www.ascens-ist.eu

ASCENS

Autonomic Service-Component Ensembles

TR-1202: Science Cloud: Modelling and Implementing the Peer-to-Peer DHT protocol 'Chord'

Grant agreement number: **257414**
Funding Scheme: **FET Proactive**
Project Type: **Integrated Project**
Latest version of Annex I: **7.6.2010**

Author(s): **Philipp Zormeier (LMU), Annabelle Klarl (LMU), Christian Kroiß (LMU), Philip Mayer (LMU)**

Periodic report:
Periodic covered:
Date of technical report: **July 25, 2012**
Revision: **V1**
Classification: **[CO]**

Project coordinator: **Martin Wirsing (LMU)**
Tel: **+49 89 2180 9154**
Fax: **+49 89 2180 9175**
E-mail: **wirsing@lmu.de**

Partners: **LMU, UNIPI, UDF, Fraunhofer, UJF-Verimag, UNIMORE, ULB, EPFL, VW, Zimory, UL, IMT, Mobsya, CUNI**

SEVENTH FRAMEWORK PROGRAMME

Abstract

The ASCENS project is an EU-funded integrated project within the 7th framework program with the goal of handling open-ended, highly parallel, massively distributed systems by breaking them down into self-aware, self-adaptive and self-expressive autonomic components which are grouped into so-called *ensembles*.

One of the case studies to evaluate the ASCENS results is the *Science Cloud* case study which describes a peer-to-peer voluntary computing cloud for scientific usage which consist of an arbitrary number of nodes running the science cloud software. The nodes interact to ensure that applications deployed in the cloud are kept running in case of disappearing nodes, changing load conditions, and different cloud layouts.

One possible option for implementing communication within a distributed peer-to-peer system is the Chord protocol which induces an overlay ring-like structure within the cloud to ensure both resilience and fast access to values in the node-based distributed hash table.

In this document, we describe both the modelling and the implementation of Chord as a test-case for the usability within the Science Cloud case study. Modelling of Chord is achieved using SCEL (Service Component Ensemble Language), the primary formal modelling language of ASCENS. The implementation of Chord is created using the Java programming language and is kept close to the SCEL model.

Contents

1 Introduction

The ASCENS *Science Cloud* case study [vRA+11] realizes a peer-to-peer platform-as-a-service by an arrangement of autonomic, self-aware service components. A service component, running on a physical or virtual machine, continually monitors its own status as well as the status of other components associated to the cloud. Components may also be grouped into service component ensembles based on certain properties of the individual components or the links between them. The main aim of the science cloud is ensuring that deployed applications are executed reliably and while keeping to their SLA, even if indivdual service components fail or disappear.

One option for realizing distributed communication between the individual components of the Science Cloud is the *Chord* protocol [SMLN+03]. Chord is a distributed lookup protocol providing simple ID assignment to both nodes and data and ensures efficient lookups. We have extended the basic protocol for our purposes with regard to predecessor handling, leaving nodes, and data handling.

In this document, we formalize the basic Chord protocol with SCEL [GLPT12], a language for modelling autonomic service components and their ensembles, and provide an implementation of Chord in Java as a proof of its practical use.

This document is structured as follows. We first give a short introduction to the Chord protocol in section 2, followed by a description of SCEL in section 3. Afterwards, we discuss our main findings, i.e. the modelling of Chord in SCEL (section 4) and the Java implementation (section 5). We conclude in section 6.

2 About Chord

The Chord protocol implements a distributed lookup protocol for locating nodes which store a certain piece of data. Chord thus provides a mapping of some data key to a node which is responsible for this data. The protocol therefore implements a distributed hash table.

The members of a Chord network are logically arranged in a circle which is described by m-bit addresses. A node is identified by an address n between 0 and $2^m - 1$. The address is generated by hashing its IP address.

Data is then stored at the node to which a key associated to the data item maps (key/value pair). If a data item with key m is queried along the network, the next node n in clockwise direction called *successor(m)* (possibly m) is responsible for locating the data.

To establish communication around the network, the independent nodes have to be aware about a couple of other nodes. In a so called *finger table*, each node n stores information about m other subsequent successors. *finger[i].node* is *successor*($n+2^{i-1}$) ($i \in \{1, ..., m\}$). *finger[1].node* therefore is the direct successor of n. Though for correct behavior only information about the next successor is needed, the finger table ensures efficient communication (logarithmic complexity). In addition, the main procedures assume knowledge of the node's predecessor.

If any node needs to access data with key m, it looks up the nearest element in its finger table. This node either is directly responsible for the key or passes it to its own nearest finger.

A joining node initializes its finger table by means of an existing arbitrary node in the network (which must be known). That node locates all m successors via its own fingers.

After the finger table is set, the new node updates all preceding nodes which might contain it in their finger tables. This procedure ensures valid lookup functionality.

Note that leaving nodes are not specified by the basic protocol. To react to node failures, concurrent events and other challenges, further techniques have to be applied. A Chord-based application might, for example, establish data redundancy and mechanisms that maintain lookup information.

For our model and implementation of Chord, we had to add some enhancements to the original Chord protocol. First, we will describe an inconsistency of the protocol observed during implementation (subsection 2.1). Afterwards we discuss new functionality: Leaving nodes in subsection 2.2 and data handling in subsection 2.3.

2.1 Excluding and including predecessors

While testing the implementation we noticed an inconsistency in the Chord protocol, which may lead to invalid finger tables. This situation may occur if a node joins or leaves the network whose id matches the start id of a finger of a preceding node. We will discuss the joining case here as our leaving behavior is yet to be defined.

The procedure *update_others* notifies nodes which might have to update their finger table with the new node with a call of *update_finger_table*.

n.update_others()
for (i = 1 **to** m)
p = *find_predecessor(*$n - 2^{i-1}$*)*;
p.*update_finger_table(n,i)*;

To determine the node that should check its i-th finger table entry, the predecessor of id $n - 2^i$ is calculated. This node then eventually updates the entry and passes the message to its direct predecessor.

n.update_finger_table(s,i)
if (s $\in$ [n,finger[i].node))
finger[i].node = s;
p = predecessor;
p.*update_finger_table(s,i)*;

Within *find_predecessor*, a predecessor of an id is determined excluding the id itself. If the predecessor of id n is required and there actually is a node with id n, the predecessor of that node will be returned.

n.find_predecessor(id)
n' = n;
while (id $\notin$ (n', n'.successor])
n' = n'.*closest_preceding_finger(id)*
return n';

If n'.successor is matching the given id, the while loop stops and n' instead of n'.successor is returned. If this is the case, however, the procedure *update_others* will skip the node whose finger i is already (perfectly) pointing to the new node.

To solve this issue, we provide a variation of *find_predecessor*. This variation simply checks if n'.successor is a perfect match.

n.find_predecessor_incl(id)
n' = n;
while (id $\notin$ (n', n'.successor])
n' = n'.*closest_preceding_finger(id)*
if (id == n'.successor)

```
        return n'.successor;
    return n';
```

This new procedure is called instead of *find_predecessor* in the join and leave procedures. In all other cases the original procedure is adequate.

The other node lookup and joining behavior was not modified but enhanced by data management, which is described below.

We use the *stabilize* and *fix_finger* procedure as defined in the Chord paper, the only difference being that we do not limit the join procedure to setting the successor as proposed there.

2.2 Leaving nodes

We define the following leaving behavior:

```
n.leave()
    finger[1].node.add_data(dataMap);
    finger[1].node.predecessor = predecessor;
    for (i = 1 to m)
        p = find_predecessor(n − 2^(i−1));
        p.clean_finger(n,i,finger[1].node);

n.clean_finger(s,i,c)
    if (finger[i].node == s)
        finger[i].node = c;
        p = predecessor;
        p.clean_finger(s,i,c);
```

A leaving node updates the predecessor of its successor and afterwards notifies preceding nodes which might have to update their finger table. This behavior is analog to the behavior in *update_others* in *join*, the only difference being a candidate node being passed along with the message. The candidate to replace the original finger node is its successor.

2.3 Data management

As mentioned above, we use Chord to implement a map of key ids to data. This can be done by simply adding local hash maps to the nodes. To transfer keys between the nodes during joins and leaves, these maps have to offer methods like put, get, remove, extract and merge.

If data at a given key is requested, the successor node of the key is responsible for the data. This leads to the following procedures:

```
n.lookup_data(key)
    if (key ∈ dataMap.domain())
        return dataMap.get(key)
    else
        n' = find_successor(key);
        return n'.lookup_data(key);

n.store_data(key,data)
    if (key ∈ dataMap.domain())
        dataMap.put(key,data)
    else
```

n' = *find_successor(key)*;
n'.*store_data(key,data)*;

These procedures check if the given id is within the local key space, which is similar to the one of the data map. In this case, the data is returned or stored. In the other case, the responsible node is calculated and the procedure is started there.

