

Thialfi: A Client Notification Service for Internet-Scale Applications

Atul Adya

Gregory Cooper

Daniel Myers

Michael Piatek

{adya, ghc, dsmyers, piatek}@google.com

Google, Inc.

ABSTRACT

Ensuring the freshness of client data is a fundamental problem for applications that rely on cloud infrastructure to store data and mediate sharing. Thialfi is a notification service developed at Google to simplify this task. Thialfi supports applications written in multiple programming languages and running on multiple platforms, e.g., browsers, phones, and desktops. Applications register their interest in a set of shared objects and receive notifications when those objects change. Thialfi servers run in multiple Google data centers for availability and replicate their state asynchronously. Thialfi's approach to recovery emphasizes simplicity: all server state is soft, and clients drive recovery and assist in replication. A principal goal of our design is to provide a straightforward API and good semantics despite a variety of failures, including server crashes, communication failures, storage unavailability, and data center failures.

Evaluation of live deployments confirms that Thialfi is scalable, efficient, and robust. In production use, Thialfi has scaled to millions of users and delivers notifications with an average delay of less than one second.

Categories and Subject Descriptors

C.2.4 [Computer-Communications Networks]: Distributed Systems; D.4.5 [Operating Systems]: Reliability

General Terms

Distributed Systems, Scalability, Reliability, Performance

1. INTRODUCTION

Many Internet-scale applications are structured around data shared between multiple users, their devices, and cloud infrastructure. Client applications maintain a local cache of their data that must be kept fresh. For example, if a user changes the time of a meeting on a calendar, that change should be quickly reflected on the devices of all attendees. Such scenarios arise frequently at Google. Although infrastructure services provide reliable storage, there is currently no general-purpose mechanism to

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SOSP '11, October 23-26, 2011, Cascais, Portugal.

Copyright © 2011 ACM 978-1-4503-0977-6/11/10 ... \$10.00.

notify clients that shared data has changed. In practice, many applications periodically poll to detect changes, which results in lengthy delays or significant server load. Other applications develop custom notification systems, but these have proven difficult to generalize and cumbersome to maintain.

This paper presents Thialfi, a highly scalable notification system developed at Google for user-facing applications with hundreds of millions of users and billions of objects. Thialfi provides sub-second notification delivery in the common case and clear semantics despite failures, even of entire data centers. Thialfi supports applications written in a variety of languages (C++, Java, JavaScript) and running on a diverse set of platforms such as web browsers, mobile phones, and desktops. To achieve reliability, Thialfi relies on clients to drive recovery operations, avoiding the need for hard state at the server, and our API is structured so that error handling is incorporated into the normal operation of the application.

Thialfi models shared data as versioned objects, which are stored at a data center and cached at clients. Clients *register* with Thialfi to be notified when an object changes, and the application's servers notify Thialfi when updates occur. Thialfi propagates notifications to registered clients, which synchronize their data with application servers. Crucially, Thialfi delivers only the latest *version number* to clients, not application data, which simplifies our design and promotes scalability.

Thialfi's implementation consists of a library embedded in client applications and two types of servers that run in Google data centers. *Matchers* are partitioned by object and receive and forward notifications; *Registrars* are partitioned by client and manage client registration and presence state. The client library communicates with the servers over a variety of application-specific channels; Thialfi protocols provide end-to-end reliability despite channel losses or message reordering. Finally, a best-effort replication protocol runs between Thialfi data centers, and clients correct out-of-date servers during migration.

A principal feature of Thialfi's design is reliability in the presence of a wide variety of faults. The system ensures that clients eventually learn of the latest version of each registered object, even if the clients were unreachable at the time the update occurred. At large scale, ensuring even eventual delivery is challenging—Thialfi is designed to operate at the scale of hundreds of millions of clients, billions of objects, and hundreds of thousands of changes per second. Since applications are replicated across data centers for reliability, notifications may need to be routed over multiple unreliable communication channels to reach all clients. During propagation, a client may become unavailable or change its server affinity. Clients may be offline. Servers, storage systems, or even entire data centers may become temporarily unavailable. Thialfi handles these issues internally, freeing application developers from the need to cope with them as special cases. Indeed, Thialfi remains correct even when all server state is discarded. In our API, *all failures* manifest as signals that objects or registrations have become stale and should be refreshed, and this process reconstructs state at the server if necessary.

Like many infrastructure services, Thialfi is designed for operational simplicity: the same aspects of our design that provide reliability (e.g., tolerating data center failures) also make the system easier to run in production. Our techniques emphasize simplicity but do not provide perfect availability. While Thialfi remains correct, recovering from some failures results in partial unavailability, and we discuss these scenarios in our design.

Thialfi is a production service that is in active use by millions of people running a diverse set of Google's applications. We focus on two: Chrome and Contacts. These show the diversity of Thialfi usage, which includes desktop applications synchronizing data with the cloud (Chrome) as well as web/mobile applications sharing data between devices (Contacts). In both cases, Thialfi has simplified application design and improved efficiency substantially.

Further evaluation of Thialfi confirms its scalability, efficiency, and robustness. In production use, Thialfi has scaled to millions of users. Load testing shows that Thialfi's resource consumption scales

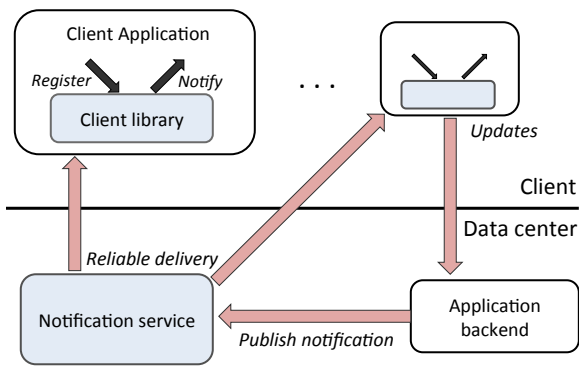


Figure 1: An abstraction for a client notification service.

directly with usage. Injecting failures shows that the cost of recovery is modest; despite the failure of an entire data center, Thialfi can rapidly migrate clients to remaining data centers with limited over-provisioning.

To summarize, we make the following contributions:

- We provide a system robust to the full and partial failures common to infrastructure services. Thialfi is one of the first systems to demonstrate robustness to the complete failure of a data center and to the partial unavailability of infrastructure storage.
- Our design provides reliability at Internet scale without hard server state. Thialfi ensures that clients eventually learn the latest versions of registered objects even if all server state is dropped.
- Thialfi’s API unifies error recovery with ordinary operation. No separate error-handling code paths are required, greatly simplifying integration and reasoning about correctness.
- We integrate Thialfi with several Google applications and demonstrate the performance, scalability, and robustness of our design for millions of users and thousands of notifications per second.

2. MOTIVATION AND REQUIREMENTS

This section describes an abstraction for a notification service with requirements drawn from our experience at Google. Figure 1 shows the abstraction. Since Internet applications are separated into server and client components, the service includes both an infrastructure component and a client library. At the client, developers program against the library’s API and make updates that modify shared data. At the server, applications publish notifications, which the service routes to appropriate clients. The remainder of this section describes how we arrived at this abstraction.

2.1 A Case for a Notification Service

Applications that share data among users and devices have a common need for notifications when data has changed. For example, the Google Contacts application allows users to create, edit, and share contact information through web, mobile, and desktop interfaces that communicate with servers running in Google’s data centers. If a contact changes, other devices should learn of the change quickly. This is the essence of a notification service: informing interested parties of changes to data in a reliable and timely manner.

Throughout the paper, we refer to application data as *objects*: named, versioned entities for which users may receive notifications. For example, a contacts application might model each user’s address book as an object identified by that user’s email address, or the application may model each contact as a separate object. Contacts may be shared among users or a user’s devices. When the contact list is changed, its version number increases, providing a simple mechanism to represent changes.

In the absence of a general service, applications have developed custom notification mechanisms. A widely used approach is for each client to periodically poll the server for changes. While conceptually

Configuration	Choices
Channel	HTTP, XMPP, internal RPC (in DC)
Language	Java, C++, JavaScript
Platform	Web, mobile, native desktop apps
Storage	Storage with inter-DC sync or async replication

Table 1: Configurations supported by Thialfi.

simple and easy to implement, polling creates an unfortunate tension between timeliness and resource consumption. Frequent polling allows clients to learn of changes quickly but imposes significant load on the server. And, most requests simply indicate that no change has occurred.

An alternative is to push notifications to clients. However, ensuring reliability in a push system is difficult: a variety of storage, network, and server failures are common at Internet scale. Further, clients may be disconnected when updates occur and remain offline for days. Buffering messages indefinitely is infeasible. The server’s storage requirements must be bounded, and clients should not be overwhelmed by a flood of messages upon wakeup.

As a result of these challenges, push systems at Google are generally best-effort; developers must detect and recover from errors. This is typically done via a low-frequency, backup polling mechanism, again resulting in occasional, lengthy delays that are difficult to distinguish from bugs.

