UiT THE ARCTIC UNIVERSITY OF NORWAY

Faculty of Science and Technology
Department of Computer Science

ColdNotify

A Notification Service For A Distributed Arctic Observatory

—

Petter Kraabøl

INF-3990 Master's Thesis in Computer Science – May 2019

Abstract

One of the key challenges in the Distributed Arctic Observatory (DAO) project is designing infrastructure to reliably interact with remote, configurable observation units that capture and provide observation data from challenging environments. DAO's infrastructure is a work in progress and researching alternative strategies for interacting with observation units is necessary to gain experience and knowledge about limitations and requirements.

In client-server models, a common approach to keeping clients up to date is continuous polling, however, this may cause unnecessary stress and bandwidth as DAO scales to hundreds or thousands of observation units. Another approach to this is server-initiated publishing methods, where back-end applications provide new data to observation units. This, however, requires per-application implementations that have to keep track of which observation unit has received what, handle unreachable units and potential state loss.

This thesis has explored how notification services can help back-end application reliably interact with observation units in future deployments, to keep them up to date with configurations, perform remote operations or gather data, as DAO scales.

ColdNotify is an application-neutral notification service, based on Thialfi by Google, that aims to reliably deliver notifications to observation units, despite unreliable connectivity and state loss.

Acknowledgements

I would like to thank my supervisor Professor John Markus Bjørndalen and co-supervisor Professor Otto Anshus for the idea of combining Thialfi, and their Distributed Arctic Observatory (DAO) project.

Another thanks for the fellow students working on different areas of DAO, who has provided feedback and discussion for design, implementation and more around ColdNotify.

Contents

List of Figures

List of Tables

/1 Introduction

Distributed arctic observatory (DAO) [22] is a platform for observing and gathering climate and wildlife data in arctic environments at scale. Observation units (OUs) are physical devices, set out to measure and monitor their surroundings through various sensors and deliver observation data to back-end storage systems for processing.

One of the key challenges for the DAO project is developing infrastructure that helps automating data transfers, OU configurations, health checks, software updates, and more, which used to be done by hand. Limited and unreliable connectivity in arctic regions challenges the infrastructure design to provide reliable, fault-tolerant communication strategies, such that OUs can communicate with back-end applications.

OUs may have multiple processes running locally for applications such as configuring certain sensors and for doing local data processing. One approach to keeping these applications up to date with configurations is to have them poll back-end applications for changes at fixed intervals. However, as the number of OUs and applications increase, more stress is added on the servers and unnecessary bandwidth is used when requests return empty due to no updates.

Publish-subscribes (pub/subs) messaging is a popular approach to solve this, by letting clients subscribe to data streams (channels) which publishers can push data through. Despite solving the polling problem, this requires a pub-

/sub implementation that reliably delivers data to subscribers, even if they temporarily go offline. Moreover, additional back-end applications may have to re-implement the same functionality, and pub/sub implementations are prone to flood clients as they come back online with data they have missed out on.

Notification services can be seen as light-weight pub/sub services that notifies subscribers, sometimes with just a ping, when an event has occurred. These service are commonly used in distributed systems that are meant to scale, for applications such as cache-invalidation, where content servers invalidate cache on their cache servers using small notifications, and the cache servers will refresh their data.

DAO's infrastructure is an early work in progress, such that exploring alternative communication methods key. This thesis provides an early exploration of how DAO, developed at the University of Tromsø (UiT), can benefit of a notification service as part of its infrastructure, that lets any back-end application reliably notify OUs applications with small notifications, regardless of application-specific data structures.

I present the design and implementation of ColdNotify, a notification service based on Google's Thialfi [2], adapted to DAO.

In the following chapters, I review relevant notification services and closely related technologies, followed by a review of limitations and requirements of implementing a notification service for DAO. Furthermore, I describe the design and implementation of ColdNotify and present its performance evaluation during high loads and failure recoveries with synthetic OUs. The final chapters discusses future work and alternative implementations to expand on ColdNotify's features and requirements as the DAO project evolves.

1.1 Contributions

- An application-neutral notification service for the DAO project at UiT.
- An implementation of Thialfi's in-memory design (adapted to DAO) that relies on open-source software.
- Discussion of how notification services can benefit DAO and their use cases.
- Discussion of future work for further adapting a notification service as the

DAO project evolves and new requirements and limitations are revealed.

2 Notification Services

Notification services are often used to tackle scaling challenges for pushing data or providing cache-invalidation in distributed systems, ensuring that clients (subscribers) stay up to date with servers (publishers). Rather than sending arbitrary application data, like pub/sub services might provide, notification services typically send small messages that serve as a ping to subscribers, informing them that an event has occurred and leaves it up to them to act upon this.

Similar to pub/sub services, clients subscribe to specific event *channels*, usually identified by a unique identifier to separate notification channels among subscribers. Multiple clients may subscribe to the same notification channels, such that a publisher can multicast notifications. In use cases such as cache-invalidation, cache-servers can subscribe to channels identified by a directory

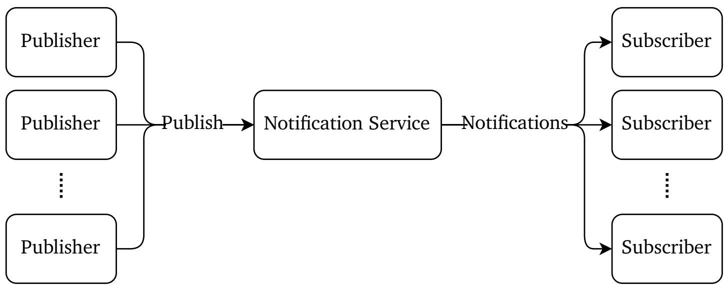


Figure 2.1: A typical notification service with publishers and subscribers.

or file name, such that content servers can publish notifications through these channels, informing the cache server to refresh specific content.

The notification payload varies among notification services and their purpose. Some service, like Siena [5], allows for small untyped application-specific payloads that give more context to the event that occurred. Other services may only provide a ping or application-neutral metadata, such as a sequence number, version number or a timestamp to differentiate notifications.

2.1 Related work

This section reviews related work on notification services and similar technologies to get an overview of various design alternatives and how they can relate to DAO.

2.1.1 AnNotify

AnNotify [17] is a private notification service that focuses on anonymity among subscribers and has been used in cache systems. Notifications are routed through mix networks (mixnets) to make them hard to trace, as they are being delivered to subscribers. The notification service also focuses on low cost in terms of bandwidth and processing power, while scaling to millions of clients, which can be relevant for the OUs limited hardware and bandwidth.

2.1.2 Thialfi by Google

In 2011, Google published their research and design of Thialfi [2], a client notification service that promises eventual notification delivery to clients across their applications, despite data-center failures and unreliable clients. Thialfi is a general-purpose notification service that lets back-end applications notify client applications when a data object has changed and provides only a version number for this object. The version numbers are used to order notifications to determine whether a client has already received a notification or not. When clients are notified about an update, they are responsible for acting on these and potentially download the actual data object directly from the back-end application.

Google does not provide a formal model of Thialfi, however, Figure 2.2 shows my interpretation of its architecture based on the paper [2] and a presentation by one of the authors [1]. The design and implementation of ColdNotify closely

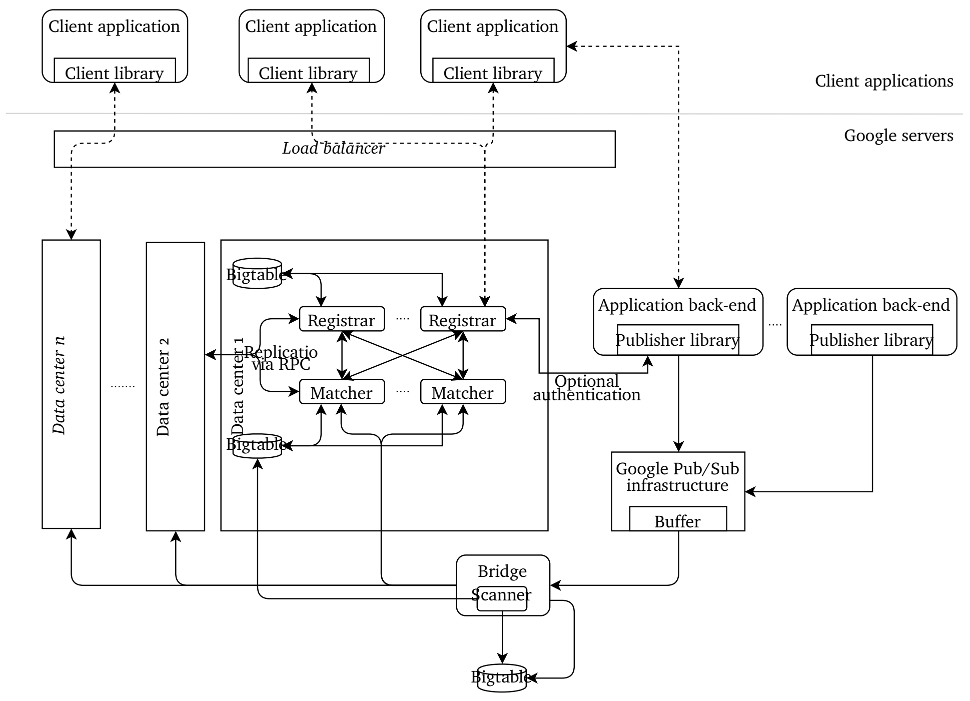


Figure 2.2: My interpretation of Thialfi's architecture within data centers.

follows Thialfi's design, due to its fault tolerance and handling of unstable clients that may remain offline for long periods or lose state.

2.1.3 NotServers at Dropbox

Dropbox, a personal cloud storage service, initially used polling to synchronize files across users' devices. As the number of users increased, polling the servers for updates was no longer viable, which lead to notification servers, called NotServers [14], that notify devices when a file has been updated. This eliminates polling, as devices will now request data only when notified by a NotServer.

To deliver notifications, HTTP long polling is used, where clients send HTTP requests to the NotServers, which are only returned when a notification is available. New requests are subsequently made to receive new notifications. To identify devices and provide separate notification channels, each device includes a unique identifier in their requests [8].

As opposed to AnNotify, NotServers do not have the same level of requirements for privacy. NotServer-client communication is unencrypted over HTTP, due to computational expenses of TLS encryption and the lack of private or secret content in notifications' payload. Using SSL could also cause unnecessary strain on OUs, and the notifications themselves can therefore benefit from being non-descriptive and seemingly pointless to external observers.

In 2012, NotServers could handle 170 thousand clients per instance and several instances run on machines with shared memory.

2.1.4 Wormhole by Facebook

Wormhole [21] is a pub/sub service that monitors writes across distributed database systems at Facebook. When content, such as a new user post, is written to a database, interested applications (subscribers) receive relevant data from Wormhole.

