Optimizations of XQuery in peer-to-peer distributed XML databases

Bogdan Butnaru

THÈSE

de

Doctorat de l'Université de Versailles Saint-Quentin-en-Yvelines

Présentée et soutenue publiquement par

Bogdan BUTNARU

Pour obtenir le grade de

Docteur en Informatique de l'Université de Versailles Saint-Quentin-en-Yvelines

Titre de la thèse :

Optimisation de requêtes XQuery dans des bases de données XML distribuées sur des réseaux pair-à-pair

Rapporteurs :
Dario Colazzo, MCF, HDR, *Laboratoire de Recherches en Informatique, UMR 8623, Université de Paris Sud et INRIA, projet LEO*
Talel Abdessalem, MCF, HDR, *Laboratoire Traitement et Communication de l'Information, UMR 5141, Telecom ParisTech*
Examinateurs :
Dan Vodislav, Pr., *Laboratoire Equipes Traitement de l'Information et Systèmes, UMR 8051, Université de Cergy-Pontoise*
Emmanuel Bruno, MCF, *Laboratoire des Sciences de l'Information et des Systèmes, UMR 6168, Université Sud-Toulon-Var*
Directeur :
Georges Gardarin, Pr. Émérite, *PRiSM, UMR 8144, Université de Versailles Saint-Quentin-en-Yvelines*
Encadrant :
Benjamin Nguyen, MCF, *PRiSM, UMR 8144, Université de Versailles Saint-Quentin-en-Yvelines et INRIA, projet SMIS*

Sommaire

Dans cette thèse nous proposons une architecture pour les bases de données XML distribuées basées sur les réseaux pair-à-pair. Notre approche est unique parce qu'elle est axée sur le traitement global du langage XQuery plutôt que l'étude d'un langage réduit spécifique aux index utilisés.

Le système XQ2P présenté dans cette thèse intègre cette architecture ; il se présente comme une collection complète de blocs de logiciels fondamentaux pour développer des applications similaires. L'aspect pair-à-pair est fourni par P2PTester, un « framework » fournissant des modules pour les fonctionnalités P2P de base et un système distribué pour des tests et simulations.

Une version de l'algorithme TwigStack adapté au P2P, utilisant un index structurel basé sur le numérotage des nœuds, y est intégré. Avec le concours d'un système de pré-traitement des requêtes il permet à XQ2P l'évaluation efficace des requêtes structurelles sur la base de données distribuée. Une version alternative du même algorithme est aussi utilisée pour l'évaluation efficace de la plupart des requêtes en langage XQuery.

L'une des nouveautés majeures de XQuery 3.0 est l'étude des séries temporelles. Nous avons défini un modèle pour traiter ce type de données, utilisant le modèle XML comme représentation des valeurs et des requêtes XQuery 3.0 pour les manipuler. Nous ajoutons à XQ2P un index adapté à ce modèle ; le partitionnement horizontal des longues séries de données chronologiques, des opérateurs optimisés et une technique d'évaluation parallèle des sous-expressions permettent l'exécution efficace d'opérations avec des volumes de données importants.

DISSERTATION

Submitted by

Bogdan BUTNARU

In support of candidature for the degree of

Doctor of Informatics of the University of Versailles Saint-Quentin-en-Yvelines

Title:

Optimizations of XQuery in peer-to-peer distributed XML databases

Thesis committee:
Dario Colazzo, MCF, HDR, *Laboratoire de Recherches en Informatique, UMR 8623, Université de Paris Sud* and *INRIA, projet LEO*
Talel Abdessalem, MCF, HDR, *Laboratoire Traitement et Communication de l'Information, UMR 5141, Telecom ParisTech*

Examining comittee:
Dan Vodislav, Pr., *Laboratoire Equipes Traitement de l'Information et Systèmes, UMR 8051, Université de Cergy-Pontoise*
Emmanuel Bruno, MCF, *Laboratoire des Sciences de l'Information et des Systèmes, UMR 6168, Université Sud-Toulon-Var*

Thesis supervisor:
Georges Gardarin, Pr.emer., *PRiSM, UMR 8144, Université de Versailles Saint-Quentin-en-Yvelines*

Thesis advisor:
Benjamin Nguyen, MCF, *PRiSM, UMR 8144, Université de Versailles Saint-Quentin-en-Yvelines* and *INRIA, projet SMIS*

Contents

Chapter 1

Introduction

Today's applications rely increasingly on the peer-to-peer model when exchanging information. This trend is sustained by many factors: Growth in the number of people with Internet access, and in their Internet consumption, makes centralized services ever harder to maintain.[1] Proliferation of Internet-enabled devices—desktop computers, laptops, TVs, game consoles, smart-phones, tablets, e-book readers and even digital cameras and thermostats—leads even to individual users having to rely on some self-organization of their increasingly complex personal networks. Such devices become ever more powerful; at the same time, ever more of the world's information is produced, rather than only consumed, by users. The edges of Internet become denser, more powerful, and increasingly sources rather than only sinks of information. All these trends push ever further away from the client-server model of communication. Even services that maintain use of the client-server model use peer-to-peer techniques in the ever-larger clusters that implement their "servers".

A parallel and related trend is the increasing dependence on XML as a format for storing and especially exchanging information. The growing importance of ease of interoperability, implied by the diversity of the devices at the network edge as well as that of the services offered to them, as well as its versatility, are key causes of this trend. The development of peer-to-peer techniques such as distributed hash tables added powerful and scalable primitives for organizing and locating information in peer to peer networks. However, most widely distributed applications are often limited to keyword search, and sometimes restricted, application-specific languages. There is a growing need

[1] Note that the growth in the number of services offered does *not* compensate for this; on the contrary, *all* new services need to be able to support the ever-growing number of users.

for widespread support of a versatile, powerful language such as XQuery in peer-to-peer distributed systems.

The topic of this work is to study the efficient evaluation of queries expressed in the XQuery language on XML databases distributed over peer-to-peer networks. We present XQ2P, an extensible XQuery evaluation engine oriented towards exploiting Distributed Hash Tables and XML indexing techniques adapted to DHTs for efficient processing. In contrast to most related research, where a new indexing algorithm is introduced and a query language is developed depending on the constructs the index can evaluate efficiently, XQ2P was designed in a top-down fashion: it attempts to support, as much as possible, the entire XQuery language with a simple though unoptimized implementation, and then extend this by adding indexing and efficient query algorithms to optimize select operations. We believe this approach is more favorable to widespread adoption of XQuery, with desirable effects such as better interoperability and increased competition.

In Chapter 2 we present a summary of each of the fundamental concepts used in the rest of this work: First, the XML language and data model, used by XQ2P to represent the data it manages. Following that, we describe XQuery, the language used for querying an XQ2P database. We then introduce the peer-to-peer approach to distributed applications, followed by a description of overlay networks and key-based-routing protocols and distributed hash tables based on the latter. XQ2P uses key-based-routing in a fashion analogous to DHTs to publish distributed indexes and answer queries. Finally, several common practical issues encountered while developing peer-to-peer applications and protocols are presented, pointing out the necessity and complexities of testing throughout each phase of development.

Chapter 3 introduces XQ2P, the main contribution of this work. XQ2P is a peer-to-peer–distributed XML/XQuery database system intended to support research through simplicity and extensibility. We adopt a generic architecture for distributed databases, in which the following steps are followed to support novel indexing and querying methods: first, documents are *published* by distributing among participating nodes indexed information; then, at query execution time, data relevant to the query is *located* using the index and *retrieved* from the nodes that own it; finally, the query is *evaluated* over the retrieved data to produce an answer.

We supply a complete suite of software modules to support this approach: A parsing module offers a complete parser for XQuery 1.0[2] and a type-safe parsed representation of queries. An implementation of the XPath/XQuery

[2]Partial support for window queries using the XQuery 3.0 syntax is also included as an optional extension.

type hierarchy, including nodes, values, sequences, and types[3] composes the data model. We supply a static analyzer which performs all the mandatory and some of the optional static analysis required by conforming XQuery implementation, easily extended thanks to use of the visitor pattern.

XQuery evaluation is supported by a set of operators derived from the semantics of each XQuery construct. A tree-shaped execution plan composed of such operators is generated after static analysis; the execution plan can be supplied with a dynamic context (containing, among other things, accessors for XML data) for immediate execution. Alternatively, it is possible to transform the execution plan, for example to apply reordering optimizations or to replace specific parts of it with index-based operators. The suite is completed by implementations of all standard XPath and XQuery functions and operators; adding user-defined functions, implemented either in Java or in XQuery, is also supported.

Peer-to-peer development is supplied by P2PTester, a framework for P2P applications which is not limited to XQ2P. P2PTester includes standardized interfaces for fundamental peer-to-peer services to support modularity. Master/slave tester programs allow running tests and simulations of P2P applications, including single-process, single-machine, and network-distributed tests controlled from a single workstation, with or without user interaction. Test scenarios can be written in Java for maximum expressivity and control over tested application nodes. Finally, a suite of ready-built modules for functionality such as network communication, event tracing and logging, as well as key-based routing and DHT protocols allows the development of higher-level peer-to-peer applications without needing to spend time on low-level details, and encourages code reuse.

In Chapter 4 we adopt a pre-post node numbering scheme to construct structural indexes of XML documents. Indexes are published over a Chord-like key-based routing layer. We adapt the TwigStack holistic join algorithm to KBR-distributed indexes, and we augment the XQ2P kernel described in Chapter 3 with TwigStack-based operators, which provide efficient execution of structural queries over local data, and also allow execution of queries with structural components over distributed data published on the overlay.

We then introduce in Chapter 5 a model for processing time-series data expressed in XML, and describe a value-based index system that supports the efficient execution of some useful queries on such data, again as extensions to XQ2P. The index partitions large time-series horizontally among peers; in most cases, operations can be performed directly on the index data, without needing to contact the peers holding the original documents. An important

[3]Notable exceptions, however, are support for XML Schema and user types.

distinction from the techniques of Chapter 4 is that some important but expensive operations on time-series can be executed in a distributed manner, by subdividing the task among peers.

Chapter 6 describes related work, mirroring the structure of Chapter 2. We conclude in Chapter 7.

In summary, we highlight the contributions of this thesis:

(i) A generic architecture for P2P-distributed XML databases, focused on supporting XQuery as a query language rather than more limited index-specific queries. (See Chapter 3.)

(ii) XQ2P, an open-source application using the above architecture, intended both as a proof of the architecture's suitability and as a complete collection of fundamental software building-blocks for developing similar applications. Supporting the peer-to-peer end is P2PTester, a framework providing pluggable modules for lower-level P2P functionality and a distributed testing and simulation system. (See Chapter 3.)

(iii) An version of the TwigStack algorithm adapted to P2P using a KBR-based node-labeling indexing scheme. A query preprocessing scheme which enables XQ2P to efficiently evaluate structural queries over the distributed database using this type of structural indexing. The optimized algorithm is integrated with XQ2P's local processing to allow support of most of the XQuery language for user queries. (See Chapter 4.)

(iv) An model for processing time-series data using the XML model with XQuery expressions. An implementation of a value-based index for horizontal partitioning of large time-series data over a DHT-like overlay. A further addition to XQ2P which enables it to efficiently evaluate many useful time-series operations distributedly. (See Chapter 5.)

Chapter 2

Peer-to-Peer XML Databases

XQ2P is an XML/XQuery–based database distributed over a peer-to-peer overlay network. This chapter presents in some detail each of the domains on the intersection of which XQ2P stands.

The first section below summarizes the XML language and describes in some detail the XML data model, used by XQ2P to represent the data it manages. Following that, *The XQuery Language* describes XQuery, the dominant language for processing XML data, used to query an XQ2P database.

Section 2.3, *P2P and KBR Systems*, introduces the peer-to-peer approach to distributed applications, overlay networks, key-based-routing protocols and distributed hash tables based on the latter. XQ2P uses key-based-routing in a fashion analogous to DHTs to publish distributed indexes and answer queries.

Finally, Section 2.4, *Testing and Simulating P2P Applications*, presents several common practical issues encountered while developing peer-to-peer applications and protocols related to the necessity of testing throughout each phase of development. To address these issues we developed XQ2P using P2PTester. Both XQ2P and P2PTester are described in the next chapter.

2.1 XML

As its name—e*X*tensible *M*arkup *L*anguage—suggests, XML originated as a standard for encoding mostly textual documents with *markup* added on sections of text, in a format that is both machine-readable and easily human-readable. Similar to HTML (and other, more powerful markup languages like SGML), XML defined a precise method for embedding structured or semi-structured "tags" in a text. In very simple terms, transforming simple text in XML entails surrounding (or "marking") fragments of the text with labeled tags (called very generally "markup"); these tags represent properties of the

```
<document type='article'>
  <metadata>
    <title>An example of XML</title>
    <author>A. Person</author>
    <author>J. Doe</author>
  </metadata>
  <section title='Introduction'>
    The first paragraph...
  </section>
  <!-- some other sections may follow -->
</document>
```

Figure 2.1: Example of a document-like XML tree.
Elements are delineated by tags surrounded by angle brackets; the word following the opening bracket is the name or *type* of tag; tags beginning with </ mark the end of an element. Name/value pairs following the element name in opening tags are attributes. The node delineated by <!-- and --> is a comment.

text they surround (in other words, they "tag" the parts of text they surround with meaning).

XML also became very popular as a format for encoding data in general, not only text-oriented documents. A variety of associated standards were developed: The XML data model defines its logical representation, basically of a tree of labeled nodes with data-bearing leaves; The XPath and XQuery standards define powerful, expressive languages for querying and manipulating this data model; XML Schema provides a standard way to define XML-based formats and data types; DOM (Document Object Model) and other APIs for manipulating XML are available in most popular programming languages. Many successful implementations of these standards, and the great ease of ensuring compatibility and interoperability of applications that use XML for data representation, as well as its flexibility, have led to XML being increasingly popular.