To maintain the right data assignment during ongoing joins and leaves, we require the following modifications to *join* and *leave*. In *join*, after the call of *update_others*, *finger[1].node.extract_data(predecessor,n)* is called and the resulting map is the new local data map:

n.extract_data(p,n')
return dataMap.*extract((p,n'])*;

The procedure *extract* is meant to return a map containing all key value pairs within the given interval and shortens the domain by the interval.

When a node leaves, it first has to transmit its data to its successor. For this reason, the whole dataMap is sent to the successor node:

n.transmit_data(otherMap)
dataMap.merge(otherMap);

The procedure *merge* extends the data map's domain and adds the key-value pairs of the given map. At the beginning of *leave*, *finger[1].node.transmit_data(dataMap)* has to be called.

All procedures in full length will be repeated at the corresponding place in section 4.

3 About SCEL

SCEL (Service Component Ensemble Language) [GLPT12] is a language for modelling autonomic service components and service component ensembles. SCEL is a formal language and built on top of solid semantic grounds. SCEL is also a parameterized language, i.e. the first step when using SCEL is settling on a concrete SCEL *dialect* by substantiating the variable parts of the language (cf. [dNFLP11]):

- *The language for policies* together with an interaction and authorization predicate.
- *The language for representing knowledge items and repositories*, i.e. handling data within the language.
- *The expression language* which is used for producing values and evaluating data.

In the following subsections, we discuss the SCEL dialect used (section 3.1), which data (or rather, knowledge) we store within the nodes (section 3.2), and a set of macros we use to ease the SCEL specification (section 3.3).

3.1 Used SCEL dialect

In order to model in SCEL we define a dialect, i.e. concrete instantiations for the three languages discussed above. We set the stage as follows:

- Our model of the Chord algorithm does not require *policies* since all logics and constraints are directly included within the modelled processes. Consequently, no policy language is required.

- The *knowledge* repositories of nodes consist of tuples. We use a key-based approach to retrieving tuples, i.e. the first element of each tuple contains a key token which is used like an identifier. The rest of the tuple is an arbitrary number of arguments, which vary based on the key or the purpose of the tuple. Tuples are added to the repository by a *put* statement, which does not overwrite tuples with the same key. A *qry* statement returns the first found tuple, and a *get* statement additionally removes the retrieved tuple of the repository. Querying and getting tuples is realized by pattern matching. The statements have to consist of a specific key and the right number of arguments. The other arguments may be certain values or define a pattern to match tuples. Alternatively, a process can bind data to local variables by inserting wild card-like expressions, for example $!x$. These expressions can be extended by patterns in brackets, for example $!x[x > 20]$. Only certain values and patterns narrow the space of results.

- *Expressions* are mainly used within tuples and can be of arbitrary data types. The tuple keys are strings, but other values might also be boolean, numeric or of other types. We use Java-like syntax within curled brackets to symbolize object manipulation. Algebraic expressions are written in plain mathematical syntax and are wrapped by curled brackets.

All other properties and syntax elements are standard SCEL as explained in [GLPT12].

3.2 Node properties

Every node knows its finger table, the address of its predecessor and the data it is responsible for. This information is stored in their knowledge repository as follows:

- dataMap - a *Map<int,data>* object containing all node data.
  methods:
  - *put(key,data)* - adds the specified key-value pair.
  - *get(key)* - returns the requested value.
  - *domain()* - returns the interval of all assigned keys.
  - *merge(Map<int,data>)* - merges the domains and adds all data of the given map.
  - *extract(domain)* - shortens the domain by the given domain and returns a map containing lost key-values.
  - *addAll(Map<int,data>* - adds all elements.

- fingers - the finger list.
  fields of the fingers:
  - start - the expected node position.
    The i-th finger has $start = n + 2^{i-1} \ mod \ 2^m$, n being the nodes identifier.
  - node - successor(start).

  methods:
  - *set(int,finger)* - sets given finger.
  - *get(int)* - returns an element.

- *pred* - the node's predecessor

3.3 SCEL macros

In order to abbreviate the specification of the SCEL model we define the following three macros.

- *Sequential composition*
  To compose two processes sequentially, we use the following syntax:
  $\mathrm{P} \triangleq \mathrm{P}_1 \cdot \mathrm{P}_2$
  Sequential composition can be realised by synchronization over the knowledge repository. Here is an example:
  $\mathrm{P}_1 \cdot \mathrm{P}_2 \triangleq \mathrm{P}_1^*[\mathrm{P}_2^*]$ with an empty policy and

$$
\begin{aligned}
\mathrm{P}_1^* \triangleq\ & \mathbf{put}(\text{“}\mathit{uniqueIdentifier1}\text{”}, \{\mathit{generateID}()\})@\mathit{self} \\
& .\mathbf{get}(\text{“}\mathit{uniqueIdentifier1}\text{”}, !\mathit{id})@\mathit{self} \\
& .\mathbf{put}(\text{“}\mathit{uniqueIdentifier2}\text{”}, \mathit{id})@\mathit{self} \\
& .P_1 \\
& .\mathbf{put}(\text{“}\mathit{ready}\text{”}, \mathit{id})@\mathit{self} \\
& .\mathrm{nil} \\
\mathrm{P}_2^* \triangleq\ & \mathbf{get}(\text{“}\mathit{uniqueIdentifier2}\text{”}, !\mathit{id})@\mathit{self} \\
& .\mathbf{get}(\text{“}\mathit{ready}\text{”}, \mathit{id})@\mathit{self} \\
& .\mathrm{P}_2
\end{aligned}
$$

  Process P_1 has to end with nil. By writing $P_1.\mathbf{put}(\text{“}\mathit{ready}\text{”}, \mathit{id})@\mathit{self}.\mathrm{nil}$, we mean to add the put action right before P_1's nil.

- *Loops*
  $(i : \mathit{start} \rightarrow \mathit{end})(P(i), Q)$ abbreviates

$$
\begin{aligned}
& ....\mathbf{put}(\text{“}i\text{”}, \mathit{start})@\mathit{self} \\
& \quad .\mathrm{For} \\
\mathrm{For} \triangleq\ & (\mathrm{Check} \cdot \mathrm{P(i)} \cdot \mathrm{Increment}) + \mathrm{Finally} \\
\mathrm{Check} \triangleq\ & \mathbf{get}(\text{“}i\text{”}, !i[|\mathit{start} - i| \leq |\mathit{start} - \mathit{end}|]) \\
& \quad .\mathrm{nil} \\
\mathrm{Increment} \triangleq\ & \mathbf{put}(\text{“}i\text{”}, \{i + (\mathit{end} - \mathit{start})/|\mathit{end} - \mathit{start}|\})@\mathit{self} \\
& \quad .\mathrm{For} \\
\mathrm{Finally} \triangleq\ & \mathbf{get}(\text{“}i\text{”}, !i[|\mathit{start} - i| > |\mathit{start} - \mathit{end}|])@\mathit{self} \\
& \quad .\mathrm{Q}
\end{aligned}
$$

  P(i) has to with process nil and Q is a process. This macro simplifies the expression of for loops. An equivalent Java example would be

$$
\begin{aligned}
& for(int\ i = start; i < end, i++)\{ \\
& \quad P(i) \\
& \} \\
& Q
\end{aligned}
$$

As usual, we can use standard comparison operators for checking the loop end condition, i.e. $<$, $<=$, $>$, and $>=$, and both increment $++$ and decrement $--$ the loop variable.

- *Parallel Processes*
  For readability we write $P_1|P_2|...|P_n$, meaning $P_1[P_2[...[P_n]...]]$ with empty policies.

4 Modelling Chord in SCEL

This section gives a model of Chord using the SCEL language as parameterized in the previous section. In the following, we assume that value m is known by every node in the network. Please note that summation is considered modulo 2^m and *self* represents both the processing node and its identifier.

