NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

FRANCISCO TOMÁS ESTEVES GASPAR TEIXEIRA MOURA

Bachelor in Computer Science and Engineering

# A LIBRARY OF PEER-TO-PEER PROTOCOLS FOR BABEL-SWARM

Dissertation Plan

MASTER IN COMPUTER SCIENCE AND ENGINEERING

NOVA University Lisbon

February, 2025

NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

DEPARTMENT OF
COMPUTER SCIENCE

# A LIBRARY OF PEER-TO-PEER PROTOCOLS FOR BABEL-SWARM

FRANCISCO TOMÁS ESTEVES GASPAR TEIXEIRA MOURA

Bachelor in Computer Science and Engineering

**Adviser:** João Leitão
*Associate Professor, NOVA University Lisbon*

Dissertation Plan
MASTER IN COMPUTER SCIENCE AND ENGINEERING

NOVA University Lisbon
February, 2025

# Abstract

Swarm computing represents an innovative paradigm in which numerous heterogeneous devices autonomously collaborate to execute tasks with minimal intervention from user. The TaRDIS European project is dedicated to making this computing model both accessible and practical by developing essential building blocks—such as decentralized membership and communication abstractions—largely inspired by peer-to-peer systems. Central to this initiative is Babel-Swarm, a Java framework for building decentralized systems, evolved from Babel, which incorporates enhanced security, self-management, and auto-configuration mechanisms to support swarm operations.

In this engineering project we will conduct the implementation and evaluation of overlay network protocols from the literature, including adaptive overlays (e.g., X-BOT and T-MAN), membership protocols (e.g., CYCLON and HyParView), DHT protocols (e.g., Chord, Tapestry, and Kadmelia) and communication protocols (e.g., Plumtree and GoCast) in the context of Babel-Swarm. A challenge on this endeavor is to efficiently provide such abstractions in a modular way, such that they can easily be replaced and interchanged by other protocols, offering the same (or similar) functionality without requiring changes to the application logic or code. We will therefore develop an extensive common API that can expose these abstractions to applications in the context of Babel-Swarm ensuring this requirement, while also allowing each abstraction to be used as an independent library.

**Keywords:** Swarm Computing, Overlay Networks, Peer-to-Peer, Distributed Systems

# Resumo

A computação em *Swarm* representa um paradigma inovador no qual numerosos dispositivos heterogéneos colaboram de forma autónoma para executar tarefas com uma intervenção mínima do utilizador. O projeto europeu TaRDIS dedica-se a tornar este modelo de computação acessível e prático através do desenvolvimento de blocos de construção essenciais - tais como a protocolos de afiliação e comunicação descentralizadas - largamente inspirados pelos sistemas *peer-to-peer*. No centro desta iniciativa está o Babel-Swarm, uma *framework* Java para a construção de sistemas descentralizados, desenvolvida a partir do Babel, que incorpora segurança reforçada, autogestão e mecanismos de auto configuração para suportar operações de *swarm*.

Neste projeto de engenharia, iremos realizar a implementação e avaliação de protocolos de redes *overlay* da literatura, incluindo *overlays* adaptativas (e.g., X-BOT e T-MAN), protocolos de afiliação (e.g., CYCLON e HyParView), protocolos DHT (e.g., Chord, Tapestry e Kadmelia) e protocolos de comunicação (e.g., Plumtree e GoCast) no contexto do Babel-Swarm. Um dos desafios deste projeto é fornecer eficientemente tais abstrações de uma forma modular, de modo que possam ser facilmente substituídas e trocadas por outros protocolos, oferecendo a mesma funcionalidade (ou semelhante) sem exigir alterações à lógica ou ao código da aplicação. Por consequente, vamos desenvolver uma *API* comum extensa que possa expor estas abstrações a aplicações no contexto do Babel-Swarm, assegurando este requisito e, ao mesmo tempo, permitindo que as abstrações sejam utilizadas de forma independente.

**Palavras-chave:** Swarm Computing, Overlay Networks, Peer-to-Peer, Distributed Systems

# Contents

# List of Algorithms

# Acronyms

**C-S** Client-Server *(pp. 3, 4)*

**DHT** Distributed Hash Table *(pp. 8, 10, 13, 23)*
**DOLR** Decentralized Object Location and Routing *(p. 13)*

**GUID** Globally Unique Identifier *(p. 13)*

**IPFS** InterPlanetary File System *(pp. 10, 24)*

**P2P** Peer-to-Peer *(pp. 1, 3, 4, 7, 8, 10, 12, 17)*

**TaRDIS** Trustworthy and Resilient Decentralised Intelligence for Edge Systems *(p. 1)*
**TCP** Transmission Control Protocol *(p. 6)*

**XOR** Bitwise Exclusive Or *(pp. 10, 11)*

# 1 Introduction

## 1.1 Motivation & Context

In today's technology-driven era, distributed systems have become a cornerstone of modern computing infrastructures. With the exponential growth of data and the increasing demand for high availability, scalability, and fault tolerance, traditional centralized systems often struggle to meet the performance and reliability requirements of contemporary applications.[19] Distributed systems, by spreading tasks across multiple interconnected nodes, offer a promising solution to these challenges by enhancing resource utilization and ensuring continuous service even in the presence of individual node failures.[19]

Trustworthy and Resilient Decentralised Intelligence for Edge Systems (TaRDIS) is an initiative whose primary goal is reduce the complexity of building correct decentralized distributed systems, providing a toolbox for supporting the development and executing of applications. TaRDIS is actively developing Babel-Swarm[1], which is a framework written in Java for easily implementing performant distributed systems.

The motivation for this engineering project is to provide a library of high-level abstractions of decentralized protocols, targeted to Babel-Swarm, in order to provide solid building blocks to users of the framework.

## 1.2 Structure

This dissertation plan is divided in the following segments:

1. In Chapter 1 we give an introduction to this project, exploring its motivation, context

2. In Chapter 2 we explore the background and related work that underpins our research. We begin by examining various Peer-to-Peer (P2P) protocols documented in the literature, then provide an overview of distributed systems frameworks, looking closely at the Babel-Swarm framework that we will target for our implementation.

[1] https://codelab.fct.unl.pt/di/research/tardis/wp6/babel-swarm

3. In Chapter 3, we go over our solution proposal, defining the main objectives and scheduling for this project.

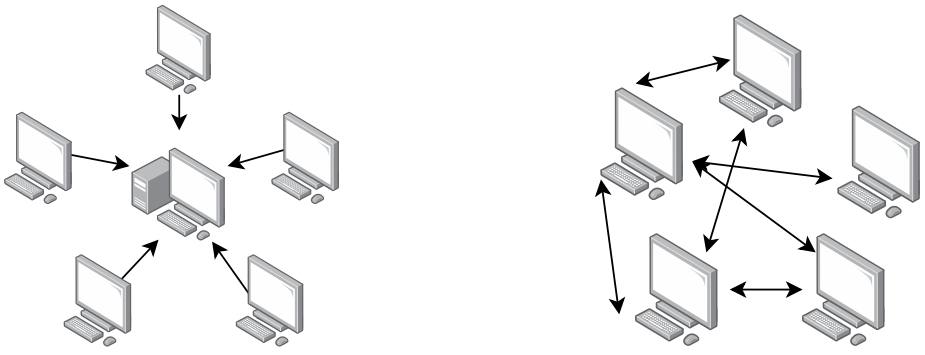
# 2

# Background & Related Work

In this chapter we present the technological context relevant to our work. First, we review some of the historical and state-of-the-art peer-to-peer protocols, and then go over frameworks for building distributed applications finalizing with the Babel-Swarm framework, within which the identified protocols of interest will be implemented in our proposed library.

## 2.1 Peer-to-peer and Swarms

P2P is a distributed networking architecture in which a computational workload in partitioned over a group of peers in a network – a "swarm". This type of architecture, differs from a Client-Server (C-S) architecture, where clients connect to a centralized server entity. Instead, it discourages the use of centralized entities and leverages the collective capabilities of individual nodes to achieve scalability, fault tolerance, and resilience guarantees[32][1].



(a) (b)

Figure 2.1: Representation of Client-Server (a) and Peer-to-Peer (b) networks.

