Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: 1 September 2025

D. Jesus
J. Leit達o
TaRDIS

28 February 2025

A Generic Framework for Building Dynamic Decentralized Systems (GFDS) draft-jesus-gfds-00

Abstract

Building and managing highly dynamic and heterogeneous decentralized systems can prove to be quite challenging due to the great complexity and scale of such environments. This document specifies a Generic Framework for Building Dynamic Decentralized Systems (GFDS), which composes a reference architecture and execution model for developing and managing these systems, while providing high-level abstractions to users.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 1 September 2025.

Copyright Notice

Table of Contents

1. Overview

Building decentralized systems is a complex and challenging task due to the inherent unpredictability and scale of such systems. These systems often consist of multiple nodes that may be located in different geographical regions and need to collaborate seamlessly to provide services or process data. The difficulty arises in managing issues like network latency, node failures, variable load distribution, or possibly node displacement in particular environments. Nevertheless, these systems need to remain highly available and responsive even when individual components experience failures, which requires robust fault tolerance and self-healing mechanisms.

While achieving data consistency, synchronization, and ensuring that all nodes have a coherent view of the system in a centralized system would be relatively easy, due to the existence of a centralized unit in charge of maintaining a "source of truth" and handling concurrency, in a decentralized system where nodes behave freely, and may at any moment have their own perspective of the system, such goal is not trivial. While centralized systems benefit from easier state management, they suffer from scalability limitations, single points of failure, and a lack of redundancy. Decentralized architectures, by their nature, mitigate some these issues but introduce added complexity due to the lack of centralized control.

This document focuses on decentralized systems and explores strategies to simplify their development and management.

Maintaining scalability and flexibility as the system evolves proves to be quite challenging. With growing demand, the system must be able to scale dynamically, by adding or removing nodes, without

disrupting ongoing operations, which requires sophisticated orchestration, auto-reconfiguration and adaptability. Moreover, dealing with the complexities of data consistency, synchronization, and ensuring that all nodes have a coherent view of the system state introduces a level of complexity that can be difficult to manage.

Developing frameworks and libraries for decentralized systems presents unique challenges. These frameworks must abstract the complexities of distributed architectures while maintaining flexibility and control for developers. While developers often need low-level access to aspects like network management, fault tolerance, and security, they also benefit from high-level abstractions that simplify common tasks such as inter-node communication and failure handling. Striking a balance between abstraction and control is crucial to ensure usability without compromising performance or system correctness.

Existing network libraries, such as [Lib2p], provide modular and flexible tools for building peer-to-peer networks but often have a steep learning curve and interoperability challenges. Moreover, while libp2p offers an extensive set of functionalities out of the box, these are not always easily adaptable to specific use cases, limiting flexibility for developers. Similarly, simulators like [PeerSim] allow rapid prototyping and testing of distributed systems but fail to accurately model real-world execution environments, leading to a _reality gap_ between simulation and real-world deployment.

To address these challenges, we propose the *Generic Framework for Building Dynamic Decentralized Systems (GFDS)*. GFDS provides a comprehensive set of tools, abstractions, and best practices to help developers design, deploy, and manage decentralized applications that are dynamic, resilient, scalable, and fault-tolerant.

The framework is composed of an execution model, which details and controls the life-cycle of protocols (the base unit in the framework), and an architecture detailing a set of managers to handle the different components and their interactions while providing common APIs for enabling inter-protocol communication between the different elements in the stack.

1.1. Document Structure

This document describes a Generic Framework for Building Dynamic Decentralized Systems and its structured as follows:

* Section 3 describes _protocols_-the base unit of interaction within the framework,

* Section 4 describes the framework architecture, its different components and their interaction,

* Section 5 details the execution model of the framework and the life cycle of protocols,

* Section 6 describes the programming interfaces offered by the framework to handle interaction with the application level, inter-protocol interactions and inter-node communication,

* Section 7 provides real-world scenarios and examples.

2. Conventions and Definitions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

The following abbreviations are used throughout the document:

* API: Application Programming Interface

* GFDS: Generic Framework for Building Dynamic Decentralized Systems

* CAS: Central Authentication Service

3. Protocol Anatomy

The main unit of interaction in the framework is the _protocol_. Protocols are assigned an unique identifier and embed the logic implemented by the developer and use the abstractions provided by the framework to interact with other protocols being executed locally, as well as handling communication with other nodes. A process may execute an arbitrary number of protocols concurrently at any given time, and protocols can communicate with each other to cooperate and delegate tasks. Moreover, as it is very common in distributed protocols to capture certain behaviours, protocols can execute actions periodically through the use of timers (e.g., execute a garbage collection function).

Each protocol is composed of the following concepts that dictate the anatomy and life cycle of a protocol:

* *state* describes the inner state of the protocol, containing the necessary data and data structures to ensure its correct behaviour. The state should be initialized in a specialized

init function and can be mutated through interaction with other local protocols, periodic timers that alter the state of the protocol and, finally, communication with other nodes running the same protocol on different machines;

* *timer handlers* are meant to execute periodic or scheduled tasks. When the timer expires, a handler is executed with user-defined logic. Additionally, protocols can cancel timers if they are no longer relevant (e.g., a timer set up by the lack of an acknowledgement can be cancelled if the acknowledged arrived in the meantime);

* *inter-protocol handlers* are in charge of managing communication between protocols running on the same machine. These handlers are divided into two categories: one-to-one _requests/replies_ and one-to-many _notifications_;

* *communication handlers* which manage incoming and outgoing communications between different nodes. At the protocol level such information arrives in the form of messages;

* *discovery and configuration handlers* which deal with peer discovery and configuration.

With most of the complexity abstracted by the framework, the developer can focus on the logic of the protocol, without having to worry about the low-level aspects associated with building large-scale systems (e.g., dealing with faults in the network layer).

3.1. Protocol Initialization

As described previously, each protocol should implement a special _init_ function. This function is meant to be executed exactly one time during the life-cycle of the protocol, and has four main purposes:

1. Initialize the protocol’s state, namely, its control variables, local data structures, etc.,

2. Setup self-configuration and adaptive handlers,

3. Choose the preferred transport preferences,

4. Register the different handlers (i.e., timer, inter-protocol, discovery and inter-node communication handlers).

A typical protocol initialization would be structured as follows:

```
init(properties):
  // 1 - Setup initial state
 ...

 // 2 - Setup discovery and configuration
 registerSelfConfiguration(properties, uponSelfConfig)

 registerAdaptiveConfiguration(properties, uponAdaptive)

 // 3 - Transport preferences
 preferences = {reliable, secure}
 registerCommunicationPreferences(preferences)

 // 4 - Handlers
 registerRequestHandler(BroadcastRequest, uponBroadcastRequest)
 ...

 registerReplyHandler(DeliverReply, uponDeliverReply)
 ...

 registerTimerHandler(GarbageCollectTimer, uponGarbageCollectTimer)
 ...

 registerDiscoveryHandler(uponParticipant)
 ...

 subscribeNotificationHandler(NeighborUpNotification, uponNeighborUpNotification)
 ...

 registerCommunicationHandler(BroadcastMessage, uponBroadcastMessage)
 ...
```

3.2. Handlers

Handlers operate as callback functions. During the initialization of a protocol, the developer is in charge of registering the handlers associated with the protocol and their respective callbacks. This guarantees that when an event is dispatched to the protocol by the framework (e.g., due to a request arriving from another protocol, a message in the network, etc.), the respective callback function associated with the event is rightfully triggered.

In this section, we will focus on how handlers work at the protocol level. Further details regarding the architecture and internal life-cycle management of handlers by the runtime are provided in Section 4.

Each type of handler has specific information according to their different nature, but, their registration mostly consists of specifying two fields: 1) a type encapsulating the incoming information arriving at the handler, and 2) the _callback function_ to be triggered when an event arrives. This can be depicted as follows, by analyzing a transmission example:

```
//Types definition
def BroadcastMessage {
 byte [] data,
 int hopCount,
 Peer origin

 //Omitting serializer
 ...
}

init(properties):
 ...
 registerCommunicationHandler(BroadcastMessage, uponBroadcastMessage)
 ...

uponBroadcastMessage(BroadcastMessage: msg, Peer: sender):
 //Handle data
```

Hence, at the protocol level, the developer solely worries about defining the correct types associated with each handler and the corresponding callback functions to manage event arrivals.

3.2.1. Timer-Handlers

Timer-handlers allow the developer to set up tasks that should be executed periodically or after a certain amount of time has passed. This is especially helpful to model certain aspects of distributed systems, such as cases where some kind of verification has to be done periodically to ensure consistency, or to time-bound actions in certain aspects of a protocol.

Thus, we propose the following API to interact with timers at the protocol level:

```
registerTimerHandler(TimerType: timerType, TimerHandler: function)

setupTimer(Timer: timer, long: timeout) -> long

setupPeriodicTimer(Timer: timer, long: first, long: period) -> long

cancelTimer(long: timerID)
```

* _registerTimerHandler_, as the name implies, should only be invoked during protocol initialization and ensures that all the timers are correctly registered. The function receives the timer type (which encapsulates the information to be passed on as an argument) and the callback function to be triggered.

* _setupTimer_ sets up a timer that will be triggered after a timeout. After the timeout has passed the timer will cease to exist. The function returns a unique timerID that can be used to cancel the timer if needed.

* _setupPeriodicSetupTimer_ sets up a timer that will be triggered periodically (as indicated by the _period_ parameter). It is possible to specify a timeout until the first trigger with the _first_ parameter. Akin to _setupTimer_, this function also returns a unique timerID.

* _cancelTimer_ allows canceling a timer by passing its unique timerID. This can be extremely helpful in cases where a periodic action is no longer needed, or, for example, if a timeout is no longer required due to the arrival of data. Protocols should be able to set up and cancel timers during their life-cycle.

Below, we depict a simple example of synchronous communication among nodes:

```
//Types definition
def GarbageCollectionTimer(){
 long interval
}

def AckTimer(){
 string msgID
}

//Omitting state for clarity

init(properties):
 ...
 registerTimerHandler(GarbageCollectionTimer, uponGarbageCollectTimer)
 timer = GarbageCollectionTimer(1m)
 setupPeriodicTimer(timer, 10s, 100s);

 registerTimerHandler(AckTimer, uponAckTimer)
 ...

uponBroadcastMessage(BroadcastMessage: message, Peer: sender, Transport: transport):
 this.blocks.add(msg)
 setupTimer(AckTimer(msg.id), 10s)

//Executed every 100 seconds
uponGarbageCollectTimer(GarbageCollectTimer: timer):
 this.blocks.remove(msg -> Time.now - msg.timestamp > timer.interval)

//Executed 10s after setupTimer invocation
uponAckTimer(GarbageCollectTimer: timer):
 if (!this.acks.contains(timer.msgID))
 // run fault checks
```

3.2.2. Inter-Protocol Handlers

Inter-protocol handlers govern the interaction among protocols being executed in the same machine. Usually, each machine in a distributed system runs a stack of protocols (i.e., protocols for membership, propagation, storage, etc.) and they interact with each other to achieve composability and describe more complex behaviours. A simple example of this can be seen in a propagation protocol (e.g., multicast) that makes use of the neighbors provided by a lower-level protocol in charge of membership on top of an overlay network.

As mentioned previously, these types of handlers are divided into two categories: _request/reply_ handlers, meant for one-to-one interaction among protocols, and _notifications_ with one-to-many semantics. The first is especially useful in scenarios where a

protocol intends to request the execution of a certain task to another protocol (e.g., the storage protocol asking for the transmission of an object to the protocol in charge of propagation) and expects to receive a reply after the said task is completed. On the other hand, the second enables protocols to subscribe to notifications, so that they are notified when a node triggers a new notification (e.g., a message arrived in a pub/sub protocol and every protocol that is interested in it should be notified).

Regarding requests and their respective replies, we propose the following API:

```
registerRequestHandler(RequestType: requestType, RequestHandler: function)

sendRequest(Request: request, Protocol: destProtocol)
sendRequest(Request: request)

registerReplyHandler(ReplyType: replyType, ReplyHandler: function)

sendReply(Reply: reply, Protocol: destProtocol)
```

* _registerRequestHandler_ should only be invoked in the _init_ function and ensures that all the requests are correctly registered. The function receives the request type (which encapsulates the information to be passed on as an argument to the handler) and the callback function to be triggered.

* _sendRequest_ allows a protocol to send a request to another protocol. The function receives an instance of the request and the corresponding destination protocol. A second variant doesn’t receive the destination protocol and is solely used to request tasks to the framework (i.e., Section 3.2.3)

* _registerReplyHandler_ is akin to _registerRequestHandler_, but for replies.

* _sendReply_ behaves similarly to _sendRequest_, thus allowing protocols to issue replies. Is to be noted that replies are usually issued as a response to requests issued by other protocols.

```
subscribeNotificationHandler(NotificationType: notificationType, NotificationHandler: fun
ction)

triggerNotification(Notification: notification)
```

* _subscribeNotificationHandler_ should only be invoked in the _init_ function and guarantees the correct subscription of the protocol to notifications. The function receives the notification type, and a callback to handle incoming notifications.

* _triggerNotification_ triggers a notification to be propagated to all protocols subscribed to it. The function receives an instance of the notification as an argument.

This semantic allows developers to fine-grain the specification of their protocols by using a declarative language that allows them to focus their efforts on implementing the algorithm logic.