4.1 Node lookup procedures

The Chord protocol contains the following lookup procedures:

```
n.find_successor(id)
    n' = find_predecessor_excl(id);
    return n'.successor;


n.find_predecessor_excl(id)
    n' = n;
    while (id ∉ (n', n'.successor])
        n' = n'.closest_preceding_finger(id)
    return n';


n.find_predecessor_incl(id)
    n' = n;
    while (id ∉ (n', n'.successor])
        n' = n'.closest_preceding_finger(id)
    if (id == n'.successor)
        return n'.successor;
    return n';


n.closest_preceding_finger(id)
    for (i = m downto 1)
        if (finger[i].node ∈ (n,id))
            return finger[i].node;
    return n;
```

We now discuss the SCEL model for each of these.

find_successor

Process findSucc waits for successor requests and calculates the response according to the Chord protocol. As this is the first SCEL process shown, we describe our modelling approach here. Within this model, procedures, which can be called locally or remotely, are processes that wait for calls.

Those calls take place via the local knowledge repositories, as well the result return mechanisms. So this findSucc waits for calls by the blocking **get**.. statement. A calling process has to *put* a tuple with the key "findSuccReq", its id for returns and the target id as arguments in the knowledge repository of the systeme running this findSucc process, to trigger calculation. The given parameters are then stored within the local variables *reqNode* and *targID*. In the following procedure *find_predecessor* is called by triggering process findPred with an analog pattern. The result is retrieved via the knowledge repository. The last but one line shows that returning values is indeed handled by putting them in knowledge repositories. The following process is findSucc itself so the procedure *find_successor* remains available.

$$\begin{aligned}\text{findSucc} \triangleq\ & \mathbf{get}(\text{“}\mathit{findSuccReq}\text{”}, !\mathit{reqNode}, !\mathit{targID})@\mathit{self}\\ & .\mathbf{put}(\text{“}\mathit{findPredExclReq}\text{”}, \mathit{self}, \mathit{targID})@\mathit{self}\\ & .\mathbf{get}(\text{“}\mathit{findPredExclResp}\text{”}, \mathit{targID}, !\mathit{targPredNode})@\mathit{self}\\ & .\mathbf{put}(\text{“}\mathit{getSuccReq}\text{”}, \mathit{self})@\mathit{targPredNode}\\ & .\mathbf{get}(\text{“}\mathit{getSuccResp}\text{”}, \mathit{targPredNode}, !\mathit{resultNode})@\mathit{self}\\ & .\mathbf{put}(\text{“}\mathit{findSuccResp}\text{”}, \mathit{targID}, \mathit{resultNode})@\mathit{reqNode}\\ & .\text{findSucc}\end{aligned}$$

get_successor

We add the process getSucc to access other nodes' successors. It realizes a getter which encapsulates finger table access.

$$\begin{aligned}\text{getSsucc} \triangleq\ & \mathbf{get}(\text{“}\mathit{getSuccReq}\text{”}, !\mathit{reqNode})@\mathit{self}\\ & .\mathbf{qry}(\text{“}\mathit{fingerTable}\text{”}, !\mathit{fingers})@\mathit{self}\\ & .\mathbf{put}(\text{“}\mathit{getSuccResp}\text{”}, \mathit{self}, \{\mathit{fingers}.\mathit{get}(1).\mathit{node}\})@\mathit{reqNode}\\ & .\text{getSucc}\end{aligned}$$

find_predecessor_excl

Procedure *find_predecessor_excl* contains a while statement, which can be expressed via nondeterministic choices with guards.

$$\begin{aligned}\text{findPredExcl} \triangleq\ & \mathbf{get}(\text{“}\mathit{findPredExclReq}\text{”}, !\mathit{reqNode}, !\mathit{targID})@\mathit{self}\\ & .\mathbf{put}(\text{“}\mathit{tmpPredNode}\text{”}, \mathit{self})@\mathit{self}\\ & .\text{findPredExclWhile}(\text{reqNode}, \text{targID})\end{aligned}$$

$$\begin{aligned}\text{findPredExclWhile}(\text{reqNode}, \text{targID}) \triangleq\ & \mathbf{get}(\text{“}\mathit{tmpPredNode}\text{”}, !\mathit{tmpPredNode})@\mathit{self}\\ & .\mathbf{put}(\text{“}\mathit{getSuccReq}\text{”}, \mathit{self})@\mathit{tmpPredNode}\\ & .(\text{findPredExclWhileBody}(\text{reqNode}, \text{targID}, \text{tmpPredNode})\\ & +\\ & \text{findPredExclResp}(\text{reqNode}, \text{targID}, \text{tmpPredNode}))\end{aligned}$$

The process, that wins the nondeterministic choice, is the one that starts with the matching *get* statement. The other process at the same time dies.

$$\begin{aligned}\text{findPredExclWhileBody(reqNode,}&\\ \text{targID, tmpPredNode)} \triangleq\ & \mathbf{get}(\text{“}\mathit{getSuccResp}\text{”}, \mathit{tmpPredNode}, !\mathit{tmpPredSuccNode}\\ & \quad [\mathit{targID} \notin (\mathit{tmpPredNode}, \mathit{tmpPredSuccNode}]])@\mathit{self}\\ & .\mathbf{put}(\text{“}\mathit{predFingerReq}\text{”}, \mathit{self}, \mathit{targID})@\mathit{tmpPredNode}\\ & .\mathbf{get}(\text{“}\mathit{predFingerResp}\text{”}, \mathit{targID}, !\mathit{tmpPredNode})@\mathit{self}\\ & .\mathbf{put}(\text{“}\mathit{tmpPredNode}\text{”}, \mathit{tmpPredNode})@\mathit{self}\\ & .\text{findPredExclWhile(reqNode, targID)}\end{aligned}$$

$$\begin{aligned}\text{findPredExclResp(reqNode,}&\\ \text{targID, tmpPredNode)} \triangleq\ & \mathbf{get}(\text{“}\mathit{getSuccResp}\text{”}, \mathit{tmpPredNode}, !\mathit{tmpPredSuccNode}\\ & \quad [\mathit{targID} \in (\mathit{tmpPredNode}, \mathit{tmpPredSuccNode}]])@\mathit{self}\\ & .\mathbf{put}(\text{“}\mathit{findPredExclResp}\text{”}, \mathit{targID}, \mathit{tmpPredNode})@\mathit{reqNode}\\ & .\text{findPredExcl}\end{aligned}$$

find_predecessor_incl

This procedure is defined analogue to *find_predecessor_excl*, but with our modification described in 2.1.

$$\begin{aligned}\text{findPredIncl} \triangleq\ & \mathbf{get}(\text{“}\mathit{findPredInclReq}\text{”}, !\mathit{reqNode}, !\mathit{targID})@\mathit{self}\\ & .\mathbf{put}(\text{“}\mathit{tmpPredNode}\text{”}, \mathit{self})@\mathit{self}\\ & .\text{findPredInclWhile(reqNode, targID)}\end{aligned}$$

$$\begin{aligned}\text{findPredInclWhile(reqNode, targID)} \triangleq\ & \mathbf{get}(\text{“}\mathit{tmpPredNode}\text{”}, !\mathit{tmpPredNode})@\mathit{self}\\ & .\mathbf{put}(\text{“}\mathit{getSuccReq}\text{”}, \mathit{self})@\mathit{tmpPredNode}\\ & .(\text{findPredInclWhileBody(reqNode, targID, tmpPredNode)}\\ & \quad +\\ & \quad \text{findPredInclResp(reqNode, targID, tmpPredNode))}\end{aligned}$$

$$\begin{aligned}\text{findPredInclWhileBody(reqNode,}&\\ \text{targID, tmpPredNode)} \triangleq\ & \mathbf{get}(\text{“}\mathit{getSuccResp}\text{”}, \mathit{tmpPredNode}, !\mathit{tmpPredSuccNode}\\ & \quad [\mathit{targID} \notin (\mathit{tmpPredNode}, \mathit{tmpPredSuccNode}]])@\mathit{self}\\ & .\mathbf{put}(\text{“}\mathit{predFingerReq}\text{”}, \mathit{self}, \mathit{targID})@\mathit{tmpPredNode}\\ & .\mathbf{get}(\text{“}\mathit{predFingerResp}\text{”}, \mathit{targID}, !\mathit{tmpPredNode})@\mathit{self}\\ & .\mathbf{put}(\text{“}\mathit{tmpPredNode}\text{”}, \mathit{tmpPredNode})@\mathit{self}\\ & .\text{findPredInclWhile(reqNode, targID)}\end{aligned}$$

$$\begin{aligned}&\text{findPredInclResp(reqNode,}\\&\text{targID, tmpPredNode)} \triangleq \mathbf{get}(\text{“}\mathit{getSuccResp}\text{”}, \mathit{tmpPredNode}, !\mathit{tmpPredSuccNode}\\&\qquad [\mathit{targID} \in (\mathit{tmpPredNode}, \mathit{tmpPredSuccNode}]])@\mathit{self}\\&\qquad .\mathbf{put}(\text{“}\mathit{tmpPredSuccNode}\text{”}, \mathit{tmpPredSuccNode})@\mathit{self}\\&\qquad .(\text{findPredInclResult(reqNode, targID, tmpPredNode)}\\&\qquad +\\&\qquad \text{findPredInclRespResultSucc(reqNode, targID, tmpPredNode))}\end{aligned}$$