Figure 2.1 illustrates the P2P and C-S architectures. In (a), nodes connect to a central server that mediates all communication, acting as the network's central "brain". In (b), a P2P network is depicted, where communication occurs directly between peers, and logic

is decentralized across the nodes.

The connections established between peers form an overlay network – a logical structure built atop the underlying physical network[32]. Depending on how that overlay network is organized and configured at the logical level, we can distinguish different types of P2P systems: unstructured P2P systems, where nodes connect randomly and resource discovery is usually performed through techniques like flooding or gossiping (e.g., early file-sharing networks such as Gnutella); structured P2P systems, which use a defined topology and predictable data placement(e.g., Chord, Kademlia, and Pastry); and hybrid P2P systems, which combine P2P and C-S models, where some nodes act as super-nodes to enhance efficiency (e.g., early versions of Spotify[13]).[32]

From this point onward, we will turn our attention to specific P2P protocols, with each section addressing a specific problem domain.

## 2.2 Membership Protocols

In this section we go over membership protocols. This kind of protocol is responsible for maintaining information about the members of a network. This allows the active members of a network to be aware of each other, and take decisions accordingly in the case of the joining or leaving of members and handling failure scenarios.[29]

A straightforward approach to provide a membership service could be employing a centralized entity to maintain and coordinate a list containing the members. However, this method may produce a bottleneck, creating a single point of failure, hindering performance and defeating the decentralized nature of a peer-to-peer system.[32] Therefore, we look for a decentralized approach to provide this information.

**Gossip-based protocols.** One decentralized approach is "gossip-based", leveraging an epidemic methodology to spread information among a network. In a gossip protocol, peers communicate their given state periodically with a random selection of a set of peers in the network. Over multiple rounds of "gossip", information can therefore be disseminated over a potentially large network of peers.[3]

This type of protocol can exist for different purposes, including event dissemination, where new events are shared among peers in the gossip rounds; for anti-entropy protocols, used to conciliate replicas in a network; aggregate computation, where results of different machines are combined to reach a system-wide value; or background state dissemination, where network membership is continuously shared among nodes[3]. This latter type applies to the relevant gossip-based membership protocols in the literature that we will now explore.

### 2.2.1 CYCLON

CYCLON is a gossip-based membership protocol designed for unstructured P2P overlays. It aims to maintain low-cost membership while preserving the inherent randomness of the network. This approach results in an overlay that is well-connected, with a low diameter and minimal clustering, and remains resilient to high churn and numerous node failures[29].

**Algorithm 1:** Shuffle Operation in CYCLON

**Input:** $\ell$: Maximum subset size for communication
$P$: Current peer
**Output:** Updated cache for peer $P$

```
1  foreach neighbor n ∈ P.cache do
2  |  n.age ← n.age +1
3  end
4  Q ← neighborWithHighestAge(P.cache);
5  toSend ← {new(age = 0, address = P.address)} ∪ randomNeighbors(P.cache,
   |  ℓ − 1, exclude=Q)
6  SendToPeer(Q, toSend)
7  filteredResponse ← {}
8  foreach e ∈ ReceiveFromPeer(Q, maxSize=ℓ) do
9  |  if e.address ≠ P.address and e ∉ P.cache then
10 |  |  filteredResponse ← filteredResponse ∪ {e}
11 |  end
12 end
13 foreach e ∈ cache do
14 |  if |filteredResponse| > 0 and isEmpty(e) then
15 |  |  n ← n ∈ filteredResponse
16 |  |  cache ← (cache \ {e}) ∪ {n}
17 |  |  filteredResponse ← filteredResponse \ {n};
18 |  end
19 end
20 foreach e ∈ toSend do
21 |  if |filteredResponse| > 0 then
22 |  |  n ← n ∈ filteredResponse
23 |  |  cache ← (cache \ {e}) ∪ {n}
24 |  |  filteredResponse ← filteredResponse \ {n};
25 |  end
26 end
```

#### 2.2.1.1 Protocol Overview

In this protocol, each node in the network at a given moment keeps a partial view of the network – its neighbors – that is continuously changing by the means of a periodic "shuffling" operation.

**Shuffling.** This protocol improves on a previous shuffling algorithm presented in a study by Stavrou et al. study in [25], in what it names as "enhanced shuffling". In the original algorithm, a node would periodically exchanged a subset of its neighbors with a random neighbor, in this enhanced version, instead of choosing a random it chooses its earliest known neighbor instead by using a new "age" parameter. The "age" parameter is used to more rapidly detect dead nodes, as older nodes are more likely to have been disconnected compared to fresh nodes.

The pseudocode for the enchanced shuffling algorithm[29] is presented in Algorithm 1.

### 2.2.2 HyParView

HyParView (**Hy**brid **Par**tial **View**) is a membership protocol proposed by Leitão et al. in 2007[15] for gossip-based broadcast that ensures high levels of reliability even in the presence of high rates of node failure. It does this by, instead of every node maintaining a single partial view of the network, having each node hold two: passive and active.[15].

HyParView was developed with the following objectives:

- Minimize the fanout size of the broadcast gossip protocol – that is, lower the number of nodes to which a message is forwarded – while maintaining reliability and fault-tolerance. In practice, this is achieved by keeping the active view that is used for the broadcasting smaller, while keeping the larger passive view as fallback.
- Address high failure scenarios, in which membership partial views may lose quality and take time to recover, by incorporating the Transmission Control Protocol (TCP) failure detector as a mechanism to aid fast-healing.

#### 2.2.2.1 Protocol Overview

In its essence, HyParView maintains two partial views of the network, the passive and active view, applying different policies for each, maintaining a larger set of nodes in the former and and a smaller amount of nodes in the latter.

**Active View.** This view defines the neighbors of a node, in which messages will be disseminated in broadcast. A reliable TCP connection is established with each one of the neighbors. A reactive strategy is used to refresh the active view, so as soon as a node suspects a failure of a neighbor, a random node from the passive view is selected for replacement and attempts to connect to it, in case of failure, this process is repeated until an adequate node is found.

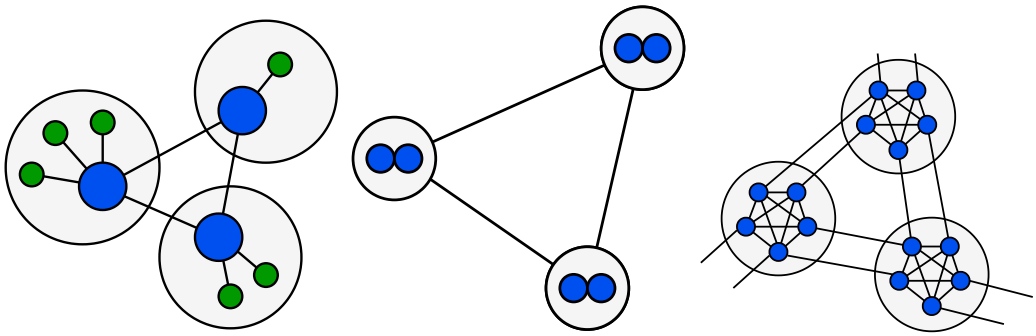
As a new connection is being established, a priority level is sent: "high" in case there are no elements in the node active view or "low" otherwise, in the former case the new

node will accept the connection even if it needs to drop another node to be able to do so, otherwise it will only accept the connection if it has a free space in its active view.

**Passive View.** This view keeps track of a set number of nodes of the network. The purpose of this view is to be a fallback to the active view, in the possibility of failures in nodes, and therefore no connection channels are kept open to reduce overhead. A cyclic strategy is used to periodically perform a shuffle operation with a random neighbor.

**Shuffling** In the shuffling operation, a node sends an exchange request to one of its neighbors in the active view. The parameters $k_a$ and $k_p$ define the number of nodes that are randomly picked from the active and passive view, respectively, and sent in the exchange message. The reason for also choosing nodes from the active view is to increase the possibility of exchanging nodes that are correct[15]. Whenever a shuffling request is accepted, both nodes will merge the received identifiers in the exchange into their passive view, and if necessary delete older identifiers randomly.

### 2.2.3 Overnesia



(a) Super-peer network (b) Replicated super-peers (c) Overnesia network

Figure 2.2: Representation of a Super-peer networks, with the blue nodes representing super-peers, green nodes representing regular peers and the gray circles representing clusters.

