Transparent Recovery for Object Delivery Services

Wyatt Lloyd and Michael J. Freedman
Princeton University

Abstract

Application-level protocols used for object delivery, such as HTTP and FTP, are built almost universally atop TCP/IP and thus inherit its host-to-host abstraction. Given that object delivery services are replicated for scalability, this host-to-host abstraction unnecessarily exposes failures of individual service replicas to their clients. While changes to both client and server applications could 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, the network, or the object delivery service. We implement TRODS as a server-side kernel module and experimentally demonstrate that its performance is competitive with unmodified HTTP services.

1 Introduction

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

We aim to enable failover for a large class of Internet services, called *object delivery services*, that give clients read-only access to a set of content objects, such as webpages, images, or videos. These services are typically replicated for scalability, *e.g.*, there are dozens or hundreds of servers that all deliver the same set of images. This replication makes these services amenable to failover: If one server fails while delivering an object, another server with access to the object should be able to continue delivering it. Our system, Transparent Recovery for Object Delivery Services (TRODS), is an efficient way to enable exactly that.

TRODS has been designed with the goal of *immediate deployability*, which brings with it several restrictions. First, clients of the service cannot be modified: they are often not under the service's control and often run different applications, browsers, and operating systems. Second, the server's application code cannot 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, requiring no changes to the client or application service.

At a high level, TRODS operates by ensuring that the minimal application-level information needed to continue a connection is available to a recovery server at failover time. This information can be preserved in two ways. First, it can be retransmitted by the client to its new server. While TRODS cannot modify the client, it uses its position in the server's kernel to manipulate a connection's TCP packets to coerce the client into retransmitting information to the new server. Second, the information can be saved to a persistent store that can 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 local to the server for persistence. It improves on previous failover schemes by requiring only a single on-path remote operation by the original server—a save to the key-value store—to guarantee any subsequent connection failover. The second version, TRODS-TS, takes this a step farther and eliminates the need for any remote operations. TRODS-TS carefully repurposes the TCP timestamp option that accompanies every packet in a connection as the persistent store.

These two TRODS approaches provide complementary functionality: TRODS-KV is more general purpose, provides greater scalability, 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-KS can handle the unpopular or exceptional cases.

Finally, TRODS makes the service operator's job easier. TRODS allows a service to use stateless load-balancers, which are easier to replicate and scale out than their stateful counterparts.

1.1 Why Not an End-to-End Solution?

End-to-end solutions to failover offer an architecturally pure approach. These include new transport layer protocols that can be designed to specifically allow failover [20, 21, 23, 24], as well as application-level solutions that modify the client so it reconnects to a new server if its current server fails. These end-to-end solutions almost by definition require client-side changes, however.

Yet, one particular application-level end-to-end solution for HTTP deserves a closer look: JavaScript is ubiquitous in web browsers and can download new client-side instructions on demand. For example, when a client requests a webpage, the returned object's JavaScript can set timeouts for downloading embedded objects which, if triggered, retry the downloads without the user's intervention.

This approach is promising, but it suffers from three problems that illuminate a few of TRODS' benefits. First, bootstrapping this approach can be problematic, as the initial page, with its JavaScript control loop, does not have the potential for failover. TRODS, by contrast, can failover any connection and does not have a bootstrapping problem. Second, failure detection needs to occur across the wide-area network, leading either to overly conservative timeouts (and significant user-perceived delays) or to aggressive retry attempts (and unnecessary service load). TRODS performs failure detection inside the datacenter and completes failover in less than a second. Third, the JavaScript approach may add failover from the wrong administrative domain, as webpages often use object delivery from third parties, such as advertisers or content delivery networks (CDNs). And bootstrapping again does not necessarily help, *e.g.*, embedded images cannot execute JavaScript themselves. TRODS allows separate administrative domains to control their own failover mechanisms, and does so in a more application-independent manner. That said, this JavaScript approach can recover from failures at the network routing or entire datacenter-level while TRODS is only designed to handle failures between machines in the same datacenter.