Wormhole handles a large bandwidth of several gigabytes per second across trillions of messages every day, which is not applicable to DAO, however, integrating a notification service with the flow of application data to automatically publish notifications is relevant. ColdNotify requires each back-end server to deliberately publish notifications as events happen, however, exploring whether Wormhole's approach is suitable is appropriate in future work.

2.1.5 MQTT Protocol

Message Queuing Telemetry Transport (MQTT)[16] is a pub/sub protocol designed for Internet of Things (IOT) devices with limited network bandwidth and power.

MQTT requires a message broker such as Eclipse Mosquitto [9] that handles all communication between subscribers and publishers. While this thesis focuses on a notification service, MQTT and existing brokers may be suitable alternative for DAOs, however, this involves passing application-specific data through a broker, which may add additional complexity for storing and data recovery may not always be available, which is not an issue for ColdNotify. ColdNotify also eliminates complexity for potentially protecting sensitive or private data with different requirements to security.

2.2 Use Cases for DAO

This section describes where a notification service can fit into DAO and its applications. The two concrete use cases given below help define the design and requirements that should be expected from ColdNotify, as they play an important role in building the infrastructure of DAO.

2.2.1 Region-specific configuration

Application processes running on OUs typically have configuration files for how the application should be run. This can for instance be an OU application that observes animals, which has configurations for how to capture data, when to capture, which sensors to use and maybe how to process the data locally.

These configurations might have to be tweaked over time and distributed to OUs from remote servers. Moreover, OUs observing animals might be distributed over large areas, such that configurations might be region-specific, for instance to account for local weather or to experiment with certain configurations within a region.

To integrate a notification service into this process, OU applications can subscribe to notification channels, identified by the configuration name. When configurations are changed on the back-end application, it can publish a notification to the OUs applications, via a notification service, letting them know which configurations have changed. OUs applications can act on these notifications by downloading configurations, based on which notification channel the notification came through.

Grouping OUs applications by regions is just one example. One might also have applications where grouping by OU type (which sensors they have on board) is more applicable.

This procedure of keeping subscribers (OUs) in sync with shared data (configurations) is similar to Thialfi and NotServers' Different procedures often comes down to how subscribers act upon notifications.

2.2.2 Observation data on demand

An alternative way to act on notifications is providing publishers with data. A significant part of the DAO project is to reliably gather observation data from OUs to use them for processing, and several DAO projects at UiT revolves around this process.

If an OU subscribes to notification channels identified by the kind of data it provides, back-end storage applications can for instance publish a notification for *humidity-data* in *region-1*. The notification service will deliver the notification to OUs within *region-1* that has subscribed to *humidity*, which can act on this by preparing and sending data to the storage application.

This use case can be extended, where publishers may request health checks, status reports, log files, etc.

3 Design

This chapter describes the design of ColdNotify to tackle limitations and requirements that are suitable for DAO and its OUs. The design is closely related to Thialfi's in-memory design and follows the same naming scheme where applicable. As several aspects of DAO's infrastructure and OU specifications are a work in progress, certain design aspects of ColdNotify, like communication methods, energy usage and security, are either based on estimations, assumptions or left out to be revised in future work.

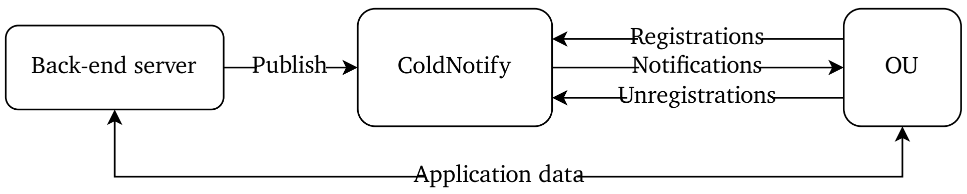


Figure 3.1: An overview of ColdNotify. Notifications are published to ColdNotify, which routes them to interested OUs. Application-specific data is transferred directly between OUs applications and servers.

3.1 Overview

Figure 3.1 shows an abstraction of how ColdNotify is used within DAO. ColdNotify follows Thialfi's model, where notification channels are identified by application-defined object names, such that a notification references a specific object. As an example, a notification channel may be named *camera-config* and notifications published through this channel indicates that a camera config object has been updated. Furthermore, application-specified version numbers are attached to each object notification, such ColdNotify can keep track of the latest version numbers and which version was last sent to an OU.

ColdNotify does not handle any application-specific data, except the object identifier and a version number. This eliminates additional complexity for data recovery or potentially storing sensitive data which should be encrypted, and makes the notification size small (10s of bytes), which is preferable for OUs. Granted, application-specific data may be transferred as a result of a notification, but the OU application will now be aware that an event has happened and can act on this when suitable, for example by scheduling downloads when they have enough battery or upgrade software during observation downtime.

Expanding the notification payload further is discussed as future work in Section 8.2, as OU applications may benefit from notifications that can carry metadata, such as data size, level of importance or expiration date. Such metadata can further help instruct the OU application to act on notifications.

To interact with ColdNotify, OU applications use a client library to register for specified objects to and receive notifications. Back-end servers publish notifications using a publisher library, which ensures that publish messages are properly structured and can be read by ColdNotify.

3.2 ColdNotify's State

ColdNotify's state holds which clients are registered to which objects and the last known version of each object. When a notification for an object is published from a back-end application, ColdNotify finds a list of clients to notify. When a client registers or unregisters for object notifications, ColdNotify has to make changes to this list of clients. To do these lookups efficiently, ColdNotify's state is split into two views: one indexed by clients and the other indexed by objects to provide a $\mathcal{O}(1)$ lookup time complexity.

The two views are held by two separate processes/servers: the *registrar* that stores registrations indexed by clients, and the *matcher* that indexes by objects.

When a client now registers for an object (Figure 4.2), the registrar simply looks up the client and adds the registration to their list of registrations. To reflect this change in the matcher's state, the registrar *propagates* (transfers) the registration to the matcher, which looks up the object and adds the client to the list of subscribers.

There is, however, a slight difference between the registrar's and matcher's state regarding the object version. The registrar holds which version number each client received last, while the matcher holds the most recent version number from the publishers. This way, the registrar does not have to resend notifications for a reconnecting client if the most recent version number in the matcher's state matches the version in the registrar's state.

Thialfi uses a third component, the *bridge*, which extracts publish messages from Google's internal pub/sub infrastructure, converts them into a readable format for the matcher and ensures that the matcher receives it. DAOs does not have any internal infrastructure like this, such that publish messages are sent directly to the matcher, eliminating the need for a bridge.

3.3 Registrar

3.3.1 Client communication

The registrar concurrently communicates with OU applications to register, unregister and send notifications. To reliably communicate with OUs, communication methods such as TCP sockets is preferred to deliver messages, but also application-level acknowledgements, such as acknowledging that a registration has successfully been stored in the registrar and matcher's state.

OUs are prone to disconnect, whether it is temporarily lack of connectivity, powering cycling to save battery or running out of battery, which may leave them offline for long periods. ColdNotify holds onto notifications for offline OUs, and delivers them when they come back online. To not flood OUs with every missed notifications on wakeup, ColdNotify will only send the most recent notification per object. This means ColdNotify does not ensure that all notifications are delivered to OU applications, but rather that they will eventually learn the latest version number.

3.3.2 Notification channel namespaces

To divide OU applications into groups, as described in Subsection 2.2.1, notification channels are namespaced by *group* identifiers. Furthermore, groups are namespaced by *domain* identifiers, which are unique to each application. As OU applications connect to the registrar, they will introduce themselves with which domain and group they belong to. Applications within the same domain and group can now register for and receive the same object notifications.

3.3.3 Registration-sync protocol

Thialfi's registration sync (reg-sync) protocol is important to ensure that OU applications and the registrar agrees on registrations and detects any mismatches caused by state loss or communication failure. A registration state is stored locally on the OU (handled by the client library), which keeps track of which registrations are acknowledged by the registrar. The local registration state and the registrar's must stay in sync, such that online OUs do not miss out on notifications.

To detect registration mismatch, a registration *digest* is appended in OU-registrar communication, such that either of them can verify the other's state. The digest is a summary of all active registrations and can for example be the hash sum of all registrations. In the event of a mismatch, the registrar drops all registrations that it has on the OU application, such that the client library can resend all registrations.

3.4 Matcher

3.4.1 Communication

The matcher accepts incoming notifications from publishers and looks up the list of OUs that should be notified and sends this to the registrar. The matcher does not request missing data, such as version numbers, from publishers, as this would require publishers to accept incoming requests and keeps a registry of version numbers, which is not always the case. If a registration is made for an object with unknown version number, the matcher responds with an *unknown* version, which always triggers a notification, such that subscribers can decide how to act on this.

Similar to the registrar, the matcher has workers for resending missed messages. Any unpropagated notifications will be added to a list of pending notifications, such that a worker can attempt to send them concurrently.

3.4.2 Recovery

In Thialfi's in-memory design, if a matcher fails, all servers, including the registrars within the data-center, are restarted and loses their state and lets the reg-sync protocol reconstruct their state. ColdNotify takes a different approach to handling matcher failures, by taking advantage of the similarity between registrar and matcher's state.

They both contain information about which object each client is registered to, so the matcher will simply tell the registrar to re-propagate all registrations to reconstruct its state. Even though the registrar keeps track of version numbers, they only represent the last version that a subscriber received and not the latest sent by a publisher, such that the version number will remain unknown until a new publish message is received.

4 Implementation

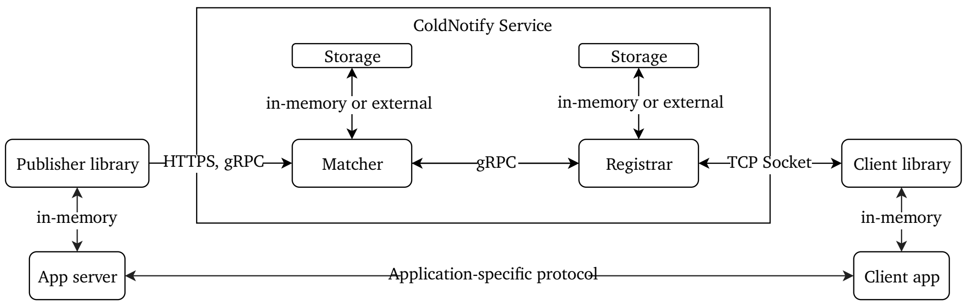


Figure 4.1: ColdNotify components and communication protocols.

This chapter describes in detail how each component of ColdNotify, shown in Figure 4.1, is implemented. The implementation closely follows Thialfi's in-memory design and naming scheme, but has over time discarded certain features and components that are not suitable or applicable for DAO.

