Decentralized Indexing of Presentities over \( n \)-Dimensional Context Information

Christian Lentfort
christian@lentfort.com
Abstract

Modern context-aware applications no longer justify their decisions based only on their own information but on the decisions and information of other applications in a similar context. Acquiring context information of other entities in a distributed system is a difficult task when using the current content-centric solutions such as DHTs. This project aims to build a distributed index that provides storage for the so-called Presentities solely based on the state of their context information. Furthermore, the stored Presentities must be efficiently accessible even if only some information of their current context is available. To fulfill these requirements the PAST DHT was extended to support range queries and modified to use points on a space-filling curve as index values. The simulation of the system has shown very good accuracy rates, on average 99%, for range queries by maintaining a logarithmic relationship to the amount of required messages sent in the DHT. Problems have emerged from the lack of load balancing implemented into the used DHT, but it is still the case that the proposed method of using space-filling curves to build a context-centric decentralized index is both sufficient and effective.

Keywords: context awareness, indexing, space-filling curves, Hilbert curve, Pastry, PAST
Acknowledgements

Thanks go to

- Jamie Walter, my supervisor at *Mid Sweden University*, for offering me the possibility to write my thesis under his supervision and his assistance and constructive advice.
Contents

Abstract i
Acknowledgements ii
Contents iii
List of Figures v
Terminology vi

1 Introduction 1
  1.1 Problem Motivation .................................. 1
  1.2 Overall Aim ........................................... 2
  1.3 Scope .................................................. 2
  1.4 Concrete and Verifiable Goals ......................... 2

2 Theory 3
  2.1 Context Aware Computing ............................... 3
    2.1.1 Sensors ........................................... 3
  2.2 Space-filling Curves ................................... 3
    2.2.1 Compact Hilbert Curves ............................. 4
  2.3 Overlay Networks ....................................... 4
    2.3.1 Peer-to-Peer Networks .............................. 4
    2.3.2 Distributed Hash Tables ............................ 5

3 Methodology 6
  3.1 FreePastry - Network Overlay .......................... 6
    3.1.1 PAST .............................................. 6
  3.2 uzaygezen - SFC Index Calculator ....................... 6
  3.3 Development Cycle ..................................... 7
  3.4 Development Environment ................................ 7
  3.5 Testing and verification ............................... 8
    3.5.1 Units testing ..................................... 8
    3.5.2 Functionality testing .............................. 8
4 Implementation
4.1 Implementation Details of the Project Aims . . . . . . . . . . 9
  4.1.1 Data Model . . . . . . . . . . . . . . . . . . . . . . . . 9
  4.1.2 Generation of SFC Indices . . . . . . . . . . . . . . . 10
  4.1.3 Finding similar context spaces based on their SFC in-
dices . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10
4.2 Changes Made to PAST . . . . . . . . . . . . . . . . . . . . . 11
  4.2.1 A Order-Preserving Hash-Function . . . . . . . . . . . 11
  4.2.2 Implementation of Range Queries . . . . . . . . . . . . 12
4.3 Test Application . . . . . . . . . . . . . . . . . . . . . . . . 12

5 Results
5.1 Problems . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14
  5.1.1 Poor Distribution of Objects in the DHT . . . . . . . . 14
5.2 Tests and Measurements . . . . . . . . . . . . . . . . . . . . . 15
  5.2.1 Messages Send in a Range Query . . . . . . . . . . . . 15

6 Conclusions
6.1 Future Work . . . . . . . . . . . . . . . . . . . . . . . . . . . 17

Bibliography

A Using the simulator

B Code: Message Propagation
# List of Figures

2.1 The first five orders of a Hilbert Curve in a 2D space ........ 4  
2.2 Classic and P2P network model .......................... 5  
3.1 The Development Cycle ..................................... 7  
4.1 Diagram of the data model ................................. 9  
4.2 Illustration of search ranges .............................. 10  
4.3 Index format .................................................. 11  
4.4 Split ID space and message propagation ................. 12  
5.1 Growing message count with growing interest area size .... 15  
A.1 Output of a simulator session ............................. 21
Decentralized Indexing of Presentities over $n$-Dimensional Context

Information

Christian Lentfort

August 30, 2012

Terminology

Terms

Presentity A presentity describes a real world object that can contain several context spaces.

