over the median of all non-failed connections during the run. We graph the $95^{th}$ percentile of the excess latency for non-failed connections. A full 75% of failed-over connections

[7]Note that we disabled system call interception for this experiment, as it is unnecessary for deterministic services.

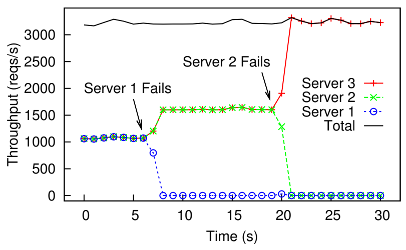
[8]We limit the cluster's throughput to ~3K reqs/s to ensure that `tcpdump`, which we use to record the experiment, does not drop any packets.



(a) Hybrid throughput experiment showing the relative performance of TRODS when both persistent stores are used. We plot the throughput of each proportion normalized against 100% KV.

| Start | End | Normal | TRODS-TS | TRODS-KV |
|---|---|---|---|---|
| SYN | SYN/ACK | 90 (95) | 89 (93) | 90 (94) |
| Req | Resp.1 | 137 (150) | 135 (49) | 256 (282) |
| SYN | FIN | 353 (372) | 362 (384) | 490 (520) |

(b) Median (and 99th percentile) latency in $\mu s$ for different sections of a single HTTP connection, for both unmodified and TRODS services. See Figure 2 for a depiction of these sections.



(c) Server and cluster throughput using TRODS-TS when undergoing failure recovery. The cluster's total throughput is unaffected by these failures, and no connections are broken.

Figure 5.

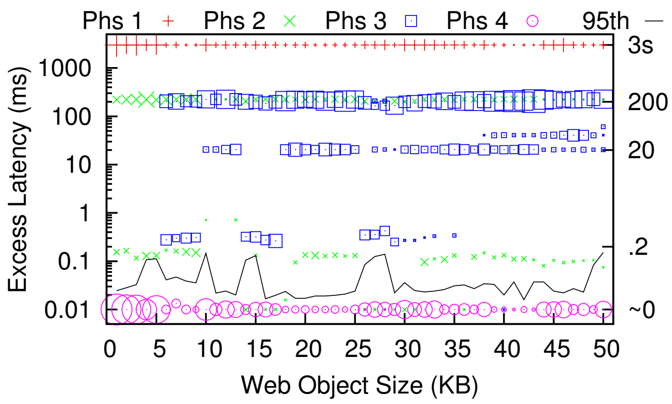


Figure 6. **Excess latency when experiencing failover using TRODS-KV. Overlapping data points that occur in the same phase of the connection are combined and the size of the shown points is proportional to the number of data points they represent. The different failover phases are shown in Figure 2.**

had excess latency above this $95^{th}$ percentile, suggesting that this excess latency is due to failover and not noise.

Each data point in Figure 6 indicates the connection phase (as specified in §III-B) during which the failure occurred. If data points in the same phase overlap (within 10% of each other), we combine them and increase the plotted size of the representative point. We find that the excess latency has a multi-modal distribution, with distinct modes at ~0 ms, .2 ms, 20 ms, 200 ms, and 3 s, as marked on the right y-axis of the figure. We briefly characterize these sources of added latency, in turn:

- **~0 ms**: Phase 4 failovers occur after the client has received the entire response and thus add no latency.
- **.2 ms**: This excess latency corresponds to the time needed to perform a key-value lookup, setup a new connection to the server, and splice in the client's connection. We observe it when failover is triggered by an in-flight packet.
- **20 ms**: Normal connections establish a large TCP window by the time they reach the middle of the download phase. When failover occurs at this point, it sets up a new connection, and thus resets the window size. This increases the number of RTTs needed to transmit the entire object and creates a cluster of failovers around one RTT (20 ms).
- **200 ms**: Our experiment uses Linux clients that have a 200 ms minimum value for their retransmit timeout. Thus, even though the RTT of the connection is well below 200 ms, we always incur at least that latency when failover is triggered by a retransmitted client packet.
- **3 s**: When a SYN packet is lost, the client does not have an estimate of the connection's RTT. Thus, it waits a conservative 3 s before retransmitting.