$$\begin{aligned}&\text{findPredInclRespResult(reqNode,}\\&\text{targID, tmpPredNode)} \triangleq \mathbf{get}(\text{“}\mathit{tmpPredSuccNode}\text{”}, !\mathit{tmpPredSuccNode}\\&\qquad [\mathit{targID} != \mathit{tmpPredSuccNode}])@\mathit{self}\\&\qquad .\mathbf{put}(\text{“}\mathit{findPredInclResp}\text{”}, \mathit{targID}, \mathit{tmpPredNode})@\mathit{reqNode}\\&\qquad .\text{findPredIncl}\end{aligned}$$

$$\begin{aligned}&\text{findPredInclResultSucc(reqNode,}\\&\text{targID, tmpPredNode)} \triangleq \mathbf{get}(\text{“}\mathit{tmpPredSuccNode}\text{”}, !\mathit{tmpPredSuccNode}\\&\qquad [\mathit{targID} == \mathit{tmpPredSuccNode}])@\mathit{self}\\&\qquad .\mathbf{put}(\text{“}\mathit{findPredInclResp}\text{”}, \mathit{targID}, \mathit{tmpPredSuccNode})@\mathit{reqNode}\\&\qquad .\text{findPredIncl}\end{aligned}$$

closest_preceding_finger

predFinger implements *closest_preceding_finger*. This is the first process that contains the loop macro.

$$\begin{aligned}\text{predFinger} \triangleq\ &\mathbf{get}(\text{“}\mathit{predFingerReq}\text{”}, !\mathit{reqNode}, !\mathit{targID})@\mathit{self}\\&.\mathbf{put}(\text{“}\mathit{break}\text{”}, \mathit{false})@\mathit{self}\\&.(i : m \rightarrow 1)(\text{predFingerFor(i, reqNode, targID)},\\&\quad \text{predFingerReturn(reqNode, targID))}\end{aligned}$$

Each iteration begins with a check of the *break* variable. If *break* is set to true, the result is already calculated returned.

$$\begin{aligned}&\text{predFingerFor(i, reqNode,}\\&\text{targID)} \triangleq \text{predFingerForBody(i, reqNode, targID)}\\&\qquad +\\&\qquad .\text{predFingerForStop}\end{aligned}$$

If *break* is false, the return criterion is checked, which may lead to setting *break* to true. predFingerForReturn matches the return criterion, predFingerForContinue continues the loop iteration.

$$\begin{aligned}&\text{predFingerForBody(i,}\\&\text{reqNode, targID)} \triangleq \mathbf{qry}(\text{“}\mathit{break}\text{”}, \mathit{false})@\mathit{self}\\&\qquad .(\text{predFingerForReturn(i, reqNode, targID)}\\&\qquad +\\&\qquad \text{predFingerForContinue(i, reqNode, targID))}\end{aligned}$$

$$\begin{aligned}\text{predFingerForReturn(i, reqNode, targID)} \triangleq\ & \mathbf{qry}(\text{“}\textit{fingerTable}\text{”}, !\textit{fingers} \\ & \quad [\textit{fingers.get}(i).\textit{node} \in (\textit{self}, \textit{targID})])@\textit{self} \\ & .\mathbf{put}(\text{“}\textit{break}\text{”}, \textit{true})@\textit{self} \\ & .\mathbf{put}(\text{“}\textit{predFingerResp}\text{”}, \textit{targID}, \\ & \quad \{\textit{fingers.get}(i).\textit{node}\})@\textit{reqNode} \\ & .\text{nil}\end{aligned}$$

$$\begin{aligned}\text{predFingerForContinue(i, reqNode, targID)} \triangleq\ & \mathbf{qry}(\text{“}\textit{fingerTable}\text{”}, !\textit{fingers} \\ & \quad [\textit{fingers.get}(i).\textit{node} \notin (\textit{self}, \textit{targID})])@\textit{self} \\ & .\text{nil}\end{aligned}$$

$$\begin{aligned}\text{predFingerForStop} \triangleq\ & \mathbf{qry}(\text{“}\textit{break}\text{”}, \textit{true})@\textit{self} \\ & .\text{nil}\end{aligned}$$

$$\begin{aligned}\text{predFingerReturn(reqNode, targID)} \triangleq\ & (\text{predFingerNoReturn} \\ & + \\ & \text{predFingerReturnSelf(reqNode, targID)})\end{aligned}$$

After the loop it has to be considered, whether self has to be returned or not.

$$\begin{aligned}\text{predFingerNoReturn} \triangleq\ & \mathbf{get}(\text{“}\textit{break}\text{”}, \textit{true})@\textit{self} \\ & .\text{predFinger}\end{aligned}$$

$$\begin{aligned}\text{predFingerReturnSelf(reqNode, targID)} \triangleq\ & \mathbf{get}(\text{“}\textit{break}\text{”}, \textit{false})@\textit{self} \\ & .\mathbf{put}(\text{“}\textit{predFingerResp}\text{”}, \textit{targID}, \textit{self})@\textit{reqNode} \\ & .\text{predFinger}\end{aligned}$$

4.2 Node Joins

A joining node runs the following procedure:

```
n.join(n')
    if (n')
        init_finger_table(n');
        update_others();
        dataMap = finger[1].node.extract_data(predecessor,n);
    else
        for (i = 1 to m)
            finger[i].node = n
        predecessor = n;

n.init_finger_table(n')
    finger[1].node = n'.find_successor(finger[1].start)
    predecessor = finger[1].node.predecessor;
    finger[1].node.predecessor = n;
    for (i = 1 to 1)
        if (finger[i + 1].start ∈ [n,finger[i].node))
```

```
            finger[i + 1].node = finger[i].node;
        else
            finger[i + 1].node = n'.find_successor(finger[i + 1].start);

n.update_others()
    for (i = 1 to m)
        p = find_predecessor_incl(n − 2^(i−1));
        p.update_finger_table(n,i);

n.update_finger_table(s,i)
    if (s ∈ [n,finger[i].node))
        finger[i].node = s;
        p = predecessor;
        p.update_finger_table(s,i);
```

We add the *extract_data* procedure. This procedure transfers all data now belonging to the new node from the successor node. It is called after *update_others()*.

```
n.extract_data(p,n')
    return dataMap.extract((p,n']);
```

In SCEL, the join procedure is modelled by the process $\mathrm{join}(\mathrm{helper})$. Helper is an arbitrary node already participating in the network and is necessary to initialize the fingers list of the new node. join(helper) is split into the following tasks:

- initFingerTable - initialize finger list via helper.
- setPred - set predecessors of self and successor.
- updateOthers - update the finger lists of preceding nodes whose fingers could be affected.
- transferKeys - get dataMap parts of successor which are now stored at the new node.

Afterwards, the main node process is started.

join

Join is modelled as follows.

$$\mathrm{join}(\mathrm{helper}) \triangleq \mathrm{initFingerTable}(\mathrm{helper}) \cdot \mathrm{setPred} \cdot \mathrm{updateOthers} \cdot \mathrm{transferKeys}$$

init_finger_table

Here the whole finger table including the first entry is set. The initialization of the predecessor happens afterwards. This order does not affect the correctness. In $\mathrm{initFingerTable}$ there is a loop setting the node values of all fingers by calling $\mathrm{findSucc}$ at the helper node.

$$\begin{aligned}\mathrm{initFingerTable}(\mathrm{helper}) \triangleq\ & \mathbf{put}(\text{“}\mathit{fingerTable}\text{”}, \{\mathit{newList}(m)\})@\mathit{self} \\ & .(i : 1 \to m)(\mathrm{setFinger}(\mathrm{i}, \mathrm{fingers}, \mathrm{helper}), \mathrm{nil})\end{aligned}$$

$$\begin{aligned}\text{setFinger(i, helper)} \triangleq\ & \mathbf{get}(\text{“}\mathit{fingerTable}\text{”}, !\mathit{fingers})@\mathit{self}\\ & .\mathbf{put}(\text{“}\mathit{findSuccReq}\text{”}, \mathit{self}, \{\mathit{self} + 2^{i-1}\})@\mathit{helper}\\ & .\mathbf{get}(\text{“}\mathit{findSuccResp}\text{”}, \{\mathit{self} + 2^{i-1}\}, \mathit{fingerNode})@\mathit{self}\\ & .\mathbf{put}(\text{“}\mathit{fingerTable}\text{”}, \{\mathit{fingers}.\mathit{set}(i, \mathit{fingerNode})\})@\mathit{self}\\ & .\text{nil}\end{aligned}$$

setPred sets the attributes *pred* and *succ* and registers the node at the successor as new predecessor.