ColdNotify is implemented in Go, to take advantage of concurrent goroutines for handling OU connections The client library is written in Python and the publisher library in Go. All communication between components are defined using Google's protocol puffers [12] to make sure any future packages for other languages follow the same message structure. Furthermore, gRPC [10] is used with protocol buffers to define RPC services. Appendix A includes instructions for how to set up and run ColdNotify.

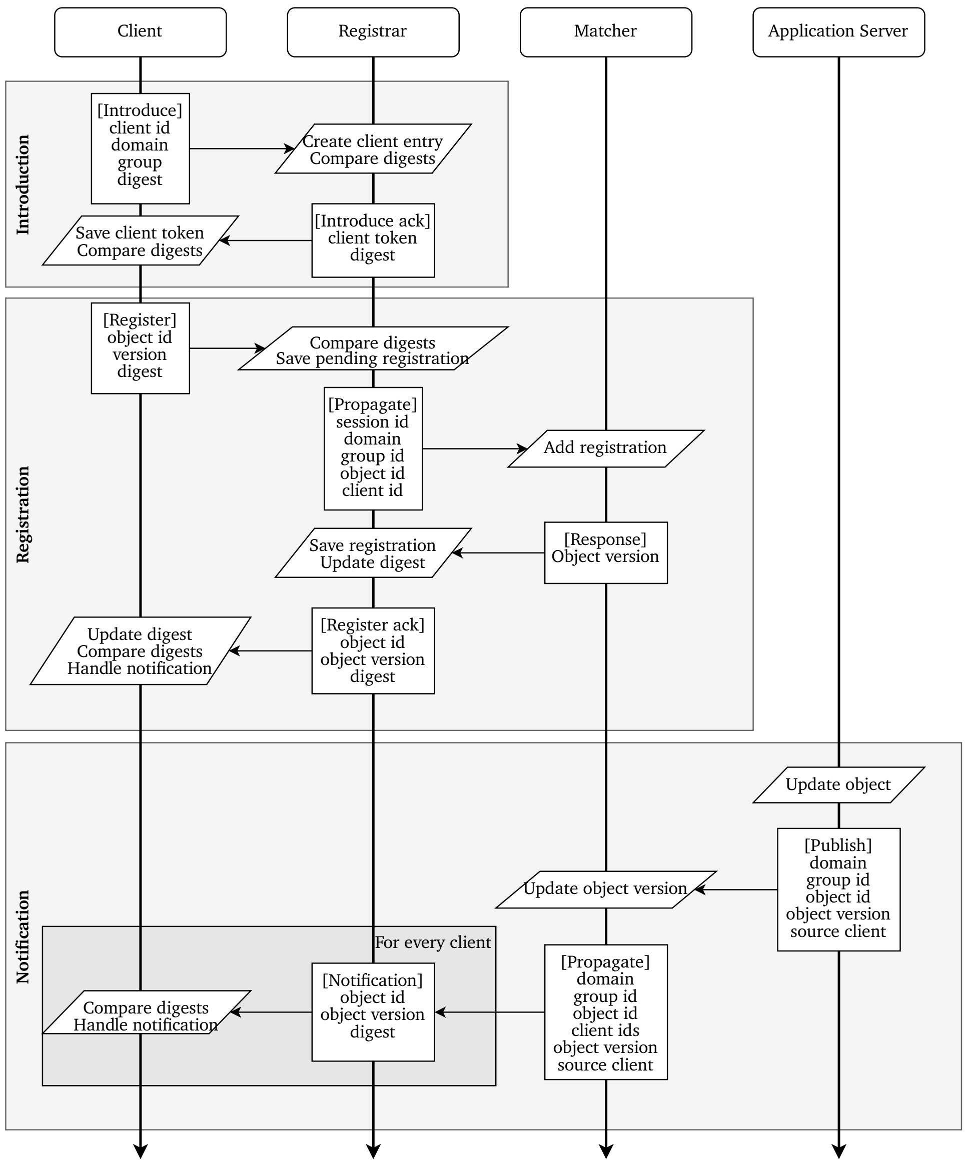

Client
Registrar
Matcher
Application Server
Introduction
[Introduce] client id domain group digest
Create client entry Compare digests
Save client token Compare digests
[Introduce ack] client token digest
Registration
[Register] object id version digest
Compare digests Save pending registration
[Propagate] session id domain group id object id client id
Add registration
Save registration Update digest
[Response] Object version
Update digest Compare digests Handle notification
[Register ack] object id object version digest
Notification
Update object
[Publish] domain group id object id object version source client
Update object version
For every client
[Notification] object id object version digest
Compare digests Handle notification
[Propagate] domain group id object id client ids object version source client

Figure 4.2: Communication model for client introduction, registration and notification delivery. Rectangles represent messages and their content, and parallelograms represent actions.

Client Token			
domain	group	client id	registrar session id

Table 4.1: Client token is a combination of application domain, group name, client ID, and registrar's session ID. Each registrar (and matcher) process has a unique session id to distinguish multiple registrars and matchers in a horizontally scaled version of ColdNotify.

Figure 4.2 gives and overview of how each component communicates during introduction, registration and notification delivery. Unregistration is similar to registration, except that the registrar and matcher removes the registration and only send an acknowledgement, with no version number, to the OUs.

4.1 The Registrar

```go

type RegistrarService struct {
	sync.RWMutex
	SessionId                  string
	Persistent                 bool
	PendingRegistrations      map[string]*matcher.Registration
	pendingRegistrationsLock  sync.RWMutex
	Clients                   map[string]*Client
}

type Client struct {
	sync.RWMutex
	Id                    string
	Group                 string
	Domain                string
	Token                 string
	Digest                uint64
	Online                bool
	LastOnline            time.Time
	PendingNotifications map[string]*Object
	Registrations        map[string]*Registration
}

```

Listing 4.1: Registrar state (simplified). Fields for storing TCP connections, matcher RPC connection, etc. has been stripped.

4.1.1 Client communication

The registrar process faces the OUs applications to handle registrations, unregistrations and provide notifications over TCP sockets. Alternatively, OU-registrar communication could be long-polled HTTP requests, like Dropbox's NotServers [14], where clients send a request to the server which is only returned whenever a notification is available.

Using a TCP connection guarantees that the data is delivered and that there is no corruption, however, application-level acknowledgements are used to confirm that registrations and unregistrations has been confirmed by both the registrar and matcher. If the registrar is unable to immediately propagate a registration to the matcher, it's added to a list of pending registrations, such that a goroutine can retry sending them and eventually send an acknowledgement back to the OU.

When a OU application connects to the registrar (introduction in Figure 4.2), an entry for the client is created and indexed by a unique *client token*, corresponding to Table 4.1. To ensure unique tokens, they consist of the *domain* name, a *group* name and a unique *client id* within that group. It's important to note that each application running on an OU has their own TCP connection with ColdNotify, with different domains.

4.1.2 Workers

The registrar process has concurrent workers (goroutines) propagate pending registrations and does garbage collection of OUs.

Some OUs go offline and never come back, however, their registration data will remain in the registrar's and matcher's state. This can happen when OUs are replaced, they break, or they may have connected to another registrar in a horizontally scaled setup. The registrar keeps track of whether a OU is online (connected) or offline and the date it last disconnected. By doing this, a registrar worker checks for offline clients that have not been connected for a long time (weeks, months), unregisters their registrations from the matcher and deletes their state.

4.2 The Matcher

```
type MatcherService struct {
    sync.RWMutex
    SessionId            string
    Persistent           bool
    NotificationInterval time.Duration
    Batching             bool
    Pending              Pending
    Objects              map[string]*Object
}

type Object struct {
    sync.RWMutex
    Id       string
    Domain   string
    Group    string
    Version  string
    Clients  []string
}

type Pending struct {
    sync.RWMutex
    notifications []*registrar.Notification
}

```

Listing 4.2: Matcher state (simplified)

The implemented matcher accepts publish notifications from back-end applications via HTTP POST requests and RPC calls. The RPC server is also used by the registrar to keep the matcher's state up to date with registrations and unregistrations from OU applications.

4.2.1 Notification batching

When a notification is published by back-end applications, the matcher's job is to lookup which clients the registrar should notify. This essentially means that one publish from back-end application for N OUs requires 1 lookup for the matcher, but requires N lookups, plus TCP communication, for the registrar. To remove some stress from the registrar, notifications can be batched by the matcher and sent in bulk. The real optimization by batching is when the matcher receives multiple notifications for the same object within a short time frame, as it will simply dismiss old notifications and only propagate the newest to the registrar.

In practice, there is a list of pending notifications using a map (associative array) as data structure, where the key uniquely identifies an object, regardless of version, such that subsequent notifications for the same object overwrites its predecessor. When batching is enabled, all notifications are added as pending notifications, otherwise this list is only used for failed propagations. A goroutine propagates the list of pending notifications to the registrar at a set interval, usually in seconds or minutes.

An alternative to the current batching implementation is an auto-tuned batching algorithm [4], originally created for pub/sub systems that are more prone to bandwidth congestion due to larger payloads. This algorithm can help alleviate congestion caused by heavy load on the registrar or slow OUs and increase latency by dynamically tuning the batching size and propagation interval according to the registrar's load. The algorithm is based on additive increase/multiplicative decrease (AIMD)[7], which is commonly used to control TCP congestion.

To implement a similar algorithm for ColdNotify, the registrar could respond to the matcher with its metrics, for instance as CPU usage, such that the matcher can dynamically change propagation interval and a batch size limit. The experiment results in Section 5.2 shows how the registrar is affected during high notification rates without batching.

4.3 Client library

```python
class State:

    def __init__(self, memory: bool, filename: str = None):
        self.memory: bool = memory
        self.filename: str = filename
        self.registrations: Dict[str, Registration] = {}


class Registration(Subject):

    def __init__(self, object_id: str, version: str, status: RegistrationStatus):
        self.object_id: str = object_id
        self.version: str = version
        self.status: RegistrationStatus = status
```

Listing 4.3: Client library registration state (simplified).

The majority of existing OU applications are written in Python, such that a Python library for ColdNotify has been first priority. Listing 4.3 shows the client library state, where registrations are indexed by their object name.

The client library takes an object-oriented approach, where a Notify singleton is created by the OU application with parameters for domain name, group and client identifier. Listing 4.4 shows how the Notify object from the client library is used.

```
from notify import Notify, State, Connection

notify = Notify(
    domain='animal.observer',
    group_id='region-1',
    client_id='observer-1',
    state=State(filename='registration-state.json'),
    connection=Connection(
        address='registrar.coldnotify.dao',
        port=443,
        ca_cert='ssl/coldnotify.crt',
        cert='ssl/client.crt',
        key='ssl/client.key'
    )
)

notify.connect()

notify.register('camera-config').subscribe(
    lambda notification: update_camera_config()
)

notify.unregister('camera-config')

notify.disconnect()
```

Listing 4.4: How an OU application interacts with ColdNotify using the client library.

