Tox4j High Level API

Objective

What are we doing? Why are we doing this? What are the goals? What are NON-goals? This section explains the project to someone not familiar with tox4j.

HLAPI aims to provide a high-level API that is directly suited for implementing a Tox client, meaning that it implements all needed platform-independent parts (friends list, file transfers, …) on top of the low-level toxcore bindings. Platform-specific operation that are specifically not implemented include storage and UI, for instance.

This will provide a common base for various Tox clients, increasing code-sharing in the ecosystem and making it easier to implement new features/protocol extensions across all existing clients.

This will *not* be a Tox client (beyond the needs of testing, and perhaps a simple demo), nor will this provide any UI. This does not replace toxcore (it is only a wrapper around it).

Moreover, it is not in scope (for GSoC) to have features specific to multi-account. However, multi-account is supported in the sense that the lack of global state (see Design principles) makes it possible to run several instances of HLAPI in the same process.

Typographic conventions

Throughout the design document, a consistent typesetting convention must be followed. This convention is defined here.

BoldCamelCase	Type names (class, trait, type, object)
ALL_CAPS	Numeric constants found in **ToxConstants**
italics	Method name
underlined	Emphasis

Background

Related work in the tox project? What similar projects exist? Links to external documents/wikipedia? Nothing about design/requirements here, just background.

There are several Tox clients all implementing more or less the same logic such as chat logs, profiles, and friend lists. This leads to fragmentation and duplication of effort.

Antox is an Android Tox client which uses the low level Tox4j API. The high level interactions are in a tightly coupled set of Android-specific implementation classes and UI classes. This tight coupling of UI and logic makes the client difficult to maintain.
A related project is the modular high level multimedia library, which provides a uniform API over various platform-dependent audio/video/desktop capture libraries. Although we do not aim to provide any platform-dependent implementations, the multimedia library's design can help us in the design of our A/V interface.

Functional programming

The design of HLAPI is done in a functional style; in particular, we leverage existing concepts, such as **Monad**s and **Lens**es, to model some of the components of HLAPI. To that end, we reuse typeclasses defined in scalaz.
This has the dual benefit of reusing a familiar interface (instead of rolling our own **Monad** trait) and giving us testable invariants for free, which are a very good fit for property-based testing.

The **Monad** that HLAPI relies the most on is **State[ToxState, _]**: state monads are very effective at describing programs in which some immutable state is modified and modified copies are passed around.

Lenses model a "location" in a datastructure. They can be thought of as a pair of a getter and a setter that returns a modified copy of the datastructure. The point of the **Lens** abstraction is that it gives better composability.
See http://eed3si9n.com/learning-scalaz/Lens.html and http://www.monadzoo.com/blog/2012/11/18/using-lenses-with-scalaz-7/.

Requirements

Who are the customers for the solution? What are their needs? What is the problem space? This section estimates scale requirements. How much data needs to be stored/processed? What kind of data? What about latency/throughput/etc network requirements? How about growth?

Customers

We are building a new Android client which uses HLAPI as its business logic and data storage layer. Antox is also planning to adopt HLAPI.
Part of this design is a high level network protocol built on top of Tox custom packets. This network protocol and any application protocols built on it need to be fully specified to enable alternative implementations.

Mobile-specific requirements

- Battery-friendliness
- Support high-latency/jitter links and not-yet-ack'd messages
- Automatic retransmit of lost messages

Scale

For estimating the scaling parameters, we consider three distinct use cases. The first two are desktop and mobile, which are similar in many ways, since they are both operated by humans. The third use case is a bot.

Bot

For the bot, we maximise all estimates, giving us an a minimum upper bound of what we can hope to support. All maxima are calculated assuming no interactions occur besides the one the calculation is for. Since maxima are heavily dependent upon external parameters, we assume the bot runs in the following environment:

Assumptions

Network	1 Gbit/sec full duplex
RAM	64 GiB
CPU	Intel Xeon E5-1650 @ 3.20GHz (6 physical cores with HT), 12MiB cache

Initial numbers

Transfer rate	1Gbit/s = 1000Mbit/s = 125MB/s = 119.20MiB/s
Protocol overhead	42B Ethernet[1] + 20B IP + 8B UDP + 28B Tox = 98 bytes/frame
Status update	1B
Location update	8B * 7 = 56B
File packet	1372B
Status updates	1/friend/minute

Derived maxima

Status packet rate	125MB/s / 99B/packet = 1.26M packets/s
Location packet rate	125MB/s / 154B/packet = 811.7K packets/s
File packet rate	125MB/s / 1470B/packet = 85K packets/s
File data rate	85K packets/s * 1372B/packet = 116MB/s = 111MiB/s
Number of friends	1.28M packets/s * (1/friend/minute) * 60 s/minute = 75.8M friends
Number of groups	75.8M peers / (1000s peer/group) = 100k groups

We use the geolocation update as an example to compute the maximum number of messages per second, assuming a single friend and no other interactions.
The numbers for status and location updates are slightly wrong, because the Tox protocol may add up to 7 bytes padding to conceal the precise packet size. This means the numbers presented here are slightly higher than what the actual network can support, but this is not an issue, since our implementation should at least support these numbers.

Desktop & mobile

For desktop and mobile, we used a modified pisg to collect statistics from real IM and IRC users who ran the script on their chat logs. The statistics the script collects are:

- IM / private chats:

[1] Headers and inter-packet gap; minimum Ethernet payload size is another 42B and our payload is more than that.

    - Average frequency of messages
    - Average & median size of a message
    - Number of peers
- IRC / group chats:
    - Total number of visited channels
    - Average frequency of messages in those channels
    - Average & median size of a message
    - Average (over time) number of active users in the channel
      Need a spec for “active” users; proposal: a user is active for X minutes after sending a message (or action); sending a new message does not accumulate time, but resets the counter.

	Desktop	**Mobile**	**Bot**
Friend list			
Number of friends	~10-100		75M
Status update freq	~1/min/friend		
Text messages			
In-flight messages[2]	1-10	10s	1000s
Frequency (mean)	1/s		800K/s
Frequency (peak)	10/s		800K/s
Message length	<100B avg., <10k usually		56B
File transfers			
Size	unbounded	5GiB	unbounded
Bandwidth	110MiB/s	10MiB/s	111MiB/s
Queue[3]	10s		memory-bound
Group chats			
Groups	100s		~100k
Peers per group	1000s		

[2] Send function returned, and the recipient has not acknowledged reception yet.
[3] The number of scheduled transfers. A transfer is a bunch of files (or directories) that were scheduled together.

Format

Some data formats are human-readable and highly compressible (those properties are usually correlated). This is especially true for natural text, and some other file formats. The HLAPI wire format should take advantage of this whenever possible.

Access patterns

- Log: Message recall/editing => append and mutate tail
  Access the bottom and full-text search
- Friend list: infrequent modifications, random access, add/remove

Confidentiality

- Fairly different for large, public[4] group chats and private conversations
  Scale constraints are also quite different in those 2 cases

Reliability

- Data corruption
  - bitrot (silent data corruption in the storage layer)
  - power failure during write
  - both seem to require cooperation from the client
- Crashes
  - Should be mostly prevented by the use of safe language features (i.e. no exceptions, no casts)
  - Can occur in the JNI binding or in toxcore itself
    - A toxcore crash is not survivable (assert failures in the JNI bindings)

Design Ideas

Overview of the design. If you have multiple viable ideas, list them all with pros and cons. Do not include code, only type signatures and explanations. Use diagrams if necessary. Major structural elements go here. Which existing technologies will be used? Which components will you write? How do they integrate? How will others integrate with them? What scaling parameters need to be considered most prominently? How will the product be rolled out to users? Implementation strategies go here, implementation does not.