Overnesia is a membeship protocol proposed by João Leitão & Luís Rodrigues in 2008[16] for building P2P overlay networks comprising of replicated super-peers.

This protocol enhances traditional "super-peer network" designs – unstructured P2P networks in which certain peers, known as super-peers, serve as servers for clients[34] (which are regular peers) and establish their own dedicated overlay network, routing querys among themselves (as seen in Figure 2.2 (a)). This approach has the potential to optimize resource management by creating a heterogeneous system. In this configuration, super-peers assume the bulk of responsibilities for routing and load balancing, while regular peers only need to interact with their designated super-peer for all operations.[34]

This solution however has deficiencies, such is the case of super-peer failures, as the peers need to find a new suitable super-peer, and informed attacks on the network are more feasible[18], as an attacker only needs to target the super-peers to incapacitate the network. Replicating super-peers ( as seen in Figure 2.2 (b)) – forming "virtual super-peers" as sets of super-peers, improves handling of failures and attacks[18].

Overnesia creates a robust solution of a virtual super-peers network, by creating a network of "islands" that are composed of fully connected virtual super-peers, and islands connect among each other[18], as represented in Figure 2.2 (c).

## 2.3 Distributed Hash Table

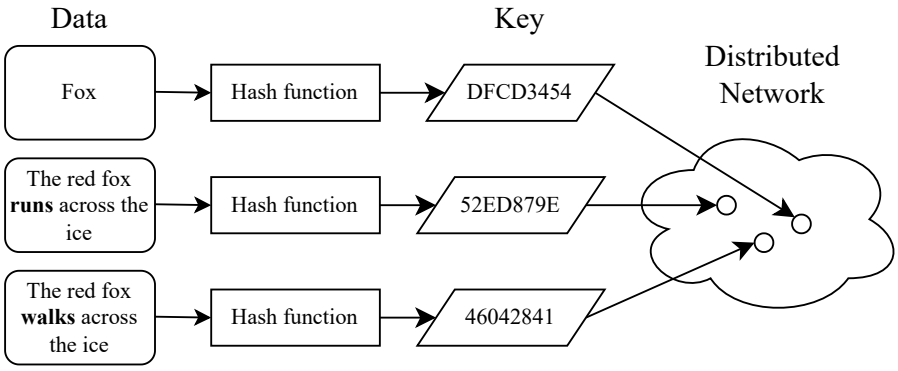



Figure 2.3: Representation of a DHT (adapted from [31]).

A DHT is a distributed system design primarily used for dealing with distributed data. It provides a service similar the one of a hash tables for looking up data over a network of nodes in a P2P network[31]. A simplified visual representation of a DHT can be seen in Figure 2.3, which illustrates how input data is hashed to generate a key and mapped to a node in the distributed network. It also demonstrates how, due to hashing, even minor changes in input data – such as altering "runs" to "walks" – can result in a drastically different key.

In a DHT network, each node is responsible for a portion of a given key-space, saving the values associated to each key. Depending on the protocol, different strategies are used for a fair distribution of keys over the nodes, commonly based on the consistent hashing[12] technique, and for effective failure handling, as for example keeping the integrity of the data in case of disconnections.

DHTs have been used to great effect as solutions for problems such as distributed file systems, search, storage, load balancing or replication[35]. In this section, we will examine various DHT protocols presented in the literature.

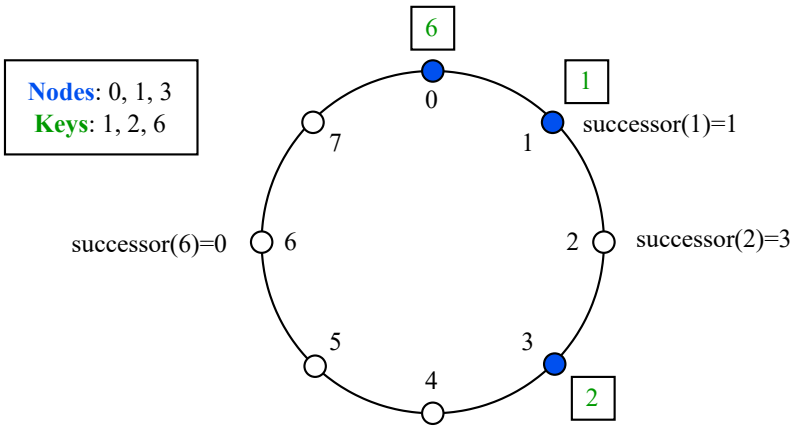



Figure 2.4: Chord identifier circle example (adapted from [26])

### 2.3.1 Chord

Chord is a distributed lookup service protocol for peer-to-peer applications presented in 2001 by Stoica et al.[26]. It attempts to distinguish itself from other look-up protocols by aiming for simplicity[26] and supporting a single operation: mapping a key to a node in the network. Other operations, such as storing values associated to keys, would be part of the specific application logic.

When routing a message, it gets to its destination in a sequence $O(log(N))$ hops. In order to maintain this efficient routing, a node needs to keep track of $O(log(N))$ other nodes at a given time, However, due to network churn, this information can quickly become outdated degrading the quality of the routing. Chord guarantees that only one routing hop information needs to be correct per node in order to successfully route a message.[26]

#### 2.3.1.1 Protocol Overview.

**Consistent Hashing.** Chord uses the consistent hashing technique[12][26]. This hashing technique enables the load balancing of the keys over the nodes, and provides that for given $n$ keys and $m$ slots, when resizing the hash table only $n/m$ keys need to be remapped.[26]

All keys in consistent hashing exist within a circular address space with modulo $2^m$, with $m$ being a number sufficiently high so that is highly unlikely two independent hashing of keys collide. The node ids are generated by hashing the IP address of the node, and the keys identifiers are generated by hashing the key.

In this scheme, a key $k$ is assigned to the node whose id numerically equals or follows its identifier: this node is named the "successor" of $k$ – `successor(k)`. Whenever a node $j$ joins the network, all keys associated to `successor(j)` are transferred to $j$, and whenever a node $l$ leaves the network its keys are mapped to `successor(l)`. Figure 2.4 presents a visual illustration of how consistent hashing is implemented to assign keys to nodes in the Chord protocol, where the nodes in the network are represented in blue in the identifier

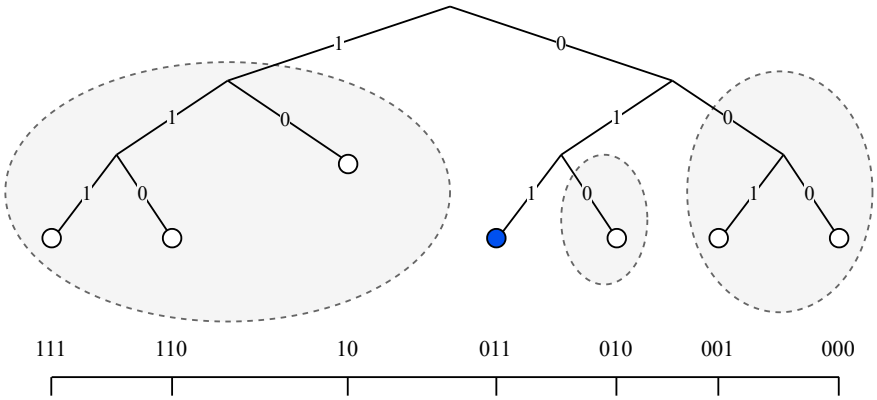



Figure 2.5: Representation of the Kademlia binary tree, with each circle representing a node in the network. The grey ovals represent the group of nodes that the blue colored node must have a contact[20].

space, and the different keys in green are mapped to each node according to its successor.

**Node state and Routing.** Every node, with $m$ as the number of bits in the identifier, keeps a routing table named "finger Table", containing at most $m$ entries, and where the $i$-th entry – named "finger" – contains the node $s$ such that $s = successor(n + 2^{i-1})$, where $1 \leq i \leq m$[26].

When searching for a key $k$ in the network, a node $n$ will first attempt to verify if $successor(k)$ exists in its finger table, if it doesn't, it searches its finger table and finds the node $p$ whose ID least precedes $k$. It then asks the node $p$ for the node it knows that least precedes $k$, and repeats for each received node, until a node $p'$ is found such that $k$ falls between $p'$ and $p'.successor$, as it can conclude $p'.successor$ is $successor(k)$.