The Notify object run the TCP connection in a thread to not interrupt application operations when communicating with ColdNotify or using the reg-sync protocol. To deliver notifications to the application, the client library follows the observer pattern by using RxPY [18] by ReactiveX. The Notify.register function returns a *Subject*[19] object that emits new notifications to subscribers.

The OU application's object registration state can either be persistent or memory-only. The registration state stores whether an object has been acknowledged as registered or unregistered by ColdNotify. Until a registration has been acknowledged, its status is either *pending registration* or *pending unregistration*.

4.4 Publisher library

The publisher library lets back-end applications publish notifications to the matcher over RPC and is significantly less complex than the client library. The library is implemented in Go, however, implementations in other languages should be straight forward, as the matcher's rpc service is defined using protocol buffers. Applications can alternatively send notifications via HTTP POST requests directly to the matcher and protocol buffers define these messages too.

```go
import "repository/coldnotify/publisher"

// Publisher service connects to matcher server
publisher.Service = publisher.NewService(
    "matcher.coldnotify.dao:443"
)

err := publisher.Publish(
    domain, group, objectId, version, source
)
if err != nil {
    // Application-specific error handling
    handlePublishError()
}
```

Listing 4.5: Code for publishing a notification to ColdNotify using the publisher library.

Listing 4.5 shows how Go applications publish notifications. Like Thialfi, the publisher library supports a *source* field, which indicates which client is responsible for an object change, such that it does not have to be notified. Most DAO applications will likely use this only in rare occasions, as most object changes are made by the application server itself instead of OUs.

If the matcher is unavailable, such that the publisher library is unable to publish a notification, the error handling is left to the application. This seems appropriate, as the application should be aware of whether a notification was actually sent and handle new attempts in its own way.

4.5 Persistent State

Experiments in the next chapter will show that registrar and matcher state losses are expensive, such that persistent state does greatly benefit ColdNotify. Selecting a proper storage platform for persistent state is left as future work, such that the current implementation uses local files just to demonstrate persistent storage.

Go can conveniently convert the registrar and matcher's state, which are implemented as structs, into JSON data and vise-versa using marshalling and unmarshalling [11]. When persistent storage is enabled, the registrar and matcher will check if a state file exists and load its JSON state into state. A goroutine worker locks the state and overwrites the file at a specified interval (seconds).

As ColdNotify scales, saving the entire state to file over and over will cause additional resource usage and will be prone to race condition failures[1]. Corrupted or invalid JSON files will not be loaded to state, however, finding a storage system that let ColdNotify update individual parts of the state is beneficial. Thialfi's persistent storage design uses local memory as a cache, while most of the state is only stored remotely in Bigtable.

4.6 Failure Recovery

4.6.1 Registrar recovery: registration-sync protocol

When the registrar is restarted as loses its state, the reg-sync protocol will be initiated, such that OUs resend their registrations.

To implement the registration-sync protocol, the 32bit CRC checksum of each registration (object identifier) is summed as an unsigned 64bit integer and used as the digest. CRC is a fast algorithm and can be used by OUs with smaller impact on CPU, compared to hashing algorithms such as SHA or MD5.

When a previously connected OU reconnects to the registrar and sends an introduction message, the registrar will drop its registration state if there is a digest mismatch. When this happens, the registrar will respond to the introduction message with a digest of 0, and the reconnecting OU will initiate the reg-sync protocol. As the client library resends registrations, the digest in the registration acknowledgements will not match the OU's, until all registrations are sent. To account for this, the client library enables *reg-sync mode*, where digest mismatch is ignored until all registrations has been resent and the

1. The registrar and matcher's state uses a mutex lock for alternating their client and object list respectively. However, the clients and objects within these lists have their own mutex lock, such that registrations, unregistrations or notifications don't lock the entire state of the registrar and matcher. To safely save the registrar's and matcher's state to disk, all clients and objects must be locked, which is not practical.

digests match again. During *reg-sync mode*, the registrar will also notice a digest mismatch, however, the registrar only drops its state during introduction.

4.6.2 Matcher state recovery

As the matcher starts, a goroutine will connect to the registrar's RPC server and request all registrations. This happens regardless of failure or a regular startup to account for any registrations that may have occurred if the registrar process is started first.

This is a best-effort recovery, hence if the matcher is unable to contact the registrar, it is assumed that the registrar is down and will perform the reg-sync protocol among its clients, which will eventually be propagated to the matcher.

4.7 Security

ColdNotify does support TLS communication between components, including communication between matcher and registrar, as they may be running on separate machines. The registrar and matcher verifies certificates on OU and back-end applications to ignore unknown users. As discussed in Subsection 2.1.3, encrypting communication plus ensuring components are legitimate, will affect ColdNotify's throughput and OUs resources.

The current state of DAO experiments with a virtual private network (VPN) where all OUs and back-end servers are connected, such that all communication will be encrypted regardless.

5 Evaluations

This chapter evaluates ColdNotify's performance in experiments that show resource usage and notification latency during high loads of clients and notifications. It also takes a look at how ColdNotify behaves during failure recovery, in experiments where the registrar and matcher loses and rebuilds their state. These experiments are similar to Google's experiments for how Thialfi's resource consumption and notification latency is affected by scaling, however, at a smaller scale in terms of online clients.

Process	**Nodes**	**CPU**	**RAM**
Registrar	1	Intel Xeon W3550 @ 3.07GHz	12GB (3 x 4GB)
Matcher	1	Intel Xeon W3550 @ 3.07GHz	12GB (3 x 4GB)
Back-end server	1	Intel Xeon W3550 @ 3.07GHz	12GB (3 x 4GB)
Clients	14	Intel Xeon W3550 @ 3.07GHz	12GB (3 x 4GB)
Clients	14	Intel Xeon E5630 @ 2.53GHz	12GB (6 x 2GB)
Clients	30	Intel Xeon E5520 @ 2.27GHz	12GB (6 x 2GB)
Clients	18	Intel Xeon E5-1620 0 @ 3.60GHz	32GB (4 x 8GB)
Clients	4	Intel Xeon W3550 @ 3.07GHz	12GB (3 x 4GB)
Clients	4	Intel Xeon W3550 @ 3.07GHz	8GB (4 x 2GB)

Table 5.1: Experiments are run on a cluster of physical LAN nodes. Clients are evenly distributed over 84 nodes.

5.1 Experiment Environment

All experiments take place on a cluster, described in Table 5.1 with synthetic clients, acting as OUs, that run a simple application for interacting with ColdNotify. The cluster consists of physical nodes connected in a LAN, such that communication latency is mostly caused by each component, rather than the network.

There are currently no large-scale deployments of OUs to do real-world testing, such that these experiments focuses on ColdNotify's performance without taking into account OU limitations, such as slower processors, less memory, bandwidth, etc. Testing ColdNotify in real environments is left as future work for when the DAO project progresses to large-scale deployments.

Location for raw experiment data and tools for processing data are described in Appendix B.

5.1.1 ColdNotify Monitoring

ColdNotify's processes, the registrar and the matcher, run on separate nodes to not interfere with each other's resource usage. This separation is important to identify how each experiment affects the two processes and gives an indication to which process might benefit of being horizontally scaled to more nodes.

The registrar and matcher are monitored using psutil 5.6.2 [20] to log CPU percentage $\frac{cpu\_percent}{cpu\_count}$ and memory (resident set size) percentage every 100 milliseconds.

5.1.2 Demo Application

An application server is implemented in Go, which publishes notifications to ColdNotify over RPC at specified intervals. Clients are Python processes that use the ColdNotify client library to register for objects and receive notifications.

Any application-specific operations is not part of these experiments, as they will vary for each application. Therefore, the client application does not download from, or interact with, the server, but is only instructed to register for objects and log when a notification is received from the client library.

5.2 Experiment: Notification Latency

This first experiment looks at how notification latency is affected when increasing the notification rate.

5.2.1 Methodology

1 000 clients connect to the registrar and registers for the same object, such that one published notification from the application server generates 1 000 notifications at the registrar.

The application server increases its publish rate from 1 to 25, in increments of 2. This makes the registrar send out 1 000 to 25 000 notifications per second to clients.

The latency is measured from the notification is sent from the application server, to when it's received by the client application. To keep track of timestamps, the application server sets the object version as the current time in milliseconds. As the client application receives the notification, it compares the version against the current timestamp to find the latency [1]. The delivery latency is gathered from each client's log and combined to find the average and the standard deviation from each notification rate.

5.2.2 Metrics

- Average end-to-end notification latency and standard deviation
- CPU usage in percentage for the registrar and matcher
- Memory usage in percentage for the registrar and matcher

1. Different hardware and potential clock skew between cluster nodes are not accounted for. Further analysis of latencies reported by individual nodes are provided in **??**, which concludes that despite a varying mean latency among nodes, it does not affect the rate at which notification latency changes.

5.2.3 Results

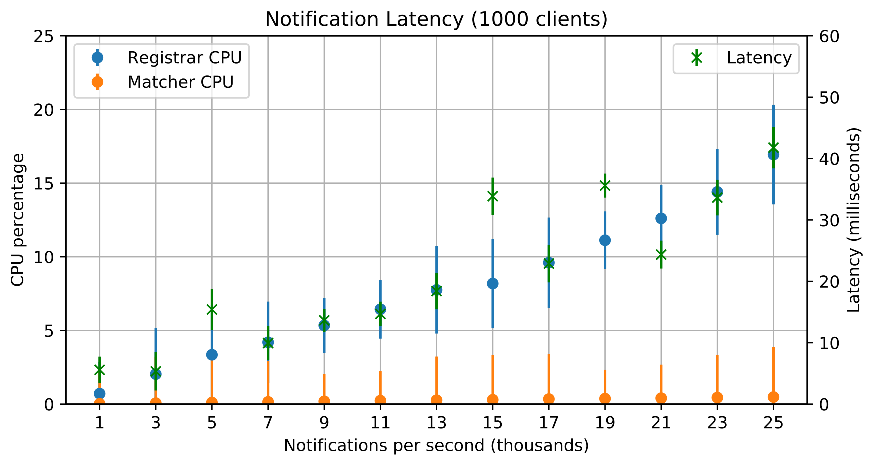


Figure 5.1: Notification latency and CPU usage from 1 000 to 25 000 notifications per second. Latency increases on average by 1.51 milliseconds per notification rate (1.51 µs per concurrent notification).