As object size increases, the proportion of phase 3 failovers also increases. This is not surprising: As a larger proportion of a connection's transfer time is spent downloading an object (i.e., in phase 3), the likelihood of failure occurring during that phase similarly increases. Notice that there are no phase 3 failovers for pages 5 KB and smaller. This occurs because objects of this size or smaller can be sent in a single TCP window with all packets within the window separated by less than 15 $\mu s$. This small gap makes it highly likely that the server will fail before (phase 2) or after (phase 4) sending them all, and we did not observe any contradictions to this in the course of our experiment.

In summary, most failovers in TRODS occurs with less than 200 ms of excess latency, sufficiently low to not noticeably impact a user's experience. While some connections experience a 3 s delay (12% of larger objects), this delay is unavoidable due to SYN retransmission timers. In either case, we argue it is still preferable to a broken connection.

## VII. Related Work

**New Transport Layers.** Several solutions for failure recovery introduce new transport layer protocols or primitives. Trickles [16] uses a new transport layer protocol and a new sockets API to make one end of a connection stateless. SCTP [19] is a transport layer protocol that, among many other things, allows a client to have connections to multiple servers it can switch between. TCP Migrate [17] can be used to migrate a connection from one server to another, which can then be used with a soft-state synchronization protocol between servers to accomplish failover [18]. M-TCP [20] is a another TCP-like transport protocol designed to support migration. All of these solutions modify the client's TCP/IP stack, TRODS does not require any client-side changes.

**TCP Failover.** Moving up the stack, there is a large body of work on failover for TCP connections that do not require changes to clients. FT-TCP [24] accomplishes TCP failover by logging (persistently storing) every packet in a TCP connection on a primary server to a backup server. Then, if the primary server fails, the (cold) backup runs through the TCP connection, and, once it catches up to the client's current position in the stream, it begins serving the client. As this can make the time to failover a connection arbitrarily long, they also describe a hot backup that processes all packets upon receiving them. FT-TCP is more general than TRODS, as it applies to all deterministic TCP services. However, FT-TCP pays for this generality with increased overhead. Every packet must be logged or processed in FT-TCP, while TRODS-KV only "logs" once per object and TRODS-TS avoids it entirely. Koch *et al.*describe a system [10] that is very similar to FT-TCP's hot backup approach. ST-TCP [14] is another primary-backup system that avoids some logging overhead placing the primary and backup on the same L2 network and having the backup snoop on the primary's traffic. Zhang *et al.* [25] describe a similar system that uses a stateful load-balancer to explicitly transmit packets to both the primary and backup. The Backdoors [21] avoids logging by using programmable NICs to extract TCP and application state from the server's memory after an OS crash.

**HTTP Failover.** CoRAL [1] is primary-backup system targeted at HTTP. All packets bound for the primary are first routed through the backup who logs them. The primary then uses application-level knowledge to identify the full reply and persistently store it on the backup. TRODS is more efficient that CoRAL because it avoids persistently storing the entire reply. Luo *et al.* [13] describes a system for HTTP failover where a "dispatcher" (load balancer) terminates the client's connections. Once a client has sent an entire request, the load balancer stores it and forwards it onto a server. Then if that server fails before fully responding, the load balancer reconnects to a new server to continue. This moves the problem of failure from the servers to the load balancer.

**TCP Timestamps.** We are not the first to use the TCP timestamp option for embedding state. Giffin *et al.* [6] use the low order bits of the TCP timestamp as a covert channel for undetectable communication. In addition, starting with version 2.6.26, the Linux kernel added support for window scaling and SACK options in SYN cookies by encoding their value in the lowest 9 bits of the TCP timestamp [7].

## VIII. Conclusion

TRODS is a fully backwards-compatible system for introducing transparent failover to object delivery services. TRODS leverages cross-layer knowledge of both application and TCP state, as well as TCP's reliable transmission mechanisms, to exert control over unmodified clients. This control allows TRODS to coerce clients into providing storage and initiating failover when needed.