### 2.3.2 Kademlia

Kademlia is a P2P DHT protocol introduced in 2002 by Petar Maymounkov & David Mazières[20]. Is employs a topology based on a Bitwise Exclusive Or (XOR) metric to measure distance between nodes. It offers robustness and performance improvement compared to the earlier DHT protocols such as Chord (2.3.1), by allowing nodes to learn information from received queries, allowing parallel requests in order to accelerate look-ups, and enabling iterative routing[20].

As of writing, the Kademlia algorithm or variations of it are used in some of the largest public DHTs, such as: InterPlanetary File System (IPFS), the Ethereum blockchain network node discovery protocol, and the mainline DHT in Bittorrent[33].

#### 2.3.2.1 Protocol Overview

Each node in Kademlia is effectively a leaf of a binary tree, with its position defined by the shortest unique prefix of its id[20]. For every sub-tree that does not contain a given

node, Kademlia makes sure that it knows at least one node, therefore guaranteeing that a node can find any other node by its ID. This is exemplified in the Figure 2.5, where a node highlighted in blue is sure to know at least one contact in each of its sub-trees that it's not part of.[20]

The Kademlia protocol defines the following operations: `ping`, for verifying if a node is online; `store`, to save a (key, value) pair in a node; `find_node`, that takes a 160-bit value as argument and receives a response of a (IP, UDP port, Node ID) for $k$ nodes or if it doesn't know about $k$ nodes, only the ones it knows about: `find_value` has the same behavior as `find_node`, with the difference being it returns a stored value for the key if present.

We will now elaborate more on how the XOR metric works, how the node state is composed, and explain the lookup procedure.

**XOR metric.** Kademlia distance metric is defined as follows: $d(x, y) = x \oplus y$.

The following properties can be inferred for this metric[20]:

$$d(x,x) = 0, \quad d(x,y) > 0 \text{ if } x \neq y, \quad \forall x, y : d(x,y) = d(y,x), \quad d(x,y) + d(y,z) \geq d(x,z)$$

Keys and node identifiers are mapped in a 160-bit identifier space. A key is assigned to the closest node according to the XOR distance metric. Whenever the tree is not complete, there might be a draw between two nodes, the closest node $n$ will therefore be the one whose identifier $n'$, obtained by flipping the bits corresponding to the empty branches in the tree, is closer to the key.

**Node state and Routing table.** For each $0 \leq i \leq 160$, a node stores a list of other nodes encompassing a distance of $2^i$ and $2^{i+1}$ from itself, and denoted as triples of (IP address, UDP port, Node ID). These lists are named $k$-buckets, and sorted by last time seen and a least-recently seen policy for removal of nodes, for a max size of the system-wide parameter $k$.[20]. The routing table can be represented as a binary tree whose leaves are k-buckets, with each one containing nodes that share the same prefix.

**Lookup.** For its operations, Kademlia uses a recursive lookup algorithm for finding $k$ closest nodes to some given node ID. First it selects $\alpha$ (a system-wide parameter for concurrency) nodes from the closest non-empty $k$-bucket, or just $\alpha$ nodes it knows if the $k$-bucket has less than $\alpha$ nodes. It then sends a `find_node` request to each one in parallel, learning from the responses, and repeating this procedure its $k$ closest nodes it, choosing $\alpha$ it hasn't yet queried. This procedure ends when a round doesn't return a new closer node, finishing with a last attempt to the $k$ closest nodes it hasn't queried.[20]

### 2.3.3 Pastry

Pastry is a structured fault-resistant, scalable and reliable peer-to-peer architecture proposed by Antony Rowstron & Peter Druschel in 2001[24], for large scale object location and

routing, leveraging a self-organizing overlay that efficiently adapts to node failures, and location-awareness for efficient routing.[24] It is expected that in a network composed of $N$ nodes, Pastry is able of routing a message in $O(log(N))$ steps, with the authors evaluating that this performance is maintained even in large-scale networks with $N$ up to 100000.

Pastry has been used effectively for building various systems such as decentralized publish/subscribe (e.g. SCRIBE), and P2P storage utilities (e.g. PAST).

#### 2.3.3.1 Protocol Overview

Pastry uses numerical prefix matching to route messages. In each routing step a message will be forwarded to the node with the most numerically close id in relation to the current node. It

**Algorithm 2:** Routing Operation in Pastry (adapted from [24])

**Input:** $D$: Destination key
$A$: Current node ID
$L$: Leaf set of the current node
$R$: Routing table of the current node
**Output:** Message forwarded to the appropriate node

**if** $L_{-\lfloor |L|/2 \rfloor} \leq D \leq L_{\lfloor |L|/2 \rfloor}$ **then**
  //D is within range of our leaf set
  forward to $L_i$, such that $|D - L_i|$ is minimal;
**end**
**else**
  // use the routing table
  // shl: length of the prefix shared among D and A
  $l \leftarrow$ `shl`$(D, A)$;
  **if** $R[l][\ D[l]\ ] \neq null$ **then**
    forward to $R[l][D_l]$;
  **end**
  **else**
    // rare case: no direct match in the routing table
    forward to $T \in L \cup R \cup M$, such that;
      `shl`$(T, D) \geq l$;
      $|T - D| < |A - D|$;
  **end**
**end**

**Node state.** Each node in Pastry keeps a state composed of a Leaf Set, a Routing Table and a Neighboring Set. The Leaf Set $L$, contains the $|L|/2$ mathematically closest smaller nodes and $|L|/2$ mathematically closest larger nodes. The Neighboring Set keeps $N$ keeps a $|N|$ that are close, according to a given proximity metric. The Routing Table $R$ having in mind that a given id is represented as a sequence of digits in base $b$, contains $\lceil \log_{2^b} N \rceil$

rows, each containing $2^b - 1$ columns. A given row $n$, shares the first $n$ digits with the current node but vary in the $n + 1$th digit

**Routing.** The routing algorithm for Pastry is represented in Algorithm 2. In a first step, it checks the Leaf Set, and if it falls within the Leaf Set range, it forwards to the mathematically closer node to the key, otherwise the Routing Table is used, and the message is forwarded to an entry that shares the same prefix but by one more digit than the present node. In case there is no appropriate entry in the Routing Table, it forwards to a node that is numerical closer to the node, while still sharing the same prefix as the present node.

**Locality.** The distance metric, allowing Pastry to determine the distance between nodes is expected to provided by the application layer, such as by geographic distance and IP routing hops[24]. This proximity metric is used to ensure when a node joins the network, it connects to a close node, and therefore initialize the Routing Table using information from nearby nodes[24]. This is an important aspect of Pastry as the routing table is meant to forward messages to nodes that not only shares a longer prefix but optimally are also relatively close to the present node.

### 2.3.4 Tapestry

Tapestry structured peer-to-peer overlay network for location-independent message routing, proposed by Zhao et al. in 2004. It builds an efficient and scalable overlay network, leveraging a self-repairing, soft-state-based routing mechanism. [36]. The Tapestry infrastructure exposes a Decentralized Object Location and Routing (DOLR)[7] interface, providing the routing of messages to nodes and object replicas[36].

**DOLR.** The DOLR abstraction provides a decentralized directory service[7]. In this system, resources – specifically, object replicas – are identified using an opaque `objectId`. This identifier contains no information about the resource's physical location, which means that the object replicas can be stored anywhere within the system[7][36]. This allows for messages to be forwarded to the nearest endpoint – a locality property not available in DHT[7]– improving efficiency, and to different object replicas in the case of failure scenarios and instability in the network[36].

#### 2.3.4.1 Protocol Overview.

Tapestry nodes are randomly assigned a `nodeId` – $N_{id}$ – from a 160-bit identifier space, and application-specific endpoints receive a Globally Unique Identifier (GUID) – $O_G$ – from the same identifier space. [36]

Due to the fact that in Tapestry efficiency increases with the size of the network[36], it is encouraged that different applications run on the same overlay network. In a scenario

where multiple applications run on the network, the $A_{id}$ identifier is used to identify the specific application.

The Tapestry API is defined as follows: `publishObject`($O_G, A_{id}$), to publish object $O$ on the local node; `unpublishObject`($O_G, A_{id}$), to attempt to remove location mappings for $O$;`routeToObject`($O_G, A_{id}$) to routes a message to location of an object with GUID $O_G$; `routeToNode`($N, A_{id}$,`exact`) to route message to application $A_{id}$ on node $N$, with exact specifying whether destination ID needs to be matched exactly[36].