Figure 5.1 shows the results as the notification rate increases by 1 000 up until 25 000 notifications per second. The matcher process remains almost unchanged by the notification rate with a mean slope of 0.02%, while the registrar increases at a rate of 0.68% on average. The latencies have little variation during each notification rate, however, they do vary between each rate.

The size of a notification was 45 bytes[2] and further analysis of experiment data and client logs confirms that all notifications were successfully delivered.

The takeaway from this experiment is that each concurrent notification contributes little (1.51 µs) to the latency and how different the matcher and registrar processes are affected. In horizontal scaling (discussed as future work in Section 8.1) the registrar will benefit from distributing its load, while the matcher has no immediate need for scaling.

The matcher can be configured to batch notifications and only propagate the most up to date notifications to the registrar at a fixed interval. This may decrease the notification rate and thereby resource usage if a back-end server is rapidly publishing notifications for the same object.

2. Notification size depends on the length of version number, object identifier and registration digest.

5.3 Experiment: Scaling Clients

In this experiment, ColdNotify is measured as the number of connecting clients increase, with a fixed notification rate.

5.3.1 Methodology

The objective of this experiment is to measure how the number of connecting clients impacts notification latency and resource utilization by the registrar and matcher.

Each client registers for the same object and the back-end server publishes a notification every second.

Two types of situations are measured in this experiment. One where ColdNotify only has to send notifications at constant numbers of online clients, and one where it has to handle clients continuously connecting at varying rate, while sending notifications.

Latency is measured in the same way as in the previous experiment (Section 5.2).

5.3.2 Metrics

- Average end-to-end notification latency and standard deviation
- CPU usage in percentage for the registrar and matcher
- Memory usage in percentage for the registrar and matcher
- Number of online clients connected to the registrar

5.3.3 Results

Figure 5.2 shows the resource usage and notification latency at fixed numbers of clients. The resource usage scales similarly to the notification latency experiment in Section 5.2. On average per client, the notification latency rises by 4.3 µs, the registrar CPU usage rises by 0.0007%, and the matcher CPU usage rises by 1e-05%. Further analysis shows that CPU usage for both the matcher and the registrar goes to 0% between notifications, as there are no operations on either processes. This indicates that resource usage in Figure 5.2 mostly, if not

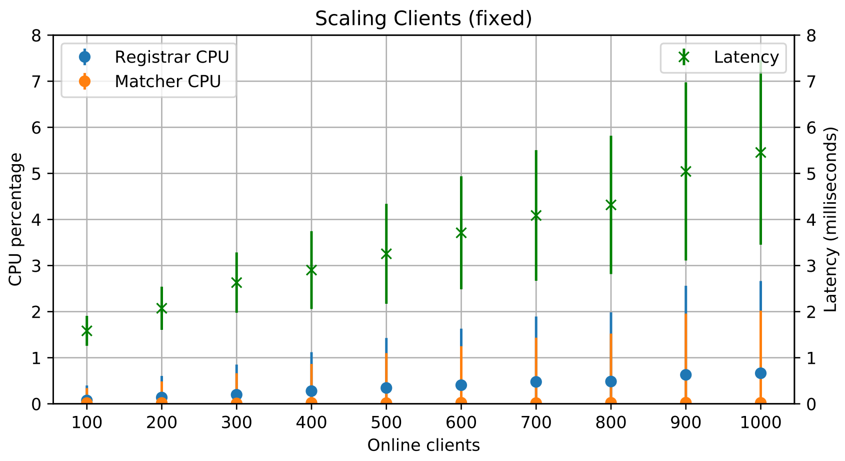


Figure 5.2: Notification latency and CPU usage with fixed numbers of online clients.

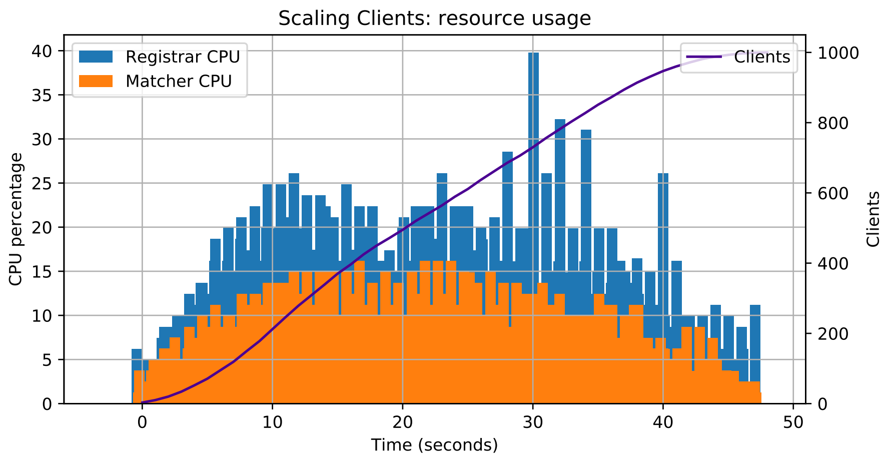
entirely, caused by sending notifications and not idle client connections.

Figure 5.3 shows a continuous timeline as clients increase from 0 to 1000 over 48.0 seconds at an average rate of 20.77 clients per second. In this situation, the registrar and matcher's CPU usage is significantly higher than previously, as each client concurrently registers 50 objects each, which on average equates to a rate of 1 038,5 registrations per second.

Despite the increase in resource usage, the average latency contribution per client is only increased from 4.3 µs to 4.8 µs per client.

5.4 Failure recovery

To evaluate ColdNotify's failure recovery, the registrar and matcher are configured to lose their state in the following two experiments. The objective of these experiments is to measure recovery time and the impact of memory and CPU usage for the registrar and matcher processes. Both experiments start with 1 000 online clients spread evenly over 84 nodes, each with 50 registrations (50 000 registrations in total).



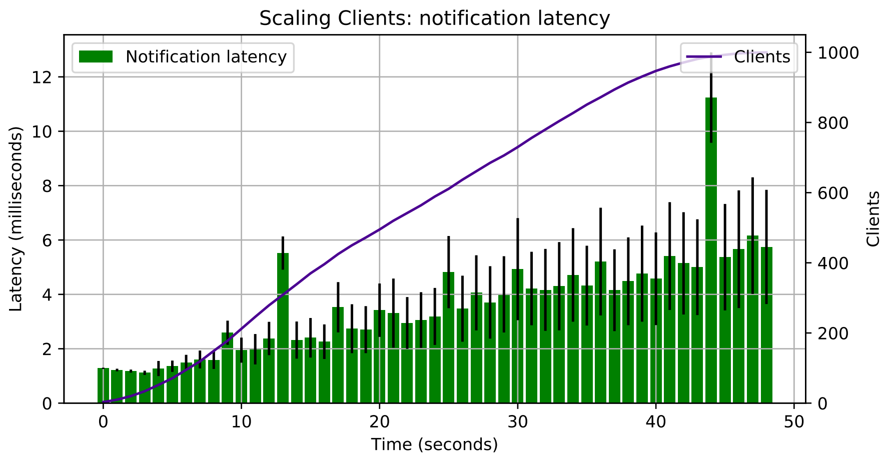


Figure 5.3: Notification latency and resource usage with a continuously increasing number of online clients from 0 to 1000 over 48.0 seconds.

5.5 Experiment: Registrar Recovery

When the registrar server fails and loses its state, all clients are disconnected from ColdNotify and their registrations are lost. The reg-sync protocol will be initiated for each reconnecting client.

The matcher is affected by this failure, as the registrar follows the normal procedure of checking the matcher's state and potentially sending notifications for every registration. For this experiment, however, no notifications are sent.

5.5.1 Methodology

1 000 online clients are connected to the registrar with 50 registrations each. Both the registrar and matcher processes are monitored as the registrar loses its state and starts recovery via the reg-sync protocol. Clients are instructed to reconnect immediately on disconnects, such that all clients will reconnect at the same time.

The recovery time is measured from the registrar process is started, until its CPU usage idles at 0%. Results from previous experiments show that matcher and registrar processes idle when there is no communication.

For comparison, a separate experiment with persistent state is made, where the registrar process loads its previous state on startup.

5.5.2 Metrics

- CPU usage in percentage for the registrar and matcher
- Memory usage in percentage for the registrar and matcher

5.5.3 Results

Figure 5.4 shows the CPU percentage of the registrar and matcher as the registrar recovers in 2,41 seconds. The first 1,5 seconds shows a large CPU impact for the registrar, as the 1 000 clients reconnects at the same time. During this phase, registration mismatches are detected and reg-syncs are initiated to synchronize the 50 000 registrations. Once all the clients have connected, the registrar stabilizes and propagates remaining registrations to the matcher.

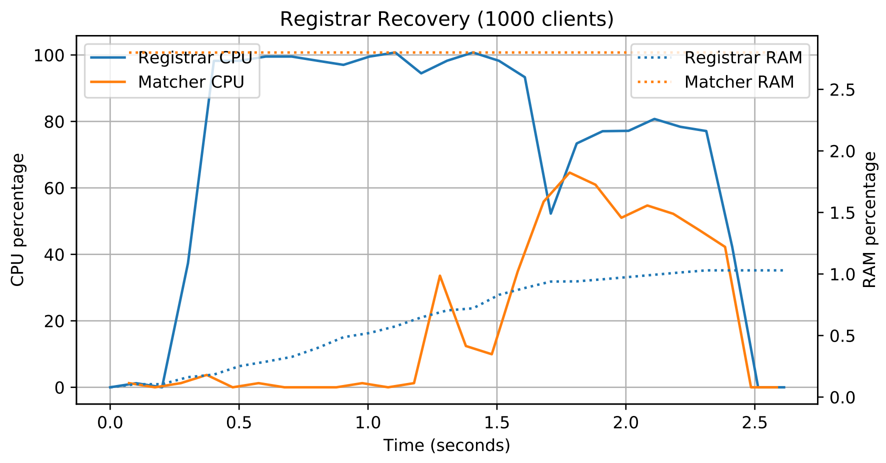


Figure 5.4: Registrar recovery with 1 000 clients, 50 registrations each. Recovery time is 2,41 seconds.

