Modern XMPP

A story based on Monal

Thilo Molitor (& Friedrich Altheide)

13.04.2022

Motivation

- Decentral
- Federated
- Secure
- Open Source
- iOS/macOS (!)

$\Rightarrow$ Just another Chat client

Requirements

Functional Requirements

- *Instant-Messaging*
- *Support Push in 1:1 and MUCs*
  - *Support OMEMO even with push*
  - *Only use already standardized and widely deployed XEPs*
- *Support for sharing images, videos etc. (even if "offline")*
- *Tolerate slow connections (32kbit/s)*

State of the art in iOS/Android

- Push notification support limited
  - No push notifications
  - Anoying message that the app is kept open
  - “New Message” notifications
  - Funky implementations that only work with specific XMPP servers

A modern XMPP approach

- Only use already standardized and widely deployed XEPs
- Support default prosody and ejabberd installs (e.g. only XEP-0357)
- Minimize needed traffic for syncs / faster syncs (XEP-0198)
- Use available background modes of iOS/Android for better UX

App decomposition in iOS

- Main App
  → Opened by user, started by some background modes
- App Extension: Notification Service Extension
  → Started in background by incoming push (max. 30 seconds)
- App Extension: Share Extension
  → Started by share action of user
- More App Extensions not used in Monal (e.g. VPN, ...)

App decomposition in iOS - Main App

- This is the main app process containing all UI parts and everything else needed to run the app
- Can be killed by swiping app away (only ~1-2 seconds time to clean up)
- Only limited runtime in background (if not swiped away):
  - 30 seconds after user closed the app (NOT swiped away!)
  - 30 seconds for incoming low priority pushes (nowadays even started in BG)
  - Up to 5 minutes if started via modern BGProcessingTask

App decomposition in iOS - Notification Service Extension

- Distinct process, started by incoming alert-type push notification
- Can modify incoming push notifications before they get delivered to the user
- Only allowed to run for 30 seconds per incoming push
- Only limited memory allowed (~30-35MiB)
- Multiple incoming pushes are serialized in iOS>=14, only sometimes in macOS
- Must deliver the (modified) notification content in this 30 second timeframe:
  - Original notification content as delivered by the push server are displayed on timeout
  - Notification can only be silenced completely (e.g. not displayed) with special entitlement granted by Apple (com.apple.developer.usernotifications.filtering)
  - Crashes of this extension deliver the original unmodified notification to the user (e.g. "New Message")

App decomposition in iOS - Share Extension

- Distinct process, started by user interaction
- Provides the UI for share actions and preforms them
- Can not run in background at all
- Fortunately able to open apps (even “own” app)

Battery Savings

On iOS processes are not allowed to run in background for a longer time and Apple can kill or freeze your process for various reasons (real app crashes being only one of them)

- **FG-BG** After background time expired, your app gets frozen (disconnecting all network connections)
- **Crash** Obviously your app does not run anymore after this
- **Device Shutdown** or **App swiped away** Your process gets killed (~1-2 seconds time to clean up)
- **Low Memory or other Apple Stuff** Your app can be killed while frozen and or freezed/killed faster

→ **Solution:** Save state, save often

Managing state is not easy

Problem:

- Various app processes can be interrupted at any time
- What about stanzas that are in flight or even already received but not processed yet?
- What about stanzas the app is waiting for (IQ responses, incoming MAM stanzas etc.)?
- How can state be migrated between the various processes?

Solution:

→ Serialize state to DB
  - Make all classes containing state serializable
  - Provide state setters and getters in non-serializable classes

→ Use (callback) handlers that can be serialized to DB

→ Use DB transactions

Managing state is not easy - DB transactions 1/3

- Sounds easy, right? Well, not really :(
- What to put into one transaction?
- How to recover from app/process freezes in the middle of a transaction?
- How to recover from failed transactions (we don't want to loose messages!)
- Can we be sure transactions don't run too long?