## 2.4 Adaptive Topologies

In this section, we will go over protocols for constructing and maintaining overlay topologies for peer-to-peer networks. The overlay topology defines how the peers are connected to each others in the network, different topologies can have severe impact in the performance of functions such as for example: data aggregation, searching, information dissemination.[11]

Protocols for adaptive topologies may be important for highly dynamic networks, where we look to adapt the topology at runtime in order to achieve better efficiency such as reduced latency or improved network bandwidth usage.

This kind of protocol may also allow for the jump-starting of other protocols, with a technique known as "bootstrapping"[22][30]. In this technique, a given desired topology can be built efficiently starting from a random unstructured overlay network. This is relevant in cases where we might need to quickly start a network from the ground up, with very little or no previous nodes from an already established network to use as a basis.[22].

### 2.4.1 T-MAN

T-MAN is a gossip-based topology management protocol proposed by Márk Jelasity and Ozalp Babaoglu in 2005[11], whose main motivation is the decentralized constructing and management of overlay topologies with known a priori properites.

In this protocol, a desired topology is formally specified by a ranking function, which nodes utilize to evaluate and prioritize potential neighbors based on their utility in achieving the desired topological configuration.

When transitioning topologies, the T-MAN protocol exhibits a convergence time that scales logarithmically with the network size, thereby rendering it a viable solution for environments that demand rapidly changing topologies or are prone to massive churn[11]

#### 2.4.1.1 Protocol Overview

**Ranking function.** The ranking function receives as arguments a list of nodes $y_1, ..., y_m$ and a base node $x$, outputting the $m$ nodes ordered by their utility to achieve the target topology.[11] The target is such that the partial view of the node, $view_x$, contains exactly

**Algorithm 3:** Active Thread

**do** at a random time once in each consecutive interval of $T$ time units;
- $p \leftarrow$ `selectPeer()`;
- $d \leftarrow$ (myAddress, myProfile);
- buf $\leftarrow$ view $\cup$ $\{d\}$ $\cup$ {rnd.view};
- `send(`$p$, $buf$`)`;
- `recv(`$p$, $buf\_p$`)`;
- buf $\leftarrow$ buf_p $\cup$ view;
- view $\leftarrow$ `selectView(`$buf$`)`;

(a)

**Algorithm 4:** Passive Thread

**do** forever;
- `recv(`$q$, $buffer\_q$`)`;
- $d \leftarrow$ (myAddress, myProfile);
- buf $\leftarrow$ view $\cup$ $\{d\}$ $\cup$ {rnd.view};
- `send(`$q$, $buf$`)`;
- buf $\leftarrow$ buf_q $\cup$ view;
- view $\leftarrow$ `selectView(`$buf$`)`;

(b)

Figure 2.6: Active and Passive Threads in T-MAN (adapted from [11])

the $c$ first elements of the the output list of the ranking function, however, due to churn in the network this problem becomes complex, so in most situations the attempt is to be as close as possible to the target.[11].

**Topology Construction.** The protocol maintains two threads, an active thread for communicating with other nodes and a passive thread that receives communications from other nodes, which are described by the pseudocode in Algorithm 3 and Algorithm 4 respectively.

The `selectPeer()` function applies the ranking function and returns the first entry that corresponds to a node that is alive, and the `selectView(nodes)` applies the ranking function to `nodes` and returns the first $c$ first elements.

The set `rnd.view` is a random selection of nodes of the entire network provided by a peer sampling service[11]. The purpose of this random sampling is to aid in convergence of the optimal set of neighbors in topologies with a large diameter, essentially allowing nodes "living" in opposite ends of a topology to "meet each other" more easily[11].

### 2.4.2 X-BOT

X-BOT (**B**ias the **O**verlay **T**opology according to a target criteria **X**) is a protocol proposed by Leitão et al. in [17] for the constant attempt of improving underlying network overlay topologies according to efficiency metrics. It does this whilst trying to maintain the node degree, as to maintain the overlay connectivity, using limited information as to not increase communication overhead between nodes and prioritizing overlay stability as much as possible to avoid disruption to the services running on top of it.

In a similar way to HyParView(see section 2.2.2), X-BOT makes use of a passive and active views, however it trades the stability of the active view in HyParView, by constantly trying to optimize it. It does this in "optimization rounds", where nodes of the active view

are swapped with nodes of the passive view in an attempt to replace existing links with more optimized links, using information provided by an oracle that estimates link "cost" according to a given metric. This oracle could be provided, for example, by calculating RTT values over nodes or by Internet Service Providers, as explored by Leitão et al. in [17].

#### 2.4.2.1 Protocol Overview.

**Unbiasing.** To preserve the desirable properties of a random overlay – such as low average path length, low clustering coefficient, and connectivity [17] – X-BOT seeks to minimize bias while maintaining randomness. It achieves this by ensuring that a portion of the nodes in the active view are "unbiased" or "high-cost nodes," as determined by the protocol parameter $\mu$. The remaining nodes, are referred as "low-cost nodes"[17].

**Optimization rounds.** The optimization rounds are performed in groups of four nodes and consist of four steps (simplified for more clarity):

1. In step one, a node $n$ randomly selects a sample from its passive view as potential targets. It then evaluates each element in its active view using the oracle to compare the cost of its current neighbors with those in the passive view. If any node $c$ from the passive view is better than one in the active view $n_a$ , an optimization request is sent to $c$ asking to exchange $n_a$ with $c$ and in case of a positive reply, it replaces $n_a$ with $c$.[17]

2. In step two, the node that previously received the optimization request – $c$ – tries to place $n$ in its active view. If it is full, it will try to swap the highest-cost neighbor $d$, sending it a replace request, with the original $n_a$ as exchange. In case of a positive reply, $d$ is replaced for $n$ in the active view.[17]

3. Step three starts with $d$ receiving a replace request from $c$ and the node $n_a$ as exchange, it consults the oracle in order to verify if the exchange is beneficial. If it is, it sends a switch request to $n_a$, and in case of successful reply, exchanges $c$ with $n_a$ in its passive view.[17]

4. Step four starts with $n_a$ verifying if $n$ is still part of its active view, and in case it is, sending a successful switch response back do $d$ and a disconnection message to $n_a$.[17]

## 2.5 Communication Protocols

In this section, we edo an overview of protocols that implement point-to-multipoint communication abstractions – specifically broadcast and multicast. Point-to-multipoint communication is a fundamental building block in distributed systems and is used across

a wide range of applications. Therefore, it is essential to employ protocols capable of delivering messages quickly and efficiently to a potentially large numbers of receivers within a network[27].

The algorithms discussed in this section aim to replace the traditional network-level IP point-to-multipoint communication mechanisms by implementing them at the application level, which allow them to not depend on network services while making them more flexible.

### 2.5.1 Scribe

Scribe is a P2P protocol proposed by Castro et al. in 2002[6], that provides an application-level multicast built on top of Pastry (see in 2.3.3). It leverages the reliability, locality and self-organization of Pastry, to build efficient and fault-tolerant multicast trees for the dissemination of messages[6].

It leverages Pastry for creating and managing the groups and for creation of trees where data is disseminated.

Scribe provides the following API: `create(credentials, groudId)`, in order to create group with a given groupId; `join(credentials, groupId, messageHandler)`, in order to join a specific group and make it so received multicast messages are passed to the specified messageHandler; `leave(credentials, groupId)` to leave the group specified with the groupId and `multicast(credentials, groupId, message)` in order to multicast a given message to the group with groupId. The credentials used for this operation are part of the acces control mechanism inherited from Pastry[6][24]

### 2.5.2 GoCast

GoCast (**G**ossip-enhanced **o**verlay multi**cast**) is a overlay network protocol proposed by Tang et al. in 2005 for multicast. It uses a tree embedded in the overlay for the rapid dissemination of messages, with a gossip component that exchanges summaries between nodes in the background and helps recover from failures and recover lost messages due to possible disruptions in the tree-based multicast.