Context space A group of several related context information.

Context information Any information that can be used to characterize the situation of an entity

Abbreviations

DHT Distributed Hash Table
SFC Space-filling curve
POC Proof of Concept
$\text{nD}$ $n$-dimensional
CHC Compact Hilbert Curve
P2P Peer-to-Peer
API Application Programming Interface
SHA Secure Hash Algorithm
ID Identifier
TDD Test Driven Development
URI Uniform Resource Identifier
Chapter 1

Introduction

1.1 Problem Motivation

With the success of smartphones and the “Internet of Things”[3] in our daily lives, context aware applications no longer rely only on their own information. They justify their decisions based on the information and decisions of other entities which share a similar context.

For this to work, it is important to provide access to the information in an easy and efficient way. Current solutions based simply on Distributed Hash Tables (DHT) are content centric but are, however, not context centric. This approach is not sufficient for context aware applications. To find presentities sharing a similar context, a list of entities must firstly be received that are capable of sharing the same context, after which their context information must be compared. This is a top-down approach which is not a suitable response to the problem of finding related context entities.

For efficient context aware applications a solution is required that enables the entities to be firstly located from their context proximity or their context information. One approach is to store presentities, which are closer with respects to their underlying context information, within the same space on either a distributed or centralized solutions.

Early approaches such as SenseWeb[13] and IMS[6] are dependent on the Domain Name System (DNS) as a means of locating other presentities. Issues associated with DNS availability due to Denial-of-Service attacks, configuration errors and its hierarchical design have led to research utilizing DHT type overlays such as [27] and [1] as possible replacements[21]. Building on the distributed approaches, MediaSense[14] and SCOPE[4] have been proposed as possible solutions.
1.2 Overall Aim

The overall aim in this case is to create a system that allows decentralized indexing of presentities based on the state of their context spaces. Furthermore, the system should also provide a method that allows querying the index for presentities that have spaces in a similar context.

1.3 Scope

The main scope of the project is centered on indexing the n-dimensional (nD) context information based on their position on a space-filling curve (SFC).

The work is merely a Proof of Concept (POC) and does not aim to be a full-feature system.

This thesis does not make any assumptions on the kind of information that is indexed nor will it deal with the implementation of the underlying technologies.

1.4 Concrete and Verifiable Goals

- Designing a model that helps to represent real-world objects and their context information. (Presentities, context spaces and context dimensions.)
- Deriving SFC indices from the information stored in a context space.
- Deriving SFC indices over a range of SFC indices. (Curve of sub-curve values.)
- Store and receive information identified by their SFC index in a overlay.
- Finding similar context spaces based on their SFC indices.
- Implement functionality for range queries and organization of indices in the overlay.
- Test and simulate the implementation to find possible weaknesses.
Chapter 2

Theory

2.1 Context Aware Computing

The term Context Aware Computing describes hardware or software that is capable of reacting to its context, where context is defined as “any information that can be used to characterize the situation of an entity”.[8]

These systems rely on different sources, such as sensors or human input to obtain the context information. The information can be used in different ways; modifying the behavior or the user interface of the application or finding the best solution to a given problem.

With the success of smartphones and “smart objects”[15] modern context aware applications can share their context information and include information of others to justify the decisions made.

An example for a context aware application is a smartphone that changes the background image of the screen according to the weather at the user’s location. The information can be obtained by either using the inbuilt sensors of the smartphone or by querying another application for the weather information at the current position.

2.1.1 Sensors

Sensors are devices that measure specific physical quantities (e.g. temperature, humidity, pressure) and convert it into a signal that can be used by an instrument.