Given such powerful tools, it is already possible to have entire application stacks built from XML-based elements. An XML database stores the data, XQuery can manipulate the data, and the results can be presented to users with in XML format via web browsers (assisted by such standardized technologies as CSS, XSLT and XSL-FO for styling and AJAX for interactivity) and XML-based UI widgets and toolkits.

A central element of this powerful model for building applications is the XML database, the engine that stores and manipulates XML-structured data. This is the central focus of this work; to support the presentation of our contribution, the following sections will present the XML data model,[1] XQuery, XML databases, and XML indexing.

Modeling Data in XML

As discussed above, XML can be used as a model for representing data, independent of its textual format. Very briefly, the model is that of a tree of nodes; the tree's leaves contain data (e.g., numbers, text strings), and the structure of the tree defines relationships between them.

The complete XML standard however is slightly more complex. This section describes in more detail its elements. Note that [79] defines very precisely the data model used by the XPath and XQuery languages. We adhere to this model, but the presentation below will use the term "XML data model", or even "data model" for brevity. Not all features and details are mentioned here; the reader is invited to read the relevant standards for more details.

Nodes and Atomic Values The data model makes a basic distinction between *nodes* and *atomic values*. Atomic values, as the name implies, encode data elements. Nodes in general have a parent and a list of children; the structure of the document is defined via these parent-child relationships and those derived from them (like descendant-ancestor, or sibling).

Sequences Data is modeled as sequences of items. A sequence is simply an ordered list of nodes and/or atomic values. All operations are described as operating on, and returning, such sequences. Sequences have an item type and a cardinality.

A sequence may contain no items at all; if so, it is referred to as the "empty sequence". In general, an operation that has no result (but that is correct, in the sense that no errors are encountered), will return the empty sequence.[2]

A single item and the sequence containing that item (be it node or atomic value) are treated at equivalent. Common terminology makes use of this fact

[1] The *textual* representation of XML—i.e., the standard way of encoding XML data as text—is irrelevant to this work; consult [74] for details.

[2] The empty sequence behaves similarly to how a `NULL` value or a `NaN` floating-point value behaves in some programming languages. For example, many operations, e.g., comparison, will return the empty sequence when one of their operands is the empty sequence.

often: An operation will commonly be said to return an item when the logical result is a single value (for example, a comparison results in a boolean value); this is understood as equivalent to a sequence containing exactly one element.

The item type of a sequence denotes the kinds of items it may contain. The most general is sequence of "items", meaning any kind of node or value, but some sequences may contain only nodes (or even nodes of a certain type), or only atomic values of some kind (e.g., only numbers). The languages using this model (e.g., XQuery or XPath) have syntax for declaring the type and cardinality of sequences, but we need not describe it here.

The `document` node The root of an XML document tree is always a `document` node. As such, document nodes never have a parent. The document node may have any number of children nodes; however, it must have exactly one `element` node as a child.[3]

Document nodes have a `document-uri` property, denoting the location of the document (as the name implies, this is an atomic value of the `URI` type), but this may be empty (e.g., for documents constructed in memory).

The `element` nodes define the structure of the document. All `element` nodes have exactly one parent node. Their parent may be the `document` node or another `element` node. They may have any number of children, of any node type except `document`.[4]

All `element` nodes have a *name*. In general, the name is a string denoting the kind of element; its contents are restricted to specific characters (e.g., it must start with a letter, it contains no whitespaces nor most punctuation marks), similar to the syntactic restrictions on identifiers in most programming languages. A description of the mechanism is beyond the scope of this document, but a very short summary is that any element may declare a default namespace with an `xmlns` attribute, and associate other namespaces with a prefix with attributes named `xmlns:<PREFIX>`; default and prefixed namespaces are scoped to that element and its descendants (but descendants may re-declare both); unprefixed names encountered in a document are interpreted as belong-

[3] All element nodes in a document are either the unique `element` child of the `document` node, or a descendant thereof. As such, this `element` node is sometimes regarded as the "root" of the tree of elements; the reader is cautioned to be aware of the distinction between the "root of the element tree" and the "root of the XML document".

[4] However, mechanisms like DTD or XML Schema may restrict the kind and number of children of specific element types.

ing to the default namespace, names of the form `<PREFIX>:<NAME>` are interpreted as name `<NAME>` in the namespace associated with prefix `<PREFIX>`. For details the reader is referred to the relevant standards.[74][75]

The `attribute` nodes Attributes can be seen as "lightweight element". Similar to element nodes, attributes have a name. They differ in structured, however: First, the parent of an attribute node is always an `element` node. Second, the content of attribute nodes is always a string (though possibly zero-length). The data model formally treats this content as a property of the attribute, and attributes never have child nodes, i.e., they are always leaves of the XML tree. (However, it is sometimes useful to regard the content of an attribute as the unique text node child of the attribute.)

Due to their particular restrictions, attributes of an element are considered separate from its other children. Languages for processing XML have specific syntax for referring to such nodes separately from the other descendants.

The `text` nodes These are unlabeled nodes that contain text. They can be children of `document`, `element` or `attribute` nodes, and have no descendants. Text nodes are thus always leaves of the XML tree. The text contained in text nodes may be interpreted as a sequence of values (with the empty sequence and singleton sequences as special cases), for example as a number, or a list of dates.

Other node types In addition to the above there exist `comment` nodes and `processing instruction` nodes. These are used to annotate parts of the document (as their names imply, the former is destined for humans, and the latter for programs). Such nodes may have only document or element nodes as parent, and they don't have children.

XML fragment When processing XML-formatted data, there is often the need to manipulate nodes that are not part of a document. (For example, one may copy an element, with all its descendants, from a document, manipulate it for a while, then perhaps attach it to a newly-constructed document.) Such nodes, together with their descendants, are called "XML fragments". Such fragments respect the general rules described above in the sense that they have the same properties described above, except that they do not have a parent node.

Document order The data model regards the XML tree as *ordered*. The children for element and document nodes (the only node kinds that have children)

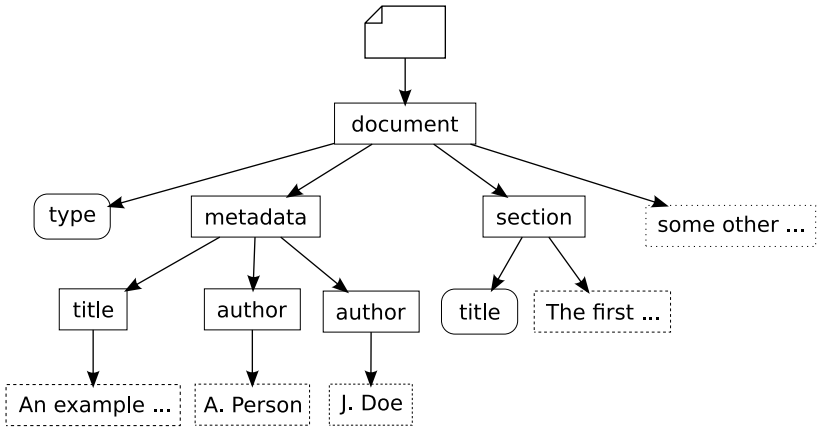


Figure 2.2: Tree model of the document in Figure 2.1.
Black rectangles represent elements, rounded boxes attributes, dashed boxes text nodes. The box with a folded corner is the document node, and the gray box to the right is a comment node. The arrows go from parent to children nodes.
Document order is top-down and left-to-right, beginning with the document node, the `document` element, `type` attribute, `metadata` element, `title` element, and so on.

and the attributes for elements form an ordered list rather than a generic collection. The XML data model extends this to total ordering for nodes: within a document, a node is defined to *precede* all its descendants, and *follow* its ancestors. Between documents nodes (or, when necessary, the roots of XML fragments) the data model requires that there be a strict ordering, but the exact order is implementation-defined.

2.2 The XQuery Language

XQuery is functional language designed for querying, processing and generating XML documents. XQuery is a superset of XPath, a smaller language

A verbose path expression:

```
/descendant-or-self::element(e)/child::b[./
   child::c]/attribute::a
```

An equivalent abbreviated path expression:

```
//e/b[c]/@a
```

Figure 2.3: Typical XPath syntax. In the first expression, the first step selects all elements with tag name `e`; the second step selects all their children with tag name `b`; the predicate eliminates those that do not have a `c` child; the final step selects the `a` attribute of remaining nodes.
The second expression is an equivalent showing typical abbreviations: the default axis is `child::`, `@` is used to denote the attribute axis, and `//` means `descendant-or-self::`.

intended primarily for retrieving parts of XML documents or collections.[5]

The main way of accessing XML content in XPath (and XQuery) is the use of path expressions. Path expressions consist of a series of steps, and operate on sequences of nodes (see the previous section). *Axis steps* applied to a sequence of nodes result in the sequence of all nodes of a certain kind reachable from any in the input sequence by traversing the XML tree on a certain *axis*. Examples of axes are `parent::`, `descendant::` or `attribute::` (the double-colon is XPath syntax separating axis names from the rest of the path expression). The kind of nodes retrieved by the step is chosen by a `node test`, which discriminates, for example, between text nodes or different types of element nodes. Each step may contain one or more predicates; predicates filter out of the resulting expression the items[6] for which they evaluate to false. Axis steps are separated by a forward-slash; syntactic sugar ensures common kinds of paths can be written quickly.

Besides path expressions, XPath also allows expressing familiar program-

[5]XPath is deliberately kept simpler, and is intended to be embedded in applications or larger languages; web browsers are examples of the former; XQuery is the canonical example of the latter. Note that through the several released versions some features have passed from one language to the other.

[6]Note that the last step of a path expression may result in values rather than nodes.

```
for $book:=//book, $auth:=//author
let $aid:=$auth/author-id
 where some $id in $aid satisfies $id = $auth/@id
 order by $auth/@name
return <result author='{$auth/@name}'
               title='{$book}/@title}' />
```

Figure 2.4: Example of FLWOR expression.
The `for` clause iterates over all pairs of books and authors; for each author `let` binds its ID to name `$aid`; the `where` clause joins the `$book` and `$auth` sequences to pair each book with all of its authors. The pairs are ordered by author name and a sequence of `result` elements is returned holding the book title and author name as attributes. The XML-like expression following `return` uses direct constructors.

ming constructs like literals, conditional expressions and arithmetic operations, sequence manipulation and function calls.

XQuery augments XPath with the FLWOR expression, an XML-specific analog of SQL's "select ... from ... where", and other features like syntax for defining functions, modules and output formatting.

FLWOR expressions are primarily used to execute joins between sequences. The abbreviation reflects the clauses that may form such an expression: `for`, `let`, `where`, `order by`, and `return`.

Aspects of XQuery

It is very important to observe that XQuery, despite its origins and name, is not *only* a query language, but a Turing-complete programming language. Although its syntax is designed for easily expressing queries on XML data, it also allows expressing arbitrary programs, even completely unrelated to XML. There are vast opportunities to apply programming language and compiler–related research to the subject.[7]

[7]Note that other database query languages, for example SQL, usually provide rather complex features for manipulating queried data, and compiler research applies to them too. However, in general they do not allow expressing arbitrary programs; language extensions or related languages like PL/SQL are needed in those cases.

However, the focus of our research is querying XML data. Accordingly, this work examines and contributes to querying techniques; matters related to XQuery's general programming features will only be discussed, briefly, when necessary to provide context.

A query expressed as XQuery (as opposed to any XQuery program in general) can be considered to express two conceptually distinct operations: First, to filter and retrieve one or several sets of data items from a collection of XML documents. Second, to manipulate the retrieved data in various ways—aggregating, transforming, or formatting it. In the same spirit as the previous paragraph, we focus on the first of these operations.[8] We will often refer to it simply as querying.

Querying XML data consists in retrieving some particular nodes from a collection of XML trees. XQuery often refers to this as "filtering": the semantic of many XQuery operations is usually defined in terms of receiving a sequence of nodes as input (starting with the entire collection), and removing ("filtering out") those that do not satisfy a particular condition. Based on the kind of conditions used to filter nodes, queries are typically divided further:

Structural queries filter nodes based on their properties related to the structure of the XML tree, seen as an ordered and labeled tree. These are the element types (expressed by their names, or labels), their positional relationships in the tree (parent/child, ancestor/descendant, sibling, attribute), and document order.

Value queries filter nodes based on their other properties, their "values". Values are represented by leaf nodes; because of the origins of XML as a (text) document markup language, these can be seen as text nodes. However, they can be interpreted as numbers, dates or other data types, not just as strings of characters, either explicitly (at the user's demand) or implicitly via automatic conversions or schema definitions.

2.3 P2P and KBR Systems

The traditional model of distributed applications is client-server: The server owns, controls and manages a certain set of resources—storage capacity, processing capability, data—which it makes available to multiple clients. This

[8] Note that it is not possible in general to separate the two based on syntactic notions. Most of XQuery's syntax can participate in both stages. Needless to say, this flexibility is advantageous to the user, but does complicate matters a bit for the researcher and developer.

model is in general applicable to any situation where resources need to be shared by several processes; the clients and servers may all coexist on a single machine, or may run on separate hosts, communicating via network connections (it is this latter case that concerns us). In such a system, all communication happens between the server and a client. Information that needs to go from a client to another, if any, will pass through the server.

Despite its usefulness and popularity, the client-server model cannot be used when its central assumption—that the control of the shared resource can be separated from its users and reside in a single entity, the server—does not apply. An alternative for such cases is the peer-to-peer, or P2P, model.