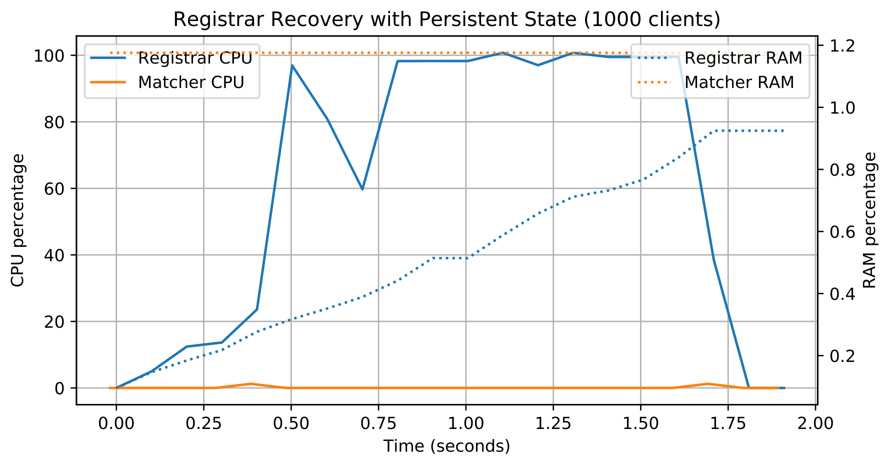


Figure 5.5: Registrar recovery with persistent state. 1 000 clients with 50 registrations each. Recovery time is 1,81 seconds.

Enabling persistent state (Figure 5.5) eliminates the reg-sync protocol from being initiated, thereby leaving the matcher untouched and decreasing the recovery time to 1,81 seconds. However, the surge of reconnecting clients will still take a hit at the CPU usage. With persistent storage, the average CPU usage by registrar goes from 69,37% to 61,12%, and from 19,61% to 0,06% for the matcher.

The registrar is currently set to accept new connections without delay, such that limiting the connection rate can help ease recovery at the cost of longer recovery time.

5.6 Experiment: Matcher Recovery

When the matcher process loses its state, ColdNotify loses the most recent version of each object. The matcher does not request versions numbers from back-end servers, but gets help from the registrar to partially recover its state, which includes all registrations. Online clients remain unaffected by this, as no new notifications are made, however, reconnecting clients will be notified about unknown versions.

5.6.1 Methodology

1 000 are connected to the registrar and registers for 50 objects. The matcher's state now contains 50 objects, each with a list of 1000 registered clients. This state is deleted and the matcher process is restarted, such that recovery time is measured from the matcher process restarts, until it idles at 0% CPU usage. No notifications are made during this recovery.

Similar to the registrar recovery experiment, a separate experiment is done with persistent state, where the matcher loads its previous state on startup.

5.6.2 Metrics

- CPU usage in percentage for the registrar and matcher
- Memory usage in percentage for the registrar and matcher

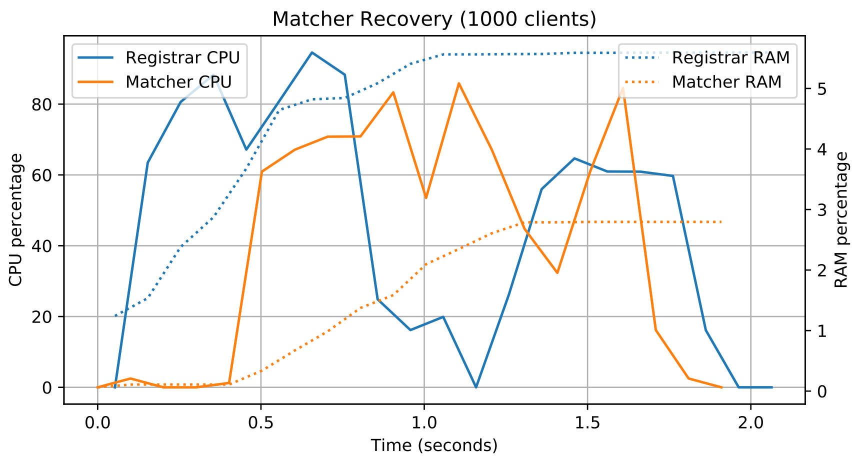


Figure 5.6: Matcher recovery with 1 000 clients, 50 registrations each. Recovery time is 1,81 seconds.

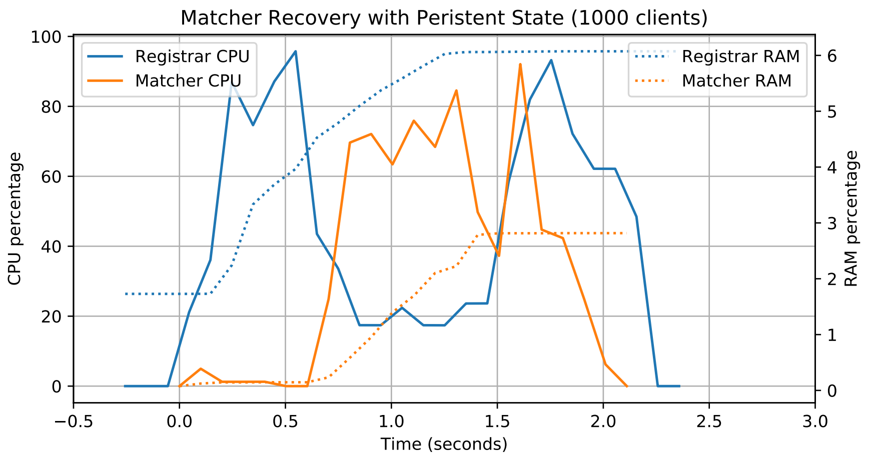


Figure 5.7: Matcher recovery with persistent state. 1 000 clients with 50 registrations each.
Recovery time is 2,01 seconds.

5.6.3 Results

Figure 5.6 shows the recovery timeline and the impact on the processes. The 1,81 second matcher recovery is faster than the registrar's recovery, however, both are impacted higher CPU usage. The registrar is instructed to resend all 50 000 registrations as fast as possible, making the CPU usage rise until the matcher starts processing them.

The registrar has an increase in RAM usage, despite not storing any new data. This is caused by memory allocation for goroutines that send registrations. Further inspection shows that repeated matcher recoveries converge memory usage to about 6,9%, which indicates that Go does free this extra allocated memory during new allocations.

Enabling persistent state should not affect recovery process, as the matcher always request registrations from the registrar on startup, in case it has missed some. However, Figure 5.7 shows an extended recovery time, which is caused by loading the previous state from file and continuously locking state to write to disk. During these two tests, enabling persistent state made the matcher CPU average go from 40,23% to 34,76%, and the registrar from 46,12% to 40,61%.

Implementing a Merkle tree strategy for recovering the matcher could be done, where the matcher provides its hashes to the registrar when requesting recovery[3]. This, however, requires the registrar to transpose its own state, such that it matches the matcher's structure to generate its own Merkle tree. This is not a scalable solution and the problem is left unsolved for now.

3. Building a Merkle tree of the matcher state is done by treating each object entry as a data block. A block's hash will be $hash(id + domain + group) + \Sigma_{i=1}^{N} hash(client_i)$.

6 Discussion

Experiments show that ColdNotify scales in terms of resource usage and notification latency at increased notification rate and connecting clients. When only increasing the notification rate, each concurrent notification adds on average 1.51 µs latency to every delivery, while increasing clients adds 4.3 µs.

Even though these experiments give a performance measure of ColdNotify, they tell little about real-world latencies or throughput, as these depend on how fast OUs can receive notifications, which varies by network latency, software, and hardware. Such testing environments were not available at the time, hence testing ColdNotify in real environments is left for future work.

Despite this, clients that use ColdNotify are reliably notified that a remote event has occurred, despite state loss and offline clients. TCP communication, application-level acknowledgments and the registration-sync protocol ensure that clients and ColdNotify are always in sync with registrations. Using Thialfi's design

6.1 A Notification Service For A Distributed Arctic Observatory

Developing ColdNotify in the current state of DAO is challenging, as there are currently no real-world deployments of OUs, other than prototypes local to the university. Despite this, an application-neutral notification service, which ColdNotify aims to be, benefits from being able to provide notifications for most existing and future applications without relying on their data structures or data transfer methods. The small notifications (10s of bytes) are beneficial to OUs with limited connectivity, where low-bandwidth communication technologies such as LoRaWAN or NB-IoT are used.

Real-world deployments of OUs and new DAO applications may reveal additional challenges, requirements or edge-cases to ColdNotify, such as battery drain caused by TCP connections with application-layer acknowledgements, notification payloads, or the notification channel structure. This is similar to what the Thialfi team experienced, as limitations were revealed over time as it was adapted by various applications. Thialfi has over time made compromises by exposing a pub/sub API, because version-only notifications were not applicable for some applications[1].

As DAO evolves, edge-cases might appear where other types of infrastructure services are more suitable or may be used alongside ColdNotify, for instance pub/sub services or MQTT.

6.2 Compared to Thialfi

Directly comparing ColdNotify to Thialfi's results are difficult, as Thialfi's experiments are configured with large amounts of clients, different notification rates and a heartbeat rate, which is not part of ColdNotify.

When Thialfi goes from 1 000 notifications per second to 13 000 with 1,4 million clients, the CPU usage is increased by a factor of 5.5, whereas ColdNotify (registrar and matcher combined) has a factor of 10.98 with 1 000 clients.

In terms of notification latency, Thialfi's results show that median latency is stable at around 0,6-0,7 seconds, regardless of notification rate, with batching done by the bridge server. For ColdNotify on the other hand, without batching, notification latency for *13000* notifications per second is 18.4 ms on average and the median is 5.1 ms.

7 Conclusion

ColdNotify is an implementation of Google's notification service, Thialfi, adapted for DAO at UiT. Its design ensures that OUs are eventually notified, even if the notification service loses its state or is temporarily unable to reach offline OUs. Small notification messages make for a lighter wake-up for OUs, instead of being flooded with data, allowing them to schedule downloads for later.

Notification services eliminate the need for polling to relieve stress on back-end servers as the number of clients scale, and is being used in several distributed systems that face similar scaling challenges. As ColdNotify does not handle application-specific data, it is a general, application-neutral, notification service that can be used by distinct applications, which deprecates any per-application implementation to reliably keep their clients in up to date.

Experiments show that ColdNotify can recover within seconds and send out thousands of notifications per second with minor latency caused by the registrar and matcher due to their transposed state for a $\mathcal{O}(1)$ lookup of clients and objects respectively. The implementation of ColdNotify relies on open-source software and can be adapted for horizontal scaling and persistent memory in future work as the DAO project progresses.

This thesis is an early exploration of using notification services in DAO, as several aspects of the project is a work in progress and real-world deployments are yet to be made. The thesis can also be seen as the starting point for discussions on how notification systems can benefit DAO, which is important for making future decision to infrastructure, which is a critical part of DAO's goal to observe and measure arctic climates. Future work advises to revisit and explore new use cases and requirements as DAO deploys OUs and applications adapt ColdNotify.

8 Future Work

8.1 Horizontal Scaling

Experiments show that high CPU usage is caused by concurrent communication between clients, registrar and the matcher, which makes failure recovery expensive, especially for the registrar during registrar recovery. Support for horizontal scaling where clients are spread over multiple registrars will benefit recovery time and CPU usage per process. Multi-registrar deployments using reverse proxy to distribute loads with Nginx, Consul [13] and Registrator [15] in Docker has been experimented with, but left for future work (see Section A.6).