$$\begin{aligned}\text{setPred} \triangleq\ & \mathbf{qry}(\text{“}\mathit{fingerTable}\text{”}, !\mathit{fingers})@\mathit{self}\\ & .\mathbf{put}(\text{“}\mathit{succ}\text{”}, \{\mathit{fingers}.\mathit{get}(1).\mathit{node}\})@\mathit{self}\\ & .\mathbf{get}(\text{“}\mathit{succ}\text{”}, !\mathit{succ})@\mathit{self}\\ & .\mathbf{qry}(\text{“}\mathit{pred}\text{”}, !\mathit{pred})@\mathit{succ}\\ & .\mathbf{put}(\text{“}\mathit{pred}\text{”}, \mathit{pred})@\mathit{self}\\ & .\mathbf{put}(\text{“}\mathit{pred}\text{”}, \mathit{self})@\mathit{succ}\\ & .\text{nil}\end{aligned}$$

update_others

updateOthers notifies all preceding nodes, that might have to update their finger table entries. To determine the procedure callees, it uses the process fingerPredlIncl.

$$\text{updateOthers} \triangleq (i : 1 \to m)(\text{updateOther(i)}, \text{nil})$$

$$\begin{aligned}\text{updateOther(i)} \triangleq\ & \mathbf{put}(\text{“}\mathit{findPredInclReq}\text{”}, \mathit{self}, \{\mathit{self} - 2^{i-1}\})@\mathit{self}\\ & .\mathbf{get}(\text{“}\mathit{findPredInclResp}\text{”}, \{\mathit{self} - 2^{i-1}\}, !\mathit{targNode})@\mathit{self}\\ & .\mathbf{put}(\text{“}\mathit{updateFinger}\text{”}, \mathit{self}, i)@\mathit{targNode}\\ & .\text{nil}\end{aligned}$$

transfer_keys

Process transferKeys retrieves all data from the successor, that is now managed at this node.

$$\begin{aligned}\text{transferKeys} \triangleq\ & \mathbf{put}(\text{“}\mathit{dataMap}\text{”}, \{\mathit{newMap}((\mathit{pred}, \mathit{self}])\})@\mathit{self}\\ & .\mathbf{qry}(\text{“}\mathit{dataMap}\text{”}, !\mathit{map})@\mathit{self}\\ & .\mathbf{qry}(\text{“}\mathit{fingerTable}\text{”}, !\mathit{fingers})@\mathit{self}\\ & .\mathbf{put}(\text{“}\mathit{succ}\text{”}, \{\mathit{fingers}.\mathit{get}(1).\mathit{node}\})@\mathit{self}\\ & .\mathbf{get}(\text{“}\mathit{succ}\text{”}, !\mathit{succ})@\mathit{self}\\ & .\mathbf{put}(\text{“}\mathit{getDataReq}\text{”}, \mathit{self}, \{\mathit{map}.\mathit{domain}()\})@\mathit{succ}\\ & .\mathbf{get}(\text{“}\mathit{getDataResp}\text{”}, \{\mathit{fingers}.\mathit{get}(1).\mathit{node}\}, !\mathit{data})@\mathit{self}\\ & .\mathbf{put}(\text{“}\mathit{dataMap}\text{”}, \{\mathit{map}.\mathit{addAll}(\mathit{data})\})@\mathit{self}\\ & .\text{nodeProcess}\end{aligned}$$

auxiliary processes

To ensure the correct joining behavior, every node has to run the processes updateFinger and getDataHelper. updateFinger is executed at nodes which might need to update their finger list.

$$\begin{aligned}\text{updateFinger} \triangleq\ & \mathbf{get}(\text{“}\mathit{updateFinger}\text{”}, !\mathit{newID}, !i)@\mathit{self} \\ & .(\text{updateFingerBody(newID, i)} \\ & + \\ & \text{updateFingerEnd(newID, i))}\end{aligned}$$

$$\begin{aligned}\text{updateFingerBody(newID, i)} \triangleq\ & \mathbf{get}(\text{“}\mathit{fingerTable}\text{”}, !\mathit{fingers}[\mathit{newID} \in \\ & \quad [\mathit{self}, \{\mathit{fingers}.\mathit{get}(i).\mathit{node}\})])@\mathit{self} \\ & .\mathbf{put}(\text{“}\mathit{fingerTable}\text{”}, \\ & \quad \{\mathit{fingers}.\mathit{set}(i, \mathit{newID})\})@\mathit{self} \\ & .\mathbf{qry}(\text{“}\mathit{pred}\text{”}, !\mathit{pred})@\mathit{self} \\ & .\mathbf{put}(\text{“}\mathit{updateFinger}\text{”}, \mathit{newID}, i)@\mathit{pred} \\ & .\text{updateFinger}\end{aligned}$$

$$\begin{aligned}\text{updateFingerEnd(newID, i)} \triangleq\ & \mathbf{qry}(\text{“}\mathit{fingerTable}\text{”}, !\mathit{fingers}[\mathit{newID} \notin \\ & \quad [\mathit{self}, \{\mathit{fingers}.\mathit{get}(i).\mathit{node}\})])@\mathit{self} \\ & .\text{updateFinger}\end{aligned}$$

getDataHelper provides functionality for extracting data out of dataMap to distribute it to new nodes.

$$\begin{aligned}\text{getDataHelper} \triangleq\ & \mathbf{get}(\text{“}\mathit{getDataReq}\text{”}, !\mathit{reqNode}, \\ & \quad !\mathit{mapDomain})@\mathit{self} \\ & .\mathbf{qry}(\text{“}\mathit{dataMap}\text{”}, !\mathit{map})@\mathit{self} \\ & .\mathbf{put}(\text{“}\mathit{getDataResp}\text{”}, \mathit{self}, \\ & \quad \{\mathit{map}.\mathit{extract}(\mathit{mapDomain})\})@\mathit{reqNode} \\ & .\text{getDataHelper}\end{aligned}$$

4.3 Leaving nodes

A leaving node executes process leave to update the predecessor node of the successor and transmit all data. The pseudocode we defined, looks like this.

```
n.leave()
    finger[1].node.transmit_data(dataMap)
    finger[1].node.predecessor = predecessor;
    for (i = 1 to m)
        p = find_predecessor(n − 2^(i−1));
        p.clean_finger(n,i,finger[1].node);

n.clean_finger(s,i,c)
    if (finger[i].node == s)
```

finger[i].node = c;
p = predecessor;
p.*clean_finger(s,i,c)*;

n.transmit_data(otherMap)
dataMap.*merge(otherMap)*;

In the following, we describe the SCEL processes.

leave

Procedure *put_data* is realized by simply putting the data to the succeeding node. The receiving process is modelled below. Then the preceding nodes are notified about the leave.

$$\begin{aligned}\text{leave} \triangleq\ & \mathbf{qry}(\text{“}\mathit{fingerTable}\text{”}, !\mathit{fingers})@\mathit{self}\\ & .\mathbf{qry}(\text{“}\mathit{dataMap}\text{”}, !\mathit{map})@\mathit{self}\\ & .\mathbf{put}(\text{“}\mathit{succ}\text{”}, \{\mathit{fingers}.\mathit{get}(1).\mathit{node}\})@\mathit{self}\\ & .\mathbf{get}(\text{“}\mathit{succ}\text{”}, !\mathit{succ})@\mathit{self}\\ & .\mathbf{put}(\text{“}\mathit{predData}\text{”}, \mathit{map})@\mathit{succ}\\ & .\mathbf{qry}(\text{“}\mathit{pred}\text{”}, !\mathit{pred})@\mathit{self}\\ & .\mathbf{put}(\text{“}\mathit{pred}\text{”}, \mathit{pred})@\mathit{succ}\\ & .(i : 1 \to m)(\text{cleanUpNotify(i, succ)}, \text{nil})\end{aligned}$$

cleanUpNotify works similar to updateOthers in the joining case.