In a peer-to-peer system, each host is at the same time a supplier and a consumer of the shared resource. Logically all peers have the same capabilities[9] and responsibilities; each peer makes some of its resources available for the others, and peers coordinate with each other without any central entity to arbitrate. A few examples:

The problem that initially led to the popularity of the peer-to-peer model was file-sharing. In the client-server model, shared files reside on the server, and clients connect to the server and download the files they are interested in from it; this describes the structure of the extremely popular World Wide Web. However, this necessitates that the server have enough storage capacity to hold all the files, and enough bandwidth to deliver them to the clients. Systems like Napster[87] and Direct Connect[86] address this problem by having each peer share the files it holds; if a peer desires a file it doesn't own, it locates one of the peers that owns it and downloads it from there. In some systems—for example BitTorrent[57]—a peer can simultaneously download pieces of the same file from many peers at the same time, including peers that are *still* downloading that file at the same time; thus, a file can be downloaded from a single source by several peers, while the source only spends the bandwidth necessary to send the file once: the source sends a different piece to each peer that needs the file, and the peers then exchange pieces among themselves until each has the complete file.[10]

In other systems the resource that cannot be centralized is *trust*: Systems like Tor[23, 72] route messages through several hosts between the peers so as to hide who communicates with whom, even from the peers that intermediate the communication. Freenet[21] extends the anonymity to storage: as in other file-

[9]Qualitatively, not quantitatively. Peers can (and usually do) have very different *amounts* of the shared resource.

[10]Note that these systems are not *pure* peer-to-peer systems: although the peers share their resources, they don't manage them strictly by coordination among themselves; instead, a centralized server is used to arbitrate between them.

sharing systems, peers hold (pieces of) the shared files, but encrypted such that a peer *doesn't* know what files it hosts; only peers that know the necessary keys and assemble the files can determine their content.[11] (Note that even though these systems also implicitly share storage and bandwidth, it is the distribution of trust that is the *reason* for their existence.)

Overlay Networks

In any distributed system, a host intending to make use of a resource must communicate with the host that controls that resource; in general, this means finding its network address. In the client-server model all resources reside on the single server; clients can be safely assumed to know how to contact it.[12]

In a peer-to-peer system this assumption no longer holds: Indeed, a typical peer-to-peer system has a very large number of peers; a participant would need to hold a very large list in order to know all other peers. More importantly, peer-to-peer networks are very dynamic; peers join and leave the network very often, and notifying each peer of one's arrival or leaving would require a prohibitive amount of communication.[13]

For this reason most peer-to-peer systems organize themselves into an *overlay network*, distinct from the physical network used to communicate. All peers participating are nodes in the overlay network, and peers that know each other are said to be *linked* in the overlay. (Note that *all* peers are linked in the physical network, in the sense that each peer can communicate with any other peer. Depending on the system, a peer needing to communicate with another peer it doesn't know—that is, one that it isn't connected to in the overlay—will use the overlay network to either find its address or to relay its message to it; the answer is usually direct, which is made possible by attaching the "return" address to the initial message.)

---

[11] This feature is intended to provide, among other things, plausible deniability: a client hosting a piece of the file cannot be shown to know what it contains, and thus might argued it is free from responsibility for its contents. Furthermore, due to the particular way Freenet splits files into fragments and distributes them through the network, a popular file is very difficult to remove; this is intended to prevent censorship.

[12] The use of, e.g., the Domain Name System does not contradict this. DNS is needed for identifying a server among many; the client-server model concerns interactions with each server individually, i.e., what happens *after* the server is found.

[13] The clients of a client-server system are also very dynamic, but they don't need to communicate among themselves; a server only communicates with a client when the client requests it. Thus, a client needs to communicate its presence only to one server, but a peer would need to communicate it to all other peers.

Although conceptually distinct, the organization of peers into overlay networks is intricately connected to the problem of resource discovery: the way peers are connected dictates how they can find the resources they need, or, more precisely, how they find a peer that controls a particular resource. Accordingly, peer-to-peer systems are classified according to the type of overlay network they use:

Structured P2P networks organize their peers (or the links between them) according to precise rules, allowing specific algorithms to find required information with certain performance guarantees. These are discussed in more detail in the next section.

Unstructured P2P networks do not impose or require a precise organization on their overlay. While usually not completely arbitrary, the topology of unstructured networks doesn't depend on the precise properties of the nodes or their resources.

A subtly different classification can be made according to the homogeneity of the peer roles: In *pure P2P networks*, all nodes have the same status; they participate equally and identically in all operations. In a *hybrid P2P network*, some nodes (often dubbed *supernodes*) are distinguished; "ordinary" nodes are always connected to a supernode, and the supernode intermediates contact between its subordinates, and between its subordinates and those of other supernodes. Usually, such hybrid networks are *hierarchical*, with the supernodes themselves being connected to and managed by higher-level supernodes. Finally, a *centralized* P2P network uses a separate host, analogous to a server, to manage and intermediate contact between the peers.[14] Note that this second classification usually applies only to unstructured P2P networks: structured overlays are almost universally purely peer-to-peer.

Some overlay networks, regardless as how they are classified above, allow the topology of the underlying physical network to influence that of their overlay: links are established preferentially between nodes that are "close" in some sense, e.g., by having high bandwidth or low latency links between them. Nevertheless, such preferential links do not in themselves confer "structuredness" to the overlay that use them; peer-to-peer networks are considered structured if their overlay has structure by itself, not if they just imitate that of the under-

[14]Note that a centralized peer-to-peer system is distinguished from a client-server system by the fact that the peers communicate and cooperate between themselves, albeit with the help of the "central node". Compare a P2P system like BitTorrent, where peers exchange files between themselves under the direction of a *tracker*, with a distributed computing system like SETI@Home. In the latter case, although individual computing nodes do work "for the group", they do so individually, without communicating between themselves.

lying structure.

Structured P2P networks

A mechanism for connecting to peers and sharing resources with them is generally not sufficient for a peer-to-peer application.[15] The different participants are usually interested in specific resources—for example, in a peer-to-peer file-sharing network, each peer is interested in downloading specific files, which often are not under the control of its immediate neighbors in the overlay—so it is usually necessary to have a mechanism for locating resources based on their characteristics (e.g., their name), and implicitly the peers controlling the needed resources.[16]

Early peer-to-peer networks were mostly unstructured. The choice of mechanisms for resource location was very restricted. A centralized system can make use of a server that indexes all shared resources (e.g., in a file-sharing network, a list of file names, perhaps with other meta-data like size and modification date), and that answers queries from peers. But in many cases the centralized solution is not available—often for non-technical reasons—and an unstructured network offers no other solution than simply broadcasting a query from peer to peer, hoping that it will eventually reach a peer owning the requested resource. This practice is usually termed "flooding" in this context; the metaphor is quite appropriate, and though the technique does work it is quite inefficient for many purposes. For example, rare content is harder (and usually much slower) to find than common content.[17] In the worst case, searching for content that does not exist, a common situation, would cause the query to propagate to the largest amount of peers. Even the basic problem of avoiding to send the same query repeatedly to the same peer (since an unstructured network may have multiple multi-hop paths between any two peers) is not easy.

[15] Although exceptions do exist. If the shared resource has high liquidity, i.e., resources shared by different peers are interchangeable for the general purpose of the network, then a peer needs only find a certain amount of resources, not specific ones, so they can just pick the first it finds.

[16] Once a resource and its controlling peer are located, the network's structure is no longer relevant: the two peers can communicate directly.

[17] Finding rare content is usually the more interesting feature to research; for example, a file-sharing application for content that is widespread is less interesting, because it is implicit in being widespread that a solution already exists for distributing it. That said, some peer-to-peer applications are intended specifically to aid the distribution of very popular content, with BitTorrent as an obvious example.

Key-Based Routing

Though many solutions to the problem described above are possible, one of the most fertile has been the concept of *Key-Based Routing*, often abbreviated KBR. Very generally:

In a KBR system there is a set of *keys* (details vary considerably, but, for illustration purposes, a popular choice would be a large set of consecutive integers, e.g., the set of 160-bit numbers).

The keys are logically distributed among peers in a certain way (again, details vary, but for our example each peer might "own" a $1/N$-long interval of the key space, with N being the number of peers).

The network then maintains the links between the peers—the overlay—in such a way that an algorithm (the *routing* algorithm) exists that allows any peer to contact the "owner" of a key, following only links of the overlay network, with some guarantees regarding the maximum number of "hops" that must be done.[18]

The routing algorithm is usually different depending on the key set and the way keys are logically assigned to the network nodes, and often several variations of an algorithm are available. Thus, a KBR system is a triplet (key set, key assignment, routing algorithm); although all have the three elements, and similarities are often great, there are almost always at least subtle differences between the corresponding levels of different KBR systems.

The generic explanation above might be easier to understand with an example. Because much of the research in this document is based on a specific KBR system—Chord—we will describe it briefly in the next subsection.

KBR with Chord

Chord[71] is a key-based routing system that uses the set of 160-bit integers as its keys.[19] The keys are logically considered ordered, such that 1 succeeds 0, 2

[18]Network latency is limited by light speed, and current networks are relatively close to this limit. In contrast, bandwidth depends more on technological advances, and the amount of bandwidth available for a certain cost is constantly increasing. Thus, the cost of just opening a connection rises relative to the cost of sending a certain amount of data per connection.

[19]The constant 160 was chosen with the intent of using the SHA-1 cryptographic hashing algorithm for operations using the KBR layer; these will be explained later. For the purposes of this subsection, the only important requirement is that the total number of keys be much larger than the number of nodes in the network; 2^{160} is much higher even than the 2^{128} number of IPv6 addresses.

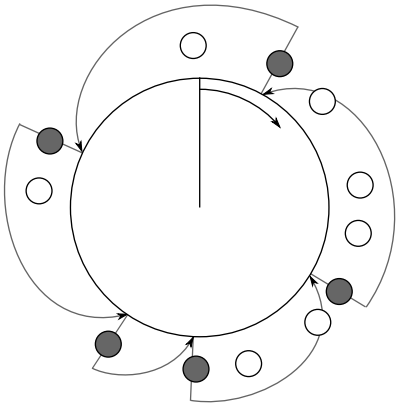


Figure 2.5: Illustration of a (very sparse) Chord overlay. Gray circles represent peer nodes and white circles key/value pairs. The vertical radius marks the origin of the ID circle; in this illustration key/ID values increase clockwise. The gray anti-clockwise arrows mark the section of the key space each node is responsible for.

succeeds 1, and arranged on a circle, such that `0` succeeds $2^{160} - 1$.

Each peer node is randomly[20] assigned an identifier (ID) from the same set of keys. The ID of the node can be thought of as its logical "address" on the circle of keys.

For any of the 2^{160} keys, Chord defines the `successor` of that key to be the first node after it on the ordered circle. (In most cases this means the node with the lowest ID that is still numerically greater than the key; that is not true, however, because the keys form a circle; for example, 0 is a successor of $2^{160}-1$.) The `predecessor` of a key is defined analogously. Chord considers each key to be the responsibility of its successor node. Given the random distribution of node IDs, in practice this means the key-space is *on average* evenly distributed between nodes. Figure 2.5 shows a schematic of these conventions.

Each Chord node maintains a list of connections to its peers, called "fingers". If a node has the ID k, its fingers will point to the nodes that own the keys $k + 2^n$, with $n = 1..160$. The fingers of a node are, as a consequence, pointers to progressively "further" points of the circle, each pointing twice as far from the node as the previous one. (Chord also keeps connections to one or

[20] A SHA-1 hash of the node's physical network address is the usual method.

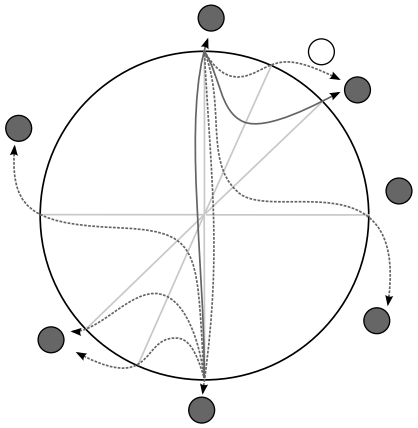


Figure 2.6: A Chord overlay with the same orientation as that in Figure 2.5. The light gray lines mark the reference positions for the finger table entries of the top node (right semicircle) and the bottom one (left semicircle); note they divide the key space in successive halves. The darker gray arrows point to the nodes that populate each table entry for the two nodes.
Each arrow intersects the circle in "target" position of its finger table entry; note that the node populating each finger table entry is the first node following the reference point, i.e., is its successor.
The continuous arrows mark the route taken by a query from the bottom node towards the key (white circle): first to the top node, then to the top-right node; the latter node is the successor of the key, so it will answer the query.
Observe that some of the finger table entries point to the same node; this happens because the identifier space is much larger than the number of nodes, and the first few fingers table entries of a node divide its neighborhood very finely

more successor and predecessor nodes, and perhaps other connections cached as a result of previous operations. But it is the fingers that are essential to the Chord algorithm.) These connections form the Chord overlay network. See Figure 2.6 for an illustration.

Consider now that a Chord node needs to find the owner of a certain key—the basic operation of key-based routing. The node will consult its finger-list, and it will direct its query to the furthest node in the list that is before

the position of the looked-for key on the circle. That node, in turn, will recursively use the same procedure. However, given the specific distribution of fingers, each node will get progressively closer: if node X decides to send its query to one of its fingers Y, then Y will never propagate the query further than the next finger in X's list. Consider:

The key, in general, can be anywhere on the Chord key-circle. Each time a node uses the above procedure to select a finger, the search space is at worst halved, i.e., at least half of the remaining peers are determined not to own the looked-for key. It takes at most $log_2(N)$ halvings to reduce the total search space to 1, thus finding the owner of a key requires in the worst case $log(N)$ hops, with N the number of peer nodes in the network. (For comparison, looking for a node in an unstructured network will lead to contacting *on average* half the nodes.)