We provide a brief example of how this works, with a simple broadcast application:

```
//Types definition
def BroadcastRequest(){
 byte [] data
}

def DeliverReply(){
 byte [] data
}

def NeighborUpNotification(){
 Peer: peer
}

//Protocol state
state = {
 neighbors: Set<Peer>,
 myself: Peer
}

init(properties):
 ...
 registerRequestHandler(PingRequest,uponPingRequest)
 subscribeNotification(NeighborUpNotification,uponNeighborUpNotification)
 ...

uponPingRequest(PingRequest: request, Protocol: sourceProto):
 msg = BroadcastMessage(request.msg, this.myself)
 this.neighbors.forEach(peer -> sendMessage(msg, peer))
 sendReply(DeliverReply(request.msg), sourceProto)

uponNeighborUpNotification(NeighborUpNotification: notification):
 this.neighbors.add(notification.peer)
```

Note that, in the previous example, there was no registration of DeliverReply. This is because only the protocols wishing to receive that information, namely the protocol that sent the request, are obliged to register such handlers.

3.2.3. Discovery Handlers

Discovery handlers are in charge of facilitating the discovery of nodes within the system, which is particularly relevant in decentralized systems where nodes need to connect dynamically without a central authority.

At the protocol level, the framework should provide ways of informing the protocol when new participants are discovered:

registerDiscoveryHandler(DiscoveryHandler: function)

* The _registerDiscoveryHandler_ function should only be invoked within the init function and is responsible for registering the handler that receives discovery events from the event manager. This function processes a notification containing information about a node (e.g., its identifier, status, etc.). Unlike other events that may have multiple handlers (e.g., requests, notifications), there should be only one active discovery handler per protocol.

While the framework probes the system during initialization or at regular intervals to detect new nodes, protocols may require real-time access to the latest set of participants or may need to announce their presence to other nodes. To facilitate this, each protocol declares its intent to either discover other nodes or be discoverable through a service name.

A service name serves as a label for the protocol and represents its offered functionalities. For example, a broadcast protocol might register itself with the service name "floodPropagation", signaling its role in message dissemination.

To enable this functionality, the sendRequest construct is used to instruct the discovery manager to propagate this information.

Thus, the framework provides developers with two mechanisms to report and retrieve such information:

sendRequest(Request: RequestProbe(String: serviceName))

sendRequest(Request: RequestAnnouncement(String: serviceName))

* _sendRequest(RequestProbe(serviceName))_ sends a specialized request, RequestProbe, which queries the system for other nodes that offer the specified service.

* _sendRequest(RequestAnnouncement(serviceName))_ sends a specialized request, RequestAnnouncement, which announces the node as a provider of the specified service.

It is important to note that while discovery is relevant in some protocols, it remains an optional module that a protocol may choose to use or ignore by registering accordingly during initialization. For example, in a system with a large protocol stack, one protocol (e.g., a membership protocol) may handle discovery, while others receive this information through inter-protocol interactions. If a protocol does not require discovery notifications, it simply does not register the corresponding handlers.

We can see the discovery mechanisms in action in the following example:

```
//Types definition
def AnnounceTimer {
  long interval
}

def BroadcastRequest{
 byte [] data
}

def DeliverReply(){
 byte [] data
}

//Protocol state
state = {
 neighbors: Set<Peer>,
 myself: Peer
}

init(properties):
 ...
 registerDiscoveryHandler(uponDiscovery)

 registerRequestHandler(PingRequest,uponPingRequest)
 subscribeNotification(NeighborUpNotification,uponNeighborUpNotification)

 registerTimerHandler(AnnounceTimer, uponAnnounceTimer)
 timer = AnnounceTimer(1m)
 setupPeriodicTimer(timer, 10s, 100s);
 ...

uponAnnounceTimer(AnnounceTimer: timer):
  request = RequestAnnouncement("pingPong")
  sendRequest(request)

uponPingRequest(PingRequest: request, Protocol: sourceProto):
 msg = BroadcastMessage(request.msg, this.myself)
 this.neighbors.forEach(peer -> sendMessage(msg, peer))
 sendReply(DeliverReply(request.msg), sourceProto)

uponDiscovery(DiscoveryNotification: notification):
 log("Discovery Method: {}", notification.serviceName)
 this.neighbors.add(notification.peer)
```

3.2.4. Configuration Handlers

Self-configuration and adaptability refer to the ability of a component to automatically configure itself and change based on its environment, without requiring manual intervention. The first guides to the fact that, when a node initiates, it should be able to self-configure to join the system as an active node, while the second handles the adaptation of a node during runtime to match the current state of the system.

To achieve this, we suggest the following design: Each protocol should define a set of state parameters that it considers to be _AutoConfigurable_ and _Adaptive_. This way, during registration, the protocol will pass this information to the framework and the framework will handle the proper initialization and update of these parameters during execution. Parameters tagged with _AutoConfigurable_ are meant to be configured during initialization by obtaining the configuration of other nodes already present in the system, while _Adaptive_ parameters are meant to be properly managed and adapted (i.e., reconfigured) by the framework during runtime.

With this said, at the protocol level, we propose the following functionalities and abstractions:

registerSelfConfiguration(Properties: properties, ConfigurationHandler: function)

* _registerSelfConfiguration_ indicates how the protocol will setup its initial configuration. The first argument, properties, encapsulates the parameters the node wishes to tag as _AutoConfigurable_ so that they are properly set up by the framework. Finally, the last argument receives a handler to be executed as a callback when an event regarding configuration arrives at the protocol.

registerAdaptiveConfiguration(Properties: properties, AdaptiveHandler: function)

* _registerAdaptiveConfiguration_ indicates how the framework will handle autonomic updates of the protocol parameters. The first argument, properties, encapsulates the parameters the node wishes to tag as _Adaptive_. The last argument receives a handler to be executed as a callback when an event regarding the reconfiguration of parameters arrives at the protocol.

The following example contains a simple application embedding these abstractions:

```
   //Types definition
   PayloadMessage {
    string payload
   }

   PublishRequest {
    string topic,
    string msg
   }

   state = {
    @AutoConfigurable
    long garbabeCollectTimer,

    @Adaptive
    long fanout,

    @AutoConfigurable
    @Adaptive
    long ttl,

    Set<PayloadMessage> data
   }

   init():
    //Omitting state initialization and handlers for simplicity

    ...
    registerSelfConfiguration(properties, uponSelfConfig)

    registerAdaptiveConfiguration( properties, uponAdaptive)
    ...

   uponPublishRequest(PublishRequest: request, Protocol: sourceProto):
    payload = PayloadMessage(request.msg)
    data.add(request.msg)
    sendMessage(payload, p)

   uponSelfConfig(Map<Parameter, Config>: parameters):
    for p in state as AutoConfigurable:
     p = parameters.get(p)

   uponAdaptive(Map<Parameter, Config>: parameters):
    for p in state as Adaptive:
     p = parameters.get(p)
```

3.2.5. Communication Handlers

Finally, while inter-protocol handlers are in charge of dealing with the interaction of protocols being executed in the same node, communication handlers manage data arriving from different nodes, and thus, inter-node communication.

With a great deal of complexity being abstracted by the framework, at the protocol level, we are concerned with providing a generic API to the developer which allows a node to send and receive information from other nodes without having to deal with the intricacies of managing such connections. Namely, we extend our framework to regard different technologies, ranging from the typical network-based protocols like TCP [RFC9293], UDP [RFC768], QUIC [RFC9000], etc., to short-range technologies such as Bluetooth Low Energy (BLE) [RFC7668] (more details are provided in the following sections).

To achieve this and provide a more user-friendly interface for newcomers, we propose a keyword-based approach instead of requiring developers to explicitly specify the transport protocols they wish to use for communication. In this approach, keywords represent the desired characteristics and guarantees of communication between nodes. These keywords include properties such as reliable or unreliable, secure or unsecure, lightweight, connection-oriented or connectionless, among others.

Internally, as explained in detail in later sections, the framework maps the specified set of keywords to an appropriate combination of transport protocols that satisfy those requirements.

Communication in GFDS is fundamentally message-based. Protocols interact by sending and receiving discrete messages, ensuring reliable and scalable communication while maintaining flexibility and compatibility with a wide range of transport protocols.

Thus, at the protocol level, we suggest the following abstractions:

```
registerCommunicationPreferences(Keywords[]: preferences, boolean?: parallelize)
```

* _registerCommunicationPreferences_ allows a developer to specify its preferences regarding communication (i.e., Section 4.5.1). Preferences are passed to functions as keyword arguments. If a conflict arises that prevents a valid match (e.g., specifying both secure and unsecure), the framework notifies the developer with an appropriate error message. If no preference set is provided, the framework defaults to its predefined settings. Additionally, the function accepts a boolean parameter that determines whether the framework should parallelize the opening of different transport

paths during startup. If the developer passes _true_, the framework initializes all transport paths in parallel when starting the protocol. If _false_, the framework sequentially opens the interfaces, prioritizing those at the top of the list and only initializing additional ones if the primary interfaces fail. The first approach reduces overhead during connection loss, whereas the second optimizes resource consumption. If no argument is provided, the framework defaults to its predefined configuration.

registerCommunicationHandler(MessageType: msg, MessageHandler: function)

* _registerCommunicationHandler_ should only be invoked in the _init_ function and ensures that all the transmissions/ communication arriving at the node from other nodes are properly handled. The function receives the message type (which encapsulates the information to be passed on as an argument to the handler) and the callback function to be triggered when a message arrives. It is worth noting that since these message types are meant to be sent/received through the network and other transmission media, they should implement the proper serializers and deserializers.

sendMessage(Message: msg, Peer: destination)

sendMessage(Message: msg, Peer: destination, Properties: props)

sendMessage(Message: msg, Peer: destination, String: alias, Properties: props)

sendMessage can be invoked in three distinct ways:

* A simple, default version that only requires the message and its destination as arguments. It sends the message using the preferences specified in the initializer.

* A more specialized version allows the developer to specify the message along with a set of optional properties (props). The purpose of this properties parameter is to enhance expressiveness, particularly for transport paths that require additional metadata (e.g., MQTT, where a topic must be specified for transmission).

* Finally, the last variant enables a node to send a message using a specific alias. In other words, a protocol can transmit a message while presenting an identity different from its default.

These abstractions allow protocols to have some control over how data is sent to other nodes while hiding the complexity of dealing with such. On the other hand, if a protocol is only concerned with

guaranteeing that information flows in and out of its host node, it can make use of the more simple abstractions following the default configuration embedded into the framework.

The following example illustrates what was presented:

```
//Types definition
PayloadMessage {
 string payload
}

AckMessage {
 long timestamp
}

PublishRequest {
 string topic,
 string msg
}

init():
 //Omitting request and notification registration for simplicity

 preferences = {unreliable, lightweight}
 registerCommunicationPreferences(preferences)

 registerCommunicationHandler(PayloadMessage, uponPayloadMessage)
 ...

uponPublishRequest(PublishRequest: request, Protocol: sourceProto):
 props = {characteristic: "topic"}
 payload = PayloadMessage(request.msg)
 sendMessage(payload, p, props)

uponPayloadMessage(PayloadMessage: message, Peer: sender, t: registerCommunicationPrefere
nces):
 msg = AckMessage(Time.now)
 sendMessage(msg, sender)
```

The different abstractions related to interactions with multiple protocols of different nature (i.e., subscribing to a topic in MQTT, listening to a characteristic in BLE, etc.) are still under development.

3.3. Procedures

Beyond the information defined above that specifies the use of the abstractions provided by the framework, protocols may also need to execute procedures. Procedures can range from simple calculations on specific parameters to updates on the local state. Thus, protocols should be allowed to declare an arbitrary number of procedures and invoke them inside the different handlers defined in the _init_ function. The declaration of a procedure should be as follows:

```
procedureName(args):
 //Perform computations on args

 return result;
```

Possible usage of a procedure within a protocol:

```
calcIntersection (set1, set2):
 return set1 ^ set2;

uponSetMessage(SetMessage: message, Peer: origin):
 intersect = calcIntersection(this.set, msg.set)
 this.set = intersect
 sendReply(SetUpdateReply(this.set), destProto)
```

3.4. Example

In this section, we will provide a simple but complete example of a protocol definition using the constructions stated above. Some definitions are omitted for clarity and succinctness.