2.2 Space-filling Curves

Space-filling curves are continuous, self-similar functions that map between a multi-dimensional hyper-cube and a single-dimensional interval and were originally discovered by Peano[22]. In the area of computer-science they
are used in fields such as image processing[16, 18], cryptology[17] or parallel computing [12]. An important property of such curves is good locality preserving: Points that have a small Euclidean distance in the hyper-cube are close to each other on the curve. A space-filling curve that has shown a strong locality preserving property is the Hilbert curve[19] (see Figure 2.1).

2.2.1 Compact Hilbert Curves
A special variation of the Hilbert Curve is the Compact Hilbert Curve (CHC) defined by Hamilton[10, 11]. In comparison to the traditional Hilbert Curve the CHC allows the mapped space to have dimensions of unequal length, while keeping all other advantages of the Hilbert Curve. This allows the CHC to represent indices in a more compact way.

2.3 Overlay Networks
Overlay networks are computer networks that are set on top of existing networks. The nodes in the network are connected to each other by logical links. These virtual links can span across several physical nodes in the underlying network.

A classical example for overlay networks is the Internet which in its early phase was built as an overlay on top of the telephone network[2].

2.3.1 Peer-to-Peer Networks
Peer-to-peer (P2P) networks are overlay networks in which each participant (peer) has the role of the client and the server. The key benefit of this architecture is the ability to share resources over the network without relying on a central server[26].

In comparison to the traditional model (Figure 2.2a), the reliability of a P2P network (Figure 2.2a) grows with the number of clients in the network, while the classical model scales poorly. In fact both the reliability and available resources grow with an increasing number of peers participating in a P2P network.
2.3.2 Distributed Hash Tables

Distributed Hash Tables (DHT) are a special form of overlay networks, mostly realized in the form of a P2P network. In a similar manner to that for centralized hash tables, DHTs provide lookup functionality on (key, value)-pairs.

Every node in the network is responsible for a part of the key-space and must respond to queries of other nodes requesting values falling outside their range. To achieve a good performance in lookups, each node in the network knows either the node responsible for the given key or a node that is closer to the given key.
Chapter 3

Methodology

3.1 FreePastry - Network Overlay

*FreePastry* is an modular, open source implementation of the *Pastry* [23] Peer-to-Peer (P2P) network overlay. It follows the specifications of the *Common API*[7] for structured P2P overlays, which allows the replacement of Pastry with other network overlays compatible with the *Common API*, if required.

FreePastry in version 2.1 will be used because it follows the *Common API*, is easy to simulate on a single computer and brings feature rich applications, such as PAST build on top of it. Furthermore is it well tested and is provided with excellent documentation and several tutorials.

3.1.1 PAST

The FreePastry implementation contains a powerful DHT implementation on top of it, called *PAST*[9, 24]. It implements a range of useful functions such as caching and persistent storage of values on its nodes.

In addition to PAST there are two other implementation compatible with the PAST interface.

- *PastGC*: a garbage collected version of PAST. All values stored in the DHT have a timeout, if a value is not refreshed before the timeout it will be dropped from the DHT.
- *Glacier*: A PAST implementation designed for unstable networks.

3.2 uzaygezen - SFC Index Calculator

The uzaygezen library offers mapping from a multi-dimensional space into a single-dimensional value via a CHC. It is a Java implementation of Chris
Hamiton’s libhilbert for CHC calculation with extended functionality such as calculation of search-ranges.

Uzaygezen in version 0.1 will be used because it is the only library fulfilling the requirements namely the generation of CHC indices and the calculation of search-ranges. Unfortunately, the available documentation for this library is insufficient and makes its usage highly prone to errors.

### 3.3 Development Cycle

In particular, strategies such as the Waterfall Model[25] and the V-Model are too inflexible to cope with the changing design requirements and ideas of a POC. Other more flexible approaches such as Test Driven Development [5] (TDD) are more suitable for a Proof of Concept but are difficult to use in situations that require full functionality tests.

Continuous progress on the work could change the requirements to the code, therefore the method illustrated in Figure 3.1 will be followed. The idea relies on the basic concepts of TDD, the regular unit testing. Contrary to that for TDD, unit tests will be written after the design and implementation of the desired functionality. This approach is chosen to cope with the changing requirements to the code.