This "dual-approach" comes from the authors notice of the deficiencies found in previous work using either gossip-based multicast and reliable tree-based broadcast respectively[27]. Reliable tree-based multicast, relies on retransmissions in order to treat failures, and therefore reducing the throughput considerably, as stated in a previous study[23]. Gossip-based multicast although very excelling in high failure scenarios, is particularly slow as latency is dependent of the period of gossip rounds, and its obliviousness to network topology may lead to overloaded links.[27].

### 2.5.3 Plumtree

Plumtree (**P**ush-**L**azy-**P**ush **M**ulticast tree) is a broadcast protocol for peer-to-peer networks, proposed by Leitão et al. in 2007[14] based on gossip protocols.

This protocol works by building and maintaining a broadcast tree built on top of a gossip-based overlay, leveraging unused links for detecting and recovering from failures and for tree repairing. The broadcast on the tree branches uses a "push" gossip approach, as the nodes forward the message to randomly selected neighbors as soon as they receive it, and for the links not part of the tree a "lazy-push" strategy is used, meaning that only an identifier of the message is sent to those neighbors such that they can pull the message if not received previously as within a configurable small time window.

In practice this means that less traffic is generated for dissemination of messages while retaining the reliability of a pure gossip protocol. With this novel approach, Plumtree can achieve low overhead and high reliability.

## 2.6 Development Frameworks

In this section we will go over development frameworks to build distributed computing systems. These frameworks reduce the developer burden by abstracting lower level details, such as communication, and exposing APIs that deal with common concerns in the development of distributed systems protocols.

After an overview of the Isis and Cactus frameworks, we conclude on the Babel framework and its current successor Babel-Swarm, on which we will work upon further to develop the presented protocols in the previous sections.

### 2.6.1 ISIS

ISIS is a system for building fault-tolerant distributed computing applications[5], that tries to abstract the complexity burden of building such applications from the programmer. It was historically used by companies and universities in scenarios ranging from financial trade floors and telecommunications switching systems.[28]

#### 2.6.1.1 Programming Model

The framework enables replication, by providing support for specifying $k$-resilient objects, with the state replicated on $k + 1$ instances. This provided abstract data type provides the following guarantees: consistency of operations, recovery of failures, and provided that the number of failed components in a given time frame is less or equal than $k$: availability of the system and progress of operations.[4]

The programmer writes the desired application, specifying resilient object types using a high level language provided by ISIS. This specification is turned into the implementation

of a resilient object and calls are inserted in the runtime system that ensures the mechanisms for automated replication, data consistency and fault tolerance.

### 2.6.2 Cactus

Cactus is a framework for building distributed computing applications, directed to solving problems in the science and engineering domains.[10]

In an attempt to converge the work of multi-disciplinary and geo-distributed teams of scientists, researchers and engineers, and to handle large scale scientific computations, Cactus attempts a heavy focus on modularity, portability, wide support, ease of use, and a concern on future-proofing with continuous incorporation of cutting-edge computing technology.[10]

#### 2.6.2.1 Programming Model

The Cactus framework is modeled with modular units "thorns". Thorns can be written in any language, and are defined by its "implementation", which is "defined by a group of variables and parameters that are used to implement some functionality"[2]. Thorns communicate with each other exclusively using the Cactus API and thorns that share the same implementation can be easily exchanged. One such example could be a message passing interface, which "drivers", that could represent different types of underlying protocol, could implement and therefore be exchanged.

Because of the easy interchangeability and uniformity of Cactus, and due its prevalence on the science community and its usage in large-scale experiments[10] one can build high-performance distributed systems, by composing and building upon "battle-tested" thorns.[2]

## 2.7 Babel

Babel is a Java framework for efficiently implementing performant distributed systems.[8] It uses an event-driven model that abstracts the low level details, such as concurrency management, networking and timers, thus enabling the quick prototyping of distributed algorithms.

The framework takes advantage of ideas previously explored in solutions such as Isis and Cactus(mentioned earlier). It tries to improve them and touch on some of its potential issues, such as the lack of concurrency management in Cactus, and the single-threaded concurrency model of Appia.[21]

In its essence, an application made using the Babel framework is composed of a set of protocols that may communicate with each other over notifications and requests and to other protocols in external systems over messages. To enable this paradigm, Babel exposes an API responsible for registering handlers for the possible events in the protocol

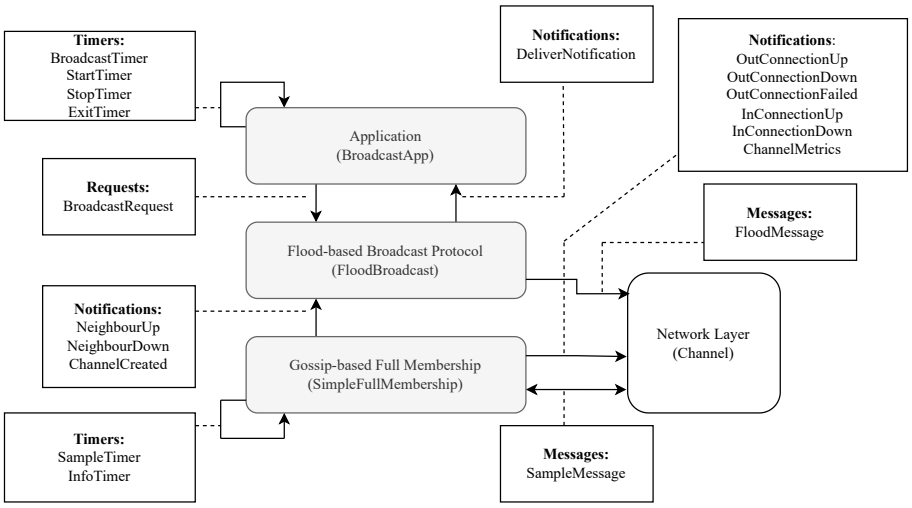



Figure 2.7: A "blown-up" diagram of a sample Babel application.

life-cycle, mediating the inter-protocol communication and abstracting the network layer using "channels".[8]

The developer is thus responsible for implementing the various protocols by extending the built-in abstract class `GenericProtocol`. This involves defining the protocol logic by implementing callbacks for the registered mechanisms, while also extending relevant abstract classes from the framework, such as `ProtoNotification`, `ProtoMessage`, `ProtoTimer`, and `ProtoRequest`, as needed.[8]

An example of a sample Babel application is present in Figure 2.7. It has three layered protocols, with `BroadcastApp` as the application at the top, a `FloodBroadcast` protocol providing a flooding broadcast protocol, and at lowest level we see `SimpleFullMembership` which provides a membership protocol. We can see the decoupling capabilities of Babel, as concerns are effectively divided between the layers, and communication is accomplished using notifications and requests.

### 2.7.1 Architecture

We will now go over the main components of the architecture of Babel. Figure 2.8 shows a high-level vision of the architecture.

**Protocols.** Each protocol operates in a single thread. To handle incoming events from the Babel core, each protocol implements an event queue, and is processed sequentially. Since all communication between protocols occurs through message passing, this approach effectively addresses concurrency issues.[8]

**Babel core.** The Babel core is the central component of the framework. It is responsible for the mediation of all communications between protocols in a system and with the

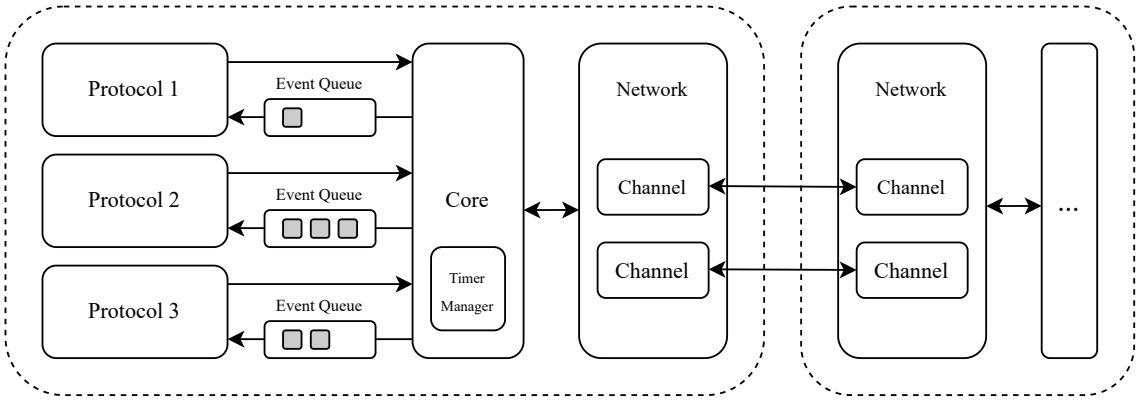



Figure 2.8: Babel architecture (adapted from [8, p. 150])

network by passing message to to the appropriate channels. It is also responsible for the timers, keeping track of them and when one is triggered, dispatching an event to its protocol.[8]

**Networking.** The networking is handled by the channels abstraction. The channels expose a simple to use interface with the following methods: `createChannel`, `openConnection`, `closeConnection`, `sendMessage`. Babel includes many types of built-in channels, for different types of protocols:

- Dedicated "client" and "server" channels, for which the server only accepts incoming connections and does not initiate them.
- A channel with automatic acknowledgments.
- A channel that implements $\phi$-accrual failure detector[8], notifying interested protocols whenever there is suspicion of failure of a peer.