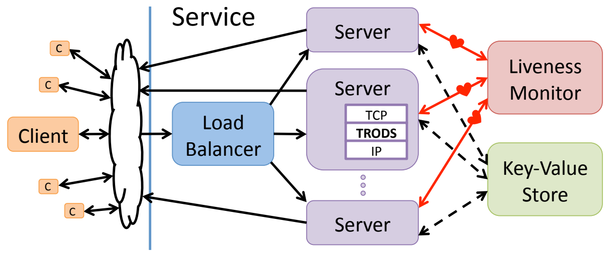

Figure 1: **A typical service architecture that uses TRODS.**

The remainder of the paper is organized as follows. Section §2 describes a typical service architecture and provides a high level view of TRODS' design. §3 gives a low level description of TRODS' design for HTTP services. §4 describes the two persistent stores for TRODS. We address some security concerns in §5 and in §6 we describe and evaluate our implementations of TRODS. §7 describes how TRODS handle non-static objects, §8 discusses related work, and §9 concludes.

2 Protocol

2.1 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 over the Internet. We use an unmodified load balancer to route 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 [14] or *mod n* hashing using the flow's 5-tuple for server affinity. The TRODS kernel module sits in these servers' network stacks, which allows TRODS to manipulate and control the connection's packets flowing in both directions. Figure 1 also includes a key-value store, which one of our designs, TRODS-KV, uses as a persistent store (discussed further in §4).

This figure illustrates one concrete example of a service architecture that uses TRODS. TRODS can work for any general object deliver 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. Furthermore, unlike typical deployments, TRODS can actually tolerate inconsistent state between load-balancers, *e.g.*, in the 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 §4.1.

2.2 The Anatomy of an ODS Connection

To fully understand 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.

2.3 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.

Otherwise, 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 [6]. Periodically, a liveness monitor sends a heartbeat packet to each server and each server responds with their own packet. If the liveness monitor does not hear from a server for longer than a threshold amount of time (25ms in our implementation), the server is declared failed.

Connection Redirection. Connections to the failed server must be rerouted to new servers so that they can be failed over. 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.

The choice of load balancing scheme affects how ongoing connections are remapped to servers. If consistent hashing [14] is used, only connections to the failed server will be reassigned elsewhere. By contract, if a hashing function that is less smooth 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 can still handle this case, 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 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 might often be outstanding packets in the network when a server fails—especially given a large TCP window size with 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 that a 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 bidirectionality of TCP and the BSD socket interface allows a client to receive the server's application-layer response, even when its request has yet to be fully acknowledged at the transport layer.

Determining the Current Phase. TRODS needs to share state between a connection's original and failover 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 a few corner cases of phase determination in §3.

Continuing Negotiations. If the connection is in the setup phase, the new server must continue the negotiation with the client. Negotiation is stateful, which would suggest TRODS needs to save state to the persistent store to continue the negotiation on a new server. 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 about HTTP setup in §3.

Determining the Object. To continue a connection in the download phase, a new server needs to determine both the object and the client's offset into that object. We assume that each service object will have a unique *objectID*, a concise identifier of the object such as a video's filename. We further assume that all objects are immutable, although we discuss TRODS' use with versioned and dynamic objects in §7. Thus, if a new server knows the objectID associated with a connection, it knows exactly what object the client is downloading. TRODS makes this objectID available to the new server by saving it to the persistent store.

Determining the Client's Offset. Once TRODS has determined what object a client is downloading, it still needs to determine how far into the download the failure occurred. 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 it uses 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.

3 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 following assumptions about a typical HTTP 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 GBs in size.
4. Neither HTTP persistent nor pipelined connections are used.
5. HTTP chunked transfer encoding is not used.

The first three assumptions hold true for the majority of HTTP connections, and TRODS takes advantage of them to improve performance. The last two assumptions simplify the basic description of TRODS. We complete our specification by relaxing each assumption in §3.3.

3.1 Normal Operation

Figure 2 shows a HTTP connection at both the application and transport layers, and table 1 briefly summarizes how TRODS interacts with this connection from its position underneath the server's TCP layer.

The connection begins with TCP's three-way handshake. During the handshake when the server sends a response SYN packet, TRODS locally stores knowledge