2.2 Requirements

Summarizing our discussion above, a general notification service should satisfy at least four requirements.

- **Tracking.** The service should track which clients are interested in what data. Particularly for shared data, tracking a mapping between clients and objects is a common need.
- **Reliability.** Notifications should be reliable. To the extent possible, application developers should not be burdened with error detection and recovery mechanisms such as polling.
- **End-to-end.** Given an unreliable channel, the service must provide reliability in an end-to-end manner; i.e., it must include a client-side component.
- **Flexibility.** To be widely applicable, a notification service must impose few restrictions on developers. It should support web, desktop, and mobile applications written in a variety of languages for a variety of platforms. At the server, similar diversity in storage and communication dependencies precludes tight integration with a particular software stack. We show the variety of configurations that Thialfi supports in Table 1.

2.3 Design Alternatives

Before describing our system in detail, we first consider alternative designs for a notification service.

Integrating notifications with the storage layer: Thialfi treats each application’s storage layer as opaque. Updates to shared objects must be explicitly published, and applications must explicitly register for notifications on shared objects. An alternative would be to track object sharing at the storage layer and automatically generate notifications when shared objects change. We avoid this for two reasons. The first is diversity: while many applications share a common need for notifications, applications use storage systems with diverse semantics, data models, and APIs customized to particular application requirements. We view the lack of a one-size-fits-all storage system as fundamental, leading us to design notifications as a separate component that is loosely coupled with the storage layer. The second reason is complexity. Even though automatically tracking object dependencies [22] may simplify the programming model when data dependencies are complex (e.g., constructing web-pages on-the-fly with data joins), such application structures are difficult to scale and rare at Google.

Requiring explicit object registrations and updates substantially simplifies our design, and our experience has been that reasoning about object registrations in our current applications is straightforward.

Reliable messaging from servers to clients: Reliable messaging is a familiar primitive for developers. We argue for a different abstraction: a reliable notification of the latest *version number* of an object. Why not reliable messaging? First, reliable messaging is inappropriate when clients are often unavailable. Lengthy queues accumulate while clients are offline, leading to a flood of messages upon wakeup, and server resources are wasted if offline clients never return. Second, message delivery is often application-specific. Delivering application data requires adhering to diverse security and privacy requirements, and different client devices require delivery in different formats (e.g., JSON for browsers, binary for phones). Instead of reliable messaging, Thialfi provides *reliable signaling*—the queue of notifications for each object is collapsed to a single message, and old clients may be safely garbage-collected without sacrificing reliability. Moreover, such an abstraction allows Thialfi to remain loosely coupled with applications.

3. OVERVIEW

This section gives an overview of the Thialfi architecture and its programming interface.

3.1 Model and Architecture

Thialfi models data in terms of object identifiers and their version numbers. Objects are stored in each application’s backend servers, not by Thialfi. Each object is named using a variable length byte string of the application’s choosing (typically less than 32 bytes), which resides in a private namespace for that application. Version numbers (currently 64-bit) are chosen by applications and included in the update published to Thialfi.

Application backends are required to ensure that version numbers are monotonically increasing to ensure reliable delivery; i.e., in order for Thialfi to reliably notify a client of an object’s latest version, the latest version must be well-defined. Synchronous stores can achieve this by incrementing a version number after every update, for example. Asynchronous stores typically have some method of eventually reconciling updates and reaching a commit point; such stores can issue notifications to Thialfi afterwards. At Google, to avoid modifying existing asynchronous backend stores, some services simply inform Thialfi when updates reach one of the storage replicas, using the current time at that replica as the version number. Although such services run the risk of missing updates due to clock skew and conflicts, this is rare in practice. Clock skew in the data center is typically low, conflicts are infrequent for many applications, and replication delay is low (seconds).

As shown in Figure 1, Thialfi is comprised of a client library and server infrastructure. We describe these components in turn.

Client library: The client library provides applications with a programmatic interface for registering for shared objects and receiving notifications. The library speaks the Thialfi protocol and communicates with the Thialfi infrastructure service running in data centers. An application uses the Thialfi library to register for objects, and the library invokes callbacks to inform the application of registration changes and to deliver notifications. For each notification, Thialfi informs the application of the modified object’s identifier and the latest version known. When the application receives a notification, it synchronizes object data by talking directly with its servers: Thialfi does not provide data synchronization.

Server infrastructure: In the data center, application servers apply updates and notify Thialfi when objects change. We provide a *Publisher* library that application backends can embed. The publisher library call:

Publish(objectId, version, source)

ensures that all Thialfi data centers are notified of the change. When present, the optional *source* parameter identifies the client that made the change. (This ID is provided by the application client

```

// Client actions
interface NotificationClient {
    Start(byte[] persistentState);
    Register(ObjectId objectId, long version);
    Unregister(ObjectId objectId);
}

// Client library callbacks
interface NotificationListener {
    Notify(ObjectId objectId, long version);

    NotifyUnknown(ObjectId objectId);

    RegistrationStatusChanged(ObjectId objectId,
        boolean isRegistered);

    RegistrationFailure(ObjectId objectId,
        boolean isTransient);

    ReissueRegistrations();

    WriteState(byte[] persistentState);
}

```

Figure 2: The Thialfi client API.

at startup and is referred to as its *application ID*.) As an optimization, Thialfi omits delivery of the notification to this client, since the client already knows about the change.

Thialfi supports multiple communication channels to accommodate application diversity. For example, native applications may use XMPP [27], while web applications typically use persistent HTTP connections [17]. This support allows Thialfi to reuse an application’s existing communication channel, an important capability given the high cost of maintaining a channel in certain contexts (e.g., mobile- or browser-based applications). Other than non-corruption, Thialfi imposes few requirements—messages may be dropped, reordered, or duplicated. Although rare, the channels most commonly used by applications exhibit all of these faults.

3.2 Security

Given the diversity of authorization and authentication techniques used by applications, Thialfi does not dictate a particular scheme for securing notifications. Instead, we provide hooks for applications to participate in securing their data at various points in the system. For example, Thialfi can make RPCs to application backends to authorize registrations. If required, Thialfi can also make authorization calls before sending notifications to clients.

Similarly, applications must provide a secure client-server channel if confidentiality is required. Thialfi does not mandate a channel security policy.

3.3 Client API and Usage

The Thialfi client library provides applications with the API shown in Figure 2, and we refer to these calls throughout our discussion.

The **NotificationClient** interface lists the actions available via the client library. The **Start()** method initializes the client, and the **Register()** and **Unregister()** calls can be used to register/unregister for object notifications. We point out that the client interface does not include support for generating notifications. **Publish()** calls must be made by the application backend.

The **NotificationListener** interface defines callbacks invoked by the client library to notify the user application of status changes. Application programmers using Thialfi’s library implement these meth-

ods. When the library receives a notification from the server, it calls `Notify()` with that object's ID and new version number. In scenarios where Thialfi does not know the version number of the object (e.g., if Thialfi has never received any update for the object or has deleted the last known version value for it), the client library uses the `NotifyUnknown()` call to inform the application that it should refetch the object from the application store regardless of its cached version. Internally, such notifications are assigned a sequence number by the server so that they can be reliably delivered and acknowledged in the protocol.

The client library invokes `RegistrationStatusChanged()` to inform the application of any registration information that it receives from the server. It uses `RegistrationFailure()` to indicate a registration operation failure to the application. A boolean, `isTransient`, indicates whether the application should attempt to retry the operation. `ReissueRegistrations()` allows the client library to request all registrations from the application. This call can be used to ensure that Thialfi state matches the application's intent, e.g., after a loss of server state.

The `WriteState()` call is an optional method that provides Thialfi with persistent storage on the client, if available. Client data storage is application-specific; e.g., some applications have direct access to the filesystem while others are limited to a browser cookie. When a client receives its identifier from the server, the client library invokes `WriteState()` with an opaque byte string encoding the identifier, which is then stored by the application and provided to Thialfi during subsequent invocations of `Start()`. This allows clients to resume using existing registrations and notification state. Clients that do not support persistence are treated as new clients after each restart.

4. DESIGN AND IMPLEMENTATION

This section describes the design and implementation of Thialfi. We highlight several key techniques.

No hard server state: Thialfi operates on *registration state* (i.e., which clients care about which objects) and *notification state* (the latest known version of each object). The Thialfi client library is responsible for tracking the registration state and updating servers in the event of a discrepancy, so loss of server-side state does not jeopardize correctness. Moreover, while Thialfi makes a substantial effort to deliver “useful” notifications at specific version numbers, it is free to deliver spurious notifications, and notifications may be associated with an *unknown* version. This flexibility allows notification state to be discarded, provided the occurrence of the drop is noted.

Efficient I/O through multiple views of state: The registration and notification state in Thialfi consists of relations between clients and objects. There is no clear advantage to choosing either client ID or object ID as the primary key for this state: notifications update a single object and multiple clients, while registrations update a single client and multiple objects. To make processing of each operation type simple and efficient, we maintain two separate views of the state, one indexed by client ID and one by object ID, allowing each type of operation to be performed via a single write to one storage location in one view. The remaining view is brought up-to-date asynchronously.