There is also the possibility for the developer to extend the program with new specialized channels when needed.[8]

### 2.7.2 Babel-Swarm

Emerging decentralized systems introduce demands in areas not addressed by the original Babel implementation. These include:

- **Automated member configuration**,
- **Dynamic system reconfiguration** throughout its operational lifecycle.
- **Security measures**, such as decentralized authentication of components and secure data channels.

Babel Swarm is an effort to extend Babel[9] by incorporating solutions to these demands. Those will be in the following:

**Automated configuration.** In Babel a node might pre-configured with essential information, such as desired parameters, and a "contact" as a gateway to the system. Babel Swarm improves on this by adding the `DiscoveryProtocol` and `DiscoverableProtocol` abstract classes. The developer can extend the former to create a protocol with discovery mechanisms, and the latter that includes methods for requesting contacts, and is handled by a different initialization routine as the protocol will not start while a contact is not found.[9]

**Automatic Parameter definition.** A new abstract class `SelfConfigurableProtocol` extending the `DiscoverableProtocol` class adds support for automatic configuration of parameters through the use of `@AutoConfigureParameter` for fields that will be dynamically configured. For use of this mechaninsm, two implementations were built: one enables the ability to copy the configuration from another active peer; and another enables DNS extraction, which reads key-value pairs from a TXT record, and apply to the parameters.[9]

**Security mechanisms.** Babel Swarm introduces authentication mechanisms, through the use of public key cryptography and self-signed certificates, channel security with support for partial and mutual authentication.[9]

# 3

# Proposed Approach

In this section we will go over some of the planning for the implementation of the protocols for Babel-Swarm, selecting the ones protocols that will be targeted, and elaborating on the testing techniques that we will be using in order to validate the implementations and test the performance.

## 3.1 Implementation details

In the Chapter 2 we went over different protocols for peer-to-peer application. We want to implement some of these protocols as a library of abstractions targeting the Babel-Swarm framework. Those are the follow:

- From the membership abstractions (in section **??**) we will the CYCLON protocol and integrating HyParView, as a previous effort has been made to implement the HyParView protocol in Babel-Swarm, so we look to find a common abstractions between the two, and make available both options for the end user.
- For the DHT abstractions we will implement Chord and Kademlia.
- For the adaptive topology abstractions we will be implementing X-BOT.
- For the communication abstractions we will be implementing both GoCast and Plumtree as broadcast and multicast options respectively.

For he development of these protocols, we aim for a high quality standard in software engineering, providing these protocols as high-level solutions and loosely coupled from application logic. Modularity will be key as we try to locate the common features between the protocols and provide them as building blocks, that will reduce code duplication and allow for easier future developments. Another important aspect is portability, as we want make it easy to easily interchange protocols, providing the proper interfaces for each of the common abstractions.

## 3.2 Evaluation and testing

For evaluating and testing our solutions we will be developing proof-of-concept applications. Such examples are:

- Information broadcast applications.
- Publish-Subscribe applications
- Distributed files system such as (e.g. IPFS).

These proof-of-concepts will serve as demonstrate of real life examples of systems built with Babel, and also include testing suites for measuring the built abstractions correctness according to underlying protocol specification, and stress test benchmarks, emulating real-life usage of the built applications, that will evaluate our solutions performance and scalability.

## 3.3 Tasks and Scheduling

In this section we go over the scheduling for this project. The Gantt chart with the detailed task distribution is presented in Figure 3.1. The different phases of the project go as follows:

1. In a first phase we will design a preliminary solution. This solution will be a working prototype of the full implementations.
2. In a second phase we will do a second iteration of the the preliminary solution, and work towards the finished deliverables.
3. In a third phase we will do a final, and more in depth, analysis and evaluation of the solutions, tweaking them as necessary.
4. The final phase, starting in July will be the writing of the dissertation and the subsequent final presentation.

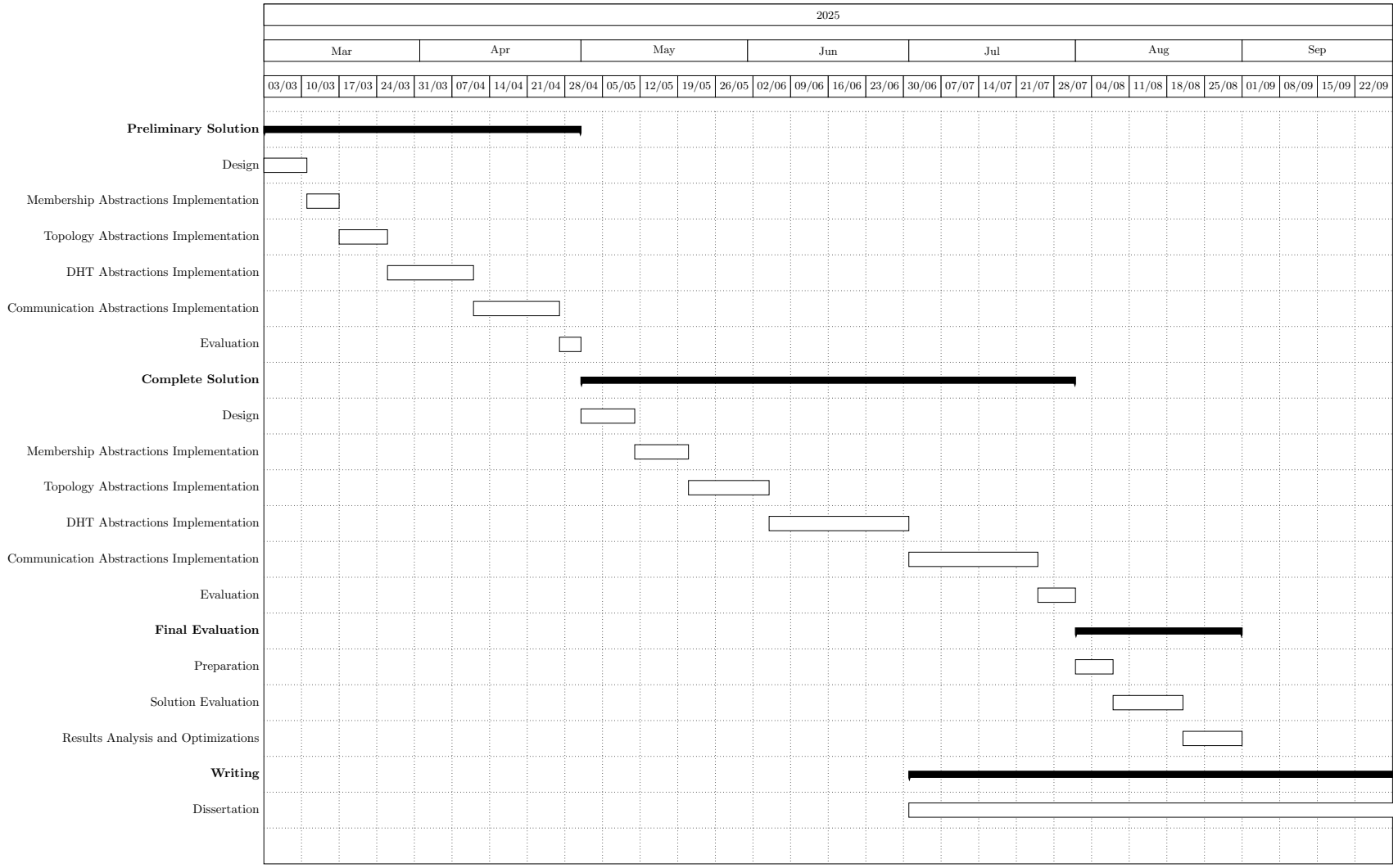



Figure 3.1: Gantt chart of the scheduling over the course of the project.