Types Definition

```
   def BroadcastRequest{
    byte [] data
   }

   def BroadcastNotification {
    byte [] data
   }

   def NeighborDownNotification {
    Peer neighbor
   }

   def BroadcastMessage {
    byte [] data,
    long timestamp,
    short ttl

    serializer(out):
    out.writeByteArray(data)
    out.writeLong(timestamp)
    out.writeShort(ttl)

    deserializer(in):
    data = in.readByteArray()
    timestamp = in.readLong(in)
    ttl=in.readShort(ttl)
    return BroadcastMessage(data, timestamp, ttl)
   }

   def GarbageCollectionTimer {
    long interval
   }
```

Protocol

```
state = {
 dataSet : Set<BroadcastMessage>,
 neighbors: Set<Peer>,
 protocolApp: Protocol

 @AutoConfigurable
 @Adaptive
 long ttl,
}
```

```
init(properties):
 this.dataSet = Set<BroadcastMessage>()
 this.neighbors = Set<Peer>()
 this.app = properties.protocol
 //ttl will be configured by the framework

 registerDiscoveryHandler(uponDiscovery)

 registerSelfConfiguration(properties, uponSelfConfig)

 registerAdaptiveConfiguration(properties, uponAdaptive)

 preferences = {reliable, connectionOriented}
 registerCommunicationPreferences(preferences)

 registerRequestHandler(BroadcastRequest, uponBroadcastRequest)
 subscribeNotification(NeighborDownNotification,uponNeighborUpNotification)

 registerTimerHandler(GarbageCollectionTimer, uponGarbageCollectionTimer)
 setupPeriodicTimer(GarbageCollectionTimer(properties.interval), uponGarbageCollectionTim
er)

 registerCommunicationHandler(BroadcastMessage, uponBroadcastMessage)

// Request/Reply Handlers
uponBroadcastRequest(BroadcastRequest: request, Protocol: sourceProtocol):
 BroadcastMessage = BroadcastMessage(request.data, Time.now)
 deliver(BroadcastMessage)
 propagate(this.neighbors, BroadcastMessage)

// Timer Handlers
uponGarbageCollectionTimer(GarbageCollectionTimer: timer):
 this.dataSet.removeIf(data -> Time.now - data.timestamp >
 timer.interval && data.ttl >= state.ttl)

// Notification Handlers
uponNeighborDownNotification(NeighborDownNotification: notification):
 this.neighbors.remove(notification.neighbor)

// Discovery and Configuration
uponDiscovery(DiscoveryNotification: notification):
 this.neighbors.add(event.peer)

uponSelfConfig(Map<Parameter, Config> parameters):
 for p in state as AutoConfigurable:
 p = parameters.get(p)

uponAdaptive(Map<Parameter, Config> parameters):
 for p in state as Adaptive:
```

```
 p = parameters.get(p)

// Communication Handlers
uponBroadcastMessage(BroadcastMessage: msg, Peer: sender):
 msg.ttl++
 deliver(msg)
 propagate(this.neighbors - sender, msg)

// Procedures
deliver(BroadcastMessage: msg):
 if(!this.dataSet.contains(msg)):
 this.dataSet.add(msg)

 notification = BroadcastNotification(msg.data)
 triggerNotification(notification)

propagate(Set<Peer> destinations, BroadcastMessage: msg):
 destinations.forEach(n -> sendMessage(msg, n))
```

4. Architecture

GFDS aims to simplify the development and management of highly dynamic decentralized systems. To achieve this, we propose an architecture where great part of the complexity is hidden underneath different layers of abstractions. The layers are in charge of different aspects of the system, ranging from inter-protocol event dispatching, to guaranteeing secure communication among nodes. The architecture overview is depicted in the following diagram:

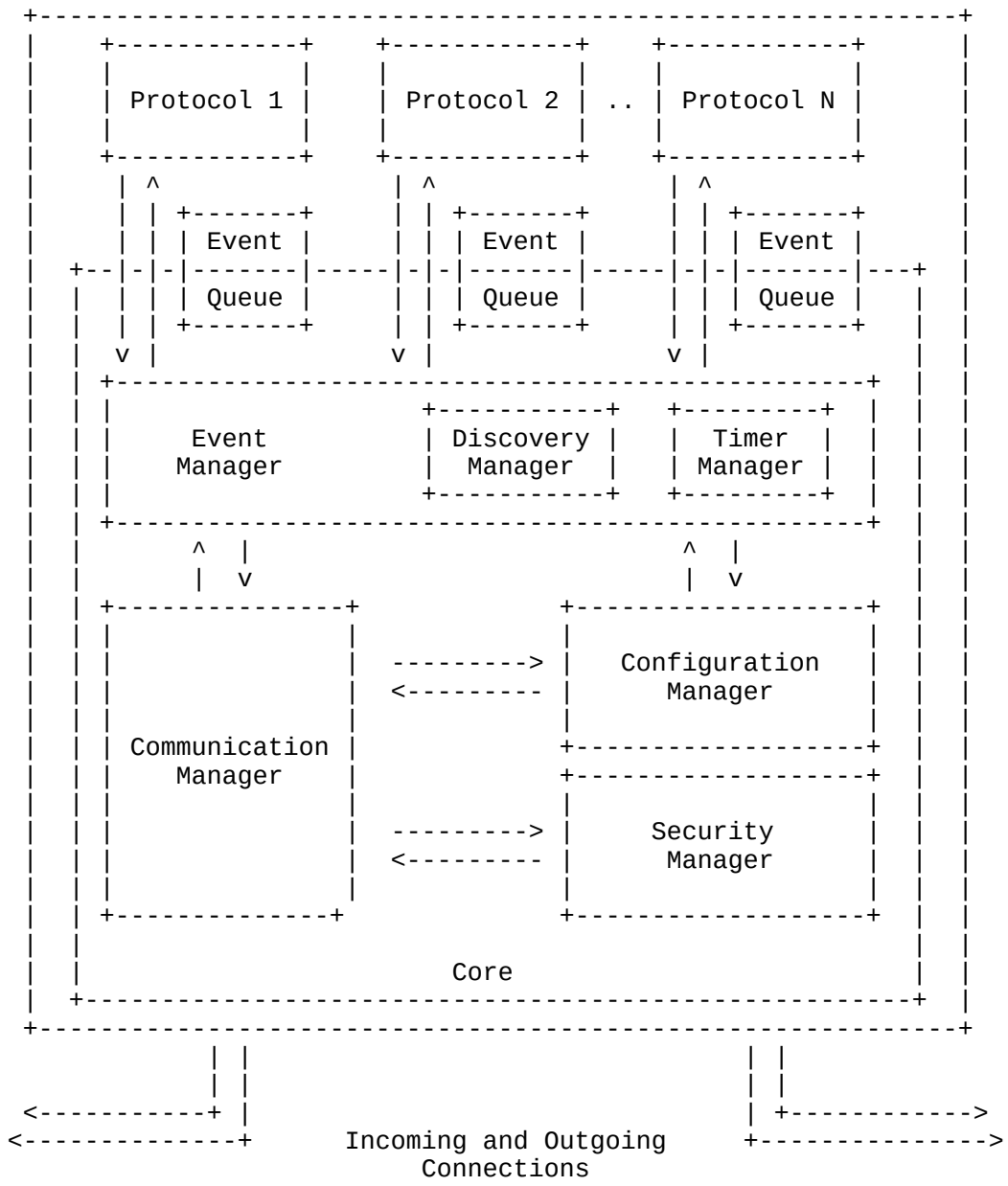

Figure 1: Framework architecture

As stated previously, the base unit of interaction is the protocol. Protocols "live" on top of the stack and developers interact with the framework by specifying protocols, implementing their logic and

defining the proper handlers to receive events (i.e., timers, inter-protocol, etc.) from the framework. This means that, to a developer, the interaction with the other layers is virtually non-existent. The event manager is in charge of dispatching events to the correct protocols, managing timers, and communicating with the communication manager and configuration manager. Finally, the security manager is responsible for identity management and ensuring secure communication among nodes. All of this is encapsulated in what we call the _core_.

The _core_ is a centralized component that coordinates the execution of the different protocols through an event queue. It is separated into different elements, as depicted in Figure 1, each responsible for a different task. In the following subsections, we specify the different details of each component.In the following subsections, we specify the different details of each component.

This design allows the framework to make a clear separation of concerns regarding what a developer is to interact with, and what is managed internally.

4.1. Protocols

Developers interact with the framework by specifying protocols. Protocols have access to common abstractions and APIs that allow to make use of the tools provided by the framework (i.e., explained in Section 3).

Protocols send/receive events to/from the event manager. Each of these events contains information about their type (i.e., request, reply, notification, etc.) and the necessary information to be correctly interpreted by the proper destination (e.g., a request event contains the source protocol to which the destination can reply). Thus, each protocol can be seen as a state machine whose state evolves by receiving and processing events.

Each protocol has an event queue that orders and serializes incoming events. Each protocol should control its life-cycle by advancing "time" in a manner it sees fit (i.e., per clock tick, periodically, etc.). In other words, protocols are in charge of pooling their respective queue and processing events by their order of arrival by matching the respective event with the registered handler. Events emitted by the protocol are sent to the event manager who handles their proper processing and forwarding to the correct destination.

From the developer’s point of view, a protocol is responsible for defining the callbacks used to process the different events in the queue and interact with other components in the framework by sending those events. This means that developers only worry about

registering the proper callbacks for each type and implementing their logic and interact with other protocols by sending events, while letting the framework handle event management through interaction with the queue.

Although each protocol contains its own event queue, these are mediated by the event manager, the component responsible for delivering events to each protocol queue and forwarding the events issued by protocols to their correct destination.

The following diagram depicts a protocol overview, divided by two views. A _developer view_ specifies the environment and elements to be handled by the developer (i.e., setup callbacks, manage protocol state, etc.), and an _internal view_ which is managed by the framework. With this design, developers only worry about creating the respective handlers, and the framework handles that events are properly dispatched to other components in the framework and that the protocol receives events accordingly.

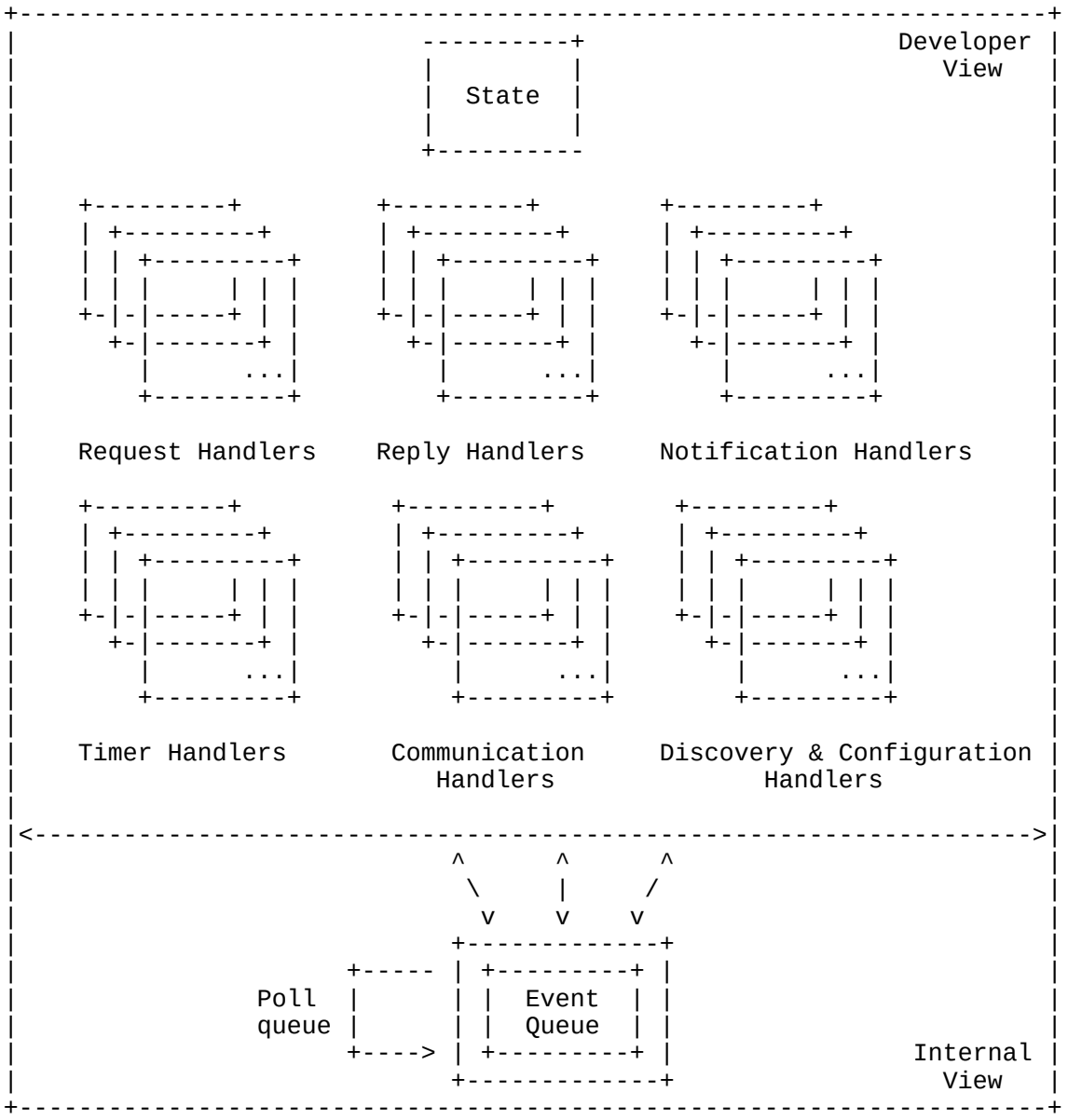

Figure 2: Protocol overview

4.2. Event Manager

The event manager is one of the most important components of the framework and the bridge between developers (i.e., protocols) and the framework's inner workings. It is responsible for the following tasks: exchanging events between respective protocols (i.e., _inter-protocol_), forwarding events coming from the communication and configuration manager to the right protocols, and handling discovery and timers.

Each protocol has a unique identifier associated with it (i.e., a random identifier issued by the framework or passed on as an argument during initialization). During protocol registration, the event manager stores the different identifiers in order to forward events to their respective destination. Moreover, since notifications are meant to ensure one-to-many semantics, the event manager maintains a mapping between notifications and subscriptions of the corresponding protocols. With this information, we are able to guarantee the proper dispatching of events to their destination.

In the upcoming subsections, we will detail the different subtasks of the event manager.

4.2.1. Inter Protocol Interaction

Inter-protocol interactions-communication between protocols in the same machine-is done through requests, replies and notifications. While requests and replies offer one-to-one semantics, notifications provide one-to-many semantics with the use of subscriptions.

Requests, and their replies, receive a destination protocol, which allows the event manager to insert these events in the appropriate protocol queues, by accessing the set of registered protocols. This simple but effective design ensures that protocols can interact with each other in an efficient way, in order to build complex behaviours through the sharing of information.

In contrast, notification triggers are meant to be delivered to all protocols that are interested in it. To achieve this, the event manager scans its mapping of notifications to subscriptions and transmits such information to all.

4.2.2. Timer Manager

The timer manager is a small module that ensures that tasks are triggered at specific time intervals or after a certain amount of time has passed.