Distributed Hash Tables

Key-based routing, as the name implies, is concerns only with locating nodes based on abstract keys. To be useful, a KBR system needs a method for associating data—or, in general, the resources shared by participants in the network—to keys (and, thus, to the nodes in the network), such that searching for a certain resource translates easily to routing a message to the node "owning" that resource.

We describe here briefly one of the approaches, the Distributed Hash Table (or DHT), partly because of its simplicity and popularity, and also because it serves as a fundamental building block for many contributions described later in this work.

In a DHT, each resource is associated a key by hashing some property of the resource—quite commonly the name—and placing responsibility of the resource to the network node that owns that key. Finding a resource is thus similar to using a hash table. (Note that a hash table is commonly said to be $O(1)$, meaning that searching for a value takes a constant time independent of the number of values in the table. For a DHT the number of hops—average or worst case—needed to reach a key is similarly cited and usually has a different value, e.g., $O(log(N))$ in Chord's case. However, the latter value makes reference to the *number of network nodes*, not values; in general, searching for a value in a DHT requires on average a constant time independent of the *number of values* stored in the network.)

Assuming a well-chosen hashing function,[21] the keys of any set of resources will be distributed evenly throughout the network, meaning that no peer will

[21] Chord uses the cryptographic hash algorithm SHA-1.

be burdened more than others. That said, a system might intentionally use an order-preserving hash function (or similar mechanism) to keep similar-valued resources close in the key space; this will lead to related values being held by one or a few successive peers, a property that might be useful for, e.g., range querying.

The resource is not necessarily moved physically from the peer that shared it to the one "owning" its key. Instead, the DHT can use as an index for locating the resource (see below). Which method is chosen depends on what exactly the application intends; for example, Freenet actually splits shared files and physically distributes the fragments redundantly among the peers in an attempt to prevent censorship: in order to forcibly remove a file from the network, one needs to find and coerce most of the large number of nodes that store the fragments, rather than the smaller number of nodes that would hold the entire file in most systems. The major part of this thesis deals with indexing systems.

For a very simple example, in a file-sharing DHT-based network, a peer will use its files' names as identifiers, hash them into a key, and then store *its own identifier* as the "value" in the DHT. Another peer looking for a file with a certain name will hash the name, retrieve the values associated with the resulting key from the DHT, which are the addresses[22] of peers owning files with that name, and then contact those peers directly to retrieve the file.

It is easy to see that, even given a certain DHT implementation and a single kind of shared resource, there are many different ways to distribute or index the resource throughout the network. Also, given particular indexing conventions, it is in general possible to use several different DHT algorithms. Thus, a DHT can be considered as a building block for peer-to-peer applications, in much the same way as hash-tables are not programs in themselves but building blocks. One consequence of this is that it is difficult to predict and compare, simply by analysis, the performance of different solutions to the same problem. Unfortunately, the distributed nature of peer-to-peer applications means that testing them is more difficult than for local programs; the next chapter discusses in detail the testing and simulation of peer-to-peer systems. The following section describes in more detail existing DHT systems.

---

[22]Observe that it is possible for the same key to be associated to different resources, and that "the same" resource (given the equivalency relation defined by the key) may reside on several peer nodes. This can be a difficulty—requiring an additional disambiguation method for content with ambiguous keys—or an opportunity, e.g., providing load-balancing or resiliency almost for free.

2.4 Testing and Simulating P2P Applications

Section 2.3, *P2P and KBR Systems*, summarizes several of the various KBR algorithms and DHT protocols in existence. All such systems offer a fundamental primitive to higher-level applications: lookup of a key, and retrieval of values associated with that key, distributed within a network self-assembled from a very large number of computing nodes, in using a number of messages sublinear in the number of peers participating.

The protocols involved are regular enough that relatively simple algorithmic analysis can provide guarantees about the worst- or average-case cost for the basic routing and retrieval operations. However, such analysis is not sufficient, for several reasons:

In the particular case of peer-to-peer networks, the number of participating nodes is limited; even a very conservative estimate of one peer active per person, at all times, results in a order of magnitude below 10^{10}, or approximately 2^{33}. In practice, the number of simultaneously active peers is much lower. Given the subunitary (usually logarithmic, or subunitary-exponential) bounds computed for the basic DHT operations, the actual orders of magnitude resulting from the cost formulas are usually close; as a consequence, the constants hidden by limits in the O notation become significant. Since the network-level operations counted by such bounds—messages—represent high costs in human terms,[23] very detailed accounting of the costs is necessary, which is much harder to do analytically.

Detailed accounting is complicated by two other issues: First, the "quantum" of cost measured by the number-of-steps formulas is far from constant. Network latency and bandwidth between peers can vary by orders of magnitude. Protocols that keep count of various network locality measures are hard to evaluate in detail, and any analyses are conditional on assumptions about the physical network structure and, usually, how the high-level application uses the DHT layer.[24]

Secondly, the various DHT protocols among which an application developer might chose usually offer different trade-offs between such properties as load-balancing, resiliency to failure, resistance to attack, or behavior during

[23] The network latency of one message exchange over the entire Internet is close to human perception, of the order of tens of milliseconds. In contrast, the basic operations of, e.g., sorting algorithms are often many orders of magnitude below human perception; thus, human-noticeable delays are encountered mainly for vast numbers of primitive operations, where the limit behavior is a more useful description.

[24] As an example, Kademlia favors long-life nodes, which improves responsivity but affects load balancing in ways that may be difficult to evaluate.

non-steady states, all of which interact in a complex way with performance. In most cases, such trade-offs may be made even within a protocol, e.g., by varying network-wide constants (like the size of k-buckets in Kademlia[50]) or even the hash function used to distribute keys.[49]

As a consequence, choosing which DHT or routing layer protocol should be decided by testing the actual behavior, after execution, rather than through analysis before implementation, perhaps using Grid'5000[7] or a similar system. Even supposing the analysis of lower level protocols were possible, analysis of the higher level application is certain to be even more complicated; and even given that higher level of analysis, the fully developed application would need to be tested, a challenging task for a large, asynchronous peer-to-peer application.

The difficulties described above apply to almost every high level peer-to-peer application. This presents an opportunity: a generic framework could abstract and implement as much as possible of the solutions, allowing a researcher or developer to focus on the implementation and study of the specific system he is working on.

Such a framework should contain generic interfaces to abstract KBR or DHT layers, allowing an application to be written, when possible, independently of a specific low-level protocol; ready-made implementations of existing protocols against these interfaces, allowing their quick assembly with an application module; and, most importantly, provide facilities for running tests and simulations of the resulting application, to observe and measure its behavior in practice.

Chapter 3

XQ2P

XQ2P, the main contribution of this work, is a research-oriented peer-to-peer–distributed XML database; it responds to queries expressed in the XQuery language.

The first section below introduces XQ2P, the problems it tries to solve and the assumptions we made when developing it.

Following that, *Architectural Overview* contains a high level presentation of XQ2P's architecture; Section 3.3, *XML Document Handling*, describes how XML documents are handled. Section 3.4, *The XQuery Processing Kernel*, details the query processing subsystem at the core of XQ2P; the XQuery kernel handles local query execution, and upon it are built the distributed query techniques presented in the next two chapters.

The final section of this chapter describes P2PTester, a framework for developing and testing peer-to-peer applications, and how we used it to build XQ2P.

3.1 Introduction

The main contributions of this paper have been implemented in XQ2P. Our intention was to explore the problems and potential opportunities arising from applying the peer-to-peer model with XML database technologies. This combination of concepts giving rise to many avenues of investigation, we attempted to limit the scope of our work to what seemed to us the most interesting direction.

Assumptions and Problem Domain

We define thus our model through the following set of assumptions:

- The users of the system form a heterogeneous community, with diverse interests;
- Each user (or a vast majority of them) owns and controls data, and is willing to share (allow use of) this data with the others users; shared data is available to every user.
- Each user is interested in executing queries in certain domains; such queries require accessing data *in that domain* shared by some of the other users;
- The number of users and the total amount of data shared are too large to allow users to simply replicate each other's data; and:
- No central entity is available to manage the shared data, e.g., by partitioning the data in domains; however:
- Typical queries need only a fraction of the data, from a relatively small number of other users (i.e., those users with shared interests *and* who shared data relevant to the query);
- All data is represented as XML documents. A user may share any number of documents, including zero;
- Queries are written in XQuery;
- Data from each domain of interest is structured, in general, in standardized ontologies. In other words, we do not consider the need to match and adapt different ontologies, except what the user can express via queries.

Observe that, from a high-level view, these statements basically describe an XML database: the user is presented with a vast collection of documents, and writes XQuery programs to find, retrieve and manipulate the assembled data.

The difference is that the system is distributed: the queried data is distributed among the (potentially vast) set of users and machines participating; XQ2P attempts to present the entire volume of data the same way as a unitary XML database.

It is important to remark on the assumption about queries' data dependence. Given our model of data ownership, queries that need to access most of the data in the database will by necessity lead to the querying node communi-

cating with most (if not all) of its peers. As such, in general a database cannot do better than contacting all peers and asking for the relevant data.[1]

In contrast, when individual queries only need to access part of the data, a different strategy is available: the system can attempt to identify the nodes owning relevant documents via one or several indexes, and then communicate with those nodes only.

3.2 Architectural Overview

In very general terms, XQ2P functions as follows:

To participate, a user must obtain and run an XQ2P node. First, the node must connect to an XQ2P network;[2] to do so, the address of any node already part of the network must be provided.[3] XQ2P is built on top of a KBR network; in principle, any KBR algorithm can be used, its details and performance being reflected in the behavior of XQ2P.

Second, a node must publish its data over the network. All shared documents are indexed, and their indexed contents are distributed throughout the overlay network, using operations very similar to those of a DHT.[4]

Finally, a node may be asked (by its owner and user) to execute queries, expressed in the XQuery language. A query is first analyzed and its components are compared with the index; Then, the distributed index is used to locate the peers that contain relevant documents; These peers are contacted, and the relevant data is retrieved from them; Finally, the initiating node aggregates the resulting data according to its original query, and returns the answer.

Note that this system is modular, and presents ample opportunity for modifications and extensions. Any KBR system—even several at the same

[1] For *particular* queries it may be possible to design a P2P system that can retrieve aggregates depending on all available data without accessing the original data sources; our system, however, does not have knowledge of which kind of queries it will need to execute.

[2] XQ2P allows and encourages the sharing of many different kinds of data over a single overlay; however, it is possible to create several separate XQ2P-based networks, for reasons external to XQ2P itself, e.g., access control.

[3] We do not address directly network discovery in this work; any of the usual methods are applicable, e.g., DNS records, "well-known" addresses, or friend's nodes.

[4] In fact, our indexing could be implemented over a DHT; however, we implement slightly different semantics than a basic DHT, to support efficient querying.

time[5]—might be used as a basis, and XQ2P's XQuery kernel may make use of varied types of indexes. We used Chord's key-based routing for our implementation; the types of indexes we implemented are presented below.

A wide variety of algorithms and indexing methods have been published that aim to optimize one of these domains of XQuery, or most often a particular subset of queries in a domain.[6] For this reason, we designed XQ2P's XQuery processor to allow the separate study of algorithms applied to these divisions.

XQ2P's query processor is designed to be easily modifiable and extensible in order to allow experimentation with many different optimization algorithms. To this end, we divided our work in two parts:

First, we implemented the entire XQuery standard in the most straightforward and simple manner possible, without any optimization. A query is received as text, and is parsed into a structure that follows closely the structure of the XQuery grammar. Then we build a tree of operators that is almost isomorphic: each syntactic construction of XQuery has an associated operator that implements directly the semantic described by the standard, except for the few elements that appear in the XQuery grammar for the purpose of textual representation.[7] This operator tree implements the entire semantic of the query, and can be invoked to execute it directly.

Second, we implement the specialized algorithms as separate operators. Such a specialized operator optimizes a certain part of the computation, and uses the generic operators for the rest. In particular, the distributed indexes and operators presented in chapters 4 and 5 are built in this fashion.

3.3 XML Document Handling

XQ2P is a peer-to-peer distributed *XML* database. As such, each node holds a collection of data in XML format.[8] In our model, a node's data is a collection

[5]This can be useful if the different overlays' structures allow indexing with different features or performance.

[6]To our knowledge there are no peer-to-peer applications that attempt to optimize the entire set of XQuery features, nor even to support the entire XQuery language with partial optimization.

[7]For example, parenthesized expressions do not have an associated operator, as they serve only syntactic purposes.

[8]It is of course possible for a particular node to not hold any data itself. Such a node can be used to query only the data shared by other nodes. But all nodes have the *ability* to host data.

of XML documents (i.e., nodes do not host XML fragments).

Because our research is focused on querying, we did not devote effort to implement a particularly complex storage system; XQ2P uses very simple in-memory storage and indexing.

A document is added to a node's data store by simply passing it the path to an XML file. The node loads the document to a structured in-memory representation and indexes it.[9] The file path serves is also remembered, serving as the document-uri property of its `document` node for queries.

In-memory representation: Documents are parsed using SAX to a structured form that is kept in memory. The data structure itself is based on the XQuery 1.0 and XPath 2.0 Data Model (XDM),[79] and is very similar to the DOM model. The main difference from the Java DOM binding is that the various node properties are expressed using the data types used by the XQuery processor—in fact, the same classes are used during XQuery processing.