When the project reaches a state that allows storing data in the network overlay basic functionality tests will be performed.

One major drawback of this concept is the quality of tests. They will be designed by the person who is also developing the actual code, therefore shortcomings of the code and tests might not be seen.

### 3.4 Development Environment

The Eclipse SDK in version 3.7.2 with the EGit plugin will be used to work on the project. Eclipse was chosen because it offers good support for devel-
Development when using the Java programming-language and, in particular, the integrated debugger and JUnit support allows for focus to be on the development while not having to deal with tasks such as running the debugger manually.

3.5 Testing and verification

3.5.1 Units testing

To verify that the code runs flawlessly unit tests are performed using JUnit. JUnit is a popular Java testing framework.

3.5.2 Functionality testing

To verify that the range query implementation is working, some basic functionality tests will be performed; including running range queries with varying search radii and queries on networks of varying size.

For this a Pastry ring will be created and filled with presentities, while inserting the presentities into the network overlay, a copy of the presentities will be hold locally. After inserting the presentities, a range query will be run simultaneously on the overlay and on the list of presentities stored locally. If both range-queries return the same results the test has been successful.
Chapter 4

Implementation

In this chapter the implementation details are explained. The implementa-
tional details of the projects aims are dealt with, followed by more in-depth
details of the changes made to the PAST DHT. Finally, the developed test-
application is described.

4.1 Implementation Details of the Project Aims

4.1.1 Data Model

To represent complex real world objects in the application a simple model
was designed. Figure 4.1 shows the model.

The model allows an object (referred to as presentity) to have several
context spaces and for which each context space can hold several context
dimensions. For global identification each presentity has an URI (Uniform
Resource Identifier).

Context spaces are used to group several related context dimensions.
They contain a unique name for each presentity, to identify a context space

![Diagram of the data model]

Figure 4.1: Diagram of the data model
Figure 4.2: Illustration of search ranges

4.1.2 Generation of SFC Indices

All calculations for the SFC indices are conducted through the library described in Section 3.2. To provide a means to easily exchange the library for another one providing similar features, the underlying API is hidden behind an interface. The interface provides all methods necessary to calculate the SFC indices and the SFC search ranges.

Based on a limitation in the used SFC library that prohibits the indexing of negative values, all dimensions are shifted before indexing so that the minimum value is the zero-point of the dimensions.

4.1.3 Finding similar context spaces based on their SFC indices

The used SFC library, described in Section 3.2 provides an easy method to calculate search ranges. A notable fact is that one search range can consist of several “sub ranges”. This is caused by the manner in which the SFC traverses through the index space. Figure 4.2 roughly shows how the area of interest (the gray circle) is covered by several “sub search ranges” (the black rectangles). A problem that arises from this is that some of the ranges are over-selective, which results in points being included that are not in the
defined search range. An example for over-selective ranges are the ranges 1 and 2 in Figure 4.2. The boundaries of the generated search ranges are combined to Pastry IDs as described in Section 4.2.1 and these IDs are then used to create Pastry ID ranges. The created ID ranges are used to query the overlay as described in Figure 4.2.2.

4.2 Changes Made to PAST

In order to fulfill the requirements of the project, two major changes had to be made to the PAST DHT.

4.2.1 A Order-Preserving Hash-Function

PAST, by default, uses the Secure Hash Algorithm 1[20] (SHA-1) to calculate hash-values of the values to be stored. Hash-values which are calculated using the SHA-1 algorithm do not reflect the order of the input, which means that Value1 > Value2 does not necessarily leads to hash(Value1) > hash(Value2).

While for a normal DHT the order of the values is unimportant, this is the case for an efficient implementation of range queries.