Idempotent operations only: Thialfi is designed so that any server-side operation can be safely repeated. Every operation commits at the server after a single write to storage, allowing aggressive batching of writes. Any dependent changes are performed in the background, asynchronously. Avoiding overwrites fosters robustness; operations are simply retried until they succeed.

Buffering to cope with partial storage availability: While data corruption is uncommon, large-scale storage systems do not have perfect availability. Writes to some storage regions may fail transiently. To prevent this transient storage unavailability from cascading to application backends, Thialfi buffers failed notification writes at available storage locations, migrating buffered state to its appropriate location when possible.

Figure 3 shows the major components of Thialfi. **Bridge servers** are stateless, randomly load-

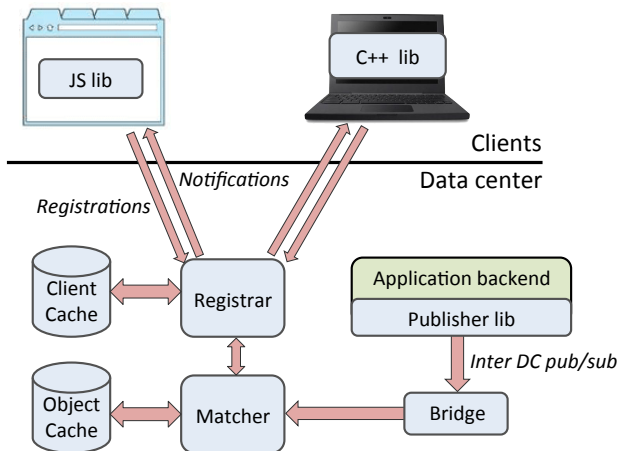


Figure 3: Overall architecture of Thialfi.

balanced tasks that consume a feed of application-specific update messages from Google’s infrastructure pub/sub service, translate them into a standard notification format, and assemble them into batches for delivery to Matcher tasks. **Matchers** consume notifications for objects, match them with the set of registered clients, and forward them to the Registrar for reliable delivery to clients. **Matchers** are partitioned over the set of objects and maintain a view of state indexed by object ID. **Registrars** track clients, process registrations, and reliably deliver notifications using a view of state indexed by client ID.

The remainder of this section describes our design in stages, starting with a simplified version of Thialfi that operates entirely in memory and in one data center only. We use this simplified design to explain the Thialfi protocol and to describe why discarding Thialfi’s server state is safe. We then extend the in-memory design to use persistent storage, reducing the cost of recovering failed servers. Finally, we add replication in order to improve recovery from the failure of entire data centers.

4.1 In-memory Design

An in-memory version of Thialfi stores client and object state in the memory of the Registrar and Matcher servers. As mentioned above, clients are partitioned over Registrar servers, and objects are partitioned over Matcher servers. In order to ensure roughly uniform distribution of load, each client and object is assigned a *partitioning key*. This key is computed by prepending a hash of the client or object ID to the ID itself. We statically partition this keyspace into contiguous ranges; one range is assigned to each server. If a server crashes or reboots, its state is lost and must be reconstructed from scratch.

Aside from lack of persistence and support for multiple data centers, this design is identical to that deployed at Google. We next describe the specific state maintained.

4.1.1 In-memory State

Registrar: For each client, the Registrar servers maintain two sets: 1) *registrations* (objects of interest to the client) and 2) *pending notifications* (notifications not yet acknowledged by the client). They also maintain a monotonically-increasing *sequence number* for each client, used to pick an ordering for registration operations and to generate version numbers for unknown-version notifications.

Matcher: For each object, Matcher servers store the latest version number provided by the application backend. Matcher servers also maintain a copy of the registered clients for each object from the

Registrar; this copy is updated asynchronously. We refer to the combined Matcher and Registrar state as the C/O-Cache (Client and Object cache).

Thialfi components that we call *Propagators* asynchronously propagate state between Matchers and Registrars. The *Registrar Propagator* copies client registrations to the Matcher, and the *Matcher Propagator* copies new notifications to the Registrar.

Both Matchers and Registrars maintain a set of pending operations to perform for objects and clients; i.e., propagation and delivery of (un)registrations and notifications. The state maintained by each server thus decomposes into two distinct parts: the C/O-Cache and a pending operation set.

4.1.2 Client Token Management

Thialfi identifies clients using *client tokens* issued by Registrars. Tokens are composed of two parts: *client identifiers* and *session identifiers*. Tokens are opaque to clients, which store them for inclusion in each subsequent message. A client identifier is unique and persists for the lifetime of the client's state. A session identifier binds a client to a particular Thialfi data center and contains the identity of the data center that issued the token.

A client acquires tokens via a handshake protocol, in which the Registrar creates an entry for the client's state. If the client later migrates to another data center, the Registrar detects that the token was issued elsewhere and informs the client to repeat the handshake protocol with the current data center. When possible, the new token reuses the existing client identifier. A client may thus acquire many session identifiers during its interactions with Thialfi, although it holds only one client token (and thus one session identifier) at any given time.

The Thialfi client library sends periodic heartbeat messages to the Registrar to indicate that it is online (a Registrar only sends notifications to online clients). In the current implementation, the heartbeat interval is 20 minutes, and the Registrar considers a client to be offline if it has not received any message from the client for 80 minutes. Certain channels inform Thialfi in a best-effort manner when a client disconnects, allowing the Registrar to mark the client offline more quickly. Superficially, these periodic heartbeats might resemble polling. However, they are designed to be extremely lightweight: the messages are small, and processing only requires a single in-memory operation in the common case when the client is already online. Thus, unlike application-level polling, they do not pose a significant scalability challenge.

4.1.3 Registration Operation

Once a client has completed the initial handshake, it is able to execute registrations. When an application calls `Register()`, the client library queues a message to send to the Registrar. (As with all protocol messages, the application dispatches outgoing registrations asynchronously using its channel.) An overview of registration is shown in Figure 4.

1. The client library sends a registration message to the Registrar with the object identifier.
2. The Registrar picks an ordering for the registration by assigning it a sequence number, using the sequence number it maintains for the issuing client. The Registrar writes the registration to the client record and adds a new entry to the pending operation set.
3. Subsequently, the Registrar Propagator attempts to forward the registration and the application ID of the registering client to the Matcher responsible for the object via an RPC, and the Matcher updates the copy of the registration in its object cache. The Registrar Propagator repeats this until either propagation succeeds or its process crashes.
4. After propagation succeeds, the Registrar reads the latest version of the object from the Matcher (which reads the versions from its object cache) and writes a pending notification for it into the client cache (i.e., updates its copy of the latest version). We call this process *Registrar post-propagation*. If no version is known, the Registrar generates an unknown-version notification for the object with the version field set using the sequence number maintained for the client.

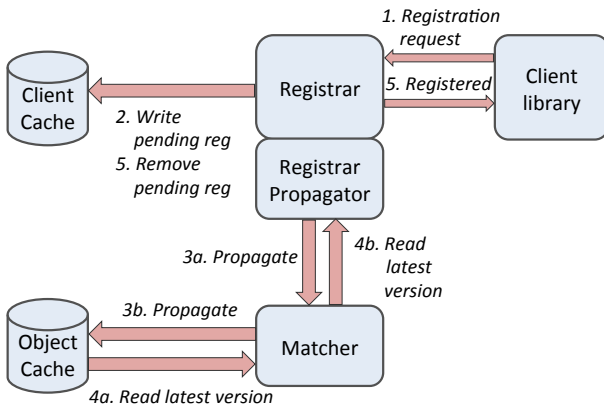


Figure 4: Object registration in Thialfi.

5. The Registrar sends a message to the client confirming the registration and removes the operation from the pending set.

Clients unregister using an analogous process. To keep the registrations at the client and the Registrar in sync, Thialfi uses a **Registration Sync Protocol**. Each message from the client contains a digest of the client’s registered objects, and each message from the server contains the digest of the client’s registrations known to the server (in our current implementation, we compute the digest using HMAC-SHA1 [10]). If the client or the server detects a discrepancy at any point, the client resends its registrations to the server. If the server detects the problem, it requests that the client resend them. To support efficient synchronization for large numbers of objects, we have implemented optional support for Merkle Trees [18], but no application currently using Thialfi has required this mechanism.

The client library keeps track of the application’s intended registrations via registration/unregistration API calls. To preserve the registration state across application restarts, the library could write all registrations to the local disk using the `WriteState()` call (Section 3.3). To simplify persistence requirements, however, Thialfi relies on applications to restate intended registrations on restart. When a client restarts, the client library invokes `ReissueRegistrations()`. The library then recomputes the digest and sends it as part of the regular communication with the server (e.g., in heartbeats). Any discrepancy in the registrations is detected and resolved using the Registration Sync Protocol discussed above. In the normal case when digests match, no registrations are resent to the server.