Managing state is not easy - DB transactions 2/3

- Use Apple callbacks to disconnect the TCP connection and make sure we are idle before we get freezed
  - This makes sure no transaction gets interrupted by the app/process freeze
  - Pause handling of already received stanzas (throw them away)
  - Special handling in NSE, explained later

Managing state is not easy - DB transactions 3/3

- Use one single transaction for every incoming stanza
  - All state changes associated with the incoming stanza in one single transaction
  - This includes the complete internal state of the app(!) (XEP-0198 queues etc.)
  - Use XEP-0198: Stream Management for stanza replay on transaction rollbacks
  - Use XEP-0313: Message Archive Management if Stream Management can not be used
  - Make sure stanza processing doesn't take too long (OMEMO etc.)
- Use one single transaction for every outgoing stanza, again includes the complete internal state of the app(!)
- Pack everything else into well chosen transactions having neither a too wide nor a too narrow scope

Managing state is not easy - Handlers overview

- No “eval” in ObjC (or Swift)
- Simple generalized concept usable throughout the app
- Leverages dynamic language features of ObjC
- Serializable callbacks to class/instance methods of classes
- Bind values (not vars) when creating a handler (e.g. to bind state to handler)
- Bind vars when calling handler

Managing state is not easy - Handler examples 1/4

Define handler method (“static” class method):

```
1 $$class_handler(handleCarbonsEnabled, $_ID(xmpp*, account), $_ID(XMPPIQ*,
        iqNode))
2
3     if([iqNode check:@"/<type=error>"]) {
4         // <snip> report error </snip>
5         return;
6     }
7     account.connectionProperties.usingCarbons2 = YES;
8 $$
```

Managing state is not easy - Handler examples 2/4

Register response/error handler when sending out IQ stanza:

```
1 XMPPIQ* carbons = [[XMPPIQ alloc] initWithType:kiqSetType];
2 [carbons addChildNode:[[MLXMLNode alloc] initWithElement:@"enable"
        andNamespace:@"urn:xmpp:carbons:2"]];
3 [account sendIq:carbons withHandler:$newHandler(self, handleCarbonsEnabled)];
```

Managing state is not easy - Handler examples 3/4

Call registered IQ Handler:

```
1 MLHandler* handler = /* <snip> get handler based on IQ id </snip> */;
2 $call(handler, $ID(account, self), $ID(iqNode));
```

Managing state is not easy - Handler examples 4/4

Define handler method (instance method):

```
1 $$instance_handler(handleMamResponseWithLatestId, account.mucProcessor, $_ID(
        xmpp*, account), $_ID(XMPPIQ*, iqNode))
2
3     NSString* stanzaId = [iqNode findFirst:@"{urn:xmpp:mam:2}fin/{http://
          jabber.org/protocol/rsm}set/last#"];
4     DDLogVerbose(@"Got latest muc stanza id to prime database with: %@",
          stanzaId);
5
6     [[DataLayer sharedInstance] setLastStanzaId:stanzaId forMuc:iqNode.
          fromUser andAccount:self->_account.accountNo];
7     [self->_account mamFinishedFor:iqNode.fromUser];
8 $$
```

Digression - Push modes in iOS: Silent Push

Silent Push

→ Delivered to Main App process

→ Even starts the process if not running/freezed in iOS >= 13

→ Not reliable, can be throttled by Apple or not delivered at all

→ Grants only 30 seconds of background time to the Main App process

Digression - Push modes in iOS: Alert-Type Push

Alert-Type Push

- → Delivered to NSE (if provided in app bundle), starts/unfreezes it
- → Does not start the Main App process
- → Reliable, every push reaches the device as soon as possible (but can be delayed somewhat in power saving modes)
- → Allows for modification or surpression of notification (depending on granted entitlements)
- → Grants only 30 seconds of background time to the NSE process

Digression - Push modes in iOS: VoIP Push

VoIP Push

→ Delivered to Main App process

→ Always starts the process, if not running/freezed, even in iOS < 13

→ Must display an "incoming call" screen to user in iOS $>= 13$

Support propper push in Monal 1/4

→ Use Alert-type pushes delivered to the NSE (recap: different process than Main App!)