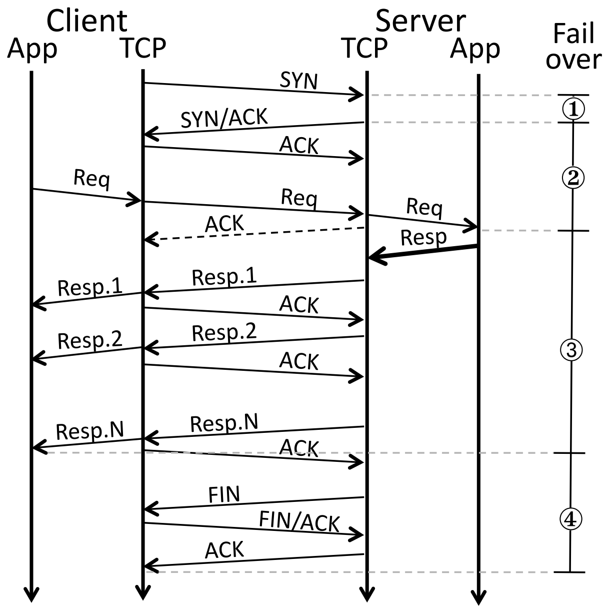

Figure 2: **A typical client-server HTTP connection at both the application and TCP layers. The dashed acknowledgment for the client's request is sent by the server's TCP stack, but dropped by TRODS. The right-most "failover" label indicates a stage of the connection; we detail how TRODS handles failover for each in §3.2.**

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 an HTTP request that fits in a single packet. From this request, TRODS extracts the objectID from the packet, which normally consists of the URI.[1] Under normal processing, the server's transport layer immediately responds to receiving this data request with an ACK, but TRODS drops this packet. If it 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.

[1]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 discuss in §7.

Cli	Srv	TRODS Operation
Syn		
	Syn	Locally store knowledge of this connection
Ack		
Req		Extract and locally save objID
	Ack	Drop
	Resp_1	Extract objISN Block until objID/objISN are durably stored Do not ack client's request
Ack		
	Resp_2	Do not ack client's request
Ack		
…	…	
	Fin	Locally store sequence number of FIN
Fin/ Ack		Delete objID/objISN from persistent storage Delete connection from local storage
	Ack	Ack client's request and Fin

Table 1: **Normal operation of TRODS during a typical HTTP connection.**

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 down in the background or during periods of low-server load; it does not block or delay an ongoing connection.

3.2 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 packet but before sending a response SYN/ACK. If this happens, the client's TCP stack will eventually timeout and retransmit the client's SYN. This SYN will be delivered to a new server and the connection will proceed normally. If a server fails after issuing a SYN/ACK but the network drops this packet, the system's behavior is identical. In later cases, we do not discuss network 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. (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.) The client's 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, close its connection to the server, delete the client's connection from the persistent store, and, if the client 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 §5.1, so that they only occur when it is plausible that their original server has failed.

3.3 Extensions

For brevity, we omit the detailed explanations of how TRODS handles HTTP connections that violate our assumptions described earlier in this section. 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, TRODS uses multiple objectIDs for it. TRODS handles persistent 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.

4 Persistent Storage

The TRODS protocol refers opaquely to a "persistent store" that assists with saving connection state necessary for failover. In this section, we describe two distinct persistent stores that we use in our prototype.

4.1 Key-Value Store

The first persistent store is a key-value storage system (*e.g.*, memcached [10]). 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 the other persistent store. Thus, if the stored information is large—for instance, when multiple response packets need to be stored before being forwarded to the client—TRODS always 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 datacenter 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 also be replicated for additional fault-tolerance. That said, even unreplicated key-value storage provides resilience to a single failure: a connection fails only when both its application and key-value server fail simultaneously. For this reason, many deployments may choose a in-memory key-value store (*e.g.*, memcached [10]) for very low latency and high throughput.

4.2 TCP Timestamps

The second persistent store is the TCP timestamp option [13] 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 must attach 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, Mac OS X, and Windows. In the rare event that a host does not use the TCP timestamp 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 [13]. 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.[2] 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,[3] so they will always be accepted by the client.

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 into the TCP timestamp option, while still adhering to these two constraints.

The 5 Bit 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 both the TCP and HTTP levels, in order 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 about connection hijacking that we address in §5.3.

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 regardless. 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 header by adding linear white space to the last header field, which HTTP clients ignore [9]. Our choice to pad to 64-byte multiples is arbitrary; we could make the response header 128-byte aligned 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.

The 7 Bit 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,

[2]The smallest ephemeral port range we could find was 3975 [26].

[3]That is, $ts_a \geq ts_b$ when $0 \leq (ts_a - ts_b) < 2^{31}$

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 MTU size of 1500 bytes, this means at most 187KB can be in flight from the server at any point. TRODS changes the client's advertised TCP window size to be conservatively smaller than this amount—causing its local TCP stack to buffer data if more than 187KB is to be sent. Because TRODS' limit is in terms of packets and not bytes, however, it also queues packets if more than 128 are outstanding. This behavior seems reasonable for most web servers, but if this limit is too low for a 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.