Design principles

Referential transparency

Almost all library functions should be referentially transparent. Referential transparency simplifies the interactions inside the library, as the observable behaviour of a function only depends on its parameters. Property-based testing using random inputs also becomes simpler, since the state each function operates on is small and independent.

[4] The private character of a group chat cannot be easily determined, so all logs should be treated as private data.

We achieve this by operating only on immutable data structures. Moreover, functions operating on immutable data structures are trivially thread-safe and reentrant, which is desirable for a library meant to be used by UI-driven applications.

Storage format independence

We should be independent of any serialisation format and provide:

- data types that accurately describe the objects being handled;
- elementary operations defined directly on those data types;
- higher level operations defined in terms of the elementary ones;
- default (de)serialisation methods (again, outputting streams rather than file-based I/O) and utility classes that turn stream I/O into file-backed I/O.

This seems important, as it enables the client to:

- inspect objects (contacts, chat logs, ...) more easily;
- pick a format adapted to the storage method, so it can, for instance, avoid storing opaque JSON in a database.

Also, this seem to be good design as it enforces separation of concerns inside the library.

Mobile-friendliness

Mobile devices and other embedded platform have tighter restrictions. Since one of our major use-cases is a mobile app, we require the design to

- have ways of notifying the UI about message acknowledgement; while not mobile-specific, packet loss, latency and jitter on mobile networks exacerbates the need to provide the UI with this information, and polling (another way of providing that information) is both cumbersome and inefficient;
- avoid unnecessary wakeups and batch network transmissions to be more energy-efficient;
- consider memory to be a scarce resource;
- avoid making assumptions about the platform in the API and implementation.

Technologies used

The library is built using Scala with support for functional programming from the Scalaz library. It provides useful concepts and syntactic sugar (such as lenses and monads, and notation for them), which make some operations less error-prone. To avoid imposing this choice on the client, we do not expose Scalaz types in the API, though we recommend the client uses Scalaz for some things (such as the **State[ToxState, _]** monad).

We use Snappy for its fast (de)compression, relatively good compression ratio (1.5x-4x is typical on human-readable data) and its native Java implementation.
The wire format for our network protocols uses Protocol Buffers. Cap'n Proto was also considered, but protobuf is already used within Tox4j. Both provide a language- and platform-independent specification language for a binary wire format, which can be used to generate the serializer and parser in a variety of languages.

Generic data structures

Streams: We take an immutable **Stream** datatype. While this has higher potential for memory leaks (leaking a **Stream** makes all subsequent **Stream**s and values leak), it

simplifies the interactions between components.[5] Moreover, this data type uses **Future**s rather than blocking when a client attempts to access yet-unknown values of the stream.

```
class Stream[+A] {
  val next: Future[(A, Stream[A])]
}

class Source[-A] {
  def put(a: A): Option[Source[A]]
  def putAll(s: GenTraversable[A]): Option[Source[A]]
}
```

Here, `+A` and `-A` denote covariant and contravariant types.
Assuming we have `Apple <: Fruit` (`Apple` is a subtype of `Fruit`), this means that any **Stream[Apple]** is a **Stream[Fruit]** (covariance), and any **Source[Fruit]** is a **Source[Apple]** (contravariance). This is useful in situations where various **Stream**s (or **Source**s) that operate on related types can be combined: for instance, it could occur when interleaving the events coming from several **Stream**s into a single **Stream**.

Futures: Use futures when wrapping a potentially-blocking operation, rather than use explicit callbacks.

Iterables: For iterable data structures, **GenTraversable[_]** was chosen over **Iterable[_]**:

- It does not involve a mutable iterator.
- It provides many useful functions (fold, …).
- It supports parallel collections.

Specific data structures

Unless specified otherwise, all data structures are immutable.

ToxState

ToxState is an immutable data structure that represents the state of HLAPI. This includes:

- module-specific state (friend list, ongoing conversations, …);
- information related to the user (nick, pubkey, …);
- callbacks that **ToxInstance** calls when I/O is performed.

Such callbacks are monadic: they have type **A => ToxState => (ToxState, B)**. If **B** is **Unit**, it is omitted and the return type is **ToxState**. For instance, **GroupMessaging**.*create* has type **ToxState => (ToxState, GroupConversation)**.

This specific type signature makes them directly suitable for use with Scalaz's State monad, as it provides the following method: **apply[S, A](f: (S) => (S, A)): State[S, A]**, so **ToxState** transformers can easily be used with the monad syntax.

```
final case class ToxState private (
    moduleStates: Map[ToxModule, Any], // Map from modules to their state
    conversationCallback: Option[UserConversation => ToxState => ToxState],
    friendCallback: Option[IncomingRequest => ToxState => ToxState]
) {
  /** Register callbacks. For HLAPI's internal use only */
```

[5] For instance, it is possible to safely share an immutable stream.

```
  private[hlapi] def registerConversation(callback: UserConversation => ToxState =>
ToxState): Option[ToxState]
  private[hlapi] def registerFriend(callback: IncomingRequest => ToxState => ToxState):
Option[ToxState]

  /** Wraps _getState and _toxState in a lens */
  private def stateLens(t: ToxModule): Lens[ToxState, t.State]

  /** Register a module.
   * This calls the module's register method with appropriate parameters. */
  def register(t: ToxModule): \/[String, (ToxState, t.ImplType)]
}
```

Callback registration is handled by the modules themselves when they are being registered. At its most basic, a **ToxModule** implements the following trait:

```
abstract class ToxModule extends Configurable with Equal[ToxModule] {
  type State
  def initial: State
  def name: String = getClass.getName

  def policy: Policy = Policy.default // Default security policy is empty

  type ImplType
  private[hlapi] def impl(lens: Lens[ToxState, State]): ImplType

  def register: ToxState => \/[String, (ToxState, ImplType)]
}
```

The implementation of a module only has access to its own state (stored in **ToxState**) through a lens, which is provided at registration. This is less error-prone than having methods for state manipulation, as external code, for example another **ToxModule** implementation, cannot directly inspect or modify the state of a given **ToxModule** without using unsafe features.

Any such interaction must be done through an API defined by the **ToxModule**. While it is possible to expose a concrete **module.State** and a **Lens[ToxState, module.State]** (or equivalent) in the module's API, doing so is not recommended. Modules should enforce separation of concerns in their API, instead.
Should a **ToxModule** depend on other **ToxModule**s, it must be given the relevant **module.Impl** objects at construction time. While this puts the burden of managing dependencies on the client (when it comes to loading the modules), it avoids dependency loops and other problems.