- Both processes use the same resources (DB, XMPP-Stream etc.)

→ Share DB (including state of whole app)

Support propper push in Monal 2/4

- Multiple processes simultaneously changing the state and/or connecting to the same XMPP-Server resuming the same XEP-0198 session? BAD!

→ Implement locking mechanism to only allow NSE to start if Main App is not running (and vice versa)

  - Needs an IPC mechanism to tell the NSE to stop if the Main App wants to start
  - Needs some locking mechanism (implemented in Monal through the same IPC mechanism)
  - Builtin IPC mechanisms in iOS can only send simple messages without any data

  → Implement own IPC mechanism based on this simple messages and an SQLite DB

Support propper push in Monal 3/4

- Check if Main App is running and exit, if so (check twice!)
- Use state from DB to resume XEP-0198 session (or do MAM)
- Handle every incoming stanza (even IQs)
- Use a timer to disconnect before our Apple-granted background time ends
- Use our entitlement to silence the incoming push notification
- Kill NSE process (clean state on next start)

Support propper push in Monal 4/4

- What about these serialized pushes?
- Don't disconnect when feeding push handler, but freeze handling of incoming stanzas
- Wait some time (1,5s) for new incoming push
- If a new push comes in, unfreeze stanza handling and proceed
- If no new push comes in, disconnect and kill NSE

iOS Background Modes revisited

- Limited resources (time, memory)
- What if time runs out?

→ Apple suggests to show a notification to the user telling him to reopen the app to finish synchronizing (WWDC talk)

- What if user has no/bad connectivity and/or swipes the app away before a messages could be sent?

→ Not only show a notificaton but try to start a BGProcessingTask

→ This task can be started anywhere between ~20 minutes to multiple hours in the future depending on Apple KI foo

Share Sheet

- Should it access the DB as well? Third process - bad!

→ Write message to special dir/DB table and open the Main App afterwards

→ The Main App manages the actual sending of our shared message

Lessons learned

- Mobile platforms (especially iOS) are really constrained
- But: Managing a full XMPP stream/session and even handling IQs is possible

→ In the end the mobile client behaves just like a traditional desktop client (as seen from other clients)

→ We don't even need new XEPs to accomplish this!

Outlook

While the XMPP backend of Monal is now in a very good shape, the UI ist not

- Some parts in the UI are still missing (create/manage groups etc.)
- Accessibility must be greatly improved
- SwiftUI is easy to maintain

→ We need a SwiftUI developer to improve our UI

→ (build missing UI parts, rewrite existing UI)

Privacy Friendly Push Design (Ongoing Project)

Motivation

- Scalable
- Available
- OpenSource
- Privacy Friendly

Requirements

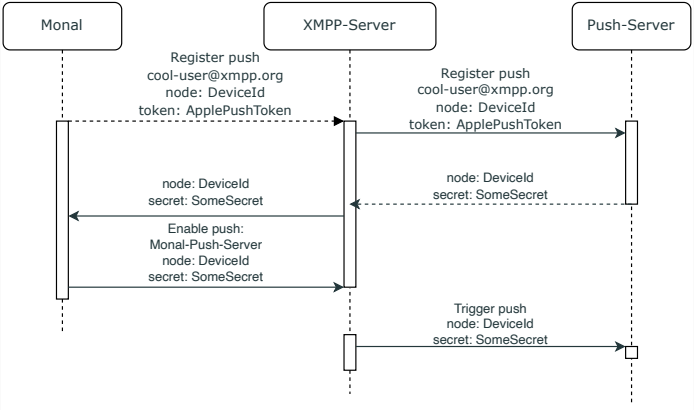
Functional Requirements

- *Should work together with XEP-0357*
- *Privacy friendly (minimal knowledge)*
- *Follow apple's push guidelines*
  - *Ratelimit pushes (save battery)*
  - *Don't repeatedly send requests to unknown devices (save resources)*