The 20 Bit ObjectID. The objectID is normally represented by a long string, such as a file path or full URL. This cannot be embedded in a timestamp, so TRODS places known objectIDs in a global array that is replicated on each server. TRODS then identifies a unique objectID by embedding its array index into the timestamp. 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 [5], 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.

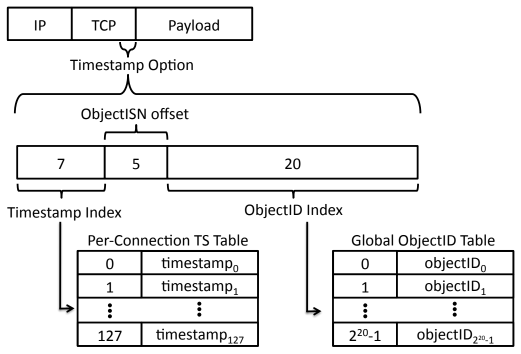

Figure 3: **The relationship between a packet, its TCP timestamp option, the fields TRODS shoehorns into that option, the per-connection timestamp table, and the global objectID table.**

5 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. In this section, we describe how TRODS mitigates each of these concerns.

5.1 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-existing 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 only initiating failover 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 and its 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 failure. Otherwise, if the packet is in the server's known range, it is dropped for being illegitimate, as it should have been in the server's local hashtable. After some quiescence period following a failover, the liveness monitor updates each affected server with new load balancing information.

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-failure.

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 TRODS onto 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 limits the damage an attacker can do 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 more difficulty in degrading the service of normal connections.

5.2 Accessing Unauthorized Content

When TCP timestamps are used for persistent storage, a client can potentially download an object they do not have permissions to access, by sending an ACK packet that will 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 §5.1, 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 protections 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.

5.3 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 ring alarm bells for anyone familiar with TCP sequence number guessing attacks [19]. 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 avoids this attack by generating the ISN from a secure hash of the client IP, port, and a private key that

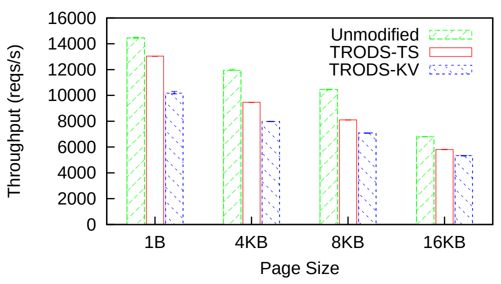

Figure 4: **HTTP throughput experiment using a single server process. We show median throughput of 5 trials of an unmodified server, a server running TRODS-TS, and a server running TRODS-KV is shown. Error bars indicate the minimum and maximum throughputs.**

is known to all the servers in the cluster. This allows TRODS to know the ISN for a connection without using any of the bits in the timestamp, while still making the ISNs appear random. Now an attacker can only learn the ISN of a connection if it is on path and thus already has access to it.

6 Evaluation

In this section we explore the cost of running TRODS. We examine this cost in terms of both decreased throughput and increased latency.

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. The current TRODS implementation handles the normal case, where none of our assumptions from section §3 are violated. We have implemented TRODS using each of the persistent stores from section §4.

Experimental Setup All of our experiments used a single client machine and a single server machine. Each machine has 8 2.3GHz cores and 8 GB of memory and is connected to a 1Gbps switch.

We use memcached 1.4.4 [10] without expiration or eviction as our key-value store and we use lighttpd 1.4.23 [16] as our web server.

6.1 Throughput

We ran two sets of experiments to examine how TRODS affects the maximum throughput of a web server. In our first experiment, shown in figure 4, we turn off all but one of the cores on the server machine and ran the web server as a single process that uses 100% CPU. In this experiment TRODS operations in the kernel steal

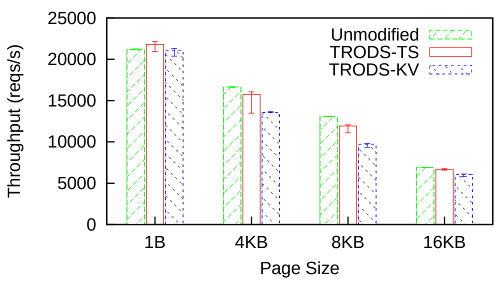

Figure 5: **HTTP throughput experiment using eight server processes. We show median throughput of 5 trials of an unmodified server, a server running TRODS-TS, and a server running TRODS-KV is shown. Error bars indicate the minimum and maximum throughputs.**

cycles that the web server could have otherwise used for serving more requests. In our second experiment, shown in figure 5, all 8 cores on the server were used by 8 server processes. Both figures show the median, minimum, and maximum throughput of 5 trials for each setup. Each trial consisted of 25 client processes continuously fetching a webpage for 40 seconds. We exclude the first and last 5 second of each trial to avoid including artifacts from the systems warming up and cooling down.