$$\begin{aligned}\text{cleanUpNotify(i, succ)} \triangleq\ & \mathbf{put}(\text{“}\mathit{findPredInclReq}\text{”}, \mathit{self}, \{\mathit{self} - 2^{i-1}\})@\mathit{self}\\ & .\mathbf{get}(\text{“}\mathit{findPredInclResp}\text{”}, \{\mathit{self} - 2^{i-1}\}, !\mathit{targNode})@\mathit{self}\\ & .\mathbf{put}(\text{“}\mathit{cleanUpFinger}\text{”}, \mathit{self}, \mathit{succ}, i)@\mathit{targNode}\\ & .\text{nil}\end{aligned}$$

Every node has to run the process leaveHelper. It simply waits for incoming data and adds it to the local data map.

$$\begin{aligned}\text{leaveHelper} \triangleq\ & \mathbf{get}(\text{“}\mathit{predData}\text{”}, !\mathit{predMap})@\mathit{self}\\ & .\mathbf{get}(\text{“}\mathit{dataMap}\text{”}, !\mathit{map})@\mathit{self}\\ & .\mathbf{put}(\text{“}\mathit{dataMap}\text{”}, \{\mathit{map}.\mathit{merge}(\mathit{predMap})\})@\mathit{self}\\ & .\text{leaveHelper}\end{aligned}$$

clean_up_finger

cleanUpFinger is the process receiving the cleanup request. It checks, if the old finger table entry is set to the leaving node. That is decided via a nondeterministic choice. If an update has to take place, the call is propageted backwards

$$\begin{aligned}\text{cleanUpFinger} \triangleq\ & \mathbf{get}(\text{“}\mathit{cleanUpFinger}\text{”}, !\mathit{leaver}, !\mathit{cand}, !i)@\mathit{self}\\ & .(\text{cleanUpFingerBody(leaver, cand, i)}\\ & +\\ & \text{cleanUpFingerEnd(leaver, cand, i)})\end{aligned}$$

$$\begin{aligned}\text{cleanUpFingerBody}(\text{leaver}, \text{cand}, \text{i}) \triangleq\ & \mathbf{get}(\text{“}\textit{fingerTable}\text{”}, !\textit{fingers}[\textit{leaver} == \\ & \{\textit{fingers}.\textit{get}(i).\textit{node}\}])@\textit{self} \\ & .\mathbf{put}(\text{“}\textit{fingerTable}\text{”}, \\ & \{\textit{fingers}.\textit{set}(i, \textit{cand})\})@\textit{self} \\ & .\mathbf{qry}(\text{“}\textit{pred}\text{”}, !\textit{pred})@\textit{self} \\ & .\mathbf{put}(\text{“}\textit{cleanUpFinger}\text{”}, \textit{leaver}, \textit{cand}, i)@\textit{pred} \\ & .\text{cleanUpFinger}\end{aligned}$$

$$\begin{aligned}\text{cleanUpFingerEnd}(\text{leaver}, \text{cand}, \text{i}) \triangleq\ & \mathbf{qry}(\text{“}\textit{fingerTable}\text{”}, !\textit{fingers}[\textit{leaver}! = \\ & \{\textit{fingers}.\textit{get}(i).\textit{node}\}])@\textit{self} \\ & .\text{cleanUpFinger}\end{aligned}$$

4.4 Data Lookup

Process lookupDataClient(key) is executed for querying the data value stored at *key*. The node responsible is identified and then asked for the data. This section is not part of Chord. We define the following procedure:

n.lookup_data(key)
if (key ∈ *dataMap.domain()*)
return *dataMap.get(key)*
else
n' = *find_successor(key)*;
return n'.*lookup_data(key)*;

lookup_data client

This is our lookupDataClient, which determines the responsible node by calling findSucc and then requests the data.

$$\begin{aligned}\text{lookupDataClient}(\text{key}) \triangleq\ & \mathbf{put}(\text{“}\textit{findSuccReq}\text{”}, \textit{self}, \textit{key})@\textit{self} \\ & .\mathbf{get}(\text{“}\textit{findSuccResp}\text{”}, \textit{key}, !\textit{targNode})@\textit{self} \\ & .\mathbf{put}(\text{“}\textit{lookupDataReq}\text{”}, \textit{self}, \textit{key})@\textit{targNode} \\ & .\mathbf{get}(\text{“}\textit{lookupDataResp}\text{”}, \textit{key}, !\textit{data})@\textit{self} \\ & .\text{nil}\end{aligned}$$

lookup_data server

In order to provide data to the network, every node has to run the process lookupDataServer. It behaves like a simple getter procedure.

$$\begin{aligned}\text{lookupDataServer} \triangleq\ & \mathbf{get}(\text{“}\mathit{lookupDataReq}\text{”}, !\mathit{reqNode}, !\mathit{key})@\mathit{self}\\ & .\mathbf{qry}(\text{“}\mathit{dataMap}\text{”}, !\mathit{map})@\mathit{self}\\ & .\mathbf{put}(\text{“}\mathit{lookupDataResp}\text{”}, \mathit{key}, \{\mathit{map}.\mathit{get}(\mathit{key})\})@\mathit{reqNode}\\ & .\text{lookupDataServer}\end{aligned}$$

4.5 Data Storage

Storing a new key-value pair is achieved through the process $\text{storeDataClient}(\text{key}, \text{data})$. After identifying the responsible node, the data is transmitted. This section is not part of Chord either. We define the following procedure:

n.store_data(key,data)
if (key ∈ *dataMap.domain()*)
dataMap.put(key,data)
else
n' = *find_successor(key)*;
n'.*store_data(key,data)*;

store_data client

The SCEL model is split up into a server and a client. The client process looks up the target and then sends the data.

$$\begin{aligned}\text{storeDataClient}(\text{key}, \text{data}) \triangleq\ & \mathbf{put}(\text{“}\mathit{findSuccReq}\text{”}, \mathit{self}, \mathit{key})@\mathit{self}\\ & .\mathbf{get}(\text{“}\mathit{findSuccResp}\text{”}, \mathit{key}, !\mathit{targNode})@\mathit{self}\\ & .\mathbf{put}(\text{“}\mathit{storeDataReq}\text{”}, \mathit{key}, \mathit{data})@\mathit{targNode}\\ & .\text{nil}\end{aligned}$$

lookup_data server

Every node has to run a storeDataServer process, waiting for new key-value pairs to arrive.

$$\begin{aligned}\text{storeDataServer} \triangleq\ & \mathbf{get}(\text{“}\mathit{storeDataReq}\text{”}, !\mathit{key}, !\mathit{data})@\mathit{self}\\ & .\mathbf{qry}(\text{“}\mathit{dataMap}\text{”}, !\mathit{map})@\mathit{self}\\ & .\mathbf{put}(\text{“}\mathit{dataMap}\text{”}, \{\mathit{map}.\mathit{put}(\mathit{key}, \mathit{data})\})@\mathit{self}\\ & .\text{storeDataServer}\end{aligned}$$

4.6 Stabilization

To react to node failures, the finger tables have to be checked periodically. The processes stabilize and fix_fingers update the successor and the other fingers.

n.stabilize()
x = finger[1].node.predecessor;
if ($x \in (n, successor)$)
finger[1].node = x;
finger[1].node.*notify(self)*;

n.notify(n')
if (predecessor **is nil or** $n' \in (predecessor, n)$)
predecessor = n';

n.fix_fingers()
i = random index > 1 into finger[];
finger[i].node = find_successor(finger[i].start);

These procedures are expressed in SCEL as follows:

stabilize

A stabilizing node checks if the predecessor of the successor is itself. If not, the new node in between is the new successor and is notified.

$$\begin{aligned}\text{stabilize} \triangleq\ & \mathbf{qry}(\text{“}\mathit{fingerTable}\text{”}, !\mathit{fingers})@\mathit{self}\\ & .\mathbf{put}(\text{“}\mathit{succ}\text{”}, \{\mathit{fingers.get}(1).\mathit{node}\})@\mathit{self}\\ & .\mathbf{get}(\text{“}\mathit{succ}\text{”}, !\mathit{succ})@\mathit{self}\\ & .(\text{newSuccNode(succ)} + \text{notifySucc(succ)})\end{aligned}$$

$$\begin{aligned}\text{newSuccNode(succ)} \triangleq\ & \mathbf{qry}(\text{“}\mathit{pred}\text{”}, !x[x \in (\mathit{self}, \mathit{succ})])@\mathit{succ}\\ & .\mathbf{get}(\text{“}\mathit{fingerTable}\text{”}, !\mathit{fingers})@\mathit{self}\\ & .\mathbf{put}(\text{“}\mathit{fingerTable}\text{”}, \{\mathit{fingers.set}(1, x)\})@\mathit{self}\\ & .\mathbf{put}(\text{“}\mathit{predNotify}\text{”}, \mathit{self})@x\\ & .\text{stabilize}\end{aligned}$$

$$\begin{aligned}\text{notifySucc(succ)} \triangleq\ & \mathbf{qry}(\text{“}\mathit{pred}\text{”}, !x[x \notin (\mathit{self}, \mathit{succ})])@\mathit{succ}\\ & .\mathbf{put}(\text{“}\mathit{predNotify}\text{”}, \mathit{self})@\mathit{succ}\\ & .\text{stabilize}\end{aligned}$$