Non-functional Requirements

- *Availability*
- *Scalability*

State of the art 1/2



- Monal registers at appserver (Monal push server) with apples push token receiving a secret
- Monal enables push on the configured XMPP servers using the secret
- XMPP server triggers push at our push server using the supplied secret

State of the art 2/2

- Ignore invalid tokens
  - Blacklist tokens that apple reported to be invalid
  - Ignore pushes for tokens that are blacklisted
- Ratelimit pushes for a token
  - If no push was recently send, trigger a push
  - If a push was just triggered, queue another push in 20 seconds
  - If a there is already a queued push, discard the push request

$\rightarrow$ Working push server following apples guidelines

$\rightarrow$ What about the privacy?

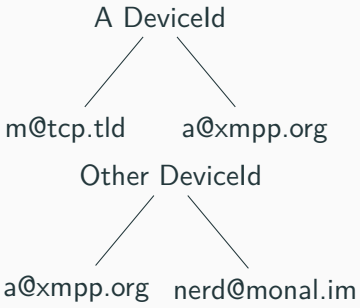
State of the art - Privacy

- On each registration we see and save
  - JID (username@domain.tld)
  - Device Id
  - Apple push token unique to the device
  - Timestamp
- For each push we see
  - The xmpp server domain
  - Our supplied push secret (Direct association with JID)
  - Timestamp

→ Account information of the same device can be joined. We know which JIDs you are using!

⇒ We know every new JID that you used. Even across devices

⇒ We don't want to know your JIDs. We need a privacy friendly solution!

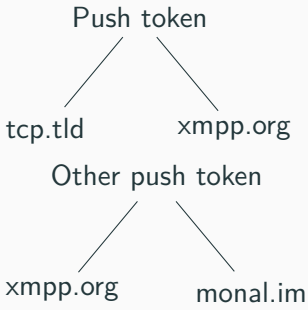


Privacy enhanced push design

- Monal registers apples push token directly into his xmpp servers
- No registration and secret at our push server needed
- We only ever see tuple consisting of a domain and apples push token
- We can only see that a device token is used on one or more domains

→ No more tracking across device IDs

- Not perfect if a domain is used by a limited number of users
- XEP compatible, no new XEP or server components needed



Recap Requirements but this time the appserver

- Ratelimit pushes for a single push token
- Ratelimit new device tokens from a domain, after too many failed pushes
- Blacklist known invalid device tokens for a few hours
- Improve availability and scalability

Push server design (single server)

1. XMPP-Component written in Rust, attachable to ejabberd and prosody
2. For each received iq, create new lightweight tokio (async runtime) thread
3. Check if domain is blacklisted, due to too many bogus requests
4. Check if token is blacklisted, and extend blocking if it is
5. Ratelimit devices pushes
6. Send push
   - Block device tokens that were reported invalid by apple
   - Block domains with too many invalid tokens in a set timeframe

→ How can we scale it to multiple servers?

Push server design (multi server)

- Same idea as a single server
  - Deploy on multiple servers accessible under the same domain (A / AAAA)
  - Use SRV records for improved steering
- Blacklists
  - Sync blacklist using an eventual consistency across all push servers
- Ratelimits
  - Only useful if a device is registered on multiple domains
  - Even eventual consistency would be too expensive
  - → Ratelimit per push server

→ Minimal sync between push servers

Summary

- Privacy-friendly push while being XEP-357 compliant
  - We will no longer see your usernames
- Design for a scalable and available appserver, with minimal sync between the nodes
  - Rust based push server component
  - Only syncing blacklist between the nodes based on eventually consistency

Outlook

- Implement new push design and appserver for apple APNS
- Test and deploy our new design
  1. Single node
  2. Multi node setup without synced blacklists
  3. Multi node setup with blacklist sync
- Extend appserver to support google FCM, ...
- Hopefully migrate other android XMPP APPs to our privacy-friendly push implementation