Protocols create timers by either invoking _setupTimer_ or _setupPeriodicTimer_. When doing this, an event is sent by the corresponding protocol to the event manager, which in turn passes it along to the timer manager. The timer manager stores all the timers issued by the different protocols and is in charge of advancing "time" (e.g., clock ticks). When the timer manager detects that a timer has come to its conclusion, it informs the respective protocol that registered it by placing an event in its queue. It is worth noting that regular timers and periodic timers work in the same manner internally, with the only difference being that one is removed from the timer manager after it's finished, while the other remains active until the end of the execution of the program, or its cancellation.

Each timer has a unique identifier associated with it at the moment of creation (i.e., Section 3.2.1). Protocols can use this identifier to cancel timers and remove them from the timer manager. This can be achieved by invoking a cancel timer event from the protocol to the event manager.

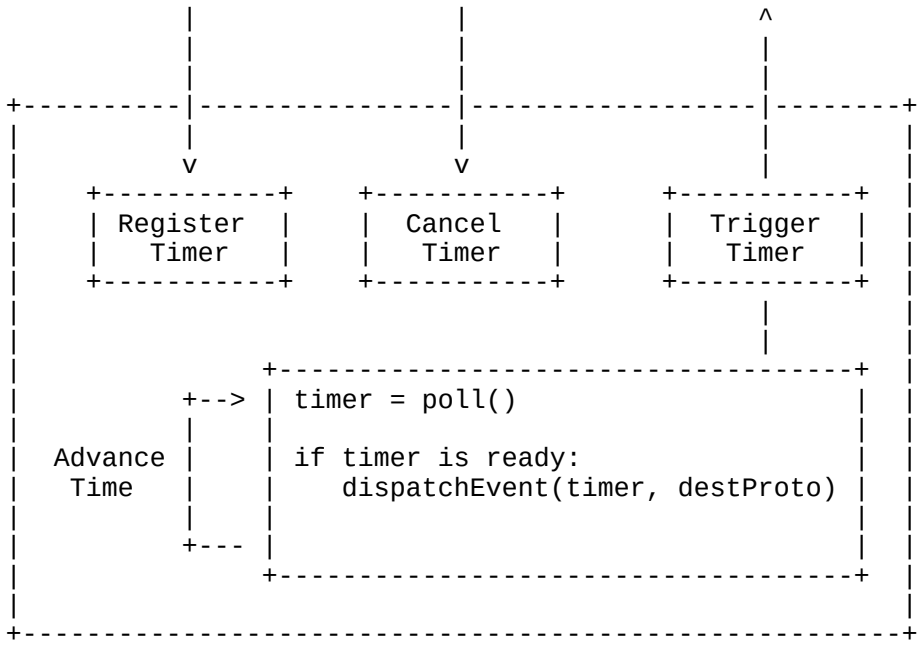

Figure 3: Timer manager overview

4.2.3. Communication and Configuration

The event manager acts as an intermediary between the protocol layer and both the communication and configuration manager. The communication manager handles connections with other nodes, while the configuration manager is responsible for self-configuration and adaptability during runtime. Consequently, the event manager has to guarantee that these events arrive at the protocols that are interested in them.

During initialization, each protocol registers handlers for external events they wish to listen to (i.e., transmissions, configuration updates, etc.). Internally, the framework maps each event to their respective consumers so that they are rightfully informed when such events happen. Subsequently, when, for instance, a message arrives from another node, the event manager can successfully generate an event to the correct protocol (or protocols) that registered it during initialization, by placing them in the protocol event queue for later processing.

Once again, at the protocol level, the protocol should only be concerned with defining the proper handlers and the framework will guarantee that information is correctly forwarded to its destination and that protocols will be rightfully apprized when an event they registered is triggered.

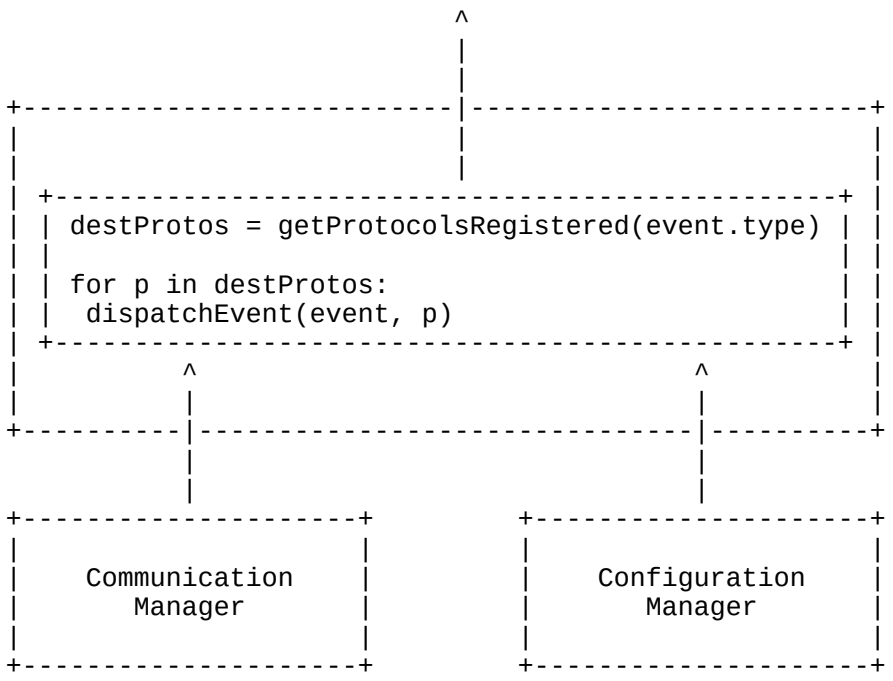

Figure 4: Overview of communication and configuration flow.

In Section 4.5 and Section 4.3 a detailed explanation is provided regarding event generation in this context.

4.2.4. Discovery Manager

In the context of decentralized systems, a frequent problem is that of identifying a contact-a node already present in the system-that can help a new node join the network. While in small systems this can be easily achieved by storing information about the different participants, in a complex and highly distributed environment this solution is no longer feasible.

The discovery manager is a subcomponent that facilitates the discovery and management of peers. During initialization, the framework provides an optional mechanism for registering the desired discovery methods. Based on this information, the discovery manager will search for nodes using the specified methods during startup. When new nodes are discovered, protocols receive notifications through the handler registered via _registerDiscoveryHandler_ (i.e., Section 3.2.3).

registerDiscoveryMethods(DiscoveryMethod []: methods)

* _registerDiscoveryMethods_ states the preferences of a developer regarding discovery methods. In other words, the developer can choose how the framework discovers other nodes by specifying which technologies and protocols are used to achieve this (e.g., mDNS [RFC6762], multicast etc.).

```
main(config):
 ...
 // Setup discovery
 discoveryMethods = {mDNS}
 framework.registerDiscoveryMethods(discoveryMethods)
 ...
```

Ex 1.: Discovery Method Registration

Moreover, when a explicit request for an announcement or probe is received, the respective action is triggered by the discovery manager. Namely, announce that the node sending the message is providing a certain service, or probe the system for nodes that offer a specific service.

The framework should specify a list of methods available for this purpose, for instance, mDNS, DHT, etc., or analogously, allow the definition of lists of contact nodes per protocol, for bootstrapping (e.g., in a peer-to-peer online video game there may exist well known players which could be used as contact points).

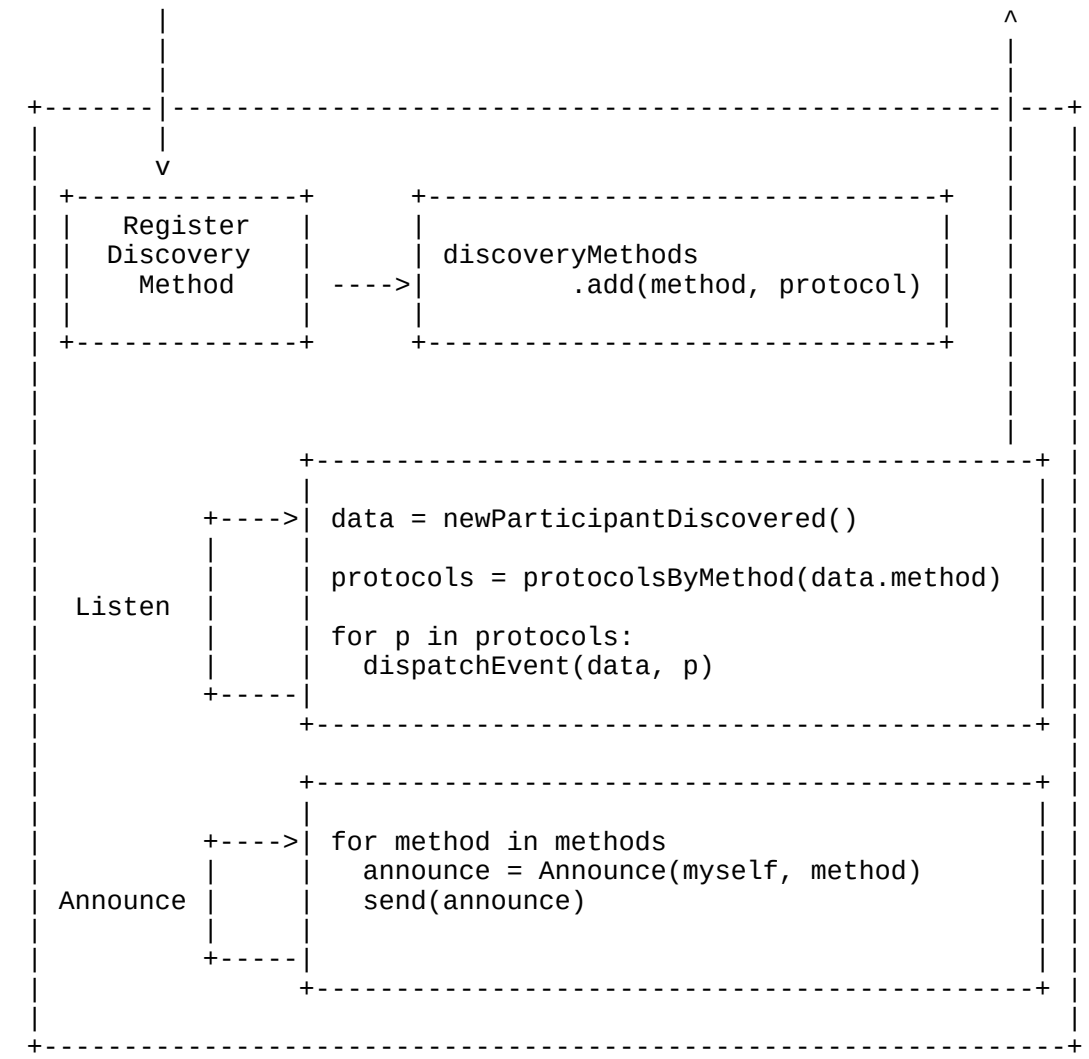

Figure 5: Discovery manager diagram

4.2.5. Framework Events

While most of the events used during execution are for inter-protocol and inter-node interaction, the framework SHOULD also provide control events for interaction with the framework. For instance, as illustrated in Section 3.2.3, a protocol may need to interact with the Discovery Manager to probe the system for neighboring nodes. To facilitate this, the framework includes a dedicated request, RequestAnnouncement, as part of its toolset.

The definition of this events is still a work in progress and it will be detailed in further versions of the document.

4.3. Configuration Manager

The configuration manager plays a critical role in managing and maintaining the configuration settings for a node, ensuring that all components function seamlessly together and adapt to the different changes happening in the system. Firstly, nodes should be capable of self-configuration during startup. Secondly, since a system's state will evolve throughout its execution, protocols should be able to adapt accordingly. Finally, since these systems may be constituted by hundreds or even thousands of nodes, manual, human interaction is unreliable, and therefore all of these changes should be conducted in a totally autonomous way.

The configuration manager achieves by receiving information from protocols (through the event manager) of which parameters it is responsible for configuring and updating. This may happen in two distinct ways: 1) based on the local metrics collected by the node (e.g., CPU usage, memory usage, latency, etc.) the framework can make informed local decisions to alter configuration parameters, or, 2) when new information arrives through the communication manager, the configuration manager processes it and forwards the respective event to the event manager so it can be correctly dispatched to the protocol that relies on it. In the future, we plan to describe the full interface for protocol configuration, as depicted in [RFC6241].

In the following subsections, we will describe the different tasks of the configuration manager.

4.3.1. Self Configuration

A fundamental challenge lies in obtaining an initial configuration for the node joining the network. Although it is possible to resort to a default configuration, in complex systems this may not be enough, as the system may have evolved up to a point that a default configuration is no longer suitable.

One possible way to solve this issue is to obtain the configuration from a node already present in the system.

As described in Section 3.2.4, protocols should tag the parameters they wish to be configured by the framework as _AutoConfigurable_. This way, during protocol initialization, the framework will register which configuration parameters it should query other nodes for. Moreover, the framework should provide a list of available methods to achieve this, such as copy and verification (i.e., contact an already

active node and copy its parameters in an autonomous way), through DNS records (i.e., analyze the TXT entries of a DNS register in order to obtain recommend values to certain parameters), to name a few.

Akin to the discovery manager, the configuration manager offers a mechanism to register the method used to ensure self-configuration of the parameters tagged as _AutoConfigurable_ by the different protocols. Thus, during the framework initialization, we provide the following function:

registerSelfConfiguration(SelfConfigMethod: method)

* _registerSelfConfiguration_ indicates how the framework will setup its initial configuration. The function receives an argument regarding the self-configuration method. This details how the framework will handle the search for a valid configuration, from the likes of copy-validate (e.g., copying the configuration from another node that has the same _AutoConfigurable_ parameters) or using a genesis node with a recommended initial configuration.

```
main(config):
 ...
 // Setup self-configuration methods
 selfConfigMethod = {copyValidate}
 framework.registerSelfConfiguration(selfConfigMethod)
 ...
```

Ex 2.: Self-Config Method Registration

The framework is in charge of communication with other entities to learn this information and communicate back its findings to the protocol, by, as usual, placing an event in its event queue for processing.