Once parsed and loaded to memory, documents are assigned a numeric identifier, unique among the documents owned by that peer.[10]

3.4 The XQuery Processing Kernel

A database is of little use unless it is possible to query the data it holds. The main focus of our work was thus to provide a powerful querying subsystem for XQ2P.

XQ2P accepts and executes queries in the XQuery language.[11] An important point that merits repeating is that we expended considerable effort attempting to build a system that handles the entire XQuery 1.0 language.[12]

Optimization techniques will, in general, support only a subset of a query language's features; a common approach when studying such techniques is to implement a processor for only the optimized subset, perhaps with some other

[9] The document is also published to the network, a process described later, at the same time. It would be possible to separate the two operations, and thus keep some documents private to a node, but we did not yet need to implement this.

[10] In principle, the document's path of origin can be used as an identifier. We prefer numeric ones, though, because path strings are much more verbose.

[11] As a proper subset of XQuery, XPath can also be used for querying, e.g., by XPath applications that use XQ2P as a storage layer.

[12] XQ2P passed above 98% of the tests included in the XQuery Test Suite[81], though the exact number varies as the test suite is updated.

useful features. A complete query module is constructed, in this approach, by adding new techniques for handling more features, hopefully resulting in a complete implementation.

This approach, although appropriate in many scenarios, has a problem of particular importance to us: Different optimization techniques will, in general, allow supporting different subsets of the complete language. As a consequence, it is difficult or impossible to compare directly two systems with different feature coverage on the same workload. Instead, tests must be done with queries tailored for each individual system tested, and comparisons will necessarily require some extrapolation of the results.

We have adopted the opposite approach: We first developed a system that supports the entire XQuery standard.[13] Rather than performance, our focus for this part was simplicity, ease of implementation, and extensibility. These goals allowed us to implement the system quickly, and thus to devote more time to investigate specific optimization techniques, and to integrate these latter with relative ease with the generic system.

With the generic querying system complete, we have then developed extensions that utilize indexing techniques and optimized algorithms to optimize subsets of the language, leaving the rest to be handled by the generic part.

Note that the generic processor does not include any notion of peer-to-peer processing. Every functionality related to the distributed nature of XQ2P is implemented as extensions to the basic system. A nice side-effect of this is that XQ2P's basic processor is also useful as a basis for researching XQuery optimizations in general—in fact, the optimization techniques studied in this paper, where possible, have been implemented first as modules for local processing; the distributed-processing modules are extensions of these, with modifications and additions appropriate to the peer-to-peer context.

The following sections of this chapter describe the generic query processor, i.e., the unoptimized but complete implementation of XQuery used as a basis for study. The following chapter describes the optimized operators we developed for structural queries, as an extension to the generic processor, and the chapter after that presents the operators that extend the query processor to operate over the peer-to-peer distributed database.

As the remainder of this chapter does not involve distributed processing concepts, in the following sections "query processor" should be taken to refer only to the generic XQuery processor component. Where distributed processing is involved, we will say so explicitly.

[13] Except for some features the standard itself classifies as optional. In particular, the Schema Import, Schema Validation and Static Typing features are not (yet) supported.

Architecture

As mentioned above, in designing the generic query processor we sought simplicity and straightforwardness rather than purely performance.

A common approach for optimizing XQuery is to define a simpler model that expresses only the subset of language features that is to be investigated, and devise a procedure for formulating supported queries in the chosen formalism, while preserving their semantic. This eases considerably the researcher's task, as XQuery is a very complex language.

We believe that this approach is not optimal for our project: We intend the query processor to be easily (or as easily as possible) extended by adding several specialized operators. A given simpler formalism will be appropriate for one or perhaps a few optimization techniques, but it risks complicating the implementation of many different ones.

For these reasons, we decided instead to structure our processor by following the structure of the XQuery language. Almost every component is a close translation in Java of a syntactical or semantic element specified by the standard. Besides being simple to implement and amenable to extension with almost any optimization method, this approach has the advantage of being exceedingly easy to read and understand by the prospective extension developer: familiarity with the XQuery language and its semantics can be easily translated to familiarity with the processor's implementation.

The XQuery standard defines queries as composed of expressions. Except for a small number of elementary expressions—e.g., constants—each type of expression is a possible way of composing other expressions; the expressions composed by another expression are called subexpressions or operands. The semantic of an expression, accordingly, is defined by describing how an expression's result is obtained from the results of its operands.

Accordingly, we built our processor in the most straightforward way: for each XQuery expression we wrote an *operator* that implements the semantic dictated by the standard. At operator instantiation subexpressions are passed as constructor parameters, and stored in member fields. Thus, an operator instance represents an entire expression tree; the operator instance corresponding to the topmost expression in a query implements the entire query's semantic.[14]

[14]That is, the expression evaluated by the query. An XQuery program also contains elements that are not expressions, e.g., options and function declarations (although these declarations may contain expressions). These are recorded in a special "query" object, of which the top-level expression is another member.

Evaluation All operator objects have an `evaluate` method, which takes as argument a *dynamic context*, and returns the `Sequence` of items, containing the result of evaluating that operator—or, more precisely, the tree of expressions rooted at it—in the given dynamic context. (Recall that the XQuery Data Model[79] defines the result of every expression to be a sequence of items. These data types are described below.) In general, when the `evaluate` method is called, an operator will call the same method of its sub-expressions (manipulating the dynamic context before passing it to them, if necessary), then will perform its own function using the results as operands, and finally will return the resulting sequence.

Errors The evaluation of an XQuery expression can also generate a dynamic error; this situation is implemented by throwing a Java exception. All raised exceptions are subclasses of `XQueryError`, itself a `RuntimeException`.[15]

Data structures

This section section presents the Java classes used to represent XQuery data. They are presented here to allow using them in the following sections.

Note that these data types are used throughout the processor: besides the values passed between operators during processors, these same data types are used for the in-memory representation of XML documents, and also for related properties of, e.g., operators. We have attempted to map the XQuery data types to Java classes and interfaces as naturally as possible; the main intent was to allow expressing XQuery semantics with natural constructions in Java.

For reference, a diagram summary of the XQuery data types is represented in Figure 3.1.

Sequences

The most general data type in XQuery is the sequence: operations are always defined in terms of input and output sequences; particular cases where an operation returns "nothing" or just one item as a result are expressed as returning the empty sequence or a cardinality-one sequence. The last part is worthy of

[15]It is perhaps unfortunate that Java and XQuery terminology differ in this respect, though there are reasons for both; we decided on `XQueryError` by reasoning that `XQueryException` can be too easily interpreted as related to Java rather than XQuery semantics, and that `XQueryErrorException` would be silly. Note that XQuery 3.0 maintains the terminology, even though it adds try/catch expressions that allow execution of a query to continue even in the presence of errors.

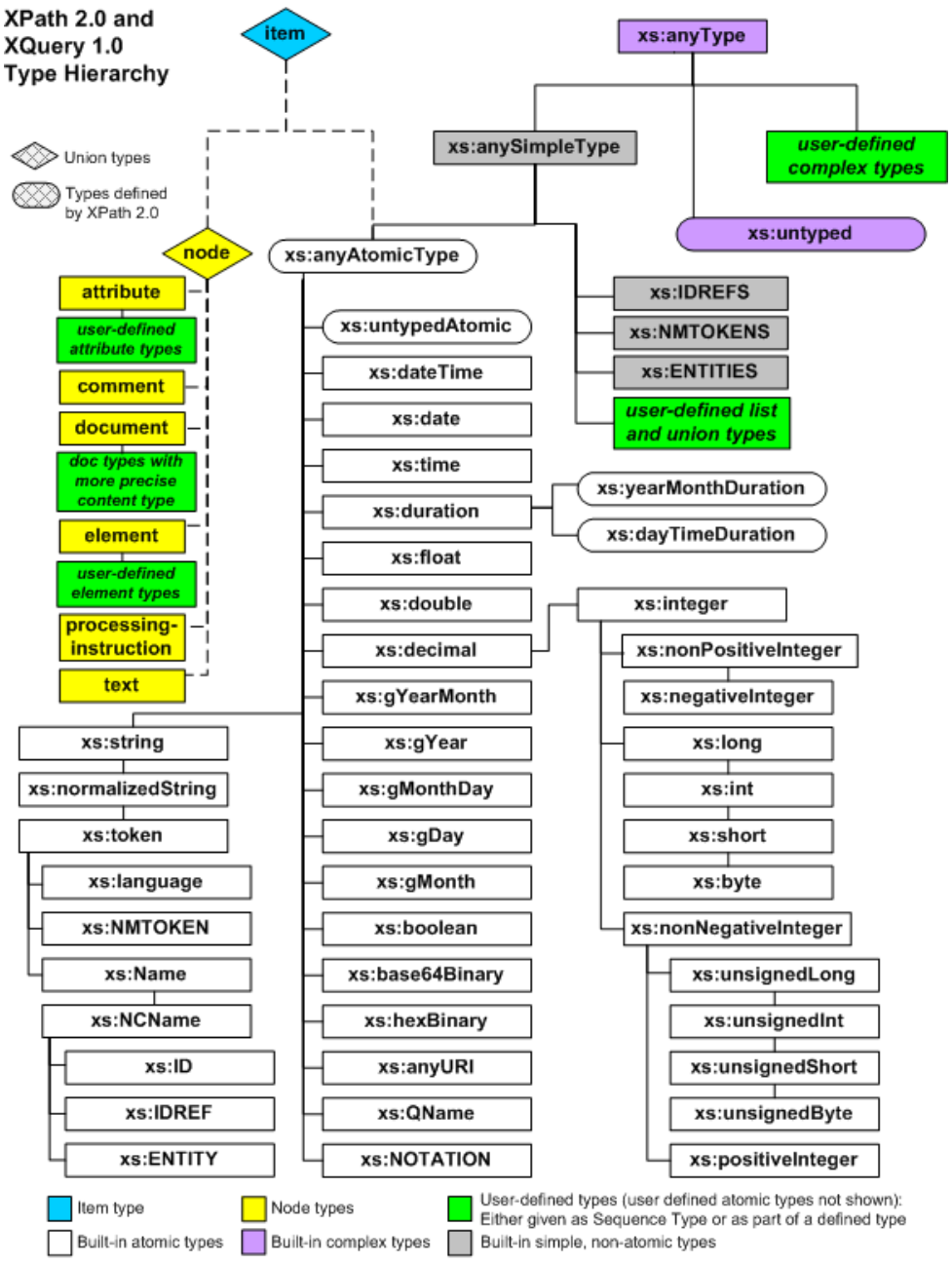


Figure 3.1: XQuery type hierarchy

Image source: *XQuery 1.0 and XPath 2.0 Data Model (XDM) (Second Edition)* [79]

mention: whereas most programming languages make an explicit distinction between an object and a container (lists and arrays being the usual container, the analogue of a sequence) containing only that object, XQuery explicitly takes the opposite position.[16]

We represent the XQuery concept of sequence with the `Sequence` Java interface. We believe that this mapping allows more flexibility than the alternative of making it a base (probably `abstract`) class for all XQuery types. Instead, the main branches of the XQuery type system—nodes and atomic values—are mapped to two different Java class hierarchies; they both implement the `Sequence` interface using helper abstract classes.

Sequences are immutable in XQuery, and this is also true of their Java representations. Appending sequences, for example, means creating a new sequence rather than adding elements to one of the operand sequences. Due to this particularity, we have chosen not to create the `Sequence` interface as a descendant of the usual Java `List` interface; instead, we created the interface from scratch with only methods that make sense for the XQuery type. However, `Sequence` does extend the `Iterable<Item>`[17] interface, mainly to allow common Java usage patterns with `for` loops. The optional `remove` operation is not supported however, attempts to use it generating an `UnsupportedOperationException`. `Sequence` does declare sequence-combining operations, like `concatenate` and `append`, but these create new sequences rather than modifying the sequence they apply to. (They are written in the interface only to allow a more natural syntax in Java.)

Besides concatenation, the usual operations XQuery does with sequences is selecting an item (the method `get(int)` and the common case `first()` helper method implement this), and sub-sequencing, i.e., creating a new sequence that contains a continuous subsequence of the first (implemented by the `before` and `after` methods, with the convenience method `rest` being the counter-part to `first`; however, note that `first` is mainly intended to be used to transfer an item in a `Sequence`-typed variable to an `Item`-typed one without a cast).

Finally, the interface specifies the exact behavior of the `equals` and `hashCode` methods, to ensure correct behavior. (In short, two sequences of the same length are equal if their items are pairwise equal.)

We mentioned before that XQuery items automatically behave as singleton sequences by implementing the interface. There are separate implementations of the class, however, for the other possible cardinalities:

---

[16] Representing no result as an empty list, by contrast, is rather more common, occurring for example in several variants of Lisp. Unlike Lisp languages, it is not possible to express a sequence of sequences in XQuery.

[17] `Item` is the base interface for XQuery items, and will be presented shortly.

The empty sequence is implemented as a singleton object, a `public static final` member of `Sequence` called `EMPTY_SEQUENCE`. All operations that would return an empty sequence will return this object. Note that this latter sentence is descriptive, not normative. In theory the existing code should work even if a different object is returned, as long as it behaves as an empty sequence. Using a singleton is useful because it avoids creating unnecessary empty objects.

"Long" sequences `NonSingletonSequence` is a somewhat unfortunately named[18] class used to represent sequences containing more than one item. It is implemented straightforwardly by wrapping a Java `List<Item>` and providing the `Sequence` operations on top of it. Special cases where the result would be the empty sequence or a single item are handled correctly, by returning either the `EMPTY_SEQUENCE` or the (unwrapped) item as appropriate (this sentence is also true for the other implementations of the interface).