If a node is notified about a potentially new predecessor is realized by predNotify.

$$\begin{aligned}\text{predNotify} \triangleq\ & \mathbf{get}(\text{“}\mathit{predNotify}\text{”}, !\mathit{newPred})@\mathit{self}\\ & .(\text{predChange(newPred)} + \text{predNoChange(newPred)})\end{aligned}$$

$$\begin{aligned}\text{predChange(newPred)} \triangleq\ & \mathbf{qry}(\text{“}\mathit{pred}\text{”}, !\mathit{pred}[\mathit{pred} == \mathit{nil} \mid\mid \mathit{newPred} \in (\mathit{pred}, \mathit{self})])@\mathit{self}\\ & .\mathbf{put}(\text{“}\mathit{pred}\text{”}, \mathit{newPred})@\mathit{self}\\ & .\text{predNotify}\end{aligned}$$

$$\text{predNoChange(newPred)} \triangleq \mathbf{qry}(\text{“}\mathit{pred}\text{”}, !\mathit{pred}[\mathit{pred}! = \mathit{nil}\ \&\&\ \mathit{newPred} \notin (\mathit{pred}, \mathit{self})])@\mathit{self}$$
$$.\text{predNotify}$$

fix_fingers

Process fixFinger checks if the node value of the finger table entry at given index is correct by calculating the successor of the start value again.

$$\begin{aligned}\text{fixFinger(index)} \triangleq\ & \mathbf{get}(\text{“}\mathit{fingerTable}\text{”}, !\mathit{fingers})@\mathit{self}\\ & .\mathbf{put}(\text{“}\mathit{findSuccReq}\text{”}, \{\mathit{self} + 2^(i-1)\}, \mathit{self})@\mathit{self}\\ & .\mathbf{get}(\text{“}\mathit{findSuccResp}\text{”}, \{\mathit{self} + 2^(i-1)\}, !\mathit{succ})\\ & .\mathbf{put}(\text{“}\mathit{fingerTable}\text{”}, \{\mathit{fingers}.\mathit{set}(\mathit{index}, \mathit{succ})\})@\mathit{self}\end{aligned}$$

4.7 Main Node Process

This process contains all auxiliary processes to ensure the Chord functionality.

$$\begin{aligned}\text{nodeProcess} \triangleq\ & \text{findSucc} \mid \text{findPredExcl} \mid \text{findPredIncl}\\ & \mid \text{getSucc} \mid \text{predFinger} \mid \text{updateFinger}\\ & \mid \text{getDataHelper} \mid \text{leaveHelper} \mid \text{lookupDataServer}\\ & \mid \text{storeDataServer} \mid \text{cleanUpFinger}\end{aligned}$$

Additionally, the following processes are called at some point by external means (timer-based or by user action).

- storeDataClient(key,data)
- lookupDataClient(key)
- leave
- join(helper)
- stabilize
- fixFinger(index)

5 Implementing Chord

When implementing the Chord protocol in Java, we focussed on keeping the program as close as possible to our SCEL model. In this section we explain the implementation starting with a short description of the chosen architecture. We finish with a short usage documentation and screenshots.

5.1 Architecture

Chord is implemented in Java using the standard Java API. No external libraries or frameworks are used to ensure that the code is easily understandable. The basic architecture is three-layered and shown in figure 1. The classes shown in the layers represent a view into the system, they are supported by others (see source code for a detailed view). The layers are discussed in the following subsections.

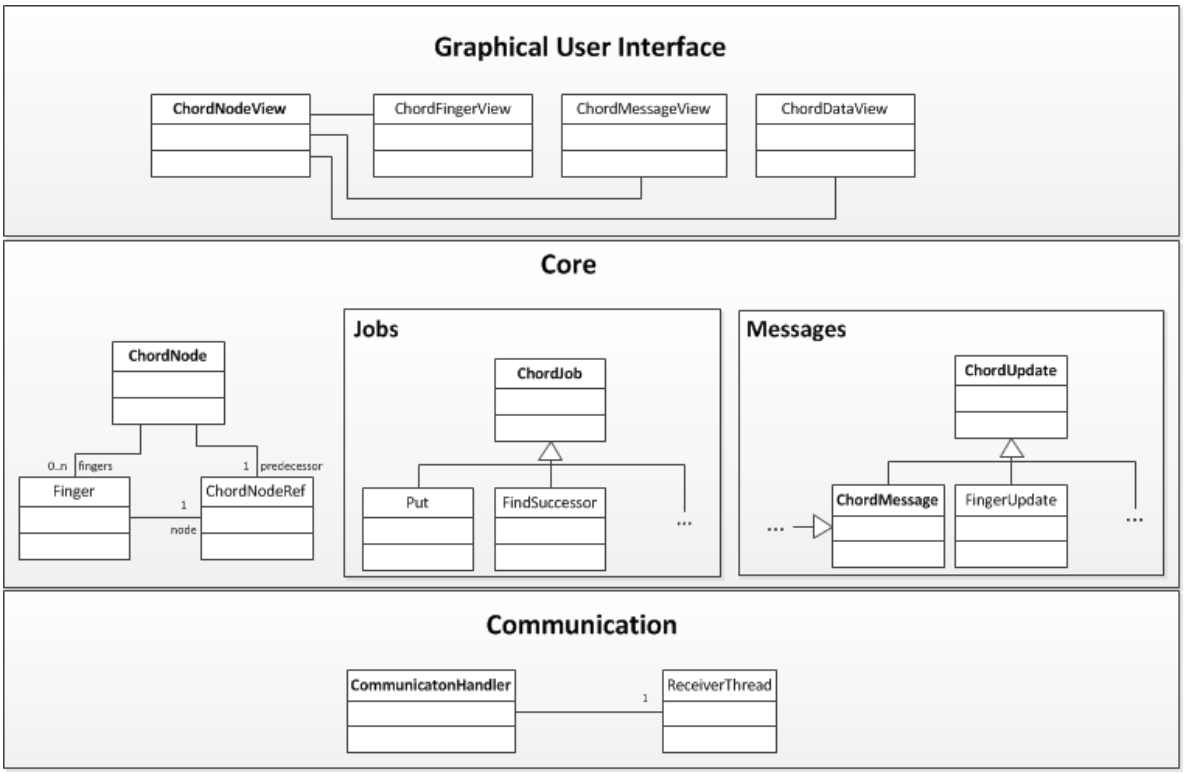


Figure 1: Layered Architecture

5.1.1 Core layer

The main part of the implementation is the *core*, which forms the Chord layer. It contains all the protocol logic. A node is represented by a *ChordNode* instance. A *ChordNode* object has links to its fingers and its predecessor. To the outside, it provides the procedures *put* and *get*, which allow adding a key value pair to the ring or removing it.

The procedures each node has to execute from time to time are *ChordJob* objects. Each job is a thread which is started individually on demand, and runs concurrent to all other jobs.

Communication at this layer is realized through serializable *ChordMessage* objects. Each message type is represented by its own class, which may contain additional information required for the receiver.

Remote procedure calls are also realized through messages. If a procedure has to be run on another node, a *ChordMessage* is used to trigger the required behavior. Procedures with result values can be accessed via queries.

If a job wants to start a query and awaits remotely calculated answers, it uses *ChordQuery* objects. Matching of query responses to waiting jobs happens at this layer. A node is identified by its Chord id at the core layer and by an ip address and a port number at the communication layer. *ChordNodeRef* objects which represent network identification contain these three values.

5.1.2 Communication layer

The layer underneath *core* is the communication layer. It provides functionality for sending UDP messages to a given receiver address. The main part in this layer is provided by *CommunicationHandler*s.

The class *CommunicationHandler* defines the method *sendMessage*, which serializes a given object and sends it to a specified address. Incoming messages are received by a dedicated *ReceiverThread*, which notifies about new packets via the observer pattern.

5.1.3 Graphical User Interface

The Chord implementation contains a graphical user interface for interacting with the implementation.