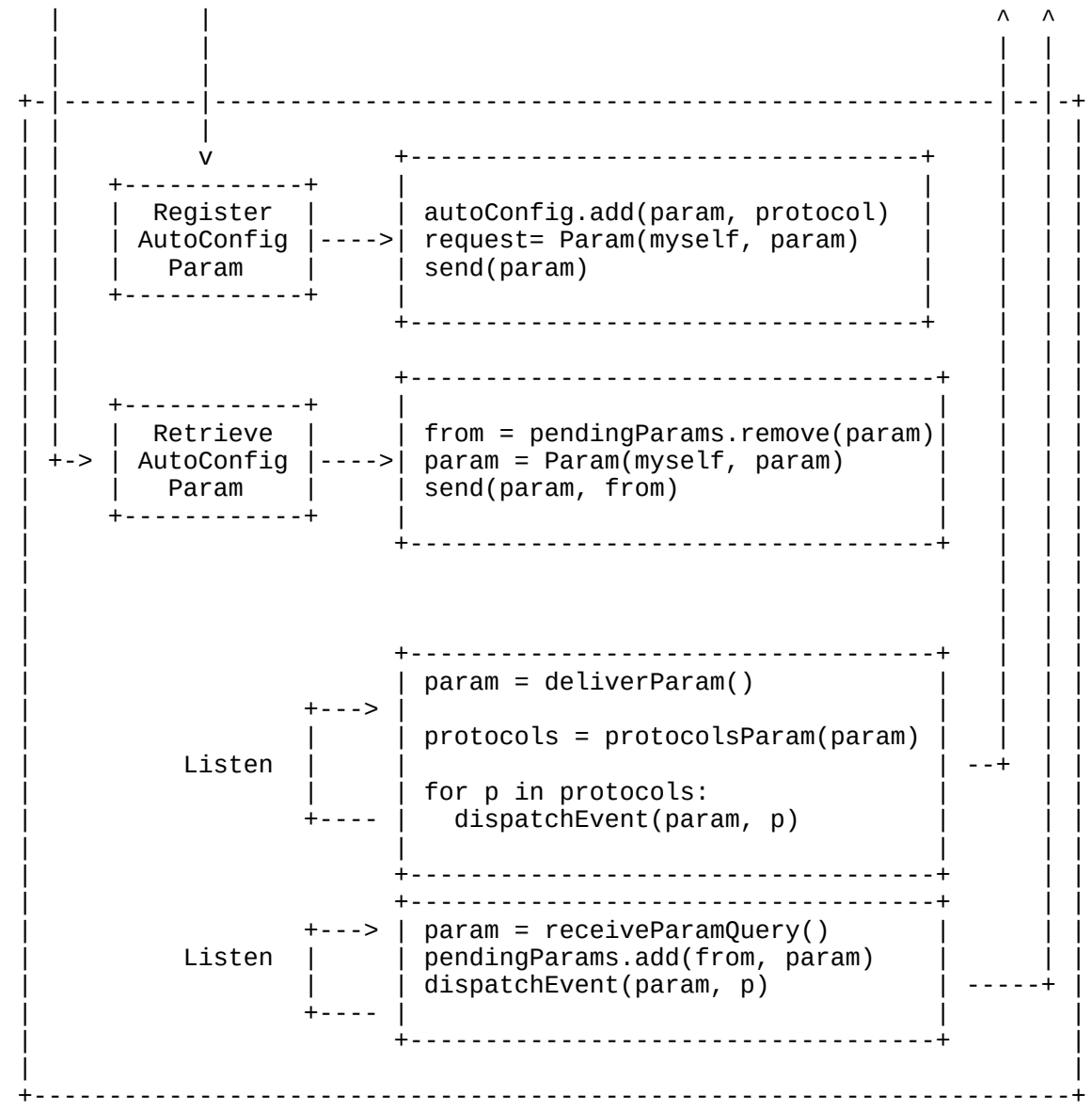

Figure 6: Self-configuration controller overview

4.3.2. Adaptability

Decentralized systems need to adapt while executing, since they operate in dynamic, and often unpredictable environments, where components may join, leave, or fail unexpectedly. As resources like nodes or services can change in real time, the system must adjust to maintain efficiency, performance, and reliability [RFC7575]. Due to these systems complexity and size, the task of managing and adapting them proves too complex for humans. The only feasible option is to have this adaptability happen in an autonomous way. While the initial configuration of a node may prove to be correct and even efficient in a preliminary state of the system, as conditions change, such configuration may become suboptimal or even incorrect. To mitigate this issue, nodes should aim to adapt during runtime, in order to keep up with the current state of the system as a whole.

These changes can be triggered locally, with values calculated by the node itself given its local perception of the network, or by the node’s neighbors that aim to converge to a common global configuration.

To accomplish this, protocols should tag the parameters they wish to be reconfigured as _Adaptive_. Parameters with this tag will be managed by the framework and updated when a reconfiguration request is issued (i.e., due to a periodic timer, reconfiguration data arriving from another node, an intelligent controller executing an AI model [I-D.irtf-nmrg-ai-challenges-04], etc.). Reconfiguration can happen through distinct methods, such as analyzing local metrics (e.g., the CPU usage of a protocol may be too high), synchronization with other nodes running the same protocol (i.e., to reach convergence of configuration), etc. So, the framework SHOULD provide a clear clarification of the methods available and how they behave.

During initialization, the developer specifies the method for automatically reconfiguring parameters tagged as _Adaptive_ by protocols by selecting one of the available methods. This is achieved using the following function:

registerAdaptiveConfiguration(AdaptiveMethod: method)

* _registerAdaptiveConfiguration_ defines how the framework will autonomously update protocol parameters. The function takes an argument specifying the adaptive method, which determines how the framework manages updates to Adaptive parameters. This may involve leveraging local or remote metrics, requesting the state of other nodes, or other adaptive mechanisms.

```
main(config):
 ...
 // Adaptive methods
 adaptiveMethod =  {localMetrics}
 framework.registerAdaptiveConfiguration(adaptiveMethod)
 ...
```

Ex 3.: Adaptive Method Registration

In the following diagrams, we show the interaction diagram regarding the requests issued to the adaptive controller and the background listeners handling incoming connections from the communication manager, respectively.

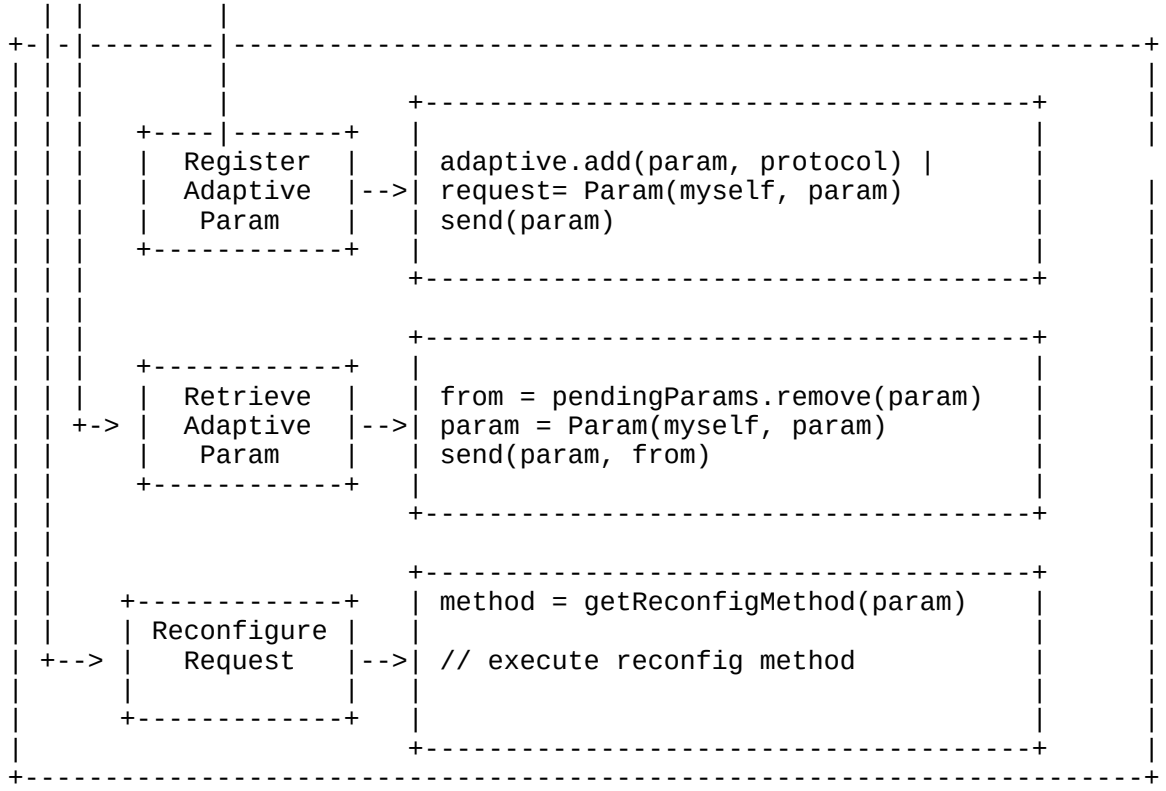

Figure 7: Adaptive controller requests overview

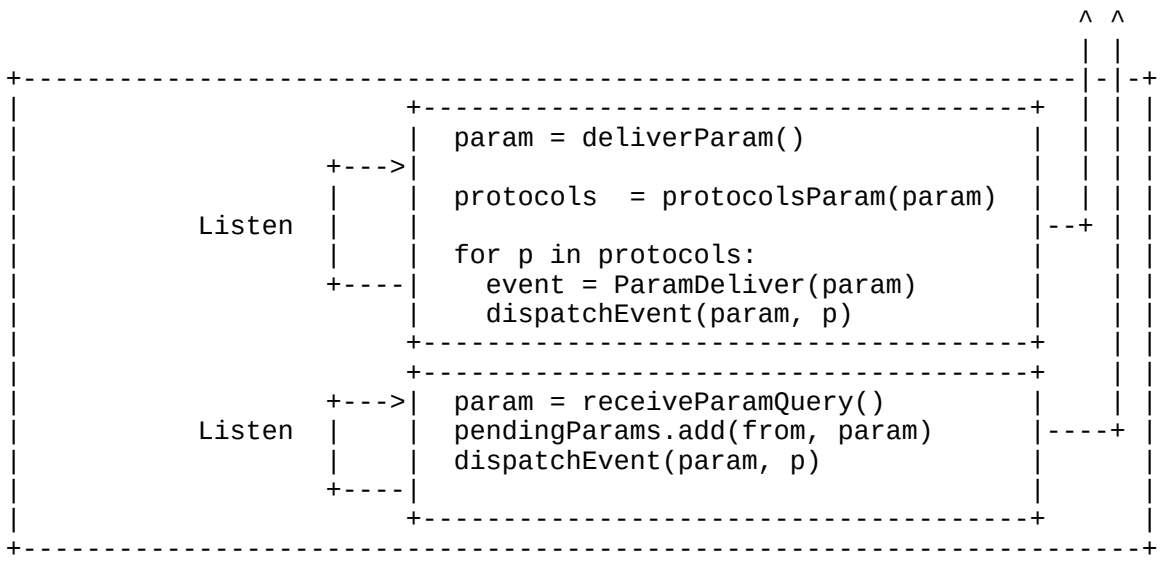

Figure 8: Adaptability listeners overview

4.4. Security Manager

One of the greatest challenges in distributed systems, particularly decentralized ones, is the ability to accurately identify and authenticate nodes in the absence of a central certification authority. Over the years, various solutions have been proposed [Authentication_Survey], including statistical trust models, auxiliary networks composed of trusted nodes, and other approaches.

To eliminate centralized components and minimize reliance on other nodes for trust, we propose a self-signed public key cryptography mechanism as the default security solution. In this approach, each node independently generates and manages its own identity using an asymmetric cryptographic scheme.

The security manager is structured into distinct components responsible for identity management and creation, secure channel establishment, and the implementation of standard security primitives. Additionally, it interacts directly with the communication manager to ensure secure communication and attestation.

4.4.1. Security Primitives

The security primitives component is responsible for managing cryptographic material, including private keys with associated certificates for identification, certificates of trusted nodes, and symmetric keys. It enables the framework to load this information into memory and persist it to disk when necessary.

This component facilitates the execution of standard security procedures, such as certificate validation, signature generation, Message Authentication Codes (MACs), and block cipher operations. These fundamental capabilities allow higher-level components within the framework to build secure abstractions for the developer.

4.4.2. Identity Manager

Properly identifying nodes in a distributed environment is inherently challenging due to factors such as the absence of a central authority, the dynamic nature of nodes, and system heterogeneity. While some decentralized trust models are more widely adopted than others, different solutions must balance security guarantees with user flexibility. As previously discussed, we propose a cryptographic model based on self-signed certificates. While this approach does not provide the same security assurances as other methods, it offers simplicity and flexibility in environments where a central authority or a predefined set of trusted nodes is not a viable option.

The identity manager is responsible for handling the identities of users, services, and system components. Its primary function is to ensure secure, consistent, and efficient identification, authentication, and authorization across all nodes.

Each node (i.e., Peer) is associated with a hash of its default certificate and maintains an internal list of aliases. An alias serves as an alternative identifier for a node, providing a simplified interface for interacting with the identity manager. This allows nodes to assume multiple identities, enabling more complex communication patterns for specific protocols when needed.