Scaling the matcher is slightly more complicated, as it requires state replication between each matcher node. Published notifications must be received by every matcher, such that the registrar can contact any matcher for registrations and unregistrations without keeping track of which matcher is responsible for which object. Thialfi uses state replication by best-effort RPC calls, which may be a viable solution for ColdNotify.

Figure 8.1 provides an model for how a horizontally scaled ColdNotify with persistent storage could look like.

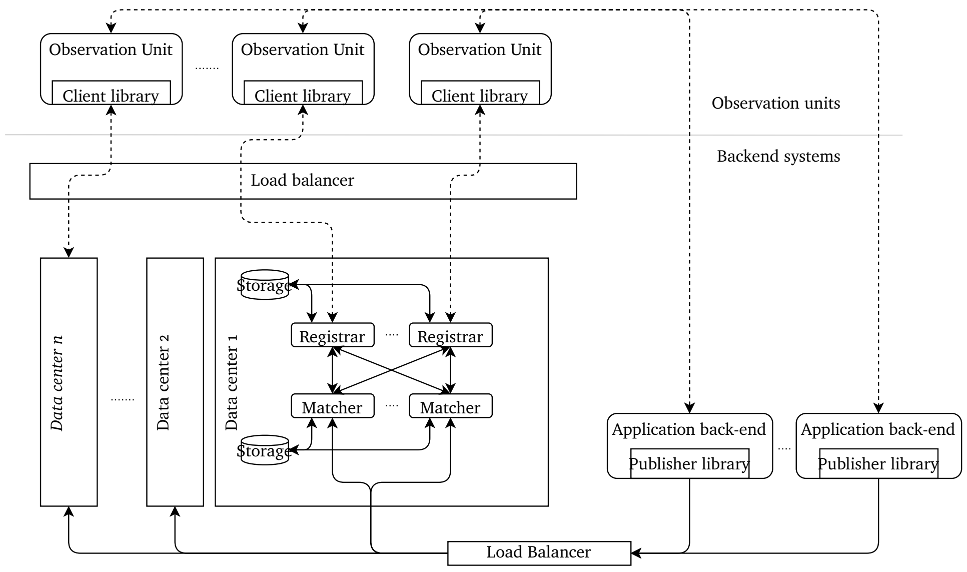


Figure 8.1: Illustration of a horizontally scaled ColdNotify, deployment with persistent storage.

8.2 Notification Payloads

ColdNotify remains a version-number-only notification service, but support for allowing small payloads have been discussed. As applications adapt ColdNotify, requirements for notification payload may appear, such that applications can execute remote-commands on OUs using notifications, include instructions for how to respond to a notification, or provide metadata about the changed data, such as size or expiration date.

As noted in Section 6.1, the Thialfi team subsequently added payloads via a pub/sub API for specific applications, which may become relevant for ColdNotify. MQTT may be used alongside ColdNotify to provide pub/sub for such applications.

8.3 Persistent Storage

Even thought the matcher and registrar can recover with 1 000 clients and 50 000 registrations in less than three seconds, having persistent storage will make recovery faster and is beneficial to application-specific notification payloads. While the Thialfi paper suggests HBase [3] as an open source alternative to BigTable [6], which Thialfi uses, other database systems or storage methods can be suitable for the scale of DAO.

The current implementation supports persistent storage by dumping the state as JSON to a file, which may be viable in small scales. Further work for persistent storage involves finding a suitable database system. DAO might over time settle on specific common database systems for all applications and services, which can be beneficial over having a separate database system that only serves ColdNotify.

8.4 Automatic Publishing

Currently, publishers must deliberately publish notifications to ColdNotify using a publisher library. Exploring alternative designs more similar to Wormhole, where ColdNotify observes data flows, can make publishing automatic and transparent to back-end applications as they write objects to databases.

Bibliography

[1] Atul Adya. "Lessons from an Internet-Scale Notification System." LADIS 2013. URL: http://2013.ladisworkshop.org/node/12.

[2] Atul Adya et al. "Thialfi: A Client Notification Service for Internet-scale Applications." In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. SOSP '11. Cascais, Portugal: ACM, 2011, pp. 129–142. ISBN: 978-1-4503-0977-6. DOI: 10.1145/2043556.2043570. URL: http://doi.acm.org/10.1145/2043556.2043570.

[3] *Apache HBase*. 2018. URL: https://hbase.apache.org (visited on 09/15/2018).

[4] S. Balasubramanian et al. "Auto-Tuned Publisher in a Pub/Sub System: Design and Performance Evaluation." In: *2018 IEEE International Conference on Autonomic Computing (ICAC)*. Sept. 2018, pp. 21–30. DOI: 10.1109/ICAC.2018.00012. URL: https://doi.org/10.1109/ICAC.2018.00012.

[5] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. "Achieving Scalability and Expressiveness in an Internet-scale Event Notification Service." In: *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*. PODC '00. Portland, Oregon, USA: ACM, 2000, pp. 219–227. ISBN: 1-58113-183-6. DOI: 10.1145/343477.343622. URL: http://doi.acm.org/10.1145/343477.343622.

[6] Fay Chang et al. "Bigtable: A Distributed Storage System for Structured Data." In: *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*. OSDI '06. Seattle, WA: USENIX Association, 2006, pp. 15–15. URL: http://dl.acm.org/citation.cfm?id=1267308.1267323.

[7] Dah-Ming Chiu and Raj Jain. "Analysis of the Increase and Decrease Algorithms for Congestion Avoidance in Computer Networks." In: *Comput. Netw. ISDN Syst.* 17.1 (June 1989), pp. 1–14. ISSN: 0169-7552. DOI: 10.1016/0169-7552(89)90019-6. URL: http://dx.doi.org/10.1016/0169-7552(89)90019-6.

[8] Idilio Drago et al. "Inside Dropbox: Understanding Personal Cloud Storage Services." In: *Proceedings of the 2012 Internet Measurement Conference*. IMC '12. Boston, Massachusetts, USA: ACM, 2012, pp. 481–494.

ISBN: 978-1-4503-1705-4. DOI: 10.1145/2398776.2398827. URL: http://doi.acm.org/10.1145/2398776.2398827.

[9] Eclipse. *Eclipse Mosquitto An open source MQTT broker*. 2018. URL: https://mosquitto.org (visited on 10/01/2018).

[10] Cloud Native Computing Foundation. *gRPC*. URL: https://grpc.io (visited on 05/05/2019).

[11] Andrew Gerrand. *JSON and Go*. Jan. 2011. URL: https://blog.golang.org/json-and-go (visited on 05/10/2019).

[12] Google. *Protocol Buffers*. 2018. URL: https://developers.google.com/protocol-buffers (visited on 11/10/2018).

[13] Hashicorp. *Consul*. URL: https://www.consul.io (visited on 03/10/2019).

[14] Rian Hunter. "Dropbox Notification Servers." In: *ACM Reflections | Projections 2012*. University of Illinois: ACM@UIUC, Oct. 2012. URL: https://www.youtube.com/watch?v=FBRIeoEr8GU.

[15] Glider Labs. *Registrator*. URL: https://github.com/gliderlabs/registrator (visited on 03/10/2019).

[16] OASIS. *MQTT*. 2019. URL: https://mqtt.org/ (visited on 10/01/2018).

[17] Ania M. Piotrowska et al. "AnNotify: A Private Notification Service." In: *Proceedings of the 2017 on Workshop on Privacy in the Electronic Society*. WPES '17. Dallas, Texas, USA: ACM, 2017, pp. 5–15. ISBN: 978-1-4503-5175-1. DOI: 10.1145/3139550.3139566. URL: http://doi.acm.org/10.1145/3139550.3139566.

[18] ReactiveX. *RxPY*. URL: https://github.com/ReactiveX/RxPY (visited on 05/08/2019).

[19] ReactiveX. *Subject*. URL: http://reactivex.io/documentation/subject.html (visited on 05/08/2019).

[20] Giampaolo Rodola. *psutil*. URL: https://github.com/giampaolo/psutil (visited on 04/26/2019).

[21] Yogeshwer Sharma et al. "Wormhole: Reliable Pub-Sub to Support Geo-replicated Internet Services." In: *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. Oakland, CA: USENIX Association, 2015, pp. 351–366. ISBN: 978-1-931971-218. URL: https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/sharma.

[22] University of Tromsø - The Arctic University of Norway. *Distributed Arctic Observatory (DAO)*. URL: https://site.uit.no/dao (visited on 01/20/2019).

A User Manual

This appendix describes how to set up and run ColdNotify locally with demo application.

A.1 Prerequisites

A.1.1 Software

- Go with $GOPATH variable for compiling Go applications
  `https://golang.org`
- Certstrap for generating SSL files
  `https://github.com/square/certstrap`
- Python 3.7 with pip
  `https://www.python.org`
- Pipenv (recommended for installing client dependencies)
  `https://github.com/pypa/pipenv`

Repository	**Directory**
Service	~/coldnotify/service
Client library	~/coldnotify/client
Demo client	~/coldnotify/demo
Registrar	$GOPATH/src/vvgitlab.cs.uit.no/coldnotify/registrar
Matcher	$GOPATH/src/vvgitlab.cs.uit.no/coldnotify/matcher
Publisher library	$GOPATH/src/vvgitlab.cs.uit.no/coldnotify/publisher
Demo server	$GOPATH/src/vvgitlab.cs.uit.no/coldnotify/demoserver

Table A.1: ColdNotify's *Service* repository contains scripts for setting up ColdNotify, which assumes these directories.

A.1.2 Directory structure

Scripts for generating SSL certificates and installing Go applications are located in the *service* directory, which assumes a directory structure described in Table A.1.

A.2 Installation

```
# Go to service directory
$ cd ~/coldnotify/service

# Install SSL certificates (no passwords)
$ sh ./ssl.sh

# Install registrar, matcher and demo server to $GOPATH/bin
$ sh ./install.sh
```

Listing A.1: Installing ColdNotify components and ssl certificates

ssl.sh generates SSL files using certstrap to ~/coldnotify/service/ssl and copies them to each directory. install.sh installs Go applications to $GOPATH/bin, which should be added to $PATH.

A.3 Usage with demo application

Now that ColdNotify components are installed, commands in Listing A.2 shows how to run ColdNotify, the demo server and a demo client. Default server ports are listed in Table A.2.