In order to implement range-queries, a hash-function was designed that preserves the order of the values. The function takes an SFC index value, which by itself is an order preserving functions, the object-type and the content-type (see Figure 4.1) of the object to index and then combines these values to a 160 bit long value. Figure 4.3 shows how the bits are set in the hash-value. The length of the hash values is 160 bit in order to maintain compatibility with hashes generated by the default hash algorithm.

By using the object-type and content-type as a prefix to the SFC index, the available ID space is split into several smaller spaces, as illustrated in Figure 4.4a.

Since PAST stores values on the nodes whose IDs have the longest common prefix with the hash value of the object, objects with the same content-type are stored on nodes that are close to each other on the Pastry’s ring.

This design has two significant advantages; objects in the same context spaces are stored close to each other and the is hash function order preserving. Both these features are necessary for the implementation of efficient range queries.
One shortcoming of this design is the probability of hash collisions. This problem was solved by storing arrays of objects instead of single objects under a certain ID.

### 4.2.2 Implementation of Range Queries

With the implementation of the order-preserving hash-functions described in Section 4.2.1 the prerequisites for range-queries are fulfilled.

To answer a range query, it is firstly forwarded to the node responsible for storing the pivot element of the range. (Query from node Origin to node 3 in Figure 4.4b.) The nodes removes the elements from the middle of the range it is responsible for, then it splits the range and forwards it to the next clockwise and counterclockwise neighbour nodes on the pastry ring. (Query from node 3 to nodes 2 and 4.) The receiving nodes will continue removing IDs from the range and forwarding the query to the next neighboring node. (E.g. Query from node 2 to node 1.) Propagation into a direction ends when the next node in the forward direction is not responsible for any of the IDs in the search range and if this is the case then a propagation end messages is sent directly to the origin of the query. (Messages from nodes 0 and 6 to node Origin.) The results of the query are reported directly to the origin of the query. (E.g. Answer message from node 5 to node Origin.)

The pseudo code in Appendix B shows the logic behind the code.

### 4.3 Test Application

For verification and functionality testing as described in Section 3.5.2 a simple application was built based on the FreePastry’s Direct Simulator.
The application provides an easy way to perform the tasks required for functionality testing. An introduction regarding how to use the simulator and an example session can be found in Appendix A.

For the simulation application the length of Pastry IDs was reduced to 32 bit. The space for the content-type was reduced to 4 bit, the object-type was dropped and the space for the SFC indices was reduced to 28 bit.
Chapter 5

Results

The data-model developed during this project enables a complex real world object to be represented and its context information to be given in a simple format. Based on this format it is easy to combine available the context information into a scalar index value.

Based on the indices derived from the data model it is possible to store and receive the data in a network overlay.

5.1 Problems

While testing the application two problems have occurred:

- The library used to generate the SFC indices is not capable of calculating search ranges for high dimensional spaces within a realistic time.

- Poor distribution of objects in the DHT.

5.1.1 Poor Distribution of Objects in the DHT

PAST does not apply any load balancing on its nodes. If there are too many similar bits in the generated IDs, all objects with a similar prefix are stored on one single node. This problem was solved in the simulation by drastically reducing the ID space to 32 bit.

This solution is not acceptable in a real world scenario, reducing the ID size leads to a reduction in the capacity of nodes in the Pastry ring. Possible solutions to this problems are to either implement a load balancing strategy into PAST or to replace PAST by another DHT.
Table 5.1: Search result accuracy

<table>
<thead>
<tr>
<th>No. of Nodes</th>
<th>No. of Presentities</th>
<th>No. of Queries</th>
<th>Avg. Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>4000</td>
<td>2000</td>
<td>250</td>
<td>98%</td>
</tr>
<tr>
<td>4000</td>
<td>2000</td>
<td>2000</td>
<td>99%</td>
</tr>
<tr>
<td>4000</td>
<td>1000</td>
<td>250</td>
<td>100%</td>
</tr>
</tbody>
</table>

Figure 5.1: Growing message count with growing interest area size

5.2 Tests and Measurements

Testing conducted using the Test Application described in Section 4.3 has shown very good accuracy rates for search results as shown in Table 5.1. The RGBA color model with 8 bit per channel was used to simulate a context space. Each color and the alpha-channel defined a single context dimension and the possible values ranged from 0 to 255.