Nodes can generate, delete, and manage aliases as required, with each alias possessing its own private key and self-signed certificate. Furthermore, when sending a message, a node can explicitly instruct the framework to use a specific alias (i.e., Section 3.2.5).

Although this design does not support verifying human-readable node information (e.g., DNS names in certificates issued by a Central Authentication Service (CAS)), it ensures that nodes can reliably reference known peers and verify that they are communicating with the same entity as before.

This simple yet powerful construct provides two essential functionalities: signing data and verifying previously signed information. These capabilities form the foundation for establishing secure communication channels.

4.4.3. Secure Communication

Secure channels are communication pathways that guarantee the confidentiality, integrity, and authenticity of messages exchanged between nodes. They prevent eavesdropping, tampering, and impersonation by malicious actors.

To establish secure channels, the framework relies on the identity manager to ensure proper signature generation and verification, ensuring that transmitted messages remain both confidential and authentic. When a node requests secure message transmission (i.e., Section 3.2.5), the communication manager coordinates with the security manager to handle the process securely.

The security manager utilizes cryptographic primitives and the node's identity to sign and encrypt the data before forwarding the encrypted payload to the communication manager for transmission.

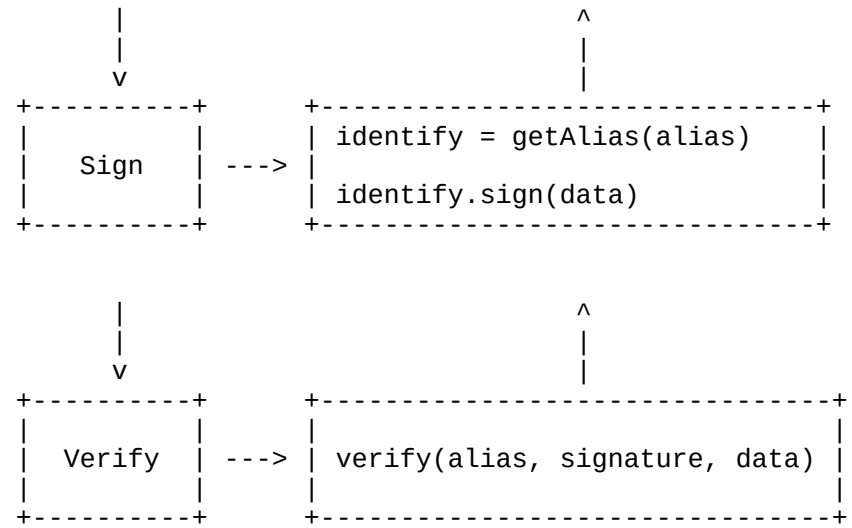

Figure 9: Identity manager and channels usage overview

It is important to note that this module is entirely optional to the developer, meaning that these functionalities are only utilized if the developer requests for the transmission of messages in a secure fashion.

4.5. Communication manager

Handling communication can prove to be extremely complex, due to the different nature of different communication interfaces. While in a robust and resilient environment it may be viable to have a predefined established communication interface to connect two nodes, in a dynamic one such an option may not be reliable. Challenges such as the ability of nodes to join and leave the system at any point, loss of connectivity due to high mobility, or heterogeneity, make having only one communication option between any pair of nodes an impractical solution. Moreover, in practice, lots of protocols are agnostic to the communication layer and are mostly worried that information arrives at their correct destination, without being concerned about how that is accomplished.

To address these challenges, we introduce a communication manager that facilitates the use and negotiation of multiple communication interfaces. These interfaces are abstracted through transports, which serve as intermediaries between protocols and communication mechanisms. Nodes communicate using the available transports, each of which corresponds to a specific communication interface (e.g., TCP, BLE, QUIC).

When a protocol is initialized (i.e., Section 3.2.5), the developer specifies connection preferences using keyword-based parameters. These keywords define the communication guarantees the developer requires. For example:

* If the preferences include {reliable, connectionOriented}, the framework will prioritize TCP as the default transport.

* If the preferences include {lightweight, connectionless}, the framework will favor UDP.

The specified keywords generate an ordered list of communication preferences. For instance, {reliable, unreliable} would result in {TCP, UDP} in that order. The exact mapping of keywords to transport protocols is still under development and will be detailed in future versions. If an invalid combination is provided, the framework will notify the protocol by throwing an error and terminating initialization (i.e., Section 4.5.3).

Using this information, the communication manager sets up a transport for each interface and establishes a priority order based on the provided preferences. The communication manager will attempt to adhere to these preferences, resorting to lower-priority transports only if those higher on the list become unavailable (e.g., if a TCP connection fails, the framework will attempt to send the message via UDP).

4.5.1. Architecture

As previously described, connections are abstracted as transports. Transports serve as an intermediary mechanism that enables nodes to interact with each other. Protocols utilize these transports in an agnostic manner to facilitate message transmission.

The communication manager is divided into different components, as depicted in the following diagram:

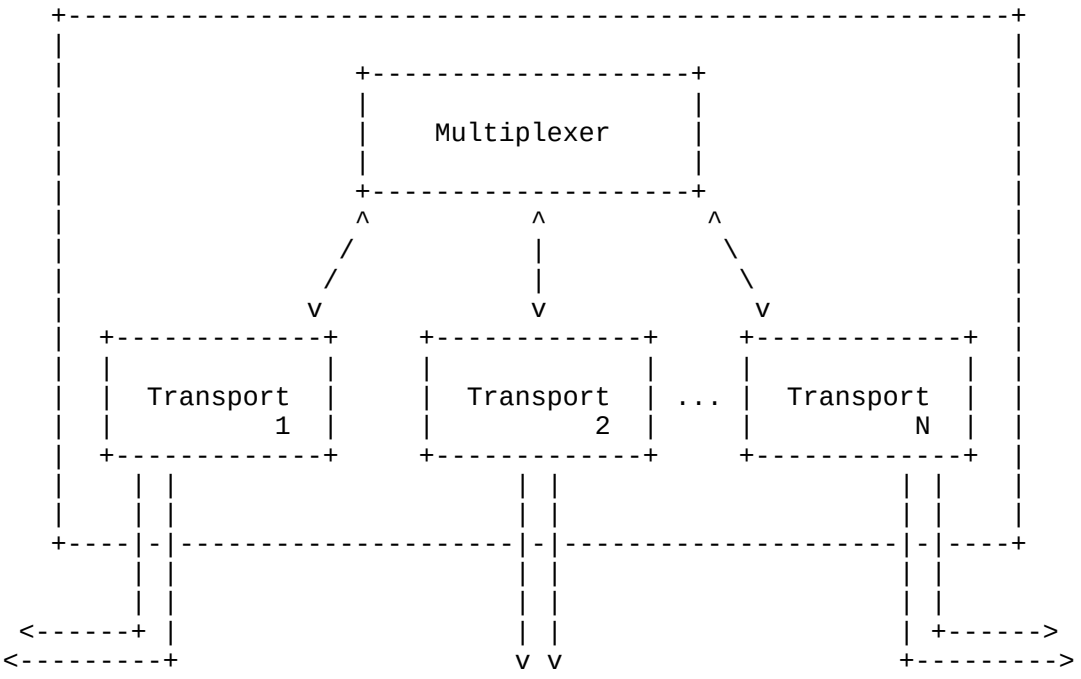

Figure 10: Communication manager architecture

The multiplexer serves as the controller responsible for selecting the appropriate transport for communication. It translates the keywords provided as preferences into concrete transport implementations, maintains the priority order established by

protocols, and forwards data accordingly. When incoming data is received, the multiplexer processes it and ensures delivery to the appropriate protocol via the event manager. Additionally, it continuously monitors transport layers for failures or errors and attempts to reestablish communication in response to abrupt connection disruptions.

Transports are coupled with a communication interface. Each transport can manage multiple parallel connections (e.g., multiple TCP connections on different ports), enabling distinct "virtual connections" for different protocols. Protocols can either use a default, pre-established connection associated with the transport-created when the transport is first requested-or establish their own dedicated connections. This aspect of the framework remains a work in progress, and a formal specification for the data format used in transport communication should be defined.

It is important to note that transports are highly flexible and modular, supporting a pluggable architecture. Developers specify the communication interfaces they intend to use, and the framework automatically assigns the appropriate transport. Additionally, new transports can be implemented and customized to enable alternative communication behaviors and guarantees. For instance, a developer may require a transport protocol not provided by the framework, which they can implement using the standardized transport API.

4.5.2. Peer Identification

Identifying peers in a dynamic distributed system is challenging due to the system’s inherent characteristics. Nodes frequently join, leave, relocate, or fail, making it difficult to maintain an accurate and up-to-date view of the system’s topology. Furthermore, in the absence of a central authority responsible for maintaining a definitive "source of truth," achieving a global perspective on peer identities is infeasible.

A naive approach would be to identify nodes using their network address, such as an IP address ([RFC791]) and port. While this may be effective in static, reliable networks, it is unsuitable for highly dynamic environments where connections can fail unexpectedly and nodes may physically relocate, resulting in loss of connectivity.

To address this challenge, we propose a peer identifier abstraction that encapsulates multiple pieces of information, ensuring that each node in the system can be uniquely identified. Each node is primarily identified by a self-signed certificate, specifically a public key. Assuming the use of a well-established public-key cryptosystem such as RSA ([RFC8017]), the likelihood of a collision

is extremely low, providing a high degree of confidence in the uniqueness of each node’s identifier. Additionally, peer identification should support reliable contact mechanisms. Given that relying on a single contact path is impractical in dynamic environments, we propose that each peer identifier includes a list of transport addresses. This redundancy reduces the risk of disconnection. For example, a node may have a TCP connection via an IP address with two distinct ports and a fallback UDP connection.

This version of GFDS provides a high-level overview of peer identifiers, which are subject to further refinement in future iterations. The proposed identifier consists of a hash of the node’s self-signed certificate and a list of transport links, inspired by [multiaddress], a self-describing network address format that enables flexible and transport-agnostic addressing.

Therefore, we present a sample model of this structure:

```
def Peer {
  //An hash of the self-signed certificate of a node
  CertificateHash identifier,
  TransportLink {key: Transport, value...: Address}: links
  //Map of the links associated with the peer
}
```

```
//Data forwarded by the security manager
privateKey, publicKey = RSA()
certificate = selfSign(privateKey, publicKey)

myself = Peer(certificate)

//Adds two connections to the node IP address through TCP on ports 8080 and 8081
myself.addTransportLink(TCP, 8080)
myself.addTransportLink(TCP, 8081)

//Adds a connections to the node IP address through UDP on port 700
myself.addTransportLink(UDP, 7000)

//Opens the node BLE adapter for incoming connections
myself.addTransportLink(BLE)

//See active connections
print(myself.getTransportLink(TCP))
//{ipv4/192.10.12.10/tcp/8080, ipv4/192.10.12.10/tcp/192.10.12.10:8081}

print(myself.getTransportLink(QUIC))
// {none}

print(myself.getTransportLink(BLE))
// {/ble/C0:26:DA:00:12:34}

//See node identification and connection (i.e., link and hash)
print(myself.getNodeIdentification(UDP))
//{ipv4/192.10.12.10/udp/700/node/QcSDSDAsxKDcseSsca}
```

It is important to note that all of this is managed internally by the framework, and the developer interacts with the communication manager by expressing the keywords englobing the behaviour he pretends to ensure during communication with other nodes.

4.5.3. Control Events

Although this layer is designed to be as agnostic as possible for developers, certain protocols may require notifications when specific changes occur in the transports they utilize. For instance, a protocol relying on a TCP connection as a failure detector should be informed if this connection is lost. To support this functionality, we propose the use of the framework’s notification events construct.

When a control event occurs in the communication manager (e.g., a connection failure), the framework generates a notification encapsulating this information. Protocols that subscribe to such notifications will receive updates, while those that do not remain unaffected and agnostic to changes in the lower layers.

The framework SHOULD define a standardized set of generic control events to encapsulate relevant transport-related information. In future iterations, we aim to develop a more comprehensive and fully extended version of this event set. The following example provides an initial illustration of these events and their intended usage:

```
// Framework Control Events
def TransportConflict {
  Throwable cause,
  string message,
}

def ConnectionUpNotification {
  Peer peer,
  Transport transport,
}

def ConnectionDownNotification {
  Peer peer,
  Throwable cause,
  string message,
  Transport transport
}
```

ConnectionDown Example

```
init(properties):
  ...
  registerRequestHandler(PingRequest,uponPingRequest)
  subscribeNotification(NeighborUpNotification,uponConnectionDown)
  ...

uponPingRequest(PingRequest: request, Protocol: sourceProto):
  msg = BroadcastMessage(request.msg, this.myself)
  this.neighbors.forEach(peer -> sendMessage(msg, peer))
  sendReply(DeliverReply(request.msg), sourceProto)

uponConnectionDown(ConnectionDownNotification: notification):
  log("Connection to node {} has failed! Cause: {}", notification.peer ,notification.caus
e)
  this.neighbors.remove(event.peer)
```

TransportConflict Example

```
init(properties):
  ...
  preferences = {reliable, secure, unsecure}
  registerCommunicationPreferences(preferences)

  registerRequestHandler(PingRequest,uponPingRequest)
  subscribeNotification(TransportConflict, uponTransportConflict)
  ...

uponTransportConflict(TransportConflict: notification)
  log.error("Impossible combination of preferences!")
  System.exit(-1)

....
```

4.5.4. Secure Communication

To guarantee message integrity, confidentiality, and protection against unauthorized access and attacks, nodes must communicate securely. As previously described, the security manager provides cryptographic primitives for signing, verifying, encrypting, and decrypting messages. The communication manager utilizes these functionalities to establish secure communication between nodes. For example, if a node has an open TCP connection and intends to transfer messages securely, the framework can use asymmetric cryptographic keys to upgrade the connection and establish communication via TLS [RFC8446] with the other node. Alternatively, it can open a new connection exclusively for TLS. The concept of transport services, explored in [RFC8922], provides a common interface for utilizing transport protocols, allowing applications to easily adopt new protocols with similar security guarantees.