A **ToxModule** also carries a **Policy** object, an immutable wrapper around **java.security.Permissions**. Its purpose is to let HLAPI run modules in appropriate security contexts, following a *least privileges* principle.

```
object Policy {
  private def copy(p: Permissions): Permissions
  def apply(p: Permissions): Policy
  val default = Policy(new Permissions())
}

final class Policy private (p: Permissions) {
```

```
  def add(perm: Permission): Policy

  def addAll(perm: GenTraversableOnce[Permission]): Policy
  def addAll(perm: Permissions): Policy
}
```

Alternatives:
An older version of the design had one **ToxState**.*registerFoo* method per component, which is not scalable. Instead, **ToxModule** now carries the *register* method, as the registration code is component-specific. **ToxState** only provides a convenience *register* method, which calls *register* on the **ToxModule**.

There used to be no difference between the module itself and it's **ImplType**. Adding this separation enforces that modules get registered. This does not prevent the client from using a **ToxState** where the module was not registered, but the monadic notation makes it hard to do accidentally.

Tox configuration

HLAPI and its components can be configured using a **SettingKey**-based approach: each module (and **ToxState** itself) possess **SettingKey** objects, which describe a single setting.

```
/** The interface implemented by setting keys. */
abstract class SettingKey {
  /** The type of the setting's value */
  type V
  /** The default value */
  val default: V
}
```

ToxModules and **ToxState** implement a common **Configurable** interface:

```
/**
  * Abstract class implemented by configurable modules.
  *
  * In particular, all [[im.tox.hlapi.core.ToxModules]] and [[ToxConfig]]
  * are configurable in this way.
  */
abstract class Configurable {
  /** The type which describes the various settings */
  type SettingKey <: SettingKey

  /** To each [[SettingKey]] there is a matching lens. */
  def getSetting(key: Setting): ToxState => key.V
  def setSetting(key: Setting)(value: key.V): ToxState => ToxState
}
```

Configurations values are heavily read during HLAPI operation. This is why **ToxModule**s (and **ToxCore**) will provide equivalent *getSetting* and *setSetting* methods which deal directly with **module.State**, for internal use.
Like their equivalents from **Configurable**, they can only be implemented inside the **Impl** object, as it holds the lens that connect the modules' state and **ToxState**.

This can be implemented using an immutable, heterogeneous map, for modules which have many settings; other modules may prefer to use a flat tuple, which should use less memory and provide faster access, at the cost of needing more code per config option.

Configuration options for **ToxState**:

```
sealed trait OptimizationTarget
final case object Memory extends OptimizationTarget
final case object Battery extends OptimizationTarget
final case object Performance extends OptimizationTarget

sealed trait ToxConfig extends SettingKey
final case object Target extends ToxConfig {
  type V = OptimizationTarget
  val default = Battery
}

final case object AwayMessage extends ToxConfig {
  type V = TextMessage
  val default
}

object ToxConfig extends Configurable {
  type Setting = ToxConfig

  def getSetting(key: ToxConfig): ToxState => key.V
  def setSetting(key: ToxConfig)(value: key.V): ToxState => ToxState
}
```

Alternatives:
It is possible to avoid the use of an heterogeneous data-structure, but at the price of type safety. Also, providing Int indexes (for instance for storing the settings in a Vector) doesn't seem doable easily (and safely). This is why we made this performance/safety tradeoff.

User profiles

```
case class User(
  // 38B pubkey, pretty-printed with 72 characters (152B in UTF-16)
  toxId:         PublicKey,
  name:          String,
  statusMessage: String,
  status:        Online | Offline
)
```

HLAPI will not support more detailed status, as they are confusing for users (not included anymore in most mobile chat apps, and may not reflect actual user status) and make optimisation for bandwidth & battery use harder (because state changes are more frequent and Tox needs to be “more sure” of the state, forcing more frequent data exchange).

Friend tag

User tags implement a N:M user group system, where one user may belong to multiple groups. In particular, a distinguished “star” group can implement the “starred user” feature.

```
class FriendTag private (
  val name: String,
  id: Int,
```

```
){
  def add(user: User): ToxState => ToxState
  def delete(user: User): ToxState => ToxState
  def users: ToxState => ToxState
}
```

Alternatives

A N:1 group system was considered, but it would be less flexible, and would require special-casing the “starred user“ feature.

ConversationId

A **ConversationId** is an immutable object that carries the information required to identify and contact a chatroom or user. It is an open case class which can be either of **core.User**, **group.GroupChat** and **group.GroupUser**.
ConversationId and **Conversation** are open case-classes to allow for future extensions. This also lets different packages implement the **Conversation** types that concern them.

```
abstract class message.ConversationId extends Serializable {
  val key: PublicKey
  val name: String
}

case class core.User(val key: PublicKey) extends ConversationId with
ValueType[PublicKey]

case class group.GroupChat(val key: PublicKey)
    extends ConversationId with ValueType[PublicKey] {
  def join: ToxState => (ToxState, Future[\/[GroupConversation, JoinError]])
}

case class group.GroupUser extends ConversationId {
  val parent: GroupChat
}
```

Alternatives

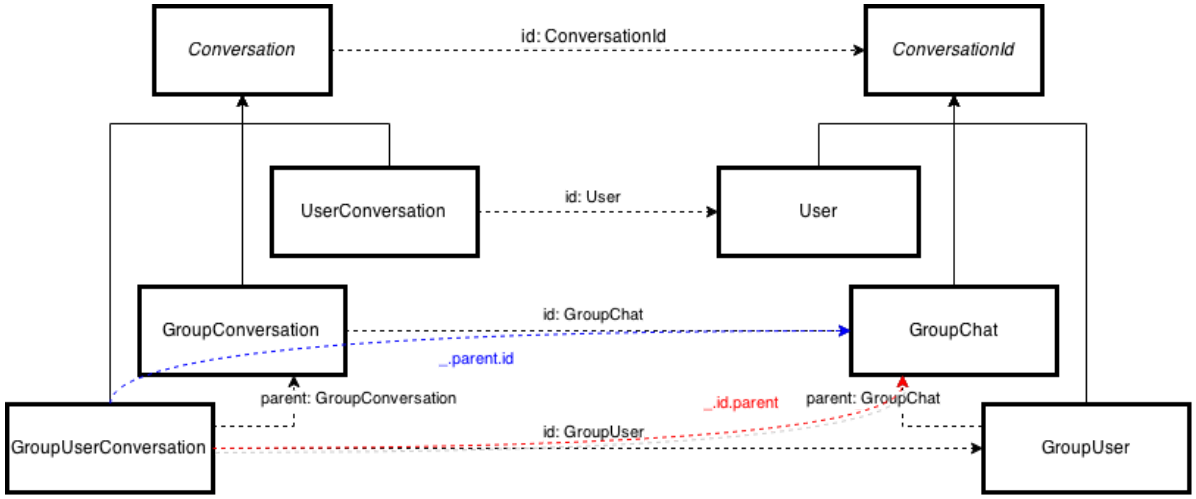
It could have been possible not to have any separation between **Conversation** and **ConversationId**. However, many functionalities (logs, friend lists, establishing conversations …) require being able to designate the recipient of a conversation, such as a **GroupChat** or a **User**, regardless of whether there is still an ongoing conversation.
Once the decision to have **GroupChat** and **User** was made, it made sense to unite those datatypes in a case class that mirrors **Conversation**.

Conversation

A **message.Conversation** is an object representing an ongoing chat session. It is actually an open case class which can be one of the following:

- **UserConversation**: a conversation with a single friend;
- **GroupConversation**: a group chat
- **GroupUserConversation**: a conversation with a single user, whose actual public key is not known, but with whom we share a group chat.