5.2.1 Messages Send in a Range Query

To measure the number of messages sent in a single range query, several queries with different interest area sizes have been executed. The initial interest area was set to the middle of each dimension (with 8 bit per dimension, the value was 128) with an radius of 10 in each direction. (For each dimensions the values from 118 to 138 formed the area of interest.) After
each successful test, the search radius was increased by 2 for each dimension. The observed growth rate of the messages required by increasing the area of interest has shown a nearly logarithmic relation. (See Figure 5.1.)

The minimum number of messages required to fulfill a range query is 4. This number consists of 1 message that starts the query and 2 messages that inform the origin that the messages propagation has ceased. When performing a range query on the SFC index a single range query can consist of up to 20 sub queries, resulting in a minimum of 60 messages in total.
Chapter 6

Conclusions

All problems mentioned in Section 1.4 have been solved successfully. The results of the test application have clearly shown that storing multi-dimensional information based on SFC indices is an effective approach.

6.1 Future Work

With more efficient ways to generate search ranges, the shortcomings associated with this implementation can easily be dealt with.

The problem of bad load balancing described in Section 5.1.1 could be solved by either extending PAST by means of a load balancing strategy or by using another DHT that implements this feature. Since PAST relies on Pastry’s routing to decide which node is responsible for storing an object, extending PAST to support load balancing is a task that will require significant amount of extra work.
Bibliography


Decentralized Indexing of Presentities over n-Dimensional Context

Information

Christian Lentfort

Bibliography

August 30, 2012


[24] Antony Rowstron and Peter Druschel. “Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility.” In: *18th ACM Symposium on Operating Systems Principles (SOSP’01)*. Chateau Lake Louise, Banff, Canada, Oct. 2001, pp. 188–201.


Appendix A

Using the simulator

>java -cp proximity-jar-extracted.jar \ 
se.mediasense.proximity.Simulator 50 100 50 15

Finished creating 50 nodes

Finished putting 100 presentities into the overlay.
Waiting 30 seconds before doing lookups

Starting 50 attempts to find similar presentities out of 100 presentities in 50 nodes:

80,80,80,80,80,80,80,80,80,80,80,80,80,80,80,80,80,80,80,80,80,80,80,80,80,80,80,
80,80,80,80,80,80,80,80,80,80,80,80,80,80,80,80,80,80,80,80,80,80,80,80,80,80,80,
80,80,80,80,80,80,80,80,80,80,80,80,80,80,80,80,80,80,80,80,80,80,80,80,80,80,80

Finished 100 tests

All tests were successful

Figure A.1: Output of a simulator session

Figure A.1 shows the output of a simulation session. The sessions runs a network of 50 nodes and inserts 100 presentities, 50 lookup tests with a search radius of 15 are done. A dot indicates a finished action.

For each tests, independent of the success the number of messages required to find similar presentities in the network is printed to the user. The message count indicates that all data was stored on one network node and
that in total 20 sub-ranges were searched.
Appendix B

Code: Message Propagation

Algorithm 1 Forwarding algorithm

1: function doLookup(range, origin)
2:   if localElements > 0 then
3:     sendLocalElements(origin)
4:     low ← lowestLocalElement − 1
5:     high ← highestLocalElement + 1
6:   else
7:     low ← range.high
8:     high ← range.low
9: end if
10: if (ccw\(^1\) neighbor is responsible) & (direction is ccw) then
11:   subrange ← range(range.low, low)
12:   forwardQuery(ccwNeighbor, subrange)
13: else
14:   sendReachedEndMessage(origin)
15: end if
16: if (cw\(^2\) neighbor is responsible) & (direction is cw) then
17:   subrange ← range(high, range.high)
18:   forwardQuery(cwNeighbor, subrange)
19: else
20:   sendReachedEndMessage(origin)
21: end if
22: end function

\(^1\)counterclockwise  \(^2\)clockwise