Through evaluation of a TRODS cluster, we demonstrate that it is both practical and efficient. In failure-free scenarios, TRODS does not significantly increase latency or decrease server throughput. When a failure occurs, TRODS transparently restores clients' ongoing connections, without adding significant latency to the connections.

## Acknowledgment

The authors would like to thank Erik Nordström, Muneeb Ali, Anirudh Badam, Nate Foster, Prem Gopalan, Rob Harrison, Michael Kaminsky, Wonho Kim, Steven Ko, David Shue, Jeff Terrace, and Vijay Vasudevan for helpful comments on earlier drafts of this paper. This work was supported by NSF CAREER Grant #0953197 (CSR) and an ONR Young Investigator Award.

## References

[1] N. Aghdaie and Y. Tamir. CoRAL: A transparent fault-tolerant web service. *Journal of Systems and Software*, 82(1), 2009.

[2] A. Broder et al. Graph structure in the web. In *Proc. World Wide Web Conference (WWW)*, May 2000.

[3] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 1996.

[4] R. Fielding et al. RFC 2616: HTTP/1.1, Jun 1999.

[5] B. Fitzpatrick. Memcached: a distributed memory object caching system. http://memcached.org/, 2009.

[6] J. Giffin, R. Greenstadt, P. Litwack, and R. Tibbetts. Covert messaging through tcp timestamps. In *Proc. PET*, Apr. 2002.

[7] Improving syncookies. http://lwn.net/Articles/277146/, Apr 2008.

[8] V. Jacobson, R. Braden, and D. Borman. RFC 1323: Tcp extensions for high performance, May 1992.

[9] D. Karger et al. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In *Proc. Symposium on Theory of Computing (STOC)*, May 1997.

[10] R. R. Koch, S. Hortikar, L. E. Moser, and P. M. Melliar-Smith. Transparent tcp connection failover. *Proc. DSN*, June 2003.

[11] E. Kohler et al. The click modular router. *ACM TOCS*, 2000.

[12] Lighttpd. http://www.lighttpd.net/, 2010.

[13] M. Luo and C. Yang. Constructing zero-loss web services. In *Proc. INFOCOM*, Apr. 2001.

[14] M. Marwah, S. Mishra, and C. Fetzer. Tcp server fault tolerance using connection migration to a backup server. In *Proc. DSN*, Jun 2003.

[15] R. Morris. A weakness in the 4.2bsd unix tcp/ip software. Technical Report TR-117, Bell Labs, 1985.

[16] A. Shieh, A. C. Myers, and E. G. Sirer. A stateless approach to connection-oriented protocols. *ACM TOCS*, 26, 2008.

[17] A. C. Snoeren and H. Balakrishnan. An end-to-end approach to host mobility. In *Proc. MobiCom*, Aug. 2000.

[18] A. C. Snoeren, D. G. Andersen, and H. Balakrishnan. Fine-grained failover using connection migration. In *Proc. USITS*, Mar. 2001.

[19] R. Stewart. RFC 4960: SCTP, Sep 2007.

[20] F. Sultan et al. Migratory tcp: connection migration for service continuity in the internet. In *Proc. ICDCS*, July 2002.

[21] F. Sultan et al. Recovering internet service sessions from operating system failures. *IEEE Internet Computing*, 9(2), 2005.

[22] B. White et al. An integrated experimental environment for distributed systems and networks. In *Proc. OSDI*, Dec. 2002.

[23] Windows ephemeral port range. http://support.microsoft.com/kb/929851, 2009.

[24] D. Zagorodnov et al. Practical and low-overhead masking of failures of tcp-based servers. *ACM TOCS*, 27(2), 2009.

[25] R. Zhang, T. F. Abdelzaher, and J. A. Stankovic. Efficient tcp connection failover in web server clusters. In *Proc. INFOCOM*, Mar. 2004.