In general the user should never need to use this class explicitly; its use is confined to the various subsequencing and concatenating operations defined by the `Sequence` interface. User code must use these exposed operations rather than creating sequences manually.

Subsequences Subsequencing is a common operation; rather than create a new sequence object for each such operation (which can be costly for long sequences), we use a specialized `SubSequence` class; provided with an initial sequence and bounds within it, a `SubSequence` will behave as expected by returning elements from the initial sequence directly rather than keeping a copy of its list.[19] (The fact that sequences are immutable considerably helps with this.) Like the other sequence classes, this one too handles special cases explicitly: besides the empty and singleton case, a subsequence of another `SubSequence` will access the original elements directly rather than creating several wrapping layers.

---

[18] The name is not quite accurate because empty sequences are also not singletons, yet are represented by a different class.

[19] It is perhaps worth mentioning that this approach is not entirely without problems: a small subsequence (at least two elements) of a large one will cause the latter to be kept in memory, even though only the small one is still accessible. It would be possible to use weak references to avoid this. We have refrained from implementing such a complex system, at least until tests show it to be necessary in practice; we expect most code to eventually disassemble sequences to individual items.

Items

XQuery sequences are composed of items. In our Java implementation, the `Item` interface serves to mark an object as an item.

Because the two kinds of items, nodes and atomic values, are quite distinct in properties, we believe an interface to be more appropriate than an abstract base class. In fact, items in general don't have *any* property except their status as (XML) nodes or atomic values. As such, the operations defined by `Item` serve only to distinguish between the two. The XQuery practice of defining an operation's effect on an item based on the item type would usually translate in Java to a rather cumbersome sequence of `instanceof` and cast operations; the methods of `Item` and `Sequence` are intended to avoid this; thus, a user is expected to write (given `Item item` as argument):

```
if(item.isNode()){
  item.asNode() // ...
}else{
  item.asAtomic() // ...
}
```

rather than the more cumbersome

```
if(item instanceof Node){
  Node n = (Node) item;
  // ...
}else{
  AtomicValue a = (AtomicValue) item;
  // ...
}.
```

This recommendation implies a few more things besides being a little more legible: First, the `asNode` and `asAtomic` methods will throw `XPTY0004`[20] rather than a class cast exception when used on the opposite item kind; thus, when implementing most operations that expect only a single kind of item the tests can be omitted, resulting in the expected XQuery type error being raised automatically. Second, it is likely to be slightly more efficient: the methods are implemented without any casting and type-checking (as they are virtual methods implemented separately for nodes and atomic values, they already know the kind of item they apply to).

[20] XPTY0004: *It is a type error if, during the static analysis phase, an expression is found to have a static type that is not appropriate for the context in which the expression occurs, or during the dynamic evaluation phase, the dynamic type of a value does not match a required type as specified by the matching rules in [subsection] SequenceType Matching.*[77]

There are two kinds of items in XQuery: nodes, representing fragments of XML trees, and atomic values, which are somewhat similar to primitive types in other programming languages. (We have attempted briefly to employ Java's generics to partially reflect XQuery sequence and item types in Java, but the results were brittle and cumbersome. We believe the "conversion" methods to be better suited to our purpose.)

Atomic Values

In XQuery, atomic values serve a purpose similar to that of primitive data types in other programming languages: they represent, for example, strings and numbers. This view can be slightly misleading; atomic values can represent rather complex objects, like dates and time intervals. Perhaps a better description would be "everything that is neither a node nor a sequence" (note that the previous sentence uses "sequence" in the XQuery sense; it is possible, e.g., to view a string as a sequence of characters, but XQuery nevertheless would treat one as an atomic item.)

We represent the XQuery atomic type hierarchy[21] with a similar hierarchy of Java types.

Unfortunately, the XQuery notion of a *derived type* is quite different from Java's. Type derivation in XQuery is a restriction rather than an extension, as in Java. As a result, the Java type hierarchy is quite convoluted; fortunately, the user of the framework will not usually need to be aware of its internal details.

All Java instances of an atomic type are immutable. All operations will create a new value rather than modifying their operands.

As a general rule, all atomic types are represented by a class with the name `XS_` followed by the (unqualified) name of the type. All these classes are derived from an abstract class named `AtomicValue`; this provides the common operations, e.g., an implementation for the type-independent methods in interfaces `Sequence` and `Item`.[22] The method `getType()` is declared, used for introspection in XQuery programs; the results of this method will be described later.

With only a few exceptions,[23], the `XS_`⟨type⟩ are `abstract` Java classes;

[21] By this we mean only the hierarchy of built-in types; user-defined types are not implemented yet.

[22] `AtomicValue` is somewhat redundant with `xs:anyAtomicType` and its Java equivalent `XS_anyAtomicType`. In fact, the latter is the sole descendant of `AtomicValue`, and has no new members itself. However, we felt it was worth having it: its name is easier to read in Java source, and it allows a clearer separation of XQuery concepts from the Java elements.

[23] For example, `xs:boolean`, which is a very simple type.

they provide general type-specific functionality via `static` methods and some checking where appropriate. Instances—created via `static` methods of the abstract classes—are actually instances of `private final` types hidden inside the abstract ones. This is intended to allow implementing XQuery-derived types without forcing the results to be Java-derived from an implementation class; the detailed description of, e.g., numeric types, below, will make this clearer for the interested reader. (Remember that knowledge of these internal details is not necessary to make use of the XQ2P framework, but only to extend it with new types.)

Each `XS_`⟨type⟩ class, in general, defines methods of several kinds, with regular names, to facilitate use throughout the rest of the framework:

Factory methods are static functions that instantiate a new value of the appropriate type. These always have names starting with "new" followed by the XML Schema name of the atomic type, with an initial capital. There is always a factory method with a single `String` argument—all non-abstract XML Schema types can be cast into from strings. Where appropriate, the overloading is used to provide factory methods with other arguments. For example, `xs:int` is associated with the Java class `XS_int`, which provides the `newInt(String)` method, but may also provide a `newInt(int)` method.

Accessor methods are usually abstract methods that extract some particular information from the atomic value as Java data types. For example, an `XS_double` has a `decimalValue()` method that returns the value it contains a Java `double` value. There is always a `stringValue()` method that returns a Java `String`. Depending on the exact type, several auxiliary methods may be provided for convenience; for example, the numeric types have `isZero()` or `isInfinite` `boolean`-valued methods, which are easier to use than extracting the value and checking manually.

Auxiliary methods are also exposed where needed, usually as `protected` `static` methods. These are used for various purposes like parsing, checking or generating canonical values, and are left non-`private` to allow their reuse in the future.

In the following paragraphs we describe in more detail some of the more important atomic value types.

String types are all based on the `XS_string` class, which doesn't do anything itself other than wrap a Java `String`. Its derived classes—e.g., `XS_normalizedString` or `XS_NCName`—in general don't do anything else

other than verifying that the string they wrap respects their respective lexical rules. We must mention that `XS_NCName` also provides a method that generates a `NCName`, a different `String`-wrapping class we developed separately for use within XQ2P's XQuery parser.[24] Similarly, `XS_QName` allows conversion to `QName`, also used in the parser.

Numeric types are the more baroque parts of the type hierarchy. A different `XS_`⟨name⟩ class is declared for each XQuery numeric type; these are derived in the same way as in XQuery. However, a separate layer of interfaces is declared that is derived in the opposite order:

`Numeric` is the base interface. It marks all numeric types, and declares `stringValue()` and `boolean` courtesy method `isZero()`.

`NumericLevelDouble` marks `xs:double` (and XQuery-derived types, though none are provided yet); since the ones below extend it, all other numeric types are also part of this level. It allows retrieving the wrapped value as a Java `double` via the `doubleValue()`, and declares the `boolean` methods `isInfinite` and `isNaN()`.

`NumericLevelFloat` marks `xs:float` and the "lower" types (i.e., all but `xs:double`). It allows retrieving the wrapped value as a Java `float` via the `floatValue()`.

`NumericLevelFloat` marks `xs:decimal` and the "lower" types (i.e., all but `xs:double` and `xs:float`). It allows retrieving the wrapped value as a `java.math.BigDecimal` via the `decimalValue()`.

`NumericLevelFloat` marks `xs:integer` and everything derived from it. It allows retrieving the wrapped value as a `java.math.BigInteger` via the `integerValue()`.

These are used to facilitate the implementation of XQuery's detailed rules for arithmetic with mixed-typed operands.

`XS_float` and `XS_double` are implemented as wrappers of the Java `float` and `double` types. `XS_decimal` and `XS_integer` use `java.math`'s `BigDecimal` and `BigInteger`[25] classes, respectively. The many numeric types

[24] We intended the different components of XQ2P to be independent of each other, to allow their being reused separately; this explains the slight duplication.

[25] Note that, while useful for calculations, the Java classes are *not* quite equivalent to the XQuery semantics; in particular, turning to and from the string representation

derived from `xs:integer` use the closest Java type of sufficient width—for example, `xs:int` wraps a Java `int`—with careful checking at each conversion for the appropriate restrictions, e.g., non-negativity.

Time types The classes associated with date and time use internally a simple numeric value (milliseconds for durations, milliseconds since the Epoch for `xs:dateTime` or since midnight for `xs:time`), together with a timezone object (which internally records "minutes from GMT"). This facilitates operations with dates, turning most of them into simple arithmetic. (Java's `GregorianCalendar` is used internally to extract date elements like the year and day, however.)

Other types We straightforwardly wrap Java types, providing methods for conversion to and from strings. `XS_base64Binary` and `XS_hexBinary` internally use `byte` arrays for storage, `XS_boolean` a simple `boolean`, and the various other types, like `XS_anyURI` or `XS_untypedAtomic`, wrap strings.

Nodes

Nodes are the other kind item that may occur in sequences in the XQuery data model. In Java they are translated to a hierarchy of interfaces, one for each kind of node: document, element, attribute, text, processing instruction and comment; the names follow the usual Java conventions, without a prefix. Interface `Node` roots the hierarchy, and two "intermediary" interfaces, `ContentNode` and `ContainerNode` are used to eliminate type-testing and casting where possible (`Attribute` is a `Node`, `Document` and `Element` are `ContainerNodes`, and all nodes except `Document` and `Attribute` are `ContentNodes`); a `getNodeKind()` method is also provided, returning an `enum`-typed value.

A parallel hierarchy of classes is used for actual implementation of these interfaces. However, their details are not exposed to the user, therefore we do not detail them here. Actual instances of nodes are obtained by parsing a document, or as results of executed queries. The intent is to hide implementation details, and allow several different implementation of the node interface (and sub-interfaces), for example for adding an optimized storage engine to XQ2P or accessing nodes on other peers via RMI-like mechanisms.

---

requires particular care, and the rules for arithmetic with the values are subtly different, especially when operands are values of different types.

`Node` extends `Comparable<Node>`, comparing nodes by document order.[26] A system of numeric document and node identifiers is available. (Future implementations of the interface must use it to ensure consistent ordering between documents and document fragments of different actual types.)

Schema Types

Several parts of XQuery require explicitly manipulating representations of types: *Schema types* are types of items, as defined by XML Schema (with a few additions from the XPath and XQuery data model). (These should not be confused with *sequence types*, which describe sequences. The two are easily confused, because they share the subtree rooted at `xs:anyAtomicType`. Sequence types are described in the next subsection.)

Although XQ2P does not support the Schema Import and Schema Validation optional features of XQuery, all non-optional operations with predefined types *are* supported. As a consequence, parallel to the hierarchy of *classes* for the predefined types, there is also a hierarchy of *objects* that identify these types. (These are also used for sequence types, see below.)

Each simple type is represented by a singleton instance of `SchemaType`, stored as a `public static final` member of that same class for ease of use via static imports. Their names all follow a standard pattern, `xs_`⟨type name⟩; this is analogous to the Java class names used for objects of each type, using a lower-case prefix.

These type objects are very simple internally; their only properties are the type name (an `ExpandedQName`, a class that will be described later) and a reference to their base type (in XML Schema terms).

Sequence Types

Sequence types describe, as their names imply, the more-or-less precise composition of sequences that are passed around between various XQuery expressions. (For example, they may be used to declare types of variables and function arguments.)

Although sequence types are represented with a rather complicated set of classes, users are not intended to need all the details: A set of `public final static` members of the `SequenceType` class are made available, both properties and factory methods, which handle the details.

---

[26] XQuery semantics demand that many expressions implicitly return node sequences in document order; ordering of non-node or mixed sequences is more complex and does not map well to Java's semantics. This explains why `Item` does not extend `Comparable<Item>`.

Exception Types

XQuery defines a large number of error conditions and the precise error types that must be raised. XQ2P translates these to Java `RuntimeExceptions`—in fact, Java's exception mechanism is used for raising XQuery errors, as will be detailed later—and for this purpose we provide in package `xquery.errors` a class for each error defined in [77].[27]

Each error kind is represented by a different class. The class name is always the unqualified name of the error, as specified by the standard, for convenience. The error types always internally contain the fully qualified name (and present it with a default prefix when needed), as well as the description of the error condition. Depending on the case, individual error classes define constructors with argument types relevant to the particular condition they represent; these are used to automatically construct more descriptive error messages.

All errors types are indirectly derived from the `XQueryError` (abstract) class, itself extending `RuntimeException`. The intermediate abstract classes `XQueryStaticError`, `XQueryTypeError` and `XQueryDynamicError` (the last with sub-type `XQueryFunctionError`) group the errors depending on their origin in different parts of the relevant standards.

Function errors are defined in [78], while the others are defined by the XQuery standard[77] itself. The reader is reminded that: static errors are those that are detected before a query starts executing—e.g., syntax errors; dynamic errors arise during processing, i.e., they cannot be detected statically; type errors are an intermediate category: XQuery implementations that support the optional Static Typing Feature may raise them statically, those that don't must raise them at execution time. XQ2P does not support static typing for now, so type errors will always be dynamic.