This approach decouples security from protocol logic while ensuring flexibility. Protocols only use the framework’s security components if explicitly requested (i.e., by invoking the relevant functions), thereby minimizing overhead for applications that do not require security assurances. Conversely, protocols that require secure communication can remain agnostic to the underlying complexities and simply use the provided abstractions.

It is important to note that while some communication interfaces offer straightforward methods for enforcing security guarantees (e.g., TLS can be layered on top of TCP using the appropriate credentials), others, such as BLE, do not have such standardized solutions. One potential approach is encrypting data at the application level and relying on an unsecured connection for transmission. However, this remains an area of active research.

5. Execution Model

As described in previous sections, developers interact with the platform by defining protocols. Protocols consist of various handlers that encapsulate user-defined logic. These handlers are triggered by events such as message transmissions, inter-protocol interactions, or timer-based triggers.

A key requirement for ensuring a seamless developer experience is that protocols should function independently, meaning that while one protocol processes a task, other protocols and the framework itself should not be blocked. Instead, each protocol should be able to make continuous progress. To achieve this, we propose the following execution model and concurrency semantics:

Each protocol is assigned a dedicated thread that handles received events sequentially via an event queue (as described in Section 4), executing the corresponding callbacks. The core framework operates on its own separate thread, managing internal operations. This design enables multiple protocols to execute concurrently (i.e., in a multi-threaded environment) while using event-based message passing to communicate. This approach eliminates concurrency-related issues for developers, as all inter-protocol interactions occur through structured event messaging. Consequently, lock-free execution is ensured for protocol interactions, while allowing multiple protocols to coexist efficiently within the same environment. This event-driven model provides developers with a clear and predictable execution flow without requiring complex concurrency mechanisms.

5.1. Initialization

Each instance of the framework (i.e., application) requires a main function, where the developer specifies the execution configuration (e.g., resource paths, default parameters, etc.) and registers the protocols to be executed. This setup enables the framework to allocate necessary resources and ensure each protocol operates within its designated execution environment. Additionally, the initialization phase is when the framework configures mechanisms for node discovery, self-configuration, and adaptability using the available framework methods. After initialization, the interaction

is governed by the protocols and the framework is left in charge of ensuring that events are interchanged between protocols, nodes can discover other nodes running in the system, timers are correctly triggered, protocols are able to self-configure and adapt as the system evolves, and last but not least, nodes can communicate with each other.

An example of initialization can be seen in the following sample:

```
main(config):
  framework = GFDS.newInstance()
  props = framework.loadConfig(config)

  // Discovery methods
  discoveryMethods = {mDNS}
  framework.registerDiscoveryMethods(discoveryMethods)

  //Self methods
  selfConfigMethod =  {copyValidate}
  framework.registerSelfConfiguration(selfConfigMethod)

  // Adaptive methods
  adaptiveMethod =  {localMetrics}
  framework.registerAdaptiveConfiguration(adaptiveMethod)

  // Protocol registration
  app = BlocksApplication()
  antiEntropy = AntiEntropyProtocol()
  broadcast = EagerPushBroadcastProtocol()
  membership = FullMembership()

  framework.registerProtocol(app)
  framework.registerProtocol(antiEntropy)
  framework.registerProtocol(broadcast)
  framework.registerProtocol(membership)

  app.init(props)
  antiEntropy.init(props)
  broadcast.init(props)
  membership.init(props)

  framework.start()
```

6. Common Interfaces

As described in previous sections, protocols interact with each other through inter-process events. While a simple program may only require a protocol to describe its logic, more complex and realistic applications combine multiple protocols and model complex inter-protocol interactions (i.e., an application supporting decentralized storage requires membership management, efficient propagation, authentication, etc.).

While this approach presents great modularity, since an application is modelled as a set of protocols in charge of different aspects of its broader logic, handling inter-protocol interactions concisely and consistently may prove to be difficult.

Following the literature, it is possible to see that, although different protocols have distinct designs and implementations, at a higher level, could aim to offer similar functionalities to the user. For example, overlay network protocols, create an abstraction layer (the "overlay") on top of an existing network infrastructure, and are divided into structured and unstructured overlays. For instance, Chord [Chord] and Kademlia [Kademlia] are structured overlay protocols that provide efficient and predictable lookups of data and nodes. On one hand, Chord uses consistent hashing to map keys to nodes in a circular fashion while Kademlia uses a XOR-based distance metric and a recursive lookup mechanism for finding nodes efficiently, but, at a higher level, both support operations for insertion, deletion and lookup of information, and hence a similar interface.

With this in mind, we propose the creation of families or groups of protocols, that, although possess diverse internal logic and design, offer similar functionalities and thus can be englobed in the same category. Protocols in the same family should implement the same API, enabling high modularity and interoperability between protocols. In this fashion, even if an application swaps a protocol for another in the same family, the way of communication between protocols remains the same.

6.1. Families

As mentioned previously, while different protocols have unique implementations, when we focus on the external view that they provide to users -the API- they can be relatively similar.

Protocols can be "stacked" on top of each other to create layers of abstractions. For instance, a storage protocol can interact with a dissemination protocol to transmit data, and the dissemination

protocol propagates data to a subset of neighbours it receives from a membership protocol. In this manner, each protocol is only focused on the interaction with other protocols from the "layer" beneath it, which forms a much simpler interaction model.

Even though this area is still a work in progress, there SHOULD be a specification of the different families and groups implemented inside the framework, with the use of already established common data types [RFC6991]. In this section, we will describe a proof of concept of such functionality which will be extended in the future. At the moment of writing this draft, we propose three distinct protocol families:

* *membership*: Membership protocols are responsible for managing and maintaining the membership of a group of nodes that participate in the system. In other words, these protocols track which nodes are currently part of the system and make decisions based on the state of these nodes (whether they are up, down, joining, leaving, etc.). A common API for this family of protocols should have events for requesting information on the membership layer regarding other neighbours, for notifying protocols when nodes join and leave the system, etc.

* *dissemination*: Dissemination protocols are designed to propagate information or updates efficiently across nodes in the system. The goal is to distribute data (e.g., state updates, configuration changes, etc.) to all or a subset of nodes within the system, ensuring that the information reaches its intended recipients in a timely and reliable manner. Not all dissemination protocols have the same semantics, for instance, a flood broadcast protocol transmits data to every node it knows, while a gossip-based protocol only disseminates data to a randomly sampled subset of nodes. Still, since all of them provide similar functionalities and in the end aim to reach the same goal, a common API should have events for dissemination of data, receiving data, etc.

* *storage*: Storage protocols are designed to manage how data is stored, accessed, and maintained within a system. Their primary functionality is to ensure that data is stored and that it can be accessed reliably by applications or other users, even in the presence of failures, network issues, or node crashes. These storage protocols could be implementing storage solutions deployed directly on the network (i.e., decentralized storage solutions) or serve as a gateway to communicate with external storage deployments (e.g., a node uses a Cassandra [Cassandra] instance running off-site to store data). A common API for storage protocols should include essential methods like storing data, retrieving data, replication management, handling failures, reconfiguring namespaces, etc.

By defining each family and a set of common APIs, when writing protocols, developers match a protocol with a respective family and use the pre-defined events of each family to describe the interaction with the different protocols.

6.2. Usage

When writing an application using the framework, the developer can use an arbitrary number of protocols to achieve such a objective. The different protocols communicate with each other through events to compose a complex application. For instance, if we are building a decentralized storage system when the storage layer receives an operation from the application to write a data item, the protocol will create an event to a protocol on the dissemination layer to propagate such information to other nodes in the system. This design allows developers to build complex dynamic and decentralized systems in a modular and efficient way, and can even pave the way for experts in the area to write protocols that can be used by other less-experienced developers.

Appendix A specifies a small set of common interfaces for the different families presented above. In the example below we showcase a sample example composing these ideas:

EagerGossipProtocol

```
init(properties):
  ...
  this.neighbors = Set()
  this.fanout = properties.fanout

  registerRequestHandler(DisseminationRequest, uponBroadcastRequest)

  registerReplyHandler(GetNeighborsSampleReply, uponNeighborsSampleReply)

  subscribeNotification(GetNeighborDownNotification, uponNeighborDown)
  ...

uponDisseminationRequest(DisseminationRequest: request, Protocol: sourceProto):
  msg = DisseminationMessage(request.payload, request.timestamp, 0)

  peers = this.neighbors.randomSubSet(this.fanout)
  for p in peers:
    sendMessage(msg, p)

uponNeighborsSampleReply(GetNeighborsSampleReply: reply):
  this.neighbors.addAll(reply.sample)

uponNeighborDown(GetNeighborDownNotification: notification):
  this.neighbors.remove(notification.peer)
```

```
                      FloodBroadcastProtocol

init(properties):
  ...
  this.neighbors = Set()

  registerRequestHandler(DisseminationRequest, uponDisseminationRequest)

  registerReplyHandler(GetNeighborsSampleReply, uponNeighborsSampleReply)

  subscribeNotification(GetNeighborDownNotification, uponNeighborDown)
  ...

uponDisseminationRequest(DisseminationRequest: request, Protocol: sourceProto):
  msg = DisseminationMessage(request.payload, request.timestamp, 0)
  for p in neighbors:
    sendMessage(msg, p)

uponNeighborsSampleReply(GetNeighborsSampleReply: reply)
  this.neighbors.addAll(reply.sample)

uponNeighborDown(GetNeighborDownNotification: notification)
  this.neighbors.remove(notification.peer)

                      FullMembershipProtocol
init(properties):
  ...
  registerDiscoveryHandler(uponDiscovery)

  registerRequestHandler(GetNeighborsSampleRequest, uponNeighborsSampleRequest)
  ...

uponNeighborsSampleRequest(GetNeighborsSampleRequest: request, Protocol: sourceProto):
  int sampleSize = request.sampleSize
  sample = this.neighbors.randomSubSet(sampleSize)

  reply = GetNeighborsSampleReply(sample)
  sendReply(reply, sourceProto)

uponDiscovery(DiscoveryNotification: notification):
  this.neighbors.add(event.peer)
```

In this example, it’s possible to see two dissemination protocols (_EagerGossipProtocol_ and _FloodBroadcastProtocol_) and a membership protocol (_FullMembershipProtocol_). The FullMembershipProtocol protocol offers an abstraction to obtain the node’s membership, and, both dissemination protocols, although having different design choices regarding the way data is propagated, use the same API

provided by the "bottom layer". This way, if an application wishes to change its dissemination protocol, it can easily do so by just registering different protocols during initialization.

6.3. Remarks

At its core, the framework aims to provide a consistent set of functionalities to protocols that adhere to a common interface design. While following this design is recommended for modularity and interoperability, it is not mandatory. Not all protocols necessarily fit into a predefined category or family, and developers have the flexibility to define their own interface specifications.

When implementing protocols, developers can design custom inter-protocol communication events (e.g., requests, notifications) tailored to their specific application needs. Although adopting the common interface design promotes high modularity and decouples implementation from interaction, developers are ultimately free to choose the approach that best suits their use case.

7. Examples

In this section, we provide a full-fledged example of the framework. The example contains the initialization of the framework with the due protocols, their implementation and the respective events. We leverage the common API presented in Section 6 to present a generic example.

The example showcases a message dissemination application which takes input from the user, and disseminates data through a network of nodes, as depicted in the following diagram:

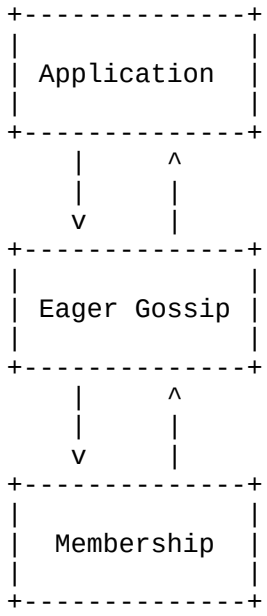

Figure 11: Protocols interaction diagram

7.1. Pseudocode

Main

```
main(config):
  framework = GFDS.newInstance()
  props = framework.loadConfig(config)

  discoveryMethods = {multicast}
  framework.registerDiscoveryMethods(discoveryMethods)

  selfConfigMethod =  {copyValidate}
  framework.registerSelfConfiguration(selfConfigMethod)

  adaptiveMethod =  {localMetrics}
  framework.registerAdaptiveConfiguration(adaptiveMethod)

  app = MessageApplication()
  dissemination = EagerGossipProtocol()
  membership = FullMembershipProtocol()

  framework.registerProtocol(app)
  framework.registerProtocol(dissemination)
  framework.registerProtocol(membership)

  app.init(props)
  dissemination.init(props)
  membership.init(props)

  framework.start()
```

Application

```
//Events
def OperationTimer {
  long timeout
}

init(props):
    this.disseminationProtocol = props.disseminationProtocol

    registerNotificationHandler(DeliveryNotification, uponDeliveryNotification)
    registerTimerHandler(OperationTimer, uponOperationTimer)

    setupPeriodicTimer( OperationTimer(props.period), props.firstTimer, props.period)

uponOperationTimer(OperationTimer: timer):
  in = input("Insert message:")
  request = DisseminationRequest(myself, in, Time.now())
  sendRequest(request, this.disseminationProtocol)

uponDeliveryNotification(DeliveryNotification: notification):
  log("Message received! Data: {}", notification.payload)
```

The pseudo code for the EagerGossip and FullMembership protocols can be found in Appendix B.