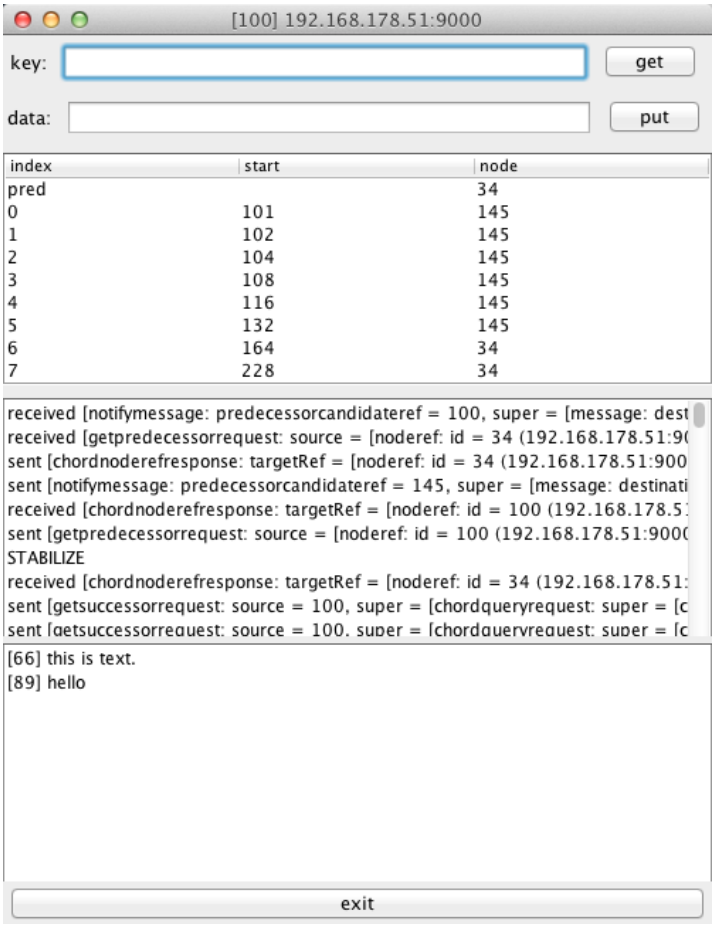


Figure 2: Screenshot of a Chord Node

Figure 2 shows the GUI of Chord nodes. The first section with textfields and buttons allows to get and put data. The string in the key textfield will be hashed to a key.

The table below shows current predecessor and finger nodes of the node. Each finger is displayed with its start value and the actual node which is addressed here.

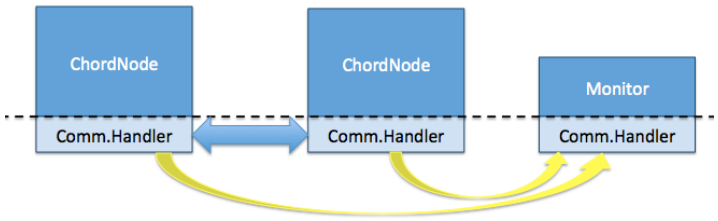
The two lists below show all received and sent message as well as some updates (example: 'STABILIZE') and all data that is stored at this node in the form [key] data.

The exit button on the bottom of the screen allows controlled leaves. Pressing it starts the Chord job *leave* and then terminates the program.

5.2 Monitoring the System

To be able to figure out what is going inside the Chord network from a global perspective, the Chord implementation contains an additional *monitoring* component. Although the main Chord protocol is strictly peer-to-peer, we add one gloabl component purely for monitoring which receives all updates sent within the network as well, and is able to display them graphically.

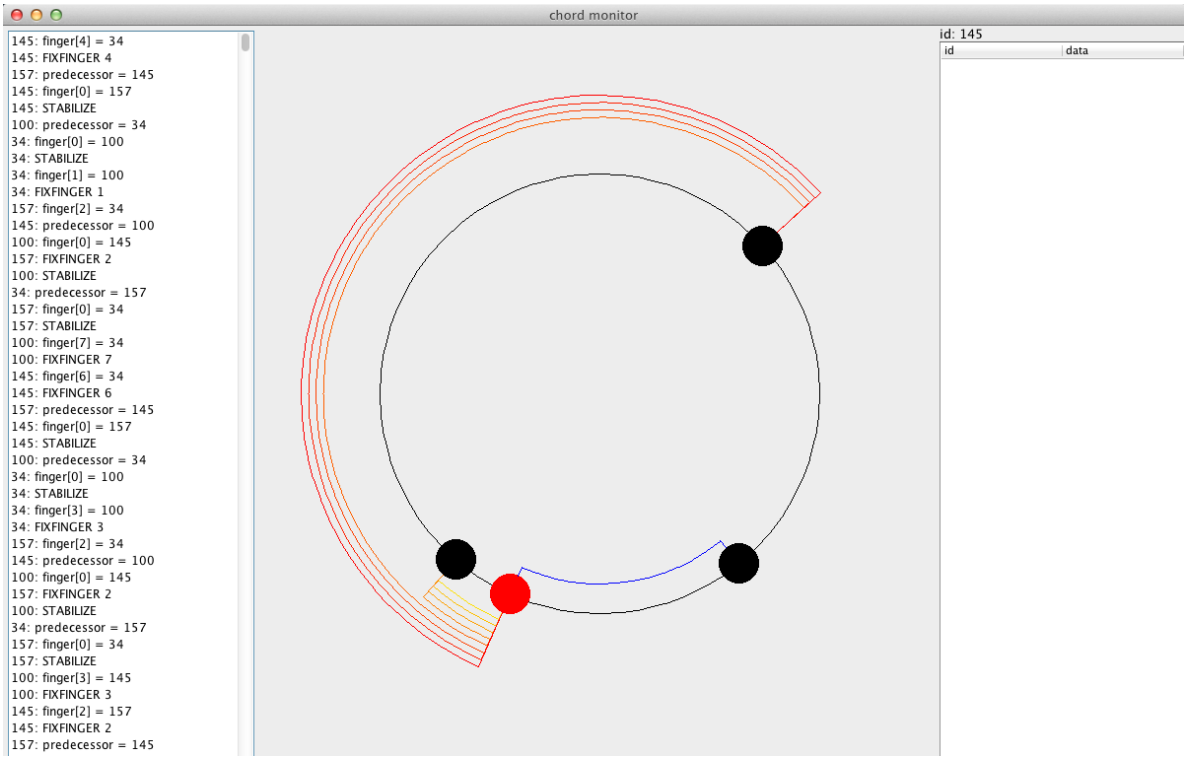
A Chord monitor runs independently on a machine in the network at a certain address and port. It receives *ChordUpdate*s, which are sent by the nodes at relevant events like joins, leaves, data changement, stabilizes and more.



A monitor is a passive component. It only listens for updates and displays them. If data information is incomplete, the monitor does not fetch data on its own. The flow of data in the Chord implementation is shown in figure 5.2.

An example of the monitor UI in action is shown in figure 5.2. The middle section shows an image of the logical ring. Here, four rings are participating. Now, node 145 is activated. The node's predecessor is shown in blue; its fingers are displayed in yellow to red around the ring. On the right, the id and stored data (currently none) are displayed.

On the left of the window, a list of incoming *ChordUpdate*s can be viewed. These updates actually provide all information the monitor has, as stated above.



6 Conclusion

This document has shown how Chord, a peer-to-peer network lookup protocol, can be modelled in the SCEL modelling language, and implemented in plain Java.

During this work, we have identified a shortcoming in the Chord protocol as discussed in section 2. We have furthermore discussed our SCEL dialect and added some macros which might be beneficial in the future for specifying other service components (section 3).

Our model of the Chord protocol in SCEL can be found in section 4, which also includes some additional Chord procedures we required for fully expressing a system based on Chord (for example, for leaving nodes). Our implementation has been discussed in section 5, where we have also described a monitor for simplifying the visualization of the Chord ring.

References

[dNFLP11] Rocco de Nicola, Gianluigi Ferrari, Michele Loreti, and Roberto Pugliese. D1.1: Language Primitives for Coordination, Resource Negotiation, and Task Description. ASCENS Deliverable, November 2011.

[GLPT12] Edmond Gjondrekaj, Michele Loreti, Rosario Pugliese, and Francesco Tiezzi. Modeling adaptation with a tuple-based coordination language. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, SAC '12, pages 1522–1527, New York, NY, USA, 2012. ACM.

[SMLN+03] Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Trans. Netw.*, 11(1):17–32, February 2003.

[vRA+11] Nikola Šerbedžija, Stephan Reiter, Maximilian Ahrens, José Velasco, Carlo Pinciroli, Nicklas Hoch, and Bernd Werther. D7.1: First Report on WP7: Requirement Specification and Scenario Description of the ASCENS Case Studies. ASCENS Deliverable, November 2011.