4.1.4 Notification Operation

As users modify data, client applications send updates to application servers in the data center. Application servers apply the updates and publish notifications to be delivered by Thialfi. Figure 5 shows the sequence of operations by which Thialfi delivers notifications to registered clients.

1. The application server updates its authoritative copy of user data and notifies Thialfi of the new version number. Applications publish notifications using a library that ensures each published notification is received by all data centers running Thialfi. Currently, we use an internal Google infrastructure publish/subscribe service to disseminate messages to data centers. The pub/sub service acknowledges the Publisher library only after a reliable handoff, ensuring eventual delivery. (During periods of subscriber unavailability, the pub/sub service buffers notifications in a persistent log.)
2. Thialfi’s Bridge component consumes the feed of published notifications in each data center and processes them in small batches. The Bridge delivers the update to the Matcher server responsible for the object.
3. The Matcher updates its record for the object with the new version number. Subsequently, using its copy of the registered client list, the Matcher propagator determines which Registrar servers

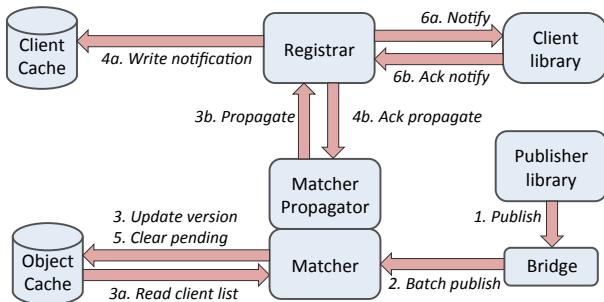


Figure 5: Notification delivery in Thialfi.

Registrar Table						
Row Key	Client State			Object State		Propagation State
	created	last-seqno	presence	reg-{oid}	log-{oid}	pending
<i>hash(user):user:uuid</i>	<i>appid@0</i>	<i>""@seqno</i>	<i>addr@seqno</i>	<i>""@seqno</i>	<i>""@version</i>	<i>""@seqno</i>

Matcher Table			
Row Key	Object State	Client State	Propagation State
	version	reg-{client-id}	pending
<i>hash(object-id):object-id</i>	<i>appid@version</i>	<i>appid@seqno</i>	<i>""@version</i>

Table 2: Bigtable layout for server-side state. *a@b* indicates a value *a* at timestamp *b*. *seqno* refers to the sequence number assigned by the Registrar for that particular client.

have clients registered for the object. It sends RPCs to each Registrar server with (client, oid, version) tuples indicating which clients need to be notified. The client identifiers are used to index the Registrar’s C/O-Cache efficiently.

- Each Registrar receiving a message stores the pending notification for the appropriate clients and responds to the RPC.
- When all Registrars have responded, the operation is removed from the Matcher pending operation set.
- Periodically, the Registrars resend unacknowledged notifications for online clients. Currently, we use a 60-second retransmission interval.

4.1.5 Handling Server Failures

We now discuss how a server reconstructs its in-memory state after a restart (an independent infrastructure system at Google monitors and restarts services that have crashed or become unresponsive). For simplicity, consider a brute-force approach: if any server fails, all servers restart, and the data center identifier is changed to a new value. Subsequent messages from clients with old tokens are detected by the Registrars, triggering a token update as described in §4.1.2. The Registration Sync Protocol then ensures that the clients reissue their registrations.

Client registration messages are sufficient to reconstruct the registration state at the Registrar. The latest-version data at the Matcher is not recovered (and pending notifications are lost) since there is no mechanism to fetch version information from the application backend. Nonetheless, correctness is not compromised. When processing client registrations, the Registrar will send unknown-version notifications for each registered object. This triggers client requests to the application backend to learn the latest version. Such an approach is conservative since the data may not have changed, but Thialfi cannot easily confirm this. After restart, Thialfi resumes normal processing of updates.

4.1.6 Handling Network Failures

There are three types of messages sent between the client and server: client token requests, registration changes, and notifications / acks. Any of these may be lost, reordered, or duplicated. Notifications are acknowledged and hence reliably delivered, and reordering and duplication are explicitly permitted by the semantics of Thialfi. All other messages are retried by the client as needed. Clients detect and ignore duplicate or reordered token grant messages from the Registrar using a nonce, and the Registration Sync Protocol ensures that client and server registration state eventually converge.

4.2 Persistent Storage

At the scale of millions of clients, recovering from failures by flushing and reconstructing state is impractical. *Some* retention of state is required to reduce work during recovery. In this section, we describe how Thialfi currently uses Bigtable [7] to address this issue. The main idea guiding our use of persistent storage is that updates to the C/O-Cache in the memory-only design translate directly into *blind writes* into a Bigtable; i.e., updating state without reading it. Because Bigtable is based on a log-structured storage system, writes are efficient and fast.

4.2.1 Bigtable Layout

Storage locations in a Bigtable (*Bigtable cells*) are named by {row key, column, version} tuples, and Bigtables may be sparse; i.e., there may be many cells with no value. We exploit this property in our storage layout to avoid overwrites. For example, in the Registrar table, for a particular client/object registration pair, we use a distinct row key (based on the client ID), column (based on the object ID), and version (based on the registration sequence number). When querying the registration status for that client/object pair, we simply read the latest version.

Adapting our in-memory representation to Bigtable is straightforward. Registrar and Matcher state is stored in separate Bigtables. The partitioning keys used in the in-memory system become the row keys used in the Bigtables, distributing load uniformly. We continue to statically partition the keyspace over the Registrar and Matcher servers. Each server is thus assigned a contiguous range of Bigtable rows.

The Bigtable schema is summarized in Table 2. Each row of the Matcher table stores the latest known version for an object, the application ID of the client that created that version, and the set of clients registered for that object. Each Registrar row stores the client's application ID, the latest sequence number that was generated for the client by the Registrar, a channel-specific address if the client is online, the object IDs that the client is registered for, and the objects for which the client has an unacknowledged notification. Each table also contains a column for tracking which rows have pending information to propagate to the other table. Note that a cell is written in the last-seqno column whenever a sequence number is used for the client. This ensures that sequence numbers always increase.

4.2.2 In-memory State

In order to improve performance, we cache a small amount of state from Bigtable in Registrar and Matcher server memory. The Registrars cache the registration digest of each online client (but not the full set of registrations). The Matchers and Registrars also cache their pending operation sets. We rely on Bigtable's memory cache for fast reads of the registrations and pending notifications. Since our working set currently fits in Bigtable's memory cache, this has not created a performance problem. (We may revisit this decision if emerging workloads change our Bigtable memory cache profile.)

The outcome of these properties is that *the in-memory state of Thialfi servers corresponds to in-progress operations and limited data for online clients only.*

4.2.3 Pushing Notifications to Clients

As with the in-memory design, reliable notification delivery to clients is achieved by scanning for unacknowledged notifications. Instead of memory, the scan is over the Registrar Bigtable. For efficiency and performance, we also introduce a *fast path*: we unreliably send notifications to online clients during Matcher propagation. While channels are unreliable, message drops are rare, so this fast path typically succeeds. We confirm this in our evaluation (§6).

Realizing that a lengthy periodic scan adversely impacts the tail of the notification latency distribution, we are currently implementing a scheme that buffers undelivered notifications in Registrar memory to more quickly respond to failures.

4.2.4 *Client Garbage Collection*

If a client remains offline for an extended period (e.g., several days), Thialfi garbage-collects its Bigtable state. This involves deleting the client's row in the Registrar Bigtable and deleting any registration cells in the Matcher Bigtable. If the client later comes back online, our use of blind writes means that the client's row may be inadvertently recreated. Although rare, some mechanism is required to detect such an entry, remove it, and notify the client that it must restart with a fresh client ID.

In order to detect client resurrection after garbage collection, Thialfi maintains a *created* cell in the client's Registrar row (Table 2). The Registrar writes this cell when it assigns an ID for a client, and the garbage collector deletes it; no other operations modify this cell. If a garbage collected client comes back online as described above, its *created* cell will be absent from the recreated row. An asynchronous process periodically scans the Registrar Table for rows without created cells. When encountered, the 'zombie' client row is deleted. Also, if the client is online, it is informed that its ID is invalid. Upon receiving this message, the client discards its ID and reconnects as a new client. This message may be lost without compromising correctness; it will be resent by the asynchronous process if the client attempts further operations.

4.2.5 *Recovery from Server Failures*

We now describe how persistent storage reduces the burden of failure recovery. The server caches of Bigtable state and of pending operations are write-through caches, so they may be restored after a restart by simply scanning the Bigtable. Since each server is assigned a contiguous range, this scan is efficient. Additionally, scanning to recover pending operations yields a straightforward strategy for shedding load during periods of memory pressure: a server aborts in-progress propagations, evicts items from its pending operation set, and schedules a future scan to recover.

If required, all Bigtable state can be dropped, with recovery proceeding as in the in-memory design. In practice, this has simplified service administration significantly; e.g., when performing a Bigtable schema change, we simply drop all data, avoiding the complexity of migration.