The following diagram represents all **Conversation** and **ConversationId** types, and their interactions:



As both **GroupUserConversation** and **GroupUser** carry a reference to their parent (resp. a **GroupConversation** and **GroupChat**), 2 paths can be used to reach the same value. HLAPI guarantees the following invariant:

```
forall GroupUserConversation guc, guc.parent.id == guc.id.parent
```

Any **Conversation** also has an associated message stream, which contains both received and sent messages: this simplifies client code, as it is enough to display all the messages from the stream, rather than add sent messages and deal with potential message transforms, like message splitting.

```
abstract class Conversation {
  val messageStream: Stream[MessageEvent]
  val id: ConversationId

  // Monadic operations
  def sendMessage(message: Message): ToxState => (ToxState, Future[Unit])
  def typing(isTyping: Boolean): ToxState => ToxState
}

case class UserConversation(val id: User) extends Conversation

case class GroupConversation(val id: GroupChat) extends Conversation {
  def members: ToxState => GenTraversable[GroupUser]
}

case class GroupUserConversation(val id: GroupUser) extends Conversation {
  val parent: GroupConversation
}
```

The **Future[Unit]** returned by *sendMessage* carries the information about message delivery: it will be resolved (either as () or as an error) when the message is acknowledged, or if delivery times out.

Alternatives: It could be possible to have instead a function which gets newly-received messages from **ToxState**, but this would require the UI to be aware of HLAPI's event loop, either directly or by setting a callback on message delivery, whereas the solution we adopted yields a pure[6] data structure that can be directly consumed in the UI in its own event loop. The same effect (needing to be mindful of HLAPI's event loop) would also manifest in tests.

No satisfying alternative was found to avoid the “diamond pattern” (`guc.parent.id == guc.id.parent`). It would be broken by removing either of the *parent* attributes, but this is difficult for various reasons. Removing it from the **GroupUser** object makes it impossible to know from which group chat a user came from, while removing it from the **GroupUserConversation** object makes it harder for the client to link back to the ongoing conversation.
The second point could be solved by maintaining a partial mapping from **GroupChat** to **GroupConversation**, but this is error-prone: it would be possible to accidentally leak memory or get a non-consistent mapping. Moreover, the information that there is one **GroupChat** corresponding to each **GroupConversation** is lost at the type level. This leads to an incomplete matching or a potentially uncaught exception in the places where the client uses the mapping (or in HLAPI).

Message event

A **MessageEvent** is represents an event in the message stream. It can either be creation, edition or deletion. **Conversation** carries a **Stream** of such events, which contain the concrete **Message**s (documented below).

```
sealed abstract class MessageEvent(final val id: MessageId)

final case class NewMessage(message: Message) extends MessageEvent(message.id)
final case class DeleteMessage(id: MessageId) extends MessageEvent(id)
final case class EditMessage(message: Message) extends MessageEvent(message.id)
```

A **NewMessage** describes the emission of a new message in an ongoing **Conversation**, while **DeleteMessage** (resp. **EditMessage**) denote the deletion (resp. replacement) of an existing message.

Message

Message is an immutable, open case class, which can be either text (**TextMessage** or **ActionMessage**), or inline media (no datatype defined yet).
As **Conversation**s carry a **Message** stream, all kind of messages can be interleaved in a single conversation. This does not cover use cases such as images embedded inside the text (such as using images for emotes, but Unicode emoji are used for this).

In particular, clients may encounter Message types they do not know how to handle. While this cannot happen for the legacy messaging protocol (which only carries **TextMessage**s and **ActionMessage**s), it is the case for the new text messaging protocol.
This is why the protocol extension must provide

- a way to signal message rejection to the sender (with an associated reason);

---

[6] While the implementation of streams may use mutation internally, it is not externally observable; basically, writing occurs in a **Future[_]**, which is write-once and can be expressed with *call/cc*.

- a way to signal which message types are supported.

HLAPI will automatically reject messages that do not conform to an advertised message type (see "Protocol extensions/Requirements"). However, messages might be unsupported for reasons that cannot be exposed to the HLAPI (image too large to decompress, unsupported media codec, …), in which case the client must reject the message, rather than silently discard.

```
abstract class Message {
  val from: User
  val time: com.github.nscala_time.time.Imports.DateTime
}
```

Both **TextMessage**s and **ActionMessage**s carry plain Unicode strings. Clients must not serialize richer formatting ADTs as Unicode strings (for instance HTML).
Three extended message types, **ImageMessage**, **SoundMessage** and **VideoMessage**, can carry respectively **Image**, **Sound** and **Video** objects. Those are currently wrappers around **Array[Buffer]**, pending definition of more appropriate datatypes.

HLAPI will provide support for richer formatting, in a different message type.

Storage

HLAPI lets its users implement only the needed requirements for the functionality they use. In particular, several components (**Logging**, **GroupChat**, **FriendList**, and **FileTransfer**) require some form of persistent storage.

Interfaces

It is unrealistic to expect all client implementers to come up with storage implementation, especially in the face of their reliability requirements. Hence, HLAPI will provide a default implementation for each, assuming the client can provide a **FileLike** storage. **FileLike** is an interface that describes storage with random-position read/write. A **FileLike** implementation is already provided, which internally uses **java.nio.MappedByteBuffer**.

FileLike directly deals with reading and writing **Slice**s (consecutive bytes) from storage at arbitrary offsets. Data is provided as individual bytes, which is well suited for existing Snappy implementations, and does not seem to bring an unreasonable performance hit (i.e. more than 1Gb/s throughput, without incurring high CPU load).

```
trait FileLike {
  // Attempt to discard part of the file (e.g using fallocate(2) hole-punching).
  // Failure MUST be handled gracefully by the caller
  //  (i.e. HLAPI storage implementation).
  //  NOTE: Haven't yet found where Java implements this.
  def discard(slice: Slice): Boolean = false

  //TODO Need proper error monad
  def apply(offset: Long, size: Int): Option[Slice]
  def size: Long
}

trait Slice extends Iterable[Byte] {
  def size: Int
```

```
  def flush() // See below

  def write(offset: Int)(data: Array[Byte]): Boolean
  def write(data: Array[Byte]): Boolean = write(0)(data)

  def get(offset: Int): Option[Byte]
  def set(offset: Int, value: Byte): Boolean
}
```

After a write operation, the data may not yet be written to persistent storage. Calling *flush* ensures the persistent write to the storage medium has happened.
discard attempts to discard part of the file. This operation may fail for arbitrary reasons, and so the trait can provide a default implementation (which always fail). Supporting this feature when possible is useful, however, because it makes garbage-collection of the **KeyValue** store faster, and reduces the amount of I/O operations to the storage, which should improve battery and flash life.
The use of **Slice** objects is to reduce the number of bound-checking (and corresponding error-handling) performed.

Alternatives:
Using **java.io.File** directly was considered, but this would be much less flexible. Also, **java.io** (and the newer **java.nio**) only support large (> 2GB) files through streams[7], whereas we need to handle random access.

Specialized storage classes

The actual storage interfaces distinguish two use-cases there: **KeyValue**, a generic interface for persisting state and **LogStorage** for storing the logs. Moreover, the **FileLike** trait describes the interface for file-like objects, which are used during file transfers and by the default **KeyValue** and **LogStorage** implementations.