XML Names

Due to its relationship with XML, identifiers used in XQuery are defined by reference to the XML Namespaces recommendation (XQ2P uses the 1.1 version[75] of the specification). XML names are strings that follow specific lexical restrictions; because these are used everywhere throughout the specification, XQ2P defines several string wrapper types that ensure well-formedness:

`NCNames` hold any string that satisfy the lexical restrictions for XML names, except those that contain colons. (The name is an derived from "no-colon name".)

---

[27] Note that we only defined the errors that the current implementation of XQ2P can raise, however. Due to the non-implemented optional features, we did not need some of the errors.

`QNames` hold all valid names for namespace-aware XML. A `QName` is formed from a local part and an optional prefix, separated by a colon; both the local part and the prefix are `NCName` instances. Note that `NCNames` and `QNames` are purely syntactic constructs, and their meaning is dependent on the namespaces that are in scope at any point they are used; it is possible for the same such name to mean different things at different points in the same document.

A `QName` is transformed into a semantic construct through the process of *expanding* it: the list of namespaces in scope is consulted to determine the namescape bound to the prefix or, for unprefixed `QNames`, the default namespace, which is associated with the local name ("local" means local to the namespace) to form an `ExpandedQName`. Strictly speaking an `ExpandedQName` is a tuple formed of an URI and a `NCName`. XQ2P however also retains the original prefix used for `ExpandedQNames` derived from a document; the prefix has no influence to the processing and evaluation of queries, except that during XML output this prefix is used if possible.[28]

The URI part of an `ExpandedQName` is an instance of `XS_anyURI`.

Note that `NCName`, `QName` and `ExpandedQName` are classes used in the implementation of *XQ2P itself*. An XQuery processed by XQ2P may also use and process *values* holding XML names, i.e., of types `xs:NCName` and `xs:QName`. Such values are held by distinct classes, `XS_NCName` and `XS_QName`, which are part of a distinct Java class hierarchy. For instance, when parsing the expression `fn:node-name(.)`, an instance of `QName` will hold the name of the `fn:node-name` non-terminal, which is then converted to an instance of `ExpandedQName` when the function is resolved; However, if the expression is later *evaluated*, and if no error is raised, it will return a (possibly empty) sequence of `xs:QName`.

XQuery parsing

For simplicity and generality, the XQuery-processing part of XQ2P accepts queries formulated in the human-readable syntax of XQuery.[29] The first step in processing such a query is parsing it into an in-memory syntactic tree.

---

[28] The XQuery standard declares the specific prefixes used during output to be implementation defined, but it encourages implementations to conserve prefixes from original data; presumably these have been chosen by humans for readability reasons.

[29] We have not made use of XQueryX, the XML syntax for XQuery[82]—which is rarely used in general, and implemented by very few commercial products— and thus we have not implemented support for it. The XML-based syntax is intended to be easy to parse, however, and support can be added very easily if needed.

As already discussed, XQ2P is intended to support (almost) the entire XQuery language; to this end, we developed a complete parser for the XQuery 1.0 language (including the few optional features XQ2P itself does not support).

We would like to stress that the parser, together with the package of classes the in-memory syntactic tree is built of, is designed to be completely separated from the rest of XQ2P. The parsing module can thus be reused in any other XQuery-related project. (Parsing XQuery is not a trivial task, and we hope this to save some time to future researchers.)

The parser The parser itself consists of about 2500 lines of JavaCC code. For the most part its structure mirrors the grammar specified in Appendix A of [77]. Only a small number of deviations were required, mainly to accommodate the extra-grammatical constraints.

Note that with respect to the `xml-version` constraint, the parser supports only XML 1.1[74] and XML Names 1.1.[75][30]

The parser relies on the Java standard library for any Unicode processing. As such, the Unicode version supported will be the same as the one in the Java library used.

A small detail of some practical importance is that the parser does not attempt to do encoding detection. Although the encoding declaration (optionally included in the version declaration of XQuery) is parsed, checked for lexical and syntactic correctness, and stored in the syntactic tree, it is otherwise completely ignored by the parser itself;[31] character set detection, even in the presence of such a declaration, is not trivial and quite beyond the scope of

[30] According to the XQuery standard, the choice to support XML 1.0 or XML 1.1, and that between XML Names and XML Names 1.1, is implementation defined. Since our decision was not motivated by any particular difference between the two—we simply picked the newest version of each—describing the differences is beyond the scope of this document. Please see the excellent summaries of and rationale for changes included in the cited standards.

[31] According to the XQuery specification, the handling of an encoding declaration is implementation-dependent; it also explicitly mentions that such a declaration may be incorect due to moving the query between environments, and that its presence or absence has no effect on the semantics of the query. (Although the latter statement, while technically true, is somewhat misleading: it is perfectly possible for a certain sequence of bytes to be parsed as several distinct but valid XQuery programs whose execution generate different results, if those bytes are transformed to characters using different character sets. However XQuery is defined in terms of Unicode characters, not bytes.)

this thesis. XQ2P uses either the platform's default charset or a user-supplied explicit one.

The syntactic tree

The result of successfully parsing a query is a syntactic tree,[32] represented with objects from the `syntax` package. This is closer to a concrete rather than an abstract syntactic tree: features like parentheses and lexical representations of literals are stored, in an attempt to support almost any usage. (Comments and non-significant whitespace, however, are not retained.)

In general, the classes in the `syntax` package are a mapping of the structure and naming patterns of the XQuery grammar. For example, the production for "if" expressions,

```
[45]  IfExpr ::= "if" "(" Expr ")"
                 "then" ExprSingle "else" ExprSingle
```

is represented by class `syntax.expr.IfExpr`, instances of which have a member `condition` of type `syntax.expr.Expr` and two member of type `syntax.expr.ExprSingle`, named `thenCase` and `elseCase`. The type `ExprSingle` is an interface, which class `IfExpr` implements, corresponding to production 32 of the grammar.[33]

The classes in general are designed to prevent, as much as possible, the construction of a parse tree that does not correspond to a correct query (or part of query). To that end, all instances are immutable, and constructors verify preconditions corresponding to the grammar constraints (for example, attempting to construct an `IfExpr` with a null `condition` will raise an exception). To make processing of such trees convenient from a Java standpoint, all members are declared `public final` rather than requiring (or even providing) accessors (collection members are all unmodifiable).

Objects corresponding to terminal symbols record the exact lexical representation that generated them—e.g., `StringLiteral`s remember if they were quote- or apostrophe-delimited, and `IntegerLiteral`s record any padding zeros—but as a convenience to the user they also have members giving them an interpretation. Integer, decimal and double literals are interpreted, respectively, as `BigInteger`, `BigDecimal` and `double` values, and string literals are given a Java `String` representation where escapes and character references in the lexical representation are replaced by their meaning.

---

[32] Specifically, an instance of `syntax.Module`.

[33] We did take a few liberties in the translation, for example with grammar structure related to operator precedence.

About names XQuery relies heavily on the XML Names[75] specification, not only for the XML elements and attributes it manipulates, but for its own identifiers, like functions and variables, including namespaces.

In the parsing tree, NCNames and QNames are represented by eponymous classes; these are simply string wrappers that verify upon construction that they are lexically correct. NCNames and QNames are syntactic elements, and they do not have a single semantic in themselves: their exact meaning depends on the context.

In XQuery processing, names are assigned meaning after parsing but before execution, during static analysis of the query: each name is assigned a namespace (possibly the empty one) depending on the namespaces in scope at the point where it occurs and its prefix (or lack thereof). The result is an expanded name. XQ2P represents expanded names as instances of `ExpandedQName`. (The prefix is also stored in the ExpandedQName, but the expanded name is only determined by the namespace and the local name.)

Static analysis

The XQuery specification defines two phases of processing: the static analysis phase and the dynamic evaluation phase. The first of these, static analysis, depends only on the query itself and the static context, not on the data that will be accessed by the query. During static analysis, the query is checked for correctness,[34] names, prefixes and types are resolved, and the operation tree is built.

Each of the various steps of static analysis can be performed separately, but XQ2P executes them all in a single pass over the parsed representation of the query.

During static analysis, the static context of each expression is determined. The static context contains all information available to the expression, for example in which namespaces and variables are in scope, or defaults that are to be used for operations that do not explicitly mention some option, e.g., collations. XQ2P holds the static context in an instance of `xquery.context.StaticContext`, which have getter and setter methods for each of the individual components of the static context.

The entire logic of static analysis is embodied in class `StaticAnalyzer` of package `xquery.core`. This class contains one method for each of the classes

[34] This means checking all the conditions that may rise a static error, with the exception of parsing errors. If a parsing error is encountered, query processing stops before the static analysis phase.

in `xquery.syntax`, named `analyze`⟨class name⟩. Each such method takes as argument an instance of its associated class plus an instance of `StaticContext`.

Static analysis is initiated by passing the entire query (an instance of `xquery.syntax.Module`), together with a static context instance freshly-initialized with the default, to the `analyzeModule` method. Each method in `StaticAnalyzer`, including `analyzeModule`, will examine the syntax (sub-)tree it was passed, alter the static context when necessary, call the appropriate methods to analyze any sub-expressions it is composed of (passing them the altered static context) and receives from them the operation sub-tree that implements them, then, assuming all is well, assembles from the results the operation tree corresponding to the syntax (sub-)tree it analyzed. The static analyzer thus performs a depth-first traversal of the syntax tree, constructing the static context of each expression when descending and building the operator tree that will execute the query while ascending.[35]

If a static error is detected during static analysis, the method that found it will throw the corresponding Java exception (see page 50). The exception will propagate backwards through the chain of method invocations and stop the processing of the query.

The somewhat complex structure of `StaticAnalyzer` (basically, one method for each syntax element, all in one[36] object), though needing a bit of care on our part during development, has the favorable side effect of enabling relatively easy modification via class derivation: The mentioned `xquery.core.StaticAnalyzer` itself builds an unoptimized operator tree, using the basic operators (see *Architecture*)(see page 39)). Adding optimized operators for some operations (or other optimizations) can be achieved by subclassing `StaticAnalyzer` and replacing only the methods corresponding to the relevant sub-part of the query features. (Alternatively, a similar pattern can be used on the resulting operator tree, which has a slightly simpler structure.)

---

[35]This system resembles somewhat the Visitor pattern; however, the methods of `StaticAnalyzer` perform the functions of both `accept` and `visit` without cooperation from the syntax tree. The `public final` members of the immutable syntax tree elements are intended to ease this process.

[36]This description is somewhat simplified; in practice, the many methods used are partitioned between several sub-classes, which provides some organization. For example, methods related to the prologue are separated from those that analyze expressions.

Processing Operators

The result of static processing is an operator tree. Operators are all Java objects with a single `evaluate` method; the method takes as its single argument an instance of `DynamicContext` and returns a `Sequence`.

During the construction of the operator tree each operator extracts from the static context any values it will need for execution; any subexpressions are also fully bound by this time. Operator trees are immutable, and they can be used to execute the same query several times, even concurrently. (In principle, the operator tree could be serialized and executed at a latter date, though we have not implemented this.)

XQuery execution is started by passing a `DynamicContext` to the root operator. Each operator may read values from the dynamic context, evaluate subexpressions, perhaps passing them a modified dynamic context, and perform some computation. In all cases, either an `XQueryDynamicError` is raised or a result sequence is returned.

Operator evaluation in general is short-circuited in the presence of exceptions, but otherwise all subexpressions are evaluated fully before their parent expression is evaluated. This is necessary to detect errors as fast as possible; given that XQ2P lacks static type analysis we considered the loss of performance an acceptable trade-off for a research engine.

In the simplest case, the operator tree is almost isomorphic with the syntax tree. The exception is that syntax-only elements are not translated. For example, parentheses are not needed to specify evaluation order because subexpressions are bound, so parenthesized expressions are not present in the operator tree.

However, structural optimizations can be implemented by traversing the initial operator tree and constructing a different operator tree. Several structural passes can be executed efficiently in series: because each sub-tree is immutable, parts of the tree that are not modified by an optimization pass can simply be linked to the new tree. This kind of transformation is used to substitute optimized TwigStack-based operators in place of the standard ones.

Note that several XQuery syntactic features are defined in terms of functions (see page 57, below) in the standard library. For example, arithmetic and logical operators are defined in terms of operator functions. We do *not* represent them as such in the operator tree, but use separate operators to maintain a more obvious relationship between the operator tree and the syntactic one. (The operators are *implemented* in terms of such functions, however. A tree-transformation step can be added if needed to make this explicit.)

Functions

Function calls are implemented by a separate operator. Only the (qualified) name of the function is bound; the actual implementation is obtained from the dynamic context at evaluation time.

User-defined functions are implemented simply by constructing the operator tree of their body expressions. When evaluating a function call, the dynamic context is augmented by the parameters, and the function's operator tree is evaluated as if it were a separate XQuery instance.

However, functions can also be implemented in Java; such functions may be added to the initial dynamic context before evaluation to make the available during XQuery execution. The entire XQuery 1.0 and XPath 2.0 Functions and Operators library[78] is implemented in this manner and included in the default environment. The time series extensions we describe in also make use of this feature.

Performance Considerations

XQ2P is intended primarily as a research engine. As such, we avoided most optimizations, preferring to focus on simplicity and obviousness of code style. We attempted, instead, to make it easy to add such optimizations in the future. For example, most user-visible structures are immutable, which allows the implementation of many parallelization techniques; interfaces are generic whenever possible, to facilitate replacement of individual components; and the operator tree structure is designed to facilitate most query plan transformations.