8. Security Considerations

This document defines the concept of secure transports to facilitate secure communication between nodes. Secure transports leverage security primitives provided by the framework, in conjunction with identity management.

For security primitives, secure transport implementations can rely on well-established cryptographic algorithms and cryptosystems. We recommend that any framework implementation utilize proven, widely accepted cryptographic libraries, as outlined in [RFC9641] and [RFC9642]. However, for specialized use cases, the framework should also support protocols that do not require secure communication and, therefore, do not adhere to these security considerations.

Identity management data should be stored in secure memory or persistent storage to protect against malicious attacks that could compromise a node’s identity or lead to impersonation.

9. IANA Considerations

This document has no IANA actions.

10. Implementation Status

At the moment of writing this draft, we have a reference implementation following the ideas presented in this document. Babel (https://codelab.fct.unl.pt/di/research/tardis/wp6/babel-swarm/babel-core-swarm) is a Java framework for easing the complexity of building and managing distributed systems. The framework is an active research effort, and it's being continuously updated to support more features and compliance with GFDS.

The Common APIs (https://codelab.fct.unl.pt/di/research/tardis/wp6/babel-swarm/babel-protocolcommons) repository materializes the design presented in Section 6, and a list of applications is available in the following repository (https://codelab.fct.unl.pt/di/research/tardis/wp6/babel-swarm/applications).

11. Acknowledgments

This work was partly funded by EU Horizon Europe under Grant Agreement no. 101093006 (TaRDIS) and FCT-Portugal under grant UIDB/04516/2020.

12. References

12.1. Normative References

[I-D.irtf-nmrg-ai-challenges-04]
Fran巽ois, J., Clemm, A., Papadimitriou, D., Fernandes, S., and S. Schneider, "Research Challenges in Coupling Artificial Intelligence and Network Management", Work in Progress, Internet-Draft, draft-irtf-nmrg-ai-challenges-04, 28 November 2024, <https://datatracker.ietf.org/doc/html/draft-irtf-nmrg-ai-challenges-04>.

[multiaddress]
"multiaddress", n.d., <https://github.com/multiformats/multiaddr>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/rfc/rfc2119>.

[RFC6241] Enns, R., Ed., Bjorklund, M., Ed., Schoenwaelder, J., Ed., and A. Bierman, Ed., "Network Configuration Protocol (NETCONF)", RFC 6241, DOI 10.17487/RFC6241, June 2011, <https://www.rfc-editor.org/rfc/rfc6241>.

[RFC6991] Schoenwaelder, J., Ed., "Common YANG Data Types", RFC 6991, DOI 10.17487/RFC6991, July 2013, <https://www.rfc-editor.org/rfc/rfc6991>.

[RFC7575] Behringer, M., Pritikin, M., Bjarnason, S., Clemm, A., Carpenter, B., Jiang, S., and L. Ciavaglia, "Autonomic Networking: Definitions and Design Goals", RFC 7575, DOI 10.17487/RFC7575, June 2015, <https://www.rfc-editor.org/rfc/rfc7575>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/rfc/rfc8174>.

[RFC8922] Enghardt, T., Pauly, T., Perkins, C., Rose, K., and C. Wood, "A Survey of the Interaction between Security Protocols and Transport Services", RFC 8922, DOI 10.17487/RFC8922, October 2020, <https://www.rfc-editor.org/rfc/rfc8922>.

[RFC9641] Watsen, K., "A YANG Data Model for a Truststore", RFC 9641, DOI 10.17487/RFC9641, October 2024, <https://www.rfc-editor.org/rfc/rfc9641>.

[RFC9642] Watsen, K., "A YANG Data Model for a Keystore", RFC 9642, DOI 10.17487/RFC9642, October 2024, <https://www.rfc-editor.org/rfc/rfc9642>.

12.2. Informative References

[Authentication_Survey] Li, Z., Xu, X., Shi, L., Liu, J., and C. Liang, "Authentication in Peer-to-Peer Network: Survey and Research Directions", IEEE, 2009 Third International Conference on Network and System Security pp. 115-122, DOI 10.1109/nss.2009.30, 2009, <https://doi.org/10.1109/nss.2009.30>.

[Cassandra] "Cassandra", n.d., <https://cassandra.apache.org/_/index.html>.

[Chord] Stoica, I., Morris, R., Karger, D., Kaashoek, M., and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications", Association for Computing Machinery (ACM), ACM SIGCOMM Computer Communication Review vol. 31, no. 4, pp. 149-160, DOI 10.1145/964723.383071, August 2001, <https://doi.org/10.1145/964723.383071>.

[Kademlia] "*** BROKEN REFERENCE ***".

[Lib2p] "Lib2p", n.d., <https://libp2p.io>.

[PeerSim] "PeerSim", n.d., <https://peersim.sourceforge.net>.

[RFC6762] Cheshire, S. and M. Krochmal, "Multicast DNS", RFC 6762, DOI 10.17487/RFC6762, February 2013, <https://www.rfc-editor.org/rfc/rfc6762>.

[RFC7668] Nieminen, J., Savolainen, T., Isomaki, M., Patil, B., Shelby, Z., and C. Gomez, "IPv6 over BLUETOOTH(R) Low Energy", RFC 7668, DOI 10.17487/RFC7668, October 2015, <https://www.rfc-editor.org/rfc/rfc7668>.

[RFC768] Postel, J., "User Datagram Protocol", STD 6, RFC 768, DOI 10.17487/RFC0768, August 1980, <https://www.rfc-editor.org/rfc/rfc768>.

[RFC791] Postel, J., "Internet Protocol", STD 5, RFC 791, DOI 10.17487/RFC0791, September 1981, <https://www.rfc-editor.org/rfc/rfc791>.

[RFC8017] Moriarty, K., Ed., Kaliski, B., Jonsson, J., and A. Rusch, "PKCS #1: RSA Cryptography Specifications Version 2.2", RFC 8017, DOI 10.17487/RFC8017, November 2016, <https://www.rfc-editor.org/rfc/rfc8017>.

[RFC8446] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <https://www.rfc-editor.org/rfc/rfc8446>.

[RFC9000] Iyengar, J., Ed. and M. Thomson, Ed., "QUIC: A UDP-Based Multiplexed and Secure Transport", RFC 9000, DOI 10.17487/RFC9000, May 2021, <https://www.rfc-editor.org/rfc/rfc9000>.

[RFC9293] Eddy, W., Ed., "Transmission Control Protocol (TCP)", STD 7, RFC 9293, DOI 10.17487/RFC9293, August 2022, <https://www.rfc-editor.org/rfc/rfc9293>.

Appendix A. APIs

The pseudo-code in this section describes a sample example of the common APIs mentioned in Section 6.

Storage Layer APIs

```
// Requests
def ExecuteRequest {
  string nameSpace,
  Operation op // Data, operation type, etc.
}

def ModifyNamespaceRequest {
  string nameSpace,
  NameSpaceConfig config //Parameters to be modified
}

// Replies
def ExecuteJSONReply{
  string jsonAsString,
  Status statusCode
}

def ExecutePayloadReply {
  byte [] payload,
  Status statusCode
}

def ModifyNamespaceReply {
  string nameSpace,
  Status statusCode
}

// Notifications
def JSONDataNotification {
  string jsonAsString,
  Status statusCode
}

def PayloadNotification {
  byte [] payload,
  Status statusCode
}

// Timers
def PersistTimer {
  long periodicTimeout
}
```

Dissemination Layer APIs

```
// Requests
def DisseminationRequest {
  Peer sender,
  byte [] payload,
  long timestamp
}

def MissingDataRequest {
  UUID uniqueID,
  Peer destination
}

// Notifications
def DeliveryNotification {
  byte [] payload,
  long timestamp
}

def DataFoundNotification{
  byte [] payload,
  UUID uniqueID,
}

// Messages
def DisseminationMessage {
  byte [] data,
  long timestamp,
  short ttl

  serializer(out):
    out.writeByteArray(data)
    out.writeLong(timestamp)
    out.writeShort(ttl)

  deserializer(in):
    data = in.readByteArray()
    timestamp = in.readLong(in)
    ttl = in.readShort(ttl)
    return DisseminationMessage(data, timestamp, ttl)
}
```

Membership Layer APIs

```
   // Requests
   def GetNeighborsSampleRequest {
     int sampleSize
   }

   // Replies
   def GetNeighborsSampleReply {
     Set<Peer> neighbors
   }

   // Notifications
   def NeighborUpNotification {
     Peer peer
   }

   def NeighborDownNotification {
     Peer peer
   }
```

Appendix B. Examples

EagerGossipProtocol

```
state = {
  this.neighbors: Set<Peer>,
  this.dataSet : Set<DisseminationMessage>,

  @AutoConfigurable
  @Adaptive
  long fanout,
}

/*
This is a simple and pragmatic implementation
of a eager push gossip-based broadcast protocol.
The implementation assumes that there exists a
membership service that uses the interface made
available in commons API.
*/
init(properties):
  this.dataSet = Set<DisseminationMessage>()
  this.neighbors = Set<Peer>()
  //fanout will be setup by the framework

  registerSelfConfiguration(properties, uponSelfConfig)
```

```
  registerAdaptiveConfiguration(properties, uponAdaptive)

  preferences = {reliable}
  registerCommunicationPreferences(preferences)

  registerRequestHandler(DisseminationRequest, uponBroadcastRequest)

  registerReplyHandler(GetNeighborsSampleReply, uponNeighborsSampleReply)

  subscribeNotification(GetNeighborDownNotification, uponNeighborDown)

  registerTimerHandler(GarbageCollectionTimer, uponGarbageCollectionTimer)
  setupPeriodicTimer(GarbageCollectionTimer(properties.interval), uponGarbageCollectionTi
mer)

  registerCommunicationHandler(DisseminationMessage, uponDisseminationMessage)

// Request/Reply Handlers
uponDisseminationRequest(DisseminationRequest: request, Protocol: sourceProto):
  msg = DisseminationMessage(request.payload, request.timestamp, 0)
  deliver(msg)
  peers = this.neighbors.randomSubSet(this.fanout)
  propagate(peers, msg)

uponNeighborsSampleReply(GetNeighborsSampleReply: reply)
  this.neighbors.addAll(reply.sample)

// Notification Handlers
uponNeighborDown(GetNeighborDownNotification: notification)
  this.neighbors.remove(notification.peer)

// Timer Handlers
uponGarbageCollectionTimer(GarbageCollectionTimer: timer):
  this.dataSet.removeIf(data -> Time.now - data.timestamp >
                    timer.interval && data.ttl >= state.ttl)


// Configuration and Adaptive
uponSelfConfig(Map<Parameter, Config> parameters):
  for p in state as AutoConfigurable:
    p = parameters.get(p)

uponAdaptive(Map<Parameter, Config> parameters):
  for p in state as Adaptive:
    p = parameters.get(p)

// Communication Handlers
uponDisseminationMessage(DisseminationMessage: msg, Peer: sender):
  msg.ttl++
```

```
  deliver(msg)

  peers = this.neighbors.randomSubSet(this.fanout)
  propagate(peers, msg)


// Procedures
deliver(DisseminationMessage: msg):
  if(!this.dataSet.contains(msg)):
    this.dataSet.add(msg)

    notification = DeliveryNotification(msg.data)
    triggerNotification(notification)

propagate(Set<Peer> destinations, DisseminationMessage: msg):
  destinations.forEach(n -> sendMessage(msg,n))
```

FullMembershipProtocol

```
/*
This is a membership protocol that should only be used
either on very small scale settings or
for testing purposes. It creates a global
membership abstraction where every single node in
the system knows every other node (except himself).
/*
init(properties):
  this.neighbors = Set<Peer>()

  registerDiscoveryHandler(uponDiscovery)

  //This is a simple protocol, so it doesn’t have self configuration or adaptive paramete
rs

  registerCommunicationPreferences({reliable, connectionOriented})

  registerRequestHandler(GetNeighborsSampleRequest, uponNeighborsSampleRequest)
  subscribeNotification(ConnectionFailedNotification, uponConnectionFailed)

uponNeighborsSampleRequest(GetNeighborsSampleRequest: request, Protocol: sourceProto):
  int sampleSize = request.sampleSize
  sample = this.neighbors.randomSubSet(sampleSize)

  reply = GetNeighborsSampleReply(sample)
  sendReply(reply, sourceProto)

uponDiscovery(DiscoveryNotification: notification):
  this.neighbors.add(event.peer)

uponConnectionFailed(ConnectionFailedNotification: notification)
  this.neighbors.remove(notification.peer)

  triggerNotification(NeighborDown(notification.peer))
```

Contributors

Rafael Matos
TaRDIS
NOVA Laboratory for Computer Science and Informatics (NOVA LINCS)
Email: rd.matos@campus.fct.unl.pt

Tomas Galv達o
TaRDIS
NOVA Laboratory for Computer Science and Informatics (NOVA LINCS)
Email: t.galvao@campus.fct.unl.pt

Felipe Carmo
TaRDIS
NOVA Laboratory for Computer Science and Informatics (NOVA LINCS)
Email: fp.carmo@campus.fct.unl.pt

Jo達o Bordalo
TaRDIS
NOVA Laboratory for Computer Science and Informatics (NOVA LINCS)
Email: j.bordalo@campus.fct.unl.pt

Jo達o Brilha
TaRDIS
NOVA Laboratory for Computer Science and Informatics (NOVA LINCS)
Email: j.brilha@campus.fct.unl.pt

Authors' Addresses

Diogo Jesus
TaRDIS
NOVA Laboratory for Computer Science and Informatics (NOVA LINCS)
Email: da.jesus@fct.unl.pt

Jo達o Leit達o
TaRDIS
NOVA Laboratory for Computer Science and Informatics (NOVA LINCS)
Email: jc.leitao@fct.unl.pt