Both **KeyValue** and **LogStorage** must provide transactional semantics, *i.e.* either an effectful operation has successfully completed or it has not happened at all, even if the whole JVM crashes (or computer shuts down, or …) before the operation returns. Read operations started during a write must observe a consistent state. This state can be from either before or after the write (thus blocking the read until the end of the write). A read performed after a write must observe the writing's effect.

KeyValue is a generic interface for a key/value store, parameterized by both of those types. It can be implemented in more-or-less arbitrary fashions, as the generic **∀ K V. KeyValue[K,V[K]]** type is never required in an interface, allowing the client to implement differently (for instance) **KeyValue[PublicKey, User]** and **KeyValue[TransferId, Transfer]**. Here, the **KeyType** trait is useful for generic implementations (as will be provided by HLAPI), as it is possible to require keys to be serializable, ordered, …). The exact **KeyType** trait is not completely defined yet, as its design will be guided by the needs of the default storage implementation.

```
trait ValueType[K] extends Serializable {
```

[7] There is support for random access, but indexing is done through 32-bits **Int**s.

```
  val key: K
}

trait KeyType extends Serializable

trait KeyValue[K <: KeyType, T <: ValueType[K]] extends GenTraversable[T] {
  // add should replace existing state with same id
  def add(id: K, obj: T)
  def lookup(id: K): Option[T]
  def delete(id: K): Boolean
}
```

LogStorage is a specialized interface for log storage, supporting per-**Conversation** storage, with insert, modify and delete operations. Its only use is implementing **LoggingReq**.

```
trait LogStorage extends GenTraversable[ConversationId] {
  def lookup(conversation: ConversationId): GenTraversable[Message]
  def append(conversation: ConversationId, message: Message)
  def modify(conversation: ConversationId, message: Message)

  def delete(conversation: ConversationId)
  def delete(conversation: ConversationId, message: Message)
}
```

Interaction with ToxState

FileLikes used for file transfers do not need to be kept strongly synchronized with the **ToxState**: the worst that can happen is a resume of a file transfer at an earlier point, causing needless data transfers. However both **KeyValue** and **LogStorage** essentially provide persistence for the component's state, and thus need more careful handling.

KeyValueWrapper provides a monadic wrapper around a **KeyValue** object, where methods produce a new instance and no actual IO is performed until the *performIO()* method is called. **ToxModule**s have a *performIO()* method which **ToxInstance** can call when IO is being performed.
As **Logging** performs IO immediately after messages from **ToxCore** are retrieved, it does not seem necessary to provide a monadic wrapper for it.

Default implementations

FileLike

HLAPI provides a **FileLike** implementation called **MappedFile**, based on Java's **MappedByteBuffer**s. A **MappedFile** can be constructed from a path, represented as a **Path** object or a **String**, and from a **RandomAccessFile**.
Each **Slice** carries its own **MappedByteBuffer** which is a mapping that starts at the specified offset, and has at least the slice's length.

KeyValue

The HLAPI default **KeyValue** implementation separates its **FileLike** into three zones: a header, 2 intent logs, and the tables. Both header and intent logs have fixed size.
The **KeyValue** implementation assumes that the keys can be efficiently hashed to a 64 bit value, and uses the keyed hash function SipHash-2-4 for that purpose.

In particular, this implementation uses hashed tries internally, and so doesn't use key ordering nor can it support ordered queries such as “all records with a key greater than X”.

Its design is influenced by Google's LevelDB and Phil Bagwell's Hashed Tries (HAMT).

Header: It contains the 32 bit magic number 0x29c35097 (here to prevent accidental use of a non-**KeyValue** file) followed by a version number (in our case, 0) and several parameters required for using the key-value store.

```
header {
  uint32 0x29c35097;
  uint32 version_number = 0;
  uint32 intent_log_size; // Default is 512kB
  uint32 intent_log_chunk_size; // Default is 512B
  uint32 table_number; // Number of tables
  uint32 _; // Padding
  uint64[2] hash_key; // Key used for hashing; taken at random when the file is created.
}
```

Intent Log: Each intent log has constant size, separated in constant-size chunks. Both of those sizes are configurable when the **KeyValue** is created. Default values are 512kB logs, with 512B chunks.
Each chunk starts with a flag that can be either FIRST (0xF0), MIDDLE (0xAA), LAST (0x0F), EMPTY (0x00) or SINGLE (0xFF). A sequence of contiguous chunks that starts with FIRST, contains any number of MIDDLE and ends with LAST contains the encoding of a (key, value) pair (called a record), as does a single chunk starting with SINGLE.

The rationale is as follows: use of the intent log avoids blocking the client while the tables undergo a concurrent operations such as compaction. Moreover, chunking the log allows for fast record skipping, supports records that can be as large as the whole intent log (minus the 2b/512B ≃ 0.05%) and allows re-synchronization in the event of disk corruption[8]. Lastly, the choice of constants was such that no two tags have less than 4 differing bits.

Tables: The main storage area, which forms the tail of the file, is structured into tables. Each table starts with a header describing the size of its sections. It can be at most 4GB large.

```
table_header {
  uint32 keys_max; // Max. size of the key chunk (values lie beyond)
  uint32 keys_alloc; // Pointer to the end of the key chunk
  uint32 values_alloc; // Pointer to the end of the value chunk
  uint64 total_size; // Used to jump to the next table
}
```

[8] Skipping blocks in the intent log will yield data loss. This circumstance cannot happen in the storage model for which this **KeyValue** implementation is designed (transactional semantics, explicit flush) except for records for which the *add* operation hasn't returned yet.

Each table is a hashed trie: during lookup, the key is hashed and the table is a prefix tree for the hashed keys. The root has degree $2^{10} = 1024$, and is represented as an array of size 1024, lying immediately after the header. Internal nodes have degree at most $64 = 2^6$.
This means the depth of the tree is at most 9, assuming a 64-bit hash function.

```
node {
  uint64 bit_set;
  int32[bitCount(bit_set)] children;
}
```

In a node, the bitset (encoded as a single int64) describes which children are non-empty. It is followed by an array whose i-th position is occupied by the offset of the i-th non-empty children[9].
For alignment reasons, and to avoid needless fragmentation, nodes are actually allocated with enough size to accommodate 2, 4, 8, 16, 32 or 64 children. bit_set MUST be set to zero when deleting a node; this allows easily checking if a node can be promoted to a larger size without copying.

The children of a node are represented by their offset. If the offset is positive, it is relative to the beginning of the key segment and points to an internal node. If the offset is negative, it is relative to the beginning of the record segment and points to a record.
This is done in this fashion for two main reasons:

- In the event where the trie is corrupted, it is possible to read sequentially all records, and reconstruct the index.
- It allows modifying the size the table to a larger size by a simple sequential copy (without modification) of the records segment.

When a table is full, a new table is created at the end of the file. The i-th table has key-segment size 2^i MB for i <= 10, and size 2GB afterwards.

TODO doc:

- freelists buckets
- algorithms for insert/delete/modify

Testing: Several sub-components of the implementation can be used as key-value stores, like the intent log, a single table, and the sequence of tables. As such, they can be tested separately using the generic tests for **KeyValue**, along with the complete implementation.