In the first experiment TRODS-TS causes a decrease in throughput from 14,455 requests/sec to 13,038 requests/sec, for minimum size 1B web pages. TRODS-KV reduces throughput even more to 10,169 requests/sec. This larger drop is due to TRODS-KV stealing more cycles that TRODS-TS because it needs to communicate with another machine in the datacenter. As the size of the webpages increases, however, the web server becomes less CPU bound and as a result TRODS-TS and TRODS-KV become more competitive with the unmodified service. For example, when pages are 16KB, the unmodified service achieves 6796 requests/sec, TRODS-TS achieves 5815 requests/sec, and TRODS-KV achieves 5335 requests/sec.

In the second experiment the web server is no longer CPU bound and TRODS processing has a smaller effect on throughput. TRODS-TS is within 1,000 requests/sec of the unmodified system for all page sizes and TRODS-KV is within 3,000 requests/sec.

6.2 Latency

Table 2 shows the median and 99th percentile latencies for different phases of 10,000 serial fetches of a 1 byte webpage. The latencies are measured by analyzing tcpdump logs of the client's connections.

Notice that despite the extra processing required when TRODS-TS is used, there is little impact on the latency of a connection. TRODS-KV similarly shows little impact from processing in most phases of a connection. It does, however, have significantly higher latency in the "request to first response packet" phase. During this phase, TRODS-KV blocks the response until the save to the persistent store has completed, which result in an additional 119 μs of latency.

	Normal	**TRODS-TS**	**TRODS-KV**
SYN-SYN	90 (95)	89 (93)	90 (94)
Req-ACK	87 (90)	-	-
Req-Resp.1	137 (150)	135 (149)	256 (282)
Total	353 (372)	362 (384)	490 (520)

Table 2: **Median latency in μs for the major phases of a single HTTP connection through an unmodified service and different implementations of TRODS. Parentheses show the 99th percentile.**

7 Non-Static Objects

Throughout the paper we have assumed that objects are entirely static. We now explore how TRODS works with the two non-static object types.

Versioned Objects. Seemingly versioned objects, such as a user's profile picture on a social networking site, are typically static content. Rather than each version being named identically (*e.g.*, *user_profile.jpg*), they are usually each named uniquely (*e.g.*, *user_profile_contenthash.jpg*). Thus, even if the latest "version" of an object changes during failover, as defined in some database or other mutable state, the new server will continue the delivery of the correct, static object because its objectID will uniquely identify it.

Dynamic Objects. Some objects never exist statically and are always generated dynamically by the service, *e.g.*, web pages produced by PHP and database lookups. TRODS could handle failover for these objects by extracting all the inputs to their generation and feeding them into the recovering server.

If the input is entirely contained in a client's request, TRODS can simply include the relevant fields in the objectID. For instance, if the input is the URI and the client's cookie, TRODS could store both of these values. Then, if TRODS needs to failover the connection, it could use both in the request it sent the new server.

If the input is not entirely contained in the request—*e.g.*, it includes the result of a `gettimeofday` or a `random` call—the service operator needs to make two modifications to use TRODS. First, the service needs to be modified to return all input it used in a special `X-TRODS` HTTP header. TRODS will remove this header from the response and save it to the key-value store along with the objectID and objectISN. Then, if the connection needs to be recovered, TRODS will include this header in the request to the new server.

Second, the service must be modified to use the input data from the optional X-TRODS header. Now, when a new server gets a request for a dynamic object that includes a X-TRODS header, it will use the same input values and deterministically generate the same object as the original server.

8 Related Work

Virtual Machines. Previous researchers have explored failover at the virtual machines level [2, 3, 4, 7, 8]. This general-purpose failover approach should work for all kinds of connections. TRODS represents a different point in the design space than these system; TRODS focuses on high performance for object delivery services, while these systems sacrifice efficiency for generality.