4.2.6 *Tolerating Storage Unavailability*

A consequence of storing state in Bigtable is that Thialfi's overall availability is limited by that of Bigtable. While complete unavailability is extremely rare, a practical reality of large-scale storage is *partial unavailability*—the temporary failure of I/O operations for some rows, but not all. In our experience, minor Bigtable unavailability occurs several times per day. Our asynchronous approach to data propagation accommodates storage unavailability. I/O failures are skipped and retried, but do not prevent partial progress; e.g., clients corresponding to available regions will continue to receive notifications.

This covers the majority of Thialfi I/O with two exceptions: 1) the initial write when accepting a client operation, e.g., a registration, and 2) the write accepting a new version of an object at the Matcher. In the first case, the client simply retries the operation.

However, accepting new versions is more complex. One possibility is to have the Bridge delay

the acknowledgement of a notification to the publish/subscribe service until the Matcher is able to perform the write. This approach quickly results in a backlog being generated for all notifications destined for the unavailable Matcher rows. Once a large backlog accumulates, the pub/sub service no longer delivers new messages, delaying notifications for *all* clients in the data center. Even in the absence of our particular pub/sub system, requiring application backends to buffer updates due to partial Thialfi storage unavailability would significantly increase their operational complexity.

Given the prevalence of such partial storage unavailability in practice, we have implemented a simple mechanism to prevent a backlog from being generated. To acknowledge a notification, the Bridge needs to record the latest version number *somewhere* in stable storage. It need not be written to the correct location immediately, so long as it is *eventually* propagated there. To provide robustness during these periods, we reissue failed writes to a distinct, scratch Bigtable. A scanner later retries the writes against the Matcher Bigtable. The Everest system [19] uses a similar technique to spread load; in Thialfi, such buffering serves to reduce cascading failures.

Specifically, for a given object, we deterministically compute a sequence of retry locations in a scratch Bigtable. These are generated by computing a salted hash over the object ID, using the retry count as the salt. This computation exploits Thialfi's relaxed semantics to reduce the amount of scratch storage required; successive version updates to the same object overwrite each other in the scratch table when the first scratch write succeeds. Storing failed updates in random locations—a simple alternative—would retain and propagate *all* updates instead of only the latest. While correct, this is inefficient, particularly for hot objects. Our scheme efficiently supports the common case: a series of Matcher writes fails, but the first attempt of each corresponding scratch write succeeds.

4.3 Supporting Multiple Data Centers

To meet availability requirements at Google, Thialfi must be replicated in multiple data centers. In this section, we describe the extensions required to support replication, completing the description of Thialfi's design. Our goal is to ensure that a site failure does not degrade reliability; i.e., notifications may be delayed, but not dropped. Clients migrate when a failure or load balancing event causes protocol messages to be routed from the Thialfi data center identified in the client's session token to a Thialfi instance in another data center.

We require that the application's channel provide client affinity; i.e., Thialfi messages from a given client should be routed to the same data center over short time scales (minutes). Over longer time scales, clients may migrate among data centers depending on application policies and service availability. Also, when a Thialfi data center fails, we require the application channel to re-route messages from clients to other data centers. These characteristics are typical for commonly used channels.

Even without replication of registration state, Thialfi can automatically migrate clients among data centers. When a client connects to a new data center, the Registrar instructs it to repeat the token-assignment handshake, by which it obtains a new token (§4.1.2). Since the new data center has no information about the client's registrations, the client and server registration digests will not match, triggering the Registration Sync Protocol. The client then reissues all of its registrations. While correct, this is expensive; a data center failure causes a flood of re-registrations. Thus, replication is designed as an optimization to decrease such migration load.

4.3.1 State Replication

Thialfi uses two forms of state replication: 1) reliable replication of notifications to all data centers and 2) best-effort replication of registration state. The pub/sub service acknowledges the Publisher library after a reliable handoff and ensures that each notification is reliably delivered to all Thialfi data centers; the Thialfi Matchers in each data center acknowledge the notification only after it has been written to stable storage.

When replicating registration state, we use a custom, asynchronous protocol that replicates only the

state we must reconstruct during migration. Specifically, we replicate three Registrar operations between Thialfi data centers: 1) client ID assignment, 2) registrations, and 3) notification acknowledgements. Whenever a Registrar processes one of these operations, it sends best-effort RPC messages to the Registrars in other data centers. At each data center, *replication agents* in the Registrar consume these messages and replay the operations. (While we have implemented and evaluated this scheme, we have not yet deployed it in production.)

We initially attempted to avoid designing our own replication scheme. A previous design of Thialfi used a synchronous, globally consistent storage layer called Megastore [2]. Megastore provides transactional storage with consistency guarantees spanning data centers. Building on such a system is appealingly straightforward: simply commit a transaction that updates relevant rows in all data centers before acknowledging an operation. Unfortunately, micro-benchmarks show that Megastore requires roughly 10 times more operations per write to its underlying Bigtables than a customized approach. For a write-intensive service like Thialfi, this overhead is prohibitive.

Although the Thialfi replication protocol is designed to make migration efficient, an outage still causes a spike in load. During a planned outage, we use an *anti-storm* technique to spread load. During a migration storm, Thialfi silently drops messages from a progressively-decreasing fraction of migrated clients at the surviving data centers, trading short-term unavailability for reduced load.

5. ACHIEVING RELIABLE DELIVERY

In this section, we describe Thialfi’s notion of reliability and argue that our mechanisms provide it. We define reliable delivery as follows:

Reliable delivery property: If a well-behaved client registers for an object X , Thialfi ensures that the client will always eventually learn of the latest version of X .

A well-behaved client is one that faithfully implements Thialfi’s API and remains connected long enough to complete required operations, e.g., registration synchronization. In our discussion, we make further assumptions regarding integrity and liveness of dependent systems. First, we assume that despite transitory unavailability, Bigtable tablets will eventually be accessible and will not corrupt stored data. Second, we assume that the communication channel will not corrupt messages and will eventually deliver them given sufficient retransmissions.

As is typical for many distributed systems, Thialfi’s reliability goal is *one-sided*. By this we mean that, while clients will learn the latest version of registered objects, notifications may be duplicated or reordered, and intermediate versions may be suppressed.

Thialfi achieves end-to-end reliability by ensuring that state changes in one component eventually propagate to all other relevant components of the system. We enumerate these components and their interactions below and discuss why state transfer between them eventually succeeds. We have not developed a formal model of Thialfi nor complete proofs of its safety or liveness; these are left as future work.

Registration state is determined by the client, from which it propagates to the Registrar and Matcher (subject to access control policies). The following mechanisms ensure the eventual synchronization of registration state across the three components:

- **Client ↔ Registrar:** Every message from the client includes a digest that summarizes all client registration state (§4.1.3). If the client-provided digest disagrees with the state at the Registrar, the synchronization protocol runs, after which client and server agree. Periodic heartbeat messages include the registration digest, ensuring that any disagreement will be detected.

- **Registrar** → **Matcher**: When the Registrar commits a registration state change to Bigtable, a pending work marker is also set atomically. This marker is cleared only after all dependent writes to the Matcher Bigtable have completed successfully. All writes are retried by the Registrar Propagator if any failure occurs. (Because all writes are idempotent, this repetition is safe.)

Notification state comes from the Publisher, which provides a reliable feed of object-version pairs via the pub/sub service. These flow reliably through the Bridge, Matcher, and Registrar to the client using the following mechanisms:

- **Bridge** → **Matcher**: Notifications are removed from the update feed by the Bridge only after they have been successfully written to either their appropriate location in the Matcher Bigtable or buffered in the Matcher scratch Bigtable. A periodic task in the Bridge reads the scratch table and resends the notifications to the Matcher, removing entries from the scratch table only after a successful Matcher write.
- **Matcher** → **Registrar**: When a notification is written to the Matcher Bigtable, a pending work marker is used to ensure eventual propagation. This mechanism is similar to that used for Registrar → Matcher propagation of registration state.

Notification state also flows from the Matcher to the Registrar in response to registration state changes. After a client registers for an object, Registrar post-propagation will write a notification at the latest version into the client's Registrar row (§4.1.3). This ensures that the client learns of the latest version even if the notification originally arrived before the client's registration.

- **Registrar** → **Client**: The Registrar retains a notification for a client until either the client acknowledges it or a subsequent notification supersedes it. The Registrar periodically retransmits any outstanding notifications while the client is online, ensuring eventual delivery.

Taken together, local state propagation among components provides end-to-end reliability. Specifically:

- A client's registration eventually propagates to the Matcher, ensuring that the latest notification received for the registered object after the propagation will be sent to the client.
- Registrar post-propagation ensures that a client learns the version of the object known to Thialfi when its registration reached the Matcher. If no version was present at the Matcher, the client receives a notification at *unknown version*.