Components

The HLAPI first lets you create a **ToxInstance** object, which contains the user's Tox key pair, along with the thread which is used for encapsulating **ToxCore**'s event loop.
Having an additional layer of abstraction between **ToxCore** and **ToxInstance**, which would provide safer wrappers for **ToxCore**'s methods, was considered. However, it was discarded as either too cumbersome (having **ToxInstance** proxy all calls to the **ToxCore** wrapper) or

[9] Indexing in this structure can be done efficiently using *java.lang.Long.bitCount*.

having too little benefit (if the HLAPI code can call the wrapper directly, it can bypass the monadic/async wrapping from **ToxInstance**).

Any non-trivial functionality is handled through optional components. Each such component is described by its own section in the following text, and implements the **ToxModule** trait.

ToxInstance operates on immutable **ToxState** objects, which carry callbacks (registrable once) and per-component state (whose type is defined by the component itself).
If the client wishes to use a given component, it must:

- construct the component (giving the constructor an implementation of the requirements, if any);
- call its *register* method on **ToxState**, which either yields a new **ToxState** value, or the name of the component which failed to register (the actual return type is V**[String, A]**); failure occurs if a component is registered more than once;
- use the new **ToxState** value for further HLAPI operations.

In the following text, each section (“Text messaging”, …) is one such component.
While registration is enforced at the type level, as a client must register a component to get the matching component.Impl object, it is possible to use a component with a **ToxState** where it was never registered (and discard the **ToxState** that registration yielded). The client must never do so, and use of the **State[ToxState, _]** monad makes this mistake less likely, because the monad handles the state-passing for the client.

Some HLAPI operations can be done without the client having to implement a specific interface (for instance, generating a new NoSpam value, setting user status, ...). Those operations are implemented as functions which operate on **ToxState**, defined in separate classes as appropriate.

Text messaging

TextMessaging has simple, single-method interfaces:

```
trait TextMessagingReq {
  def callback(newConversation: UserConversation)
}

class TextMessaging extends ToxModule {
  type State = Unit
  val initial: State = ()

  type ImplType = Impl
  private[hlapi] def impl(lens: Lens[ToxState, State]) = {
    new Impl(lens)
  }

  final class Impl(lens: Lens[ToxState, State]) extends Configurable {
    def startConversation(user: User)(tox: ToxState): (ToxState, UserConversation) = ???

    type Setting = MessageSetting
    def getSetting(key: Setting): ToxState => key.V = ???
    def setSetting(key: Setting)(value: key.V): ToxState => ToxState = ???
  }
}
```

```
sealed trait MessageSetting extends SettingKey

final case object strictEncoding extends MessageSetting {
  type V = Boolean
  val default = true
}

final case object downloadInlineMedia extends MessageSetting {
  type V = Boolean
  val default = true
}

final case object downloadInlineMediaWhenMobile extends MessageSetting {
  type V = Boolean
  val default = true
}
```

Message splitting

The messaging protocol extension supports arbitrary-sized messages. However, a message-splitting specification is still required to handle the legacy protocol, and the one suggested by STS is under-specified (in the case where a message contains no whitespace).

HLAPI will split messages, when using the legacy protocol, in the following way:

- split at the last whitespace within MAX_MESSAGE_LENGTH bytes;
- if there is no such whitespace, split at MAX_MESSAGE_LENGTH bytes;
- process the tail of the message recursively (unless empty).

Notes:

- no encoding detection is performed; all data sent/received is UTF-8, while the JRE internally uses UTF-16 encoding.
  By default, messages that are not valid UTF-8 are rejected, because detecting the encoding is in general impossible, and cannot be achieved with high confidence (making it likely that corrupted data is sent to the client, if it is attempted).
- In the case of the legacy protocol, HLAPI can be configured to replace bytes that do not decode cleanly by a Unicode replacement character; else, it will signal rejection by sending back a default rejection message (which must be ASCII-only, to ensure it can be decoded by the peer).
- In the new protocol, rejection MUST be signaled to the other end, including in the case of a wrong encoding. Such messages are not delivered to the client.
- Hook mechanism?
  need to provide configuration methods

Logs

The requirements for the logs component are a little more intricate:

```
trait LoggingReq {
  def logStore: LogStorage
  def indexFile: FileLike
}
```

The storage implementation must have atomic semantics: an operation either has happened or has not. In particular, power loss should never result in log corruption, which is why clients

are advised to use the default implementation provided under **im.tox.hlapi.storage**, which can construct **LogStorage** on top of a **FileLike** object, as explained in the "Storage" section.

In return, HLAPI provides a **Logging** instance:

```
class Logging extends ToxModule {
  type State = Unit
  val initial: State = ()

  type ImplType = Impl
  private[hlapi] def impl(lens: Lens[ToxState, State]) = {
    new Impl(lens)
  }

  final class Impl(lens: Lens[ToxState, State]) extends Configurable {
    def lookup(conversation: ConversationId)(tox: ToxState): GenTraversable[Message]
    def search(query: Query)(tox: ToxState): GenTraversable[Message]

    type Setting = SyncConfig
    def getSetting(key: Setting): ToxState => key.V
    def setSetting(key: Setting)(value: key.V): ToxState => ToxState
  }
}
```

The **SyncConfig** trait is shared with the **FriendList** module, and defines settings that are relevant for synchronizable modules (see Device synchronization):

```
sealed trait SyncConfig extends SettingKey
final case object syncPeriod extends SyncConfig {
  type V = org.joda.time.Period
  val default = StaticPeriod.minutes(10)
}

final case object syncWhenMobile extends SyncConfig {
  type V = Boolean
  val default = false
}
```

Notes:

- The exact **Query** interface is not defined yet
- No clean way was found to handle varying requirements: for instance, there is no need for *modify*, if edit is not supported. That is why support for message edit/deletion is mandatory.
- The use of Apache Lucene for indexing was considered. However, they have a hard-to-match storage API, large memory requirements, and large code size.

Friend list

The FriendList component provides a friend list, where each user can be annotated with multiple tags (see Friend tags). It has comparatively simple requirements:

```
trait FriendListReq {
  def callback(newRequest: IncomingRequest): ToxState => ToxState
  def storage: KeyValue[PublicKey, User]
}

class FriendList(req: FriendListReq) extends ToxModule {
```

```
  type State = KeyValueWrapper[PublicKey, User]
  val initial: State = KeyValueWrapper(req.storage)

  type ImplType = Impl
  private[hlapi] def impl(lens: Lens[ToxState, State]) = {
    new Impl(lens)
  }

  final class Impl(lens: Lens[ToxState, State]) extends Configurable {
    def addNoRequest(user: User)(tox: ToxState): ToxState
    def add(user: User, noSpam: NoSpam, message: String)(tox: ToxState): ToxState
    def add(address: ToxAddress, nick: Option[String], message: String)(tox: ToxState):
ToxState
    def add(address: ToxAddress, message: String)(tox: ToxState): ToxState = {
      add(address, None, message)(tox)
    }

    def delete(user: User)(tox: ToxState): ToxState

    val star: FriendTag
    def lookupTag(name: String)(tox: ToxState): Option[FriendTag]
    def createTag(name: String)(tox: ToxState): (ToxState, Option[FriendTag])
    def tags(tox: ToxState): GenTraversable[FriendTag]
    def friends(tox: ToxState): GenTraversable[User]

    type Setting = SyncConfig
    def getSetting(key: Setting): ToxState => key.V
    def setSetting(key: Setting)(value: key.V): ToxState => ToxState
  }

}
```

File transfer

The FileTransfer component also has simple requirements:

```
trait FileTransferReq {
  type T <: FileLike
  def callback(newTransfer: IncomingTransfer[T]): ToxState => ToxState
  def state: KeyValue[TransferId, Transfer[T]]
}
```

Here, **T** is a user-defined type that is used for representing file locations. Suitable choices include URIs and file paths, but this design lets clients use directly the right type for their platform.