Server	Default Port
Registrar TCP	55000
Registrar RPC	55001
Registrar HTTPS	55002
Matcher RPC	51000
Matcher HTTPS	51001
Demo server HTTPS	3000

Table A.2: Default port numbers

```
# Registrar, matcher and demo server assumes a ./ssl directory
with certificates
$ cd ~/coldnotify/service

# Running the registrar with persistent state
$ registrar -matcher localhost:51000 -persistent

# Running the matcher with persistent state
$ matcher -registrar localhost:55001 -persistent

# Running the demo server
$ demoserver -matcher localhost:51000


# Run demo clients from the demo directory
 $ cd ~/coldnotify/demo

# With Pipenv:
 $ pipenv run python . --registrar localhost --registrar-port
55000 --server localhost:3000

```

Listing A.2: ColdNotify usage with default ports

Use the -h flag to see each program's usage and how they can be configured.

A.3.1 Publish notifications

The demo server accepts POST requests to publish notifications. Curl example is given in Listing A.3.

```
    # https://localhost:3000/notify/{group}/{object}/{n}/{rate}
    # Publish 10 notifications at a rate of 1 per second
    $ curl -k -X POST https://localhost:3000/notify/group-1/object
    -1/10/1
```

Listing A.3: Publish notifications with POST requests to the demo server with curl

A.4 View application state

```
# Registrar
$ curl -k https://localhost:55002

# Matcher
$ curl -k https://localhost:51001

# Demo server
$ curl -k https://localhost:3000
```

Listing A.4: View registrar, matcher and demo state with curl

The registrar, matcher and demo server will return their current state in JSON format if a GET request is sent to their HTTP server root path (Listing A.4). When using the –persistant flag for matcher and registrar, the state is saved to *matcher.json* and *registrar.json* respectively.

A.5 Client library usage

```python
from notify import Notify, State, Connection

notify = Notify(
    domain='demo.dao',
    group_id='group-1',
    client_id='client-1',
    state=State(filename='registration-state.json')
    connection=Connection(
        address='localhost',
        port=55000,
        ca_cert='ssl/coldnotify.crt',
        cert='ssl/client.crt',
        key='ssl/client.key'
    )
)

notify.connect()

notify.register('object-1').subscribe(
    lambda notification: print(notification)
)

notify.register('object-2').subscribe(
    lambda notification: print(notification)
)

notify.unregister('object-1')

notify.disconnect()
```

Listing A.5: Client library usage

A.5.1 Publisher library usage

```go
import "vvgitlab.cs.uit.no/coldnotify/publisher"

// Publisher service connects to matcher server
publisher.Service = publisher.NewService(
    "localhost:51000"
)

domain := "demo.dao"
group := "group-1"
objectId := "object-1"
version := "123456789"
source := ""

err := publisher.Publish(
    domain, group, objectId, version, source
)
if err != nil {
    // Application-specific error handling
    handlePublishError()
}
```

Listing A.6: Publisher library usage

A.6 Notes

The *service* directory contains several configuration files for deployment experiments, such as docker-compose for Docker, Nginx and HAProxy for reverse proxies, Consul and Dockerfiles for creating Docker images. Potential future work can build on these configurations to deploy and horizontally scale ColdNotify.

/B
Experiments

Python scripts and raw data from all experiments are located in service/experiments.

B.1 Tools

A Python script for starting several clients are located in service/tools/spawn.py, which assumes the directory structure described in Table A.1.

service/tools/ utilization .py takes a process id (pid) as argument −−pid and is used to measure CPU and memory usage of the registrar and matcher processes. When measuring from process startup, for instance in failure recovery, the registrar and matcher should be run with the −delayed flag to get the pid before continuing by pressing enter.

B.2 Node analysis: latency comparison

As described in 5.1, experiments are run on a cluster and clients are distributed across nodes. However, these nodes use different hardware which could lead to clock skew, and some resources may have been used by other users, which can affect the latency. This appendix provides further analysis for latencies from the experiment in Section 5.2.

Table B.1 a full list of all 84 nodes, with latency samples, mean and standard deviation. Each node has either 11 or 12 client processes. The mean latency varies from 16.5061 to 93.8596 ms, which makes a difference of 77.3535 ms.

While I could have selected nodes with similar mean latencies when distributing clients, or accounted for their difference by weighing the results, I've decided to leave it as is. The purpose of the notification latency experiment is to show that latency increases with the notification rate, which will happen regardless of varying mean latencies.

nodes	**samples**	**mean**	**std**	**mean diff**	**std diff**
all	*909000*	*41.7885*	*89.5064*	–	–
compute-2-11	10908	16.5061	43.9771	-25.2824	-45.5293
compute-1-0	10908	16.8841	54.3984	-24.9044	-35.108
compute-2-4	10908	17.9078	45.3958	-23.8807	-44.1106
compute-0-10	10908	19.4775	43.6221	-22.311	-45.8843
compute-2-25	10908	19.5442	49.4574	-22.2443	-40.049
compute-1-2	10908	20.5416	44.1103	-21.2469	-45.3961
compute-2-2	10908	22.7073	57.0382	-19.0812	-32.4682
compute-1-3	10908	23.0788	47.1773	-18.7097	-42.3291
compute-2-16	10908	23.4845	53.61	-18.304	-35.8964
compute-2-15	10908	23.6292	68.8781	-18.1593	-20.6283
compute-0-5	10908	23.6693	61.593	-18.1192	-27.9134
compute-0-14	10908	24.5036	58.1932	-17.2849	-31.3132
compute-0-16	10908	24.8976	68.056	-16.8909	-21.4504
compute-3-16	10908	25.7823	52.9117	-16.0062	-36.5947
compute-4-0	9999	25.8744	72.8868	-15.9141	-16.6196
compute-2-20	10908	26.1269	60.7071	-15.6616	-28.7993
compute-4-7	9999	26.6025	72.423	-15.186	-17.0834
compute-1-8	10908	27.0414	63.2928	-14.7471	-26.2136
compute-2-29	10908	27.165	75.5463	-14.6235	-13.9601
compute-1-13	10908	27.5178	64.3998	-14.2707	-25.1066
compute-0-13	10908	28.0464	73.4001	-13.7421	-16.1063
compute-2-27	10908	28.087	64.3927	-13.7015	-25.1137
compute-2-22	10908	29.0173	80.8961	-12.7712	-8.6103
compute-1-6	10908	29.8243	67.7261	-11.9642	-21.7803
compute-2-13	10908	29.8476	76.1949	-11.9409	-13.3115
compute-2-18	10908	29.9574	63.2939	-11.8311	-26.2125
compute-1-4	10908	30.2715	55.5916	-11.517	-33.9148
compute-0-17	10908	30.7156	79.2593	-11.0729	-10.2471

compute-0-19	10908	30.8516	72.851	-10.9369	-16.6554
compute-3-17	10908	31.5037	69.5971	-10.2848	-19.9093
compute-3-3	10908	33.4858	75.6081	-8.3027	-13.8983
compute-2-1	10908	33.6759	58.9069	-8.1126	-30.5995
compute-3-5	10908	33.7741	61.7439	-8.0144	-27.7625
compute-1-7	10908	34.1233	79.5197	-7.6652	-9.9867
compute-0-11	10908	34.3494	63.7026	-7.4391	-25.8038
compute-1-11	10908	35.7925	72.7198	-5.996	-16.7866
compute-2-5	10908	36.1079	61.2947	-5.6806	-28.2117
compute-2-8	10908	36.4116	78.0112	-5.3769	-11.4952
compute-2-24	10908	36.4261	61.3333	-5.3624	-28.1731
compute-3-12	10908	36.8006	78.6013	-4.9879	-10.9051
compute-2-9	10908	37.3541	73.8244	-4.4344	-15.682
compute-2-3	10908	37.5826	82.1783	-4.2059	-7.3281
compute-0-9	10908	37.9536	68.1201	-3.8349	-21.3863
compute-3-8	10908	38.3192	72.3996	-3.4693	-17.1068
compute-2-21	10908	38.3457	61.7817	-3.4428	-27.7247
compute-4-2	9999	38.616	79.995	-3.1725	-9.5114
compute-2-26	10908	38.6955	92.9784	-3.093	3.472
compute-3-6	10908	38.8825	65.2931	-2.906	-24.2133
compute-4-6	9999	39.3987	99.3588	-2.3898	9.8524
compute-0-15	10908	40.2257	75.668	-1.5628	-13.8384
compute-2-23	10908	40.7126	74.6248	-1.0759	-14.8816
compute-4-3	9999	40.7829	74.0251	-1.0056	-15.4813
compute-4-5	9999	41.3784	99.7846	-0.4101	10.2782
compute-4-4	9999	41.5085	69.8404	-0.28	-19.666
compute-3-11	10908	42.3631	109.0728	0.5746	19.5664
compute-0-3	10908	43.0743	81.3351	1.2858	-8.1713
compute-0-18	10908	44.0458	65.8095	2.2573	-23.6969
compute-2-0	10908	44.0898	82.5228	2.3013	-6.9836
compute-2-28	10908	45.1225	86.2896	3.334	-3.2168
compute-3-0	10908	45.5671	90.2499	3.7786	0.7435
compute-2-12	10908	48.9945	85.8791	7.206	-3.6273
compute-0-12	10908	49.2746	90.3613	7.4861	0.8549
compute-3-4	10908	49.3857	93.0677	7.5972	3.5613
compute-3-13	10908	51.2845	104.3961	9.496	14.8897
compute-4-1	9999	51.595	93.2518	9.8065	3.7454
compute-2-10	10908	54.8355	86.5318	13.047	-2.9746
compute-2-19	10908	56.0881	101.2795	14.2996	11.7731
compute-1-9	10908	56.5069	89.3606	14.7184	-0.1458
compute-1-10	10908	57.196	170.8147	15.4075	81.3083

compute-3-10	10908	59.107	85.8713	17.3185	-3.6351
compute-2-14	10908	61.8954	99.7928	20.1069	10.2864
compute-0-4	10908	62.1732	172.5843	20.3847	83.0779
compute-2-7	10908	66.269	100.2271	24.4805	10.7207
compute-2-6	10908	66.3355	93.0805	24.547	3.5741
compute-3-1	10908	67.2957	119.8354	25.5072	30.329
compute-1-1	10908	67.739	128.5559	25.9505	39.0495
compute-3-15	10908	68.841	114.9061	27.0525	25.3997
compute-3-14	10908	72.0003	142.3691	30.2118	52.8627
compute-2-17	10908	74.8515	114.8214	33.063	25.315
compute-3-9	10908	81.7678	120.8075	39.9793	31.3011
compute-3-2	10908	84.775	120.5013	42.9865	30.9949
compute-3-7	10908	86.9977	140.477	45.2092	50.9706
compute-1-5	10908	92.7536	172.7505	50.9651	83.2441
compute-1-12	10908	93.8596	156.5089	52.0711	67.0025

Table B.1: Notification latency statistics for 25 notifications per second, reported by 1 000 clients distributed over 84 nodes. Mean and standard deviation (std) are given in milliseconds, sorted by mean latency.