The preceding discussion refers to system operation within a single data center. In the case of multiple data centers, our Publisher Library considers notification publication complete only after the notification has been accepted by the Matcher or buffered in the persistent storage of Google's infrastructure publish/subscribe service in all data centers. Thus, each application's notifications are reliably replicated to all data centers. This is in contrast to Thialfi's registration state, which is replicated on a best-effort basis. However, so long as a client is not interacting with a given data center, there is no harm in the registration state being out-of-sync there. When the client migrates to a new data center, the Registration Sync Protocol (§4.1.3) ensures that the new Registrar obtains the client's current registration state. The propagation and post-propagation mechanisms described above also apply in the new data center, ensuring that the new Registrar will reliably inform the client of the latest version of each registered object. Taken together, these mechanisms provide reliable delivery when operating with multiple data centers.

6. EVALUATION

Thialfi is a production service that has been in active use at Google since the summer of 2010. We report performance from this deployment. Additionally, we evaluate Thialfi's scalability and

fault tolerance for synthetic workloads at the scale of millions of users and thousands of updates per second. Specifically, we show:

- *Ease of adoption:* Applications can adopt Thialfi with minimal design and/or code changes. We describe a representative case study, the Chrome browser, for which a custom notification service was replaced with Thialfi. (§6.1)
- *Scalability:* In production use, Thialfi has scaled to millions of users. Load testing shows that resource consumption scales linearly with active users and notification rate while maintaining stable notification latencies. (§6.2)
- *Performance:* Measurements of our production deployment show that Thialfi delivers 88% of notifications in less than one second. (§6.3)
- *Fault-tolerance:* Thialfi is robust to the failure of an entire data center. In a synthetic fail-over experiment, we rapidly migrate over 100,000 clients successfully and quantify the over-provisioning required at remaining instances in order to absorb clients during fail-over. We also provide measurements of transient unavailability in production that demonstrate the practical necessity of coping with numerous short-term faults. (§6.4)

6.1 Chrome Sync Deployment

Chrome supports synchronizing client bookmarks, settings, extensions, and so on among all of a user's installations. Initially, this feature was implemented by piggy-backing on a previously-deployed chat service. Each online client registered its presence with the chat service and would broadcast a chat metadata message notifying online replicas that a change had committed to the back-end storage infrastructure. Offline clients synchronized data on startup. While appealingly simple, this approach has three drawbacks:

- *Costly startup synchronization:* The combined load of synchronizing clients on startup is significant at large scale. Ideally, synchronization of offline clients would occur only after a change in application data, but no general-purpose signaling mechanism was available.
- *Unreliable chat delivery:* Although generally reliable, chat message delivery is best-effort. Even when a client is online, delivery is not guaranteed, and delivery failures may be silent. In some cases, this resulted in a delay in synchronization until the next browser restart.
- *Lack of fate-sharing between updates and notifications:* Since clients issue both updates and change notifications, the update may succeed while the notification fails, leading to stale replicas. Ensuring eventual broadcast of the notification with timeout and retry at the client is challenging; e.g., a user may simply quit the program before it completes.

While these issues might have been addressed with specific fixes, the complexity of maintaining a reliable push-based architecture is substantial. Instead, Chrome adopted a hybrid approach: best-effort push with periodic polling for reliability. Unfortunately, the back-end load arising from frequent polling was substantial. To control resource consumption, clients polled only once every few hours. This again gave rise to lengthy, puzzling delays for a small minority of users and increased complexity from maintaining separate code paths for polling and push updates.

These issues drove Chrome's adoption of Thialfi, which addresses the obstacles above. Thialfi clients are persistent; offline clients receive notifications on startup only if a registered object has changed or the client has been garbage collected. This eliminates the need for synchronization during every startup. Thialfi provides end-to-end reliability over the best-effort communication channel used by Chrome, thereby easing the porting process. Finally, Thialfi servers receive notifications directly from Chrome's storage service rather than from clients, ensuring that notification delivery is fate-shared with updates to persistent storage.

Migrating from custom notifications to Thialfi required modest code additions and replaced both the previous push and polling notification support. Chrome includes Thialfi's C++ client library,

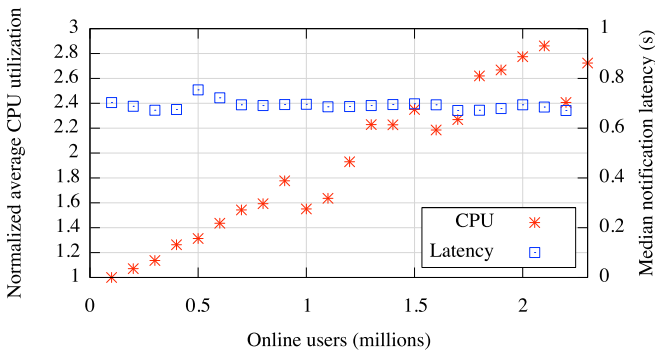


Figure 6: Resource consumption and notification latency as active users increase.

implements our API (Figure 2), and routes Thialfi notifications to appropriate Chrome components. In full, Chrome’s Thialfi-specific code is 1,753 lines of commented C++ code (535 semicolons).

6.2 Scalability

We evaluate Thialfi’s scalability in terms of resource consumption and performance. We show that resource consumption increases proportionally with increases in load. With respect to performance, we show that notification latencies are stable as load increases, provided sufficient resources. These measurements confirm our practical experience. To support increasing usage of Thialfi, we need only allocate an incremental amount of additional infrastructure resources. The two main contributors to Thialfi’s load are 1) the number of active users and 2) the rate at which notifications are published. We consider each in turn, measuring synthetic workloads on shared Google clusters. While our experiments are not performance-isolated, the results presented are consistent over multiple trials.

Increasing active users: Increasing the number of active users exercises registration, heartbeat processing, and client / session assignment. To measure this, we recorded the resource consumption of Thialfi in a single data center while adding 2.3 million synthetic users. Each user had one client (the number of clients per user does not impact performance in Thialfi). Clients arrived at a constant rate of 570 per second. Each registered for five distinct objects and issued a random notification every 8 minutes and a heartbeat message every 20 minutes. The version of each notification was set to the current time, allowing registered clients to measure the end-to-end latency upon receipt.

Figure 6 shows the results. As a proxy for overall resource consumption, we show the increasing CPU consumption as users arrive. Demand for other resources (network traffic, RPCs, memory) grows similarly. The CPU data is normalized by the amount required to support a baseline of 100,000 users. Overall, increasing active users 23-fold (from 100,000 to 2.3 million) requires $\sim 3\times$ the resources. Throughout this increase, median notification delays are stable, ranging between 0.6–0.7 seconds. (Because these synthetic clients are local to the data center, delays do not include wide-area messaging latency.)

Increasing notification rate: Increasing the notification rate stresses Matcher to Registrar propagation. In this case, we measure resource consumption while varying the notification rate for a fixed set of 1.4 million synthetic clients that have completed registrations and session assignment; all clients were online simultaneously for the duration of the experiment. As in the previous measurements, each client registered for five objects and each user had one client.

Figure 7 shows the results of scaling the notification rate. We report CPU consumption normalized by the amount required to support a baseline notification rate of 1,000 per second and increase the

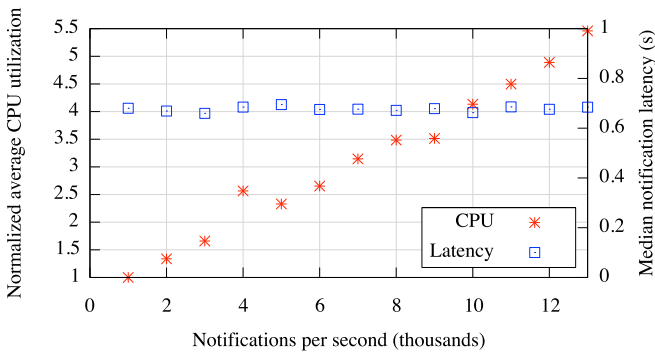


Figure 7: Resource consumption and notification latency as the notification rate increases.

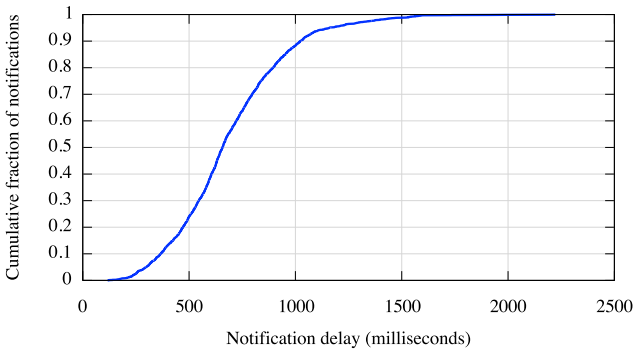


Figure 8: Cumulative distribution of notification latencies randomly sampled from our live deployment.

rate by 1,000 up to 13,000. As before, median notification delays remain stable with proportional resource consumption.

6.3 Performance

The previous measurements quantify median performance for synthetic workloads. We next examine the distribution of notification latencies observed in our production deployment. Each Thialfi component tracks internal propagation delays by appending a log of timestamps to each notification as it flows through the system.