HLAPI provides one function, and most interactions are done through the **Transfer** type:

```
final case class FileTransferring(req: FileTransferReq) extends ToxModule {
  type State = Unit
  val initial: State

  type ImplType = Impl
  private[hlapi] def impl(lens: Lens[ToxState, State]) = {
    new Impl(lens)
  }

  final class Impl(lens: Lens[ToxState, State]) extends Configurable {
    def proposeFile(file: req.T, user: User)(tox: ToxState): (ToxState,
OutgoingTransfer[req.T])
```

```
    type Setting = FileSetting
    def getSetting(key: FileSetting): ToxState => key.V
    def setSetting(key: FileSetting)(value: key.V): ToxState => ToxState
  }
}

sealed trait FileSetting extends SettingKey
final case object transferWhenMobile extends FileSetting {
  type V = Boolean
  val default = true
}
final case object simultaneousTransfers extends FileSetting {
  type V = Int
  val default = 5
}
final case object simultaneousTransfersPerUser extends FileSetting {
  type V = Int
  val default = 2
}
final case object suspendTranferUponCongestion extends FileSetting {
  type V = Boolean
  val default = true
}
```

Notes:

- Should we add per-transfer registrable callbacks (for instance on abort or on completion) ?

Group messaging

Remark: The group messaging API and feature set is not completely defined in toxcore yet, so this may be subject to change (even more than the other components)

Should the callbacks be merged in a single newConversation(Conversation) callback?

```
trait GroupConversationReq {
  def inviteCallback(chat: GroupChat)
  def privateCallback(conversation: GroupUserConversation)

  def storage: KeyValue[PublicKey, GroupChat]
}

class GroupMessaging extends ToxModule {
  type State  = Unit
  val initial = ()

  type ImplType = Impl
  private[hlapi] def impl(lens: Lens[ToxState, State]) : Impl

  trait Impl {
    def create: ToxState => (GroupChat, ToxState)
    def join(group: GroupChat): ToxState => (ToxState, Future[GroupConversation])

    type Setting = GroupSetting
    def getSetting(key: Setting): ToxState => key.V
    def setSetting(key: Setting)(value: key.V): ToxState => ToxState
  }
}
```

```
sealed trait GroupSetting extends SettingKey
```

See operations defined in
https://github.com/irungentoo/toxcore/compare/master...JFreegman:new_groupchats#diff-a58028aec4211a18b170732935eba2e3R2251

JNI binding

The JNI binding is inherited from Tox4j, and is not exposed at all in the API. The **ToxCore** object is encapsulated in a **ToxInstance**, which provides to the other HLAPI components safe methods to interact with **ToxCore**.

The existing decoupling of HLAPI and **ToxCoreImpl**, through the **ToxCore** interface, simplifies auditing and testing (JNI being unsafe), and makes it possible to use an alternative **ToxCore** implementation (either for production use or testing).

A/V

- Need to discuss client needs
- Configuration
    - Codecs when {on mobile net; on battery; plugged in}

Testing

HLAPI implementation will follow the behaviour-driven development methodology: implement a specification (in the form of a property-based test), implement the corresponding unit, and check that the tests pass.
In addition to property-based random tests, we will also require unit tests and integration tests:

- Unit tests make it easier to check corner cases where bugs are deemed more likely.
- Integration testing is (currently) the only way to test with an actual ToxCore implementation, and can detect issues where several related components interact.

Test coverage metrics

Tox4j currently uses Scoverage, which supports statement coverage. This is an appropriate metric for our needs: it enforces that all code actually gets executed (unlike line coverage), and is yet lenient enough that coverage is achievable, unlike with path coverage which is (in the strictest sense) impossible to achieve in the presence of unbounded loops and recursion.

Test framework

Tox4j currently uses ScalaTest with ScalaCheck for behavioural specifications and JUnit for specific unit-tests. These tools seem fairly well suited to behaviour-driven development, so we will keep using them.

Protocol extensions

Requirements

Protocol extensions are based on message-passing primitives, including message rejection with a protocol-provided reason (feature not supported, syntax error, …). This high-level

mechanism must support arbitrary-size “high level” messages, based on Toxcore's peer discover, cryptography and low-level message transport primitives.

The protocol extensions must be described using a language- and platform-independent syntax specification, from which efficient parsing and serialization code can easily be generated. We chose protobuf (more precisely, its proto2 version) because Protocol Buffers are already in use within Tox4j, and proto3 is[10] still in alpha, and is not yet supported by the tooling we use.
For forward-compatibility with proto3, proto2 is employed with the following restrictions:

- Fields MUST NOT be required in the format specification.
  Extensions may mandate protocol-specific message validation instead.
- Each field MUST have a default value compatible with proto3.
- Proto2 extensions MUST NOT be used.

For efficiency reasons, all repeated fields must have the attribute [packed=true].

- Optional features
  - Advertise the set of supported features
  - HLAPI will automatically discard messages related to unsupported features
  - Features themselves could have varying support (i.e. supported image types, …)
    - such features MUST define a minimal feature set;
    - they SHOULD, if possible/practical, require the client to advertise their supported feature set;
    - when such advertisement is done, HLAPI MUST automatically reject unsupported messages;
    - clients MUST reject unsupported messages in any case.
- Congestion control
  - Be able to provide back pressure (i.e. signal to peers that we are overloaded, and make them back off)
  - Prioritize some components rather than others ? (e.g. give interactive components priority over {friend list; log} synchronisation and file transfers)

Optional protocols

A message, in this protocol, carries a timestamp and one of the messages described in the following sections.

The timestamp is encoded as an **uint64** representing the number of seconds elapsed since the Unix epoch, in UTC time. The sender SHOULD round it to the nearest half-minute in pseudonymous contexts, including when sending group chat messages, to avoid (this venue of) device-fingerprinting. The receiver SHOULD set it to the reception time if it is set to a greater value.

Some security-sensitive properties, such as sender identity and originating group-chat, MUST be provided by the low-level toxcore, rather than sent in the message (because a

[10] At the time of writing, 2015-06-18.

malicious sender could falsify them). This is less error-prone than all considered alternatives (having toxcore check, or doing the checking in the high-level proto implementation).

Feature negotiation

The feature-set negotiation mechanism is based on a single **Features** message type, which contains repeated **Feature**. Those describe the message types that can be received and meaningfully interpreted by the client, they do not describe the ability to initiate an interaction: i.e. they state “I am (un)able to receive file transfers” and NOT “I am (un)able to initiate a file transfer”.