New Transport Layers. Several solutions for failure recovery introduce new transport layer protocols or primitives. Trickles [20] uses a new transport layer protocol and a new sockets API to make one end of a connection stateless. SCTP [23] 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 [21] 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 [22]. M-TCP [24] is a another TCP-like transport protocol designed to support migration. Whereas all of these solutions require changes to 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 [27] 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 [15] that is very similar to FT-TCP's hot backup approach. ST-TCP [18] 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.* [28] describe a similar system that uses a stateful load-balancer to explicitly transmit packets to both the primary and backup. The Backdoors [25] avoids logging by using programmable NICs to extract TCP and application state from the failed 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 durably store it on the backup. TRODS is more efficient that CoRAL because it avoids durably storing the entire reply. Luo *et al.* [17] describes a system for HTTP failover where the load balancer (dispatcher in their terminology) 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.* [11] 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 [12].

9 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 to exert control over unmodified clients. This control allows TRODS to coerce clients into providing storage and initiating failover.

References

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

[2] R. Bradford, E. Kotsovinos, A. Feldmann, and H. Schiöberg. Live wide-area migration of virtual machines including local persistent state. In *Proc. Virtual Execution Environments*, Jun 2007.

[3] T. C. Bressoud. Tft: A software system for application-transparent fault tolerance. In *Proc. Symposium on Fault-Tolerant Computing (FTCS)*, Jun 1998.

[4] T. C. Bressoud and F. B. Schneider. Hypervisor-based fault tolerance. *ACM Trans. Computer Systems*, 1996.

[5] A. Broder, R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopalanr, R. Stata, A. Tomkins, and J. Wiener. Graph

structure in the web. In *Proc. World Wide Web Conference (WWW)*, May 2000.

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

[7] C. Clark, K. F. S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *Proc. Networked Systems Design and Implementation (NSDI)*, May 2005.

[8] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield. Remus: high availability via asynchronous virtual machine replication. In *Proc. Networked Systems Design and Implementation (NSDI)*, Apr 2008.

[9] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. RFC 2616: Hypertext transfer protocol – HTTP/1.1, Jun 1999.

[10] B. Fitzpatrick. Memcached: a distributed memory object caching system. `http://www.danga.com/memcached/`, 2009.

[11] J. Giffin, R. Greenstadt, P. Litwack, and R. Tibbetts. Covert messaging through tcp timestamps. In *Proc. Privacy Enhancing Technologies*, Apr. 2002.

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

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

[14] D. Karger, E. Lehman, F. Leighton, M. Levine, D. Lewin, and R. Panigrahy. 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.

[15] R. R. Koch, S. Hortikar, L. E. Moser, and P. M. Melliar-Smith. Transparent tcp connection failover. *Proc. Dependable Systems and Networks (DSN)*, June 2003.

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

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

[18] M. Marwah, S. Mishra, and C. Fetzer. Tcp server fault tolerance using connection migration to a backup server. In *Proc. Dependable Systems and Networks (DSN)*, Jun 2003.

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

[20] A. Shieh, A. C. Myers, and E. G. Sirer. A stateless approach to connection-oriented protocols. *ACM Trans. Computer Systems*, 26(3), 2008.

[21] A. C. Snoeren and H. Balakrishnan. An end-to-end approach to host mobility. In *Proc. Mobile Computing and Networking Conference(MobiCom)*, Aug. 2000.

[22] A. C. Snoeren, D. G. Andersen, and H. Balakrishnan. Fine-grained failover using connection migration. In *Proc. Symposium on Internet Technologies and Systems (USITS)*, Mar. 2001.

[23] R. Stewart. RFC 4960: Stream control transmission protocol, Sep 2007.

[24] F. Sultan, K. Srinivasan, D. Iyer, and L. Iftode. Migratory tcp: connection migration for service continuity in the internet. In *Proc. Intl. Conf. Distributed Computing Systems (ICDCS)*, July 2002.

[25] F. Sultan, A. Bohra, S. Smaldone, Y. Pan, P. Gallard, I. Neamtiu, and L. Iftode. Recovering internet service sessions from operating system failures. *IEEE Internet Computing*, 9(2), 2005.

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

[27] D. Zagorodnov, K. Marzullo, L. Alvisi, and T. C. Bressoud. Practical and low-overhead masking of failures of tcp-based servers. *ACM Trans. Computer Systems*, 27(2), 2009.

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