Figure 8 shows a CDF of 2,514 notifications sampled over a 50-minute period from an active Thialfi cell. 88% of notifications are dispatched in less than one second. However, as is typical in asynchronous distributed systems operating on shared infrastructure, a minority of messages may be delayed for much longer, exceeding two seconds in our measurements.

We point out that these delays do not include delivery and acknowledgements from clients themselves; we measure only the delay within Thialfi from the receipt of a notification to the first attempt to send it to an online client. End-to-end delays vary significantly due to the variable quality of channels and the lengthy delays incurred by offline clients. In practice, network propagation adds between 30–100 ms to overall notification latency.

In practice, the majority of Thialfi’s delay is self-imposed. Our current implementation aggressively batches Bigtable operations and RPC dispatch to increase efficiency. This is illustrated in Figure 9, which shows the delay for each stage of notification delivery averaged over a 10-minute interval. This

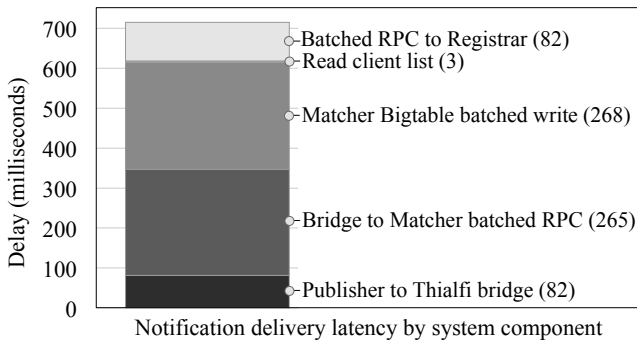


Figure 9: The average contribution to overall notification delay of each Thialfi system component.

data is drawn from our production deployment. The Publisher library appends an initial timestamp when the notification is generated by the application, and its propagation delay to Thialfi’s bridge is fundamental. Once received, the RPC sending a notification from the bridge to the Matcher is batched with a maximum delay of 500 ms. Matcher Bigtable writes are similarly batched. During propagation, the Matcher reads the active client list—this data is typically retrieved directly from Bigtable’s in-memory cache. Finally, the propagation RPC to the Registrar has a batch delay of 200 ms.

The majority of our current applications use Thialfi as a replacement for lengthy polling, and the sub-second delays associated with batching are acceptable. But, as Figure 9 shows, we can further reduce Thialfi’s delay by simply reducing the batching delay of relevant components. This increases resource demands but does not introduce any fundamental scalability bottlenecks.

6.4 Fault Tolerance

We evaluate fault tolerance in two ways. First, we examine fail-over of clients between data centers. This exercises our synchronization protocol and quantifies the over-provisioning required to cope with data center failure in practice. Second, we present a month-long trace of how often Thialfi buffers incoming notifications to cope with small periods of partial Matcher unavailability. This shows the practical necessity for our techniques.

Data center fail-over: The failure of a data center requires that clients be migrated to a new instance and their state synchronized with new servers. Migration can be expensive at the server; it requires reading the set of registered objects, computing the digest, sending pending notifications, and processing registration requests (if any). Applications with few updates and/or lengthy heartbeat intervals naturally spread migration load over a lengthy interval. Here, we consider a more challenging case: rapidly migrating tens of thousands of clients with very frequent heartbeats to ensure rapid fail-over.

We instantiated 380,000 clients spread uniformly across three distinct Thialfi data centers with a heartbeat interval of 30 seconds. Each client registered for five objects and generated random notifications yielding an incoming notification rate of roughly 11,000/sec across all clients. After allowing the system to stabilize, we halted the Thialfi instance of one data center while measuring the CPU consumption of the remaining two as well as the overall client notification rate. The failed data center was not restored for the duration of the experiment. Note that this experiment was performed using a prior version of the Registration Sync Protocol; rather than including the registration digest in each message, clients request the full registration list during migration. This modification has not significantly changed resource consumption in practice.

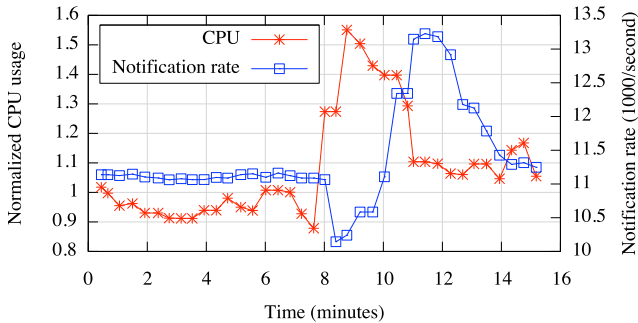


Figure 10: CPU usage and notification rate during the sudden failure of a Thialfi data center.

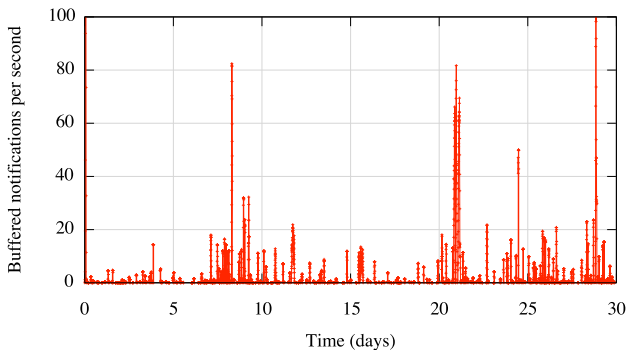


Figure 11: A month-long trace of notification buffering during Matcher unavailability or Matcher storage unavailability.

Figure 10 shows the results. We normalize CPU usage by the first observation taken in steady state. After several minutes, we fail one data center, which clients detect after three failed heartbeats. This is reflected by increased CPU consumption at the remaining instances and a sudden drop in notification receive rate corresponding to clients in the failed data center. As clients migrate, accumulated notifications are discharged as clients are brought up-to-date. Shortly after, the system stabilizes. To migrate 33% of clients over several minutes, Thialfi requires over-provisioning by a factor of ~ 1.6 .

Matcher unavailability: Thialfi’s provisions for fault tolerance arise from practical experience. For example, our implementation buffers notifications to a temporary Bigtable to cope with transient unavailability (§4.2.6). This mechanism was added after our initial deployment in response to frequent manual intervention to respond to failures. Figure 11 shows a month-long trace of notification buffering, confirming the need for error handling in practice. After deploying this solution, the number of alerts that occurred due to a backlog disappeared completely. We point out that buffering occurs not only during storage unavailability but *any* unavailability of the Matcher, e.g., during software upgrades or restarts. Support for automatically buffering notifications without manual action during these periods has greatly simplified service administration.

7. RELATED WORK

The problem of scalable event notification has received significant attention in the distributed systems community, which we draw on in our design. Thialfi differs from existing work in two principal ways. The first is the constraints of our environment. Thialfi’s design stems from the unique requirements of Internet applications, infrastructure services, and the failures they exhibit. The second difference is our goal. Our API and semantics provide developers with reliability that simplifies development, but Thialfi does not impose significant restrictions on an application’s runtime environment or software stack.

Thialfi builds on existing infrastructure services widely used at Google. We use Bigtable [7] to store object and client data. The Chubby lock service [4] provides reliable, consistent naming and configuration of our server processes. While specific to Google, the functionality of these systems is being increasingly replicated by open source alternatives for which Thialfi’s design could be adapted. For example, HBase [12] provides Bigtable-like structured storage atop the HDFS block store [13], and Zookeeper [15] provides a highly reliable group coordination service.

Thialfi’s provisions for fault-tolerance draw on emerging practical experience with infrastructure services [3, 9, 11, 21]. Our experience with performance variability and communications failures is consistent with these observations. But, unlike many existing infrastructure services, Thialfi is explicitly designed to cope with the failure of entire data centers. Megastore [2] shares this goal, using synchronous replication with Paxos [16] to provide consistent structured data storage. While early designs of Thialfi were built atop Megastore to inherit its robustness to data center failure, we eventually adopted replication and fault-tolerance techniques specific to a notification service; these increase efficiency substantially.

Our goal of providing a scalable notification service is shared by a number of P2P notification and publish / subscribe systems, e.g., Bayeux [29], Scribe [23], and Siena [6]. These systems construct multicast trees on overlay routing substrates in order to efficiently disseminate messages. While Thialfi addresses a similar problem, differences between P2P and infrastructure environments necessitate radical differences in our design. For example, P2P message delivery requires direct browser-to-browser communication that is precluded by fundamental security policies [24]. Also, message delivery is best-effort, departing from our goal of maintaining reliable delivery of notifications. Significant additional work exists on publish / subscribe systems (e.g. [1, 20, 25, 26]), but these systems provide richer semantics and target lower scale.

For web applications, Thialfi addresses a longstanding limitation of HTTP—the need for polling to refresh data. Others have observed these problems; e.g., Cao and Liu [5] advocate the use of invalidations as an alternative to polling to maintain the freshness of web documents, but their proposed protocol extensions were not taken up. Yin et al. [28] study the efficiency of HTTP polling and propose an invalidation protocol that is conceptually similar to Thialfi, although it operates on a single HTTP server only. We reexamine these problems at much larger scale. Cowling et al. [8] mention the applicability of Census, a Byzantine-fault-tolerant group membership system, to the problem of large-scale cache invalidation, but they leave the design to future work.