A **Feature** is simply an enum, which describes the presence of a single feature; unmentioned **Feature**s are implicitly not supported:

- TEXT_MESSAGING describes this client's ability to receive text and action messages.
- MEDIA_MESSAGING describe the client's ability to interpret messages that are not simply text, in a conversation. The media messaging extension provides its own mechanism for advertising which codecs are supported.
- GROUP_CHAT advertises the client's ability to take part in group chats; the set of messages it can meaningfully receive is described by TEXT_MESSAGING and MEDIA_MESSAGING.
- LOG_SYNCHRONIZATION, FRIEND_SYNCHRONIZATION and GROUP_SYNCHRONIZATION are used to advertize that this client supports synchronizing logfiles, friends lists and lists of group chats.
- LOCATION_SHARING means the client can interpret location messages.
- FILE_RECEPTION means that this client can accept incoming file transfers.

Any change in the feature-set MUST trigger sending a **Feature** message to all currently-connected friends. Joining a group-chat and observing a friend connect MUST trigger the emission of a feature message.

Instant messaging

A Messaging message carries the following:

- a message identifier (**uint64**) which MUST be present;
- a type, which can be NEW (default), MODIFY, or DELETE.
- some data, which can either be a text message or a media message.

Text messages carry:

- a **string** (ProtoBuf requires UTF-8 encoding) called *data* (empty by default);
- a *type*, which can be REGULAR (default) or ACTION.

Media messages follow a similar structure:

- *data* is a **bytes** field; implementations SHOULD reject messages with empty *data*;
- *type* can be UNKNOWN (default), IMAGE, SOUND or VIDEO.
  Implementations MUST reject messages of type UNKNOWN; it is here to allow easily checking if the field is missing or set to a value that this version of the format doesn't know.

Device synchronization

Device synchronization relies on the following prerequisites:

- each device has a unique ID; its tox pubkey (or a hash thereof) is useable as such;
- locally, events have a monotonic ID (only requires an increasing counter).

Each device maintains a vector-clock that associates with every other device the ID (on the other device) of the last synchronization event (with the current device). Whenever two devices synchronize, they exchange all events that were newer than their clock, then update their vector-clocks.

Synchronization messages can be either of the following messages:

- START_SYNC, which carries a vector clock;
- DATA, which carries repeated events;
- END_SYNC, to confirm that all DATA was received successfully (and let the peer update its vector clock).

The vector clock is relative to a (synchronizable) module, and those messages carry an enum describing which module is concerned. Any implementation MUST reject DATA messages containing events that do not conform to the appropriate type for the module.

Handling deletion

All the components that (currently) support synchronization have deletion events. However, while message deletion (in the log) is an event that is kept forever (the message history is preserved), it is not the case for the other components.
Hence, those components have to record deletion events and only garbage-collect them after all other devices (present in the vector clock) have been informed.

Friend list synchronization

Profile synchronisation

Device-list synchronisation

Log synchronisation

Location sharing

Location sharing can be done in two ways: by providing (truncated-precision) coordinates directly, or by comparing locations using a zero-knowledge protocol.
The use-case for the zero-knowledge protocol is fairly straightforward: Alice doesn't want to transmit her location to her friend at all times, yet she wants to be notified when she is near a friend (and potentially decide to reveal her current location).
As such, a location-sharing message is either an explicit coordinates message, or a zero-knowledge message. Explicit coordinates are represented as follows:

- a *precision* **uint32** field, that describes the error margin (in meters); clients SHOULD let users specify an upper-bound on the precision;

- latitude and longitude **float** fields, describing WGS 84 coordinates, as provided by GPS[11].

The zero-knowledge protocol isn't specified yet.

Implementation notes:

- Distance between locations should be computed as great-circles distance. Using the haversine formula or a specialized library is recommended. That formula is numerically stable, and accurate to within 0.5% (due to the Earth not being a perfect sphere).

Friends recommendation

A/V chat

File transfers

Real-time synchronization

- Collaborative drawing/writing
- Based on operational transforms?

Temporary Thoughts

This section contains random thoughts that do not yet have a place in any other section. Use this for braindumps and as scratch pad to jot down ideas that require more thought.

Technologies

- Nano protobuf: https://github.com/google/protobuf/tree/master/javanano
- Compression algorithm:
  - Snappy
    - https://code.google.com/p/snappy/
    - https://github.com/dain/snappy
    - https://github.com/xerial/snappy-java
- Database:
  - SQLite: https://www.sqlite.org/
  - BDB (GNU AGPL may be problematic): http://en.wikipedia.org/wiki/Berkeley_DB
  - LMDB: http://symas.com/mdb/ (Requires us to write a JNI API for it)
  - LevelDB:
    - https://github.com/google/leveldb
    - https://github.com/dain/leveldb
    - https://github.com/fusesource/leveldbjni
- Text indexing

---

[11] For values up to 360, 24 bit significand yield approximately 10^{-5} precision (in both angular coordinates), which provides ~1m precision.

    - Lucene
        - Might be possible to trim it down
          will (probably) not solve the code size issue
    - roll-our-own
        - fuzzy matching on words is not too complex; Levenshtein automata approach are simple to parallelize (“MapReduce”-like tasks) using O(text length) time for generating automata, and O(output size) for merging.
        - unfortunately, multi-term search is much more complex (n-grams ?)
        - NLP preprocessing is also quite complex
          for instance, for sound-alikes or synonyms
- Futures
    - Measure actual cost of calling JNI (and blocking) vs. creating Future/Promise

Timeline

Notes:

- higher priorities are greater
- Deadline is ETA + 1 week.
- Items in blue are scheduled for after GSoC

Action item	Prio	ETA	Done	Alloc. time (h)
Modify HLAPI doc and interface for Sync event IDs	2	26-07		10
Design message hook system	2			10
Design, test, ScalaDoc & implementation				
ToxState/**ToxInstance** message queue	1	10-07		20
~~Replace use of **Serializable** by use of the proto2 message formats~~	~~2~~	~~15-07~~	~~03-07~~	~~15~~
ToxModule unloading	2	19-07		10
congestion handling	2	03-08		15
Internationalization + setting descriptions	3	14-08		10
message log queries	2			
Tests, ScalaDoc & implementation				
1:1 messaging	1	05-08		30
Group messaging	2			
Friend list	2			
File transfer				

AV chat				
Device synchronization	2			15
Implementation				
KeyValue[_,_]	1	03-07		19
LogStorage interface and spec (+ dep interfaces)	1			6
ToxState: better **ToxModule** state storage	3			6
LogStorage	1			
Full text search	2			
Protocol: documentation & proto2 format				
wire-format for synchronization events	2			
AV chat				
File transfer				
Message chunking	2	22-07		10
Test				
CLI client				

Past TODOs

Design settings mechanism for modules (will leverage state)	2015-06-02
Investigate SettingKey-style settings	2015-06-03
Write pisg script	2015-06-07
Benchmark cost of method call for file IO	2015-06-08
Merge **AbstractFile** in **FileLike**	2015-06-09
Implement and Test **Configurable** (unif. **ToxModule**s & **ToxState**)	2015-06-21

Also done:

- Relocate classes in the right packages
- Add lenses for module state manipulation
- Finish storage wrapper experiments
- Change (again!) **Stream**
- Design message modification/deletion
- Add support for N:M friend groups

- **GroupMessaging** scala file
- Define datatype for inline media