# Bibliography


[1] [Online; accessed 18-December-2024]. URL: https://www.geeksforgeeks.org/peer-to-peer-p2p-architecture/ (cit. on p. 3).

[2] [Online; accessed 20-December-2024] (cit. on p. 19).

[3] K. Birman. "The promise, and limitations, of gossip protocols". In: *ACM SIGOPS Operating Systems Review* 41.5 (2007), pp. 8–13 (cit. on p. 4).

[4] K. P. Birman. *Isis: A system for fault-tolerant distributed computing*. Tech. rep. Cornell University, 1986 (cit. on p. 18).

[5] K. P. Birman et al. *An Overview of the Isis Project*. Tech. rep. USA, 1984 (cit. on p. 18).

[6] M. Castro et al. "SCRIBE: A large-scale and decentralized application-level multicast infrastructure". In: *IEEE Journal on Selected Areas in communications* 20.8 (2002), pp. 1489–1499 (cit. on p. 17).

[7] F. Dabek et al. "Towards a common API for structured peer-to-peer overlays". In: *International Workshop on Peer-To-Peer Systems*. Springer. 2003, pp. 33–44 (cit. on p. 13).

[8] P. Fouto et al. "Babel: A Framework for Developing Performant and Dependable Distributed Protocols". In: *2022 41st International Symposium on Reliable Distributed Systems (SRDS)*. 2022, pp. 146–155. DOI: 10.1109/SRDS55811.2022.00022 (cit. on pp. 19–21).

[9] T. Galvão et al. "Suporte ao Desenvolvimento de Novas Aplicações Descentralizadas". In: () (cit. on p. 22).

[10] T. Goodale et al. "The Cactus Framework and Toolkit: Design and Applications". In: *Vector and Parallel Processing – VECPAR'2002, 5th International Conference, Lecture Notes in Computer Science*. Berlin: Springer, 2003. URL: http://edoc.mpg.de/3341 (cit. on p. 19).

[11] M. Jelasity and O. Babaoglu. "T-Man: Gossip-based overlay topology management". In: *International Workshop on Engineering Self-Organising Applications*. Springer. 2005, pp. 1–15 (cit. on pp. 14, 15).

[12] D. Karger et al. "Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web". In: *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*. 1997, pp. 654–663 (cit. on pp. 8, 9).

[13] G. Kreitz and F. Niemela. "Spotify–large scale, low latency, P2P music-on-demand streaming". In: *2010 IEEE Tenth International Conference on Peer-to-Peer Computing (P2P)*. IEEE. 2010, pp. 1–10 (cit. on p. 4).

[14] J. Leitao, J. Pereira, and L. Rodrigues. "Epidemic broadcast trees". In: *2007 26th IEEE International Symposium on Reliable Distributed Systems (SRDS 2007)*. IEEE. 2007, pp. 301–310 (cit. on p. 18).

[15] J. Leitao, J. Pereira, and L. Rodrigues. "HyParView: A membership protocol for reliable gossip-based broadcast". In: *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)*. IEEE. 2007, pp. 419–429 (cit. on pp. 6, 7).

[16] J. Leitao and L. Rodrigues. "Overnesia: an Overlay Network for Virtual Super-Peers". In: (2008) (cit. on p. 7).

[17] J. Leitão et al. "X-bot: A protocol for resilient optimization of unstructured overlay networks". In: *IEEE Transactions on Parallel and Distributed Systems* 23.11 (2012), pp. 2175–2188 (cit. on pp. 15, 16).

[18] J. C. A. Leitão and L. E. T. Rodrigues. "Overnesia: a resilient overlay network for virtual super-peers". In: *2014 IEEE 33rd International Symposium on Reliable Distributed Systems*. IEEE. 2014, pp. 281–290 (cit. on p. 8).

[19] M. T. Liu. "Distributed computing". In: (1992), p. 560. DOI: 10.1145/131214.131287 (cit. on p. 1).

[20] P. Maymounkov and D. Mazieres. "Kademlia: A peer-to-peer information system based on the xor metric". In: *International workshop on peer-to-peer systems*. Springer. 2002, pp. 53–65 (cit. on pp. 10, 11).

[21] S. Mena et al. "Appia vs. Cactus: comparing protocol composition frameworks". In: *22nd International Symposium on Reliable Distributed Systems, 2003. Proceedings.* 2003, pp. 189–198. DOI: 10.1109/RELDIS.2003.1238068 (cit. on p. 19).

[22] A. Montresor, M. Jelasity, and O. Babaoglu. "Chord on demand". In: *Fifth IEEE International Conference on Peer-to-Peer Computing (P2P'05)*. IEEE. 2005, pp. 87–94 (cit. on p. 14).

[23] O. Ozkasap, Z. Xiao, and K. P. Birman. *Scalability of two reliable multicast protocols*. Tech. rep. Cornell University, 1999 (cit. on p. 17).

[24] A. Rowstron and P. Druschel. "Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems". In: *Middleware 2001: IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg, Germany, November 12–16, 2001 Proceedings 2*. Springer. 2001, pp. 329–350 (cit. on pp. 11–13, 17).

[25] A. Stavrou, D. Rubenstein, and S. Sahu. "A lightweight, robust p2p system to handle flash crowds". In: *10th IEEE International Conference on Network Protocols, 2002. Proceedings.* IEEE. 2002, pp. 226–235 (cit. on p. 6).

[26] I. Stoica et al. "Chord: A scalable peer-to-peer lookup service for internet applications". In: *ACM SIGCOMM computer communication review* 31.4 (2001), pp. 149–160 (cit. on pp. 9, 10).

[27] C. Tang, R. N. Chang, and C. Ward. "GoCast: Gossip-enhanced overlay multicast for fast and dependable group communication". In: *2005 International Conference on Dependable Systems and Networks (DSN'05)*. IEEE. 2005, pp. 140–149 (cit. on p. 17).

[28] *The ISIS Project*. https://www.cs.cornell.edu/Info/Projects/ISIS/. Accessed: 2024-12-19 (cit. on p. 18).

[29] S. Voulgaris, D. Gavidia, and M. Van Steen. "Cyclon: Inexpensive membership management for unstructured p2p overlays". In: *Journal of Network and systems Management* 13 (2005), pp. 197–217 (cit. on pp. 4–6).

[30] S. Voulgaris and M. Van Steen. "An epidemic protocol for managing routing tables in very large peer-to-peer networks". In: *International Workshop on Distributed Systems: Operations and Management*. Springer. 2003, pp. 41–54 (cit. on p. 14).

[31] Wikipedia contributors. *Distributed hash table — Wikipedia, The Free Encyclopedia*. [Online; accessed 22-January-2025]. 2024. URL: https://en.wikipedia.org/w/index.php?title=Distributed_hash_table&oldid=1256718648 (cit. on p. 8).

[32] Wikipedia contributors. *Peer-to-peer — Wikipedia, The Free Encyclopedia*. [Online; accessed 18-December-2024]. 2024. URL: https://en.wikipedia.org/w/index.php?title=Peer-to-peer&oldid=1260658227 (cit. on pp. 3, 4).

[33] Wikipedia contributors. *Kademlia — Wikipedia, The Free Encyclopedia*. [Online; accessed 1-February-2025]. 2025. URL: https://en.wikipedia.org/w/index.php?title=Kademlia&oldid=1270627849 (cit. on p. 10).

[34] B. B. Yang and H. Garcia-Molina. "Designing a super-peer network". In: *Proceedings 19th international conference on data engineering (Cat. No. 03CH37405)*. IEEE. 2003, pp. 49–60 (cit. on p. 7).

[35] H. Zhang et al. *Distributed hash table: Theory, platforms and applications*. Springer, 2013 (cit. on p. 8).

[36] B. Y. Zhao et al. "Tapestry: A resilient global-scale overlay for service deployment". In: *IEEE Journal on selected areas in communications* 22.1 (2004), pp. 41–53 (cit. on pp. 13, 14).