More recently, practitioners have developed a number of techniques to work around the request / reply limitations of HTTP [17]. Many approaches rely on a common technique: each client maintains an in-flight request to the server, which replies to this outstanding request only when new data is available. More recently, web sockets [14] have been proposed as a standard enabling full-duplex HTTP messaging. Thialfi supports these channels transparently, separating the implementation details of achieving push messages from the semantics of the notification service.

8. LESSONS LEARNED

In the process of designing, implementing, and supporting Thialfi we learned several lessons about our design.

For many applications, the signal is enough. Our choice to provide applications with only a notification signal was contentious. In particular, developers have almost universally asked for richer features than Thialfi provides: e.g., support for data delivery, message ordering, and duplicate suppression. Absent these more compelling features, some developers are hesitant to adopt Thialfi. We have avoided these features, however, as they would significantly complicate both our implementation and API. Moreover, we have encountered few applications with a fundamental need for them. For example, applications that would prefer to receive data directly from Thialfi typically store the data in their servers and retrieve it after receiving a notification. While developers often express consternation over the additional latency induced by the retrieval, for many applications this does not adversely affect the user experience. In our view, reliable signaling strikes a balance between complexity and system utility.

Client library rather than client protocol. Perhaps more than any other component in the system, Thialfi's client library has undergone significant evolution since our initial design. Initially, we had no client library whatsoever, opting instead to expose our protocol directly. Engineers, however, strongly prefer to develop against native-language APIs. And, a high-level API has allowed us to evolve our client-server protocol without modifying application code.

Initially, the client library provided only a thin shim around RPCs, e.g., register, unregister, acknowledge. This API proved essentially unusable. While seemingly simple, this initial design exposed too many failure cases to application programmers, e.g., server crashes and data center migration. This experience led us to our goal of unifying error handling with normal operations in Thialfi's API.

Complexity at the server, not the client. The presence of a client library creates a temptation to improve server scalability by offloading functionality. Our second client library took exactly this approach. For example, it detected data center switchover and drove the recovery protocol, substantially simplifying the server implementation. In many systems, this design would be preferable: server scalability is typically the bottleneck, and client resources are plentiful. But, a sophisticated client library is difficult to maintain. Thialfi's client library is implemented in multiple languages, and clients may not upgrade their software for years, if ever. In contrast, bug and performance fixes to data center code can be deployed in hours. Given these realities, we trade server resources for client simplicity in our current (third) client library.

Asynchronous events, not callbacks. Developers are accustomed to taking actions that produce results, and our initial client libraries tried to satisfy this expectation. For example, the register call took a registration callback for success or failure. Experience showed callbacks are not sufficient; e.g., a client may become spontaneously unregistered during migration. Given the need to respond to asynchronous events, callbacks are unnecessary and often misleading. Clients only need to know current state, not the sequence of operations leading to it.

Initial workloads have few objects per client. A key feature of Thialfi is its support for tens of thousands of objects per client. At present, however, no client application has more than tens of objects per client. We suspect this is because existing client applications were initially designed around polling solutions that work best with few objects per client. Emerging applications make use of fine-grained objects, and we anticipate workloads with high fanout and many objects per client.

9. SUMMARY

We have presented Thialfi, an infrastructure service that provides web, desktop, and mobile client applications with timely (sub-second) notifications of updates to shared state. To make Thialfi generally applicable, we provide a simple object model and client API that permit developers flexibility in communication, storage, and runtime environments. Internally, Thialfi uses a combination of server-side soft state, asynchronous replication, and client-driven recovery to tolerate a wide range of failures common to infrastructure services, including the failure of entire data centers. The Thialfi API is structured so that these failures are handled by the same application code paths used for normal op-

eration. Thialfi is in production use by millions of people daily, and our measurements confirm its scalability, performance, and robustness.

Acknowledgements

We would like to thank the anonymous reviewers and our shepherd, Robert Morris, for their valuable feedback. We are also grateful to many colleagues at Google. John Pongsajapan and John Reumann offered valuable wisdom during design discussions, Shao Liu and Kyle Marvin worked on the implementation, and Fred Akalin and Rhett Robinson helped with application integration. Brian Bershad and Dan Grove have provided support and resources over the life of the project, and Steve Lacey provided encouragement and connections with application developers. Finally, we thank James Cowling, Xiaolan Zhang, and Elisavet Kozyri for helpful comments on the paper.

10. REFERENCES

- [1] Y. Amir and J. Stanton. The Spread Wide Area Group Communication System. Technical Report CNDS 98-4, 1998.
- [2] J. Baker, C. Bond, J. C. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Léon, Y. Li, A. Lloyd, and V. Yushprak. Megastore: Providing scalable, highly available storage for interactive service. In *Proc. of CIDR*, 2011.
- [3] T. Benson, A. Akella, and D. A. Maltz. Network traffic characteristics of data centers in the wild. In *Proc. of IMC*, 2010.
- [4] M. Burrows. The Chubby lock service for loosely-coupled distributed systems. In *Proc. of OSDI*, 2006.
- [5] P. Cao and C. Liu. Maintaining strong cache consistency in the world wide web. *IEEE Trans. Comput.*, 47:445–457, April 1998.
- [6] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Trans. Comput. Syst.*, 19:332–383, August 2001.
- [7] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. In *Proc. of OSDI*, 2006.
- [8] J. Cowling, D. R. K. Ports, B. Liskov, R. A. Popa, and A. Gaikwad. Census: Location-aware membership management for large-scale distributed systems. In *Proc. of USENIX*, 2009.
- [9] J. Dean. Designs, lessons and advice from building large distributed systems. In *LADIS Keynote*, 2009.
- [10] D. E. Eastlake and P. E. Jones. US secure hash algorithm 1 (SHA1). Internet RFC 3174, 2001.
- [11] D. Ford, F. Labelle, F. I. Popovici, M. Stokely, V.-A. Truong, L. Barroso, C. Grimes, and S. Quinlan. Availability in globally distributed storage systems. In *Proc. of OSDI*, 2010.
- [12] HBase. <http://hbase.apache.org/>.
- [13] Hadoop Distributed File System. <http://hadoop.apache.org/hdfs/>.
- [14] I. Hickson. The WebSocket API. <http://dev.w3.org/html5/websockets/>.
- [15] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: Wait-free coordination for Internet-scale systems. In *Proc. of USENIX*, 2010.
- [16] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16:133–169, May 1998.
- [17] P. McCarthy and D. Crane. *Comet and Reverse Ajax: The Next-Generation Ajax 2.0*. Apress, 2008.
- [18] R. Merkle. *Secrecy, authentication and public key systems*. PhD thesis, Dept. of Electrical Engineering, Stanford University, 1979.
- [19] D. Narayanan, A. Donnelly, E. Thereska, S. Elnikety, and A. Rowstron. Everest: Scaling down peak loads through i/o off-loading. In *Proc. of OSDI*, 2008.
- [20] P. R. Pietzuch and J. Bacon. Hermes: A distributed event-based middleware architecture. In *Proc. ICDCS, ICDCSW '02*, pages 611–618, Washington, DC, USA, 2002. IEEE Computer Society.

- [21] E. Pinheiro, W.-D. Weber, and L. A. Barroso. Failure trends in a large disk drive population. In *Proc. of FAST*, 2007.
- [22] D. R. K. Ports, A. T. Clements, I. Zhang, S. Madden, and B. Liskov. Transactional consistency and automatic management in an application data cache. In *Proc. of OSDI*, 2010.
- [23] A. I. T. Rowstron, A.-M. Kermarrec, M. Castro, and P. Druschel. SCRIBE: The design of a large-scale event notification infrastructure. In *Networked Group Communication*, pages 30–43, 2001.
- [24] J. Ruderman. Same origin policy for JavaScript. https://developer.mozilla.org/En/Same_origin_policy_for_JavaScript.
- [25] R. Strom, G. Banavar, T. Chandra, M. Kaplan, K. Miller, B. Mukherjee, D. Sturman, and M. Ward. Gryphon: An information flow based approach to message brokering. In *Proc. Intl. Symposium on Software Reliability Engineering*, 1998.
- [26] R. van Renesse, K. P. Birman, and S. Maffei. Horus: a flexible group communication system. *Commun. ACM*, 39:76–83, April 1996.
- [27] Extensible Messaging and Presence Protocol. <http://xmpp.org/xmpp-protocols>.
- [28] J. Yin, L. Alvisi, M. Dahlin, and A. Iyengar. Engineering server-driven consistency for large scale dynamic web services. In *Proc. of WWW*, 2001.
- [29] S. Q. Zhuang, B. Y. Zhao, A. D. Joseph, R. H. Katz, and J. D. Kubiatowicz. Bayeux: An architecture for scalable and fault-tolerant wide-area data dissemination. In *Proc. of NOSSDAV*, 2001.