We make use of these facilities ourselves to construct optimized operators that implement structural join algorithms and substitute them into the basic query plans, as well as for implementing the subset of XQuery 3.0 windowed `for` syntax we added as an extension for time-series processing. .

3.5 P2PTester

As we argue in §2.4, the development of peer-to-peer applications benefits from —requires, even—simulations and tests throughout the entire process.

Before commencing this work we anticipated that a significant difficulty in our investigation would be the accurate characterization of the performance of different solutions to the problems to be encountered. We decided that theoretical analysis of the algorithms would not be sufficient, and that simulation and testing would be necessary.

We reviewed the available options, and decided that none of the existing systems for P2P simulation and testing were completely appropriate for our needs. We resolved to implement a new framework dedicated to this purpose. The first steps have been done as a Master's project[14]; the complete system was presented in [15] and [13].

Our goal was to develop a system with the following features:

- it should allow testing of any kind of peer-to-peer network, without being limited, e.g., to DHT systems, or even only to structured ones;
- it should include a generic event logging and recording and tools for visualization and statistic analysis of logs and test results;
- it should provide a modular architecture from building peer-to-peer applications from simpler modules, and pre-built modules for basic operations like networking, storage and even DHT;
- test scale should easily scale from simple, in-process simulations to large scale tests on many machines simultaneously, over a real physical network;
- the realism of the physical transport underlying simulations should be selectable on a run-by-run basis, and range from simple but fast in-memory inter-process communication, to programmatic simulation of network conditions, to using an actual physical network;
- the type of scenario executed should not be limited arbitrarily; fixed scenarios, recorded traces, programmed scenarios and interactive activity should all be supported;
- developed applications should not be dependent on the tester; once testing is complete, the application should be usable outside testing by just changing the command line.

We believe to have achieved all these goals.

The decision to have complete generality of the testing system, both for the type of peer-to-peer applications tested and for the test scenarios executed, led quite naturally to our splitting the framework along the two conceptual divisions, application/tester and tester/scenario. We present each of these in turn:

Applications

To allow the development of any kind of peer-to-peer network, and in particular to allow running those applications separately from the tester infrastructure, we do not impose any kind of precise structure to the tested application. As such, the user of P2PTester will simply build his or her application as any Java program.

However, during testing, the test framework—in particular, the test scenarios, as will be explained later—needs to control the detailed behavior of the application. Recall that running a test consists, in general, of: (i) instantiating several copies of the tested application, one for each network node that is simulated; (ii) instructing each node to perform certain operations at certain times, for example connecting, or publishing some data. This is accomplished by a pair of interfaces, one for controlling and one for instantiating the tested code. A third category of interfaces is provided for the purpose of logging, which will be described separately. The logical interfaces are implemented by sets of Java interfaces.

Instantiating peer nodes: Each test scenario needs to control the creation of peer nodes, in various detail. For example, scenarios that test the scaling abilities of a peer-to-peer algorithm might instantiate varying numbers of peers and compare the resulting behaviors; others might vary the order and frequency with which peers join the overlay.

At the same time, for each class of applications (peer-to-peer ones being a particular case) a specific set of objects—the set of instantiation parameters—is needed to make a new instance. In our case, some of these are the same for all instances used in a test: for example, a class implementing a KBR algorithm can, in general, use any number of network modules to communicate between peers; during a test, however, all instances tested will necessarily use the same module; the framework must thus allow arguments to be provided to each node at instantiation without involvement from the test scenario, but under the control of the test framework. Some other parameters must be controlled by the scenario: for example, the scenario might need the nodes to simulate different abilities. Still others might be independent of testing, and thus need to be provided without the involvement of the test framework.

The basic method in the Java language to specify initialization is constructors. However, constructors may not be declared for an interface, and are thus inadequate[37] for our purpose. We decided to use the *factory* design pattern to solve this problem. On the one hand, the application researcher must write a factory class with methods to instantiate a node of their application; these methods define via their argument lists the kinds of objects appropriate to their case. On the other hand, each test scenario writer must write the interface for the factory their scenario needs; the argument lists of the factory methods will define thus which instantiation parameters the scenario is willing (and able) to define. Observe that if the factory class for an application matches the factory

[37]That is not to say using them would be impossible, only that it would be more difficult than their worth.

interface for a test scenario, then that test scenario is applicable to the application (i.e., it makes sense and it is possible to test the application with that scenario).

It may be the case, however, that a specific test might not need to control some particular argument for its applications;[38] the user of P2PTester might want to supply such an argument for different test runs of the same scenario, or might need to supply the same value for different scenarios. In these cases, an *adapter factory* must be written; this is simply a class that implements the interface required by the scenario, and provides the extra arguments the application needs. In the case where some application arguments are different for each test run (but identical within a test), an extra adapter must be used to allow supplying arguments via the test harness at test execution time.

It is intended and expected that some generic factory interfaces will be developed, to be used by test scenarios that apply to many different kinds of peer-to-peer applications. Indeed, some such interfaces and several commonly used adapters (e.g., for abstracting away the networking layer) are provided, ready to use, in P2PTester's library.

Controlling peer nodes: Each type of peer-to-peer application offers several types of operations it can execute. For example, every KBR application, basically, can locate (alternatively, route a message to) the "owner" of a key. Even simpler, for all peer-to-peer application the operations for joining and leaving an overlay have in general the same signature. Each test scenario needs to control those aspects of the application that it tests; it is important to observe, though, that some generic tests will be applicable to different applications implementing the same features. For example, a scenario intended to measure how many messages are needed to connect to the overlay network will be applicable to any kind of application; similarly, a scenario used for comparing the efficiency of KBR algorithms should in general work with any kind of KBR implementation. This problem is most naturally solved by a hierarchy of Java interfaces, interface inheritance providing the required ability of writing a scenario at the necessary level of generality and executing it on a particular, perhaps more feature-full, implementation of the concept.

As it may be apparent from the example above, all interfaces that define application functionality are derived from the base interface `Peer`, which declares the fundamental ability to take part in peer-to-peer networking. On further reflection, this is not really necessary; some very specific kinds of P2P networks might possibly require a different connection model. We have never needed

[38] Indeed, for some parameters specific to an application, a sufficiently generic test might not be *able* to provide an initial value.

```
/** Chord algorithm with customizable hashing. */
class Chord<V extends Serializable>
  implements KBR<byte[], V>  {
    public Chord(MessagePipe p, MessageDigest
       hasher){
      // ...
  }
}
/** Generic DHT-over-KBR implementation. */
class StringDHT implements DHT<String,String> {
  public <K> StringDHT(KBR<K,String> adapted,
    Converter<String, K> adapter){
      // ...
  }
}

new Factory<DHT<String,String>> {
  DHT<String,String> construct(FactoryCallback c)
     {
    return new StringDHT(
      new Chord<String>(c.getMessagePipe(),
         MessageDigest.getInstance("SHA-1")),
      new Converter<String, byte[]>(){
        public byte[] convert(String s){
          return s.getBytes("UTF-8");
        }
      }
    );
  }
}
```

Figure 3.2: An example of adapter factory. Abbreviated versions of interfaces `KBR` and `DHT` are shown in Figure 3.3. `Chord` would implement the Chord algorithm for generic kinds of values and an arbitrary hashing function, and `StringDHT` would implement a DHT with `String` keys and values given a KBR implementation and a converter from strings to the KBR's type of keys. At the bottom is an example of an anonymous factory class that combines the above, using standard Chord hashing.

```
/** Base interface for peer-to-peer nodes. */
interface Peer {
  /** Create a new overlay. */
  void init();
  /** Join an existing overlay. */
  void join(PeerAddress entryPoint);
  /** Leave the overlay. */
  void leave();
  /** Obtain this 'nodes address. */
  PeerAddress getAddress();
}
/** An interface for implementations of key-based
   routing. */
interface KBR<K,M> extends Peer {
  /** Route a message to the peer owning a certain
      key. */
  void routeToKey(K key, M message);
}
/** An interface for DHT implementations. */
interface DHT<K,V> extends Peer {
  /** Store a key-value pair in the overlay. */
  void put(K key, V value);
  /** Retrieve a value from the overlay. */
  V get(K key);
}
```

Figure 3.3: Abbreviated versions of some standard service interfaces provided by P2PTester.

this, so our initial decision stands in present code; that said, the modifications required to remove the limitation are not extensive, and may be modified in future versions of the framework.[39]

It is intended that the same interface used by the test scenario to control the peers be used to connect the core of the developed application with a user interface. Since this separation between functionality and UI does not impose any significant restrictions on the application researcher—indeed, it is a recommended and common coding practice—we believe it to be a satisfactory solution[40] to the requirement that the tested application be as close as possible to the real one. An useful side-effect is that a UI written to such an interface can be used with no modification to control *different* P2P applications that share functionality.

The elements described above are the only real constraints on how a researcher must build their application to allow its testing with P2PTester. The framework provides several other modules intended to be used when building the application, but their use is optional.

Event Logging

The P2PTester framework provides a facility for logging events happening inside the tested application during a scenario. Several types of events—notably, the passing of messages between peers, explained later in more detail—are recorded automatically, but an interface to the logging facility is also exposed to the writer of test scenarios and to tested applications. This facility is intended to allow the researcher to examine in detail the internal behavior of a particular application. For example, a node in any DHT system has the ability to publish a key-value pair, but different DHT systems may accomplish

[39] A word of caution for prospective researchers on the subject: While designing pre-built components for users of the framework, the architecture also included such inheritance relations between very low level node types, like DHT, and higher level ones that build on them, e.g., an XML database that uses a DHT as a primitive operation. This might seem like a good idea: a module that makes use of another low-level P2P primitive *can* expose it with apparent ease. That said, it is not obvious why it *should* do that. We found such exposure to be more trouble than its worth; if a higher level module needs to share access to a low level module with an intermediate one, coordination is easier the higher level one is in control. We have refactored it out of some of the pre-built interfaces modules, but the user of the framework might still find traces of it in the module libraries.

[40] The alternative, namely to have the test framework control the UI of the application rather than in its internal functionality, is much more difficult and outside the scope of this research.

this by passing through different phases (e.g., the publisher might first identify the peer responsible for the key, then sending the key-value pair to that peer; or, alternatively, it may route a message with key-value pair directly through the overlay); logging the individual phases of an operation allows the application developer to analyze them separately. However, using this facility is completely optional: the interface passed to the peer may be ignored.

The interface to the event logging facility is designed to be very simple: a single method that accepts `Event` objects as arguments. The framework provides several implementations for this interface, allowing the user to chose one at scenario run time. Each uses a different technique for recording events, each influencing the performance of simulations differently. For example, an logger that serializes events to disc is necessary for test runs that will generate large numbers of events, but will introduce random delays while flushing to disc; an in-memory logger generally avoids this assuming the number of events is low enough relative to the memory available on the test hosts. A "dummy" logger is provided which ignores logged messages, intended for cases where event logging is unnecessary.[41] The advanced user is also offered the opportunity to register one or more *filter callbacks* with the logging facility; these filters are called when an event is logged and allow or deny the logging of events depending on any of their features.

The logging facility accepts as "events" any object of class `Event` of `utils.events`. The base class of `Event` (sometimes conspiring with the particular event logger used) automatically records basic information including a timestamp[42] for the event and the identity of the peer that generated it, and a set of tags (to be explained below).

Several derived classes are provided for common types of events, and the user of P2PTester is encouraged used derived classes that record events interesting to the particular application that is developed. There are no explicit limits to what kinds of information may be recorded in an event, other than that it be serializable; that said, logging to many events or events with much data attached can lower performance of test runs, and some experience and judicious application of filters may be necessary.

---

[41] In particular, this may be used after testing, to simply disable event logging in the stand-alone application. A developer concerned with performance may also use the Java trick of wrapping event generation in a `true`-returning function called from an `assert` statement, which will completely disable execution of any logging.

[42] The timestamps are generated based on the clock of the machine where the `Event` object was instantiated. Some care should be exercised when comparing timestamps for tests ran over a network, since P2PTester does not attempt to synchronize the clocks of host machines.

Event tagging A related feature is *tagging* of events. Due to the distributed and asynchronous nature of peer-to-peer networks, it is often difficult to determine which of the events in a log trace are caused by the execution of a particular command. For example, during the execution of a "publish" operation, a peer node might also receive and act on unrelated messages from the other peers, and thus the log might contain interspersed events caused by different operations. To aid in disambiguating the cause of events, the base class of `Event` accepts, via its constructors, a set of "tag" objects.

Other than implementing the interface `Watermark`, these tags may contain any kind of information, but they should be in general small, immutable and serializable, due to their use.

An application wishing to make use of this system must accept for each operation it exposes, in addition to those arguments necessary for the operation, a set of such tags, propagate them throughout[43] its internal processes, and attach them to any events generated *in order to complete that operation*. Once this is done, the writer of test scenarios may add different tags when peers are instructed to execute different operations, and then use them to identify the event trace of each operation from the event log. These tags can also be used to filter which events should be logged (see the description of the logging facility, above).

In order to make the use of this feature optional, the peer-to-peer interfaces that are included with the P2PTester framework receive the tags via a variable-length argument list. In this way, implementations that do not wish to make use of the facility may simply ignore the extra arguments, and the writer of test scenarios can add or remove any number of tags to different operations with ease. Users of the framework are encouraged to follow this practice if they need to establish their own application interfaces.

Messaging Layer

For any practical peer-to-peer application, the network layer—the part of the application that handles communication over the physical network—requires significant attention from the developer, despite the fact that usually the focus of the research is considerably higher conceptually. In our particular case, the requirements for P2PTester impose several constraints:

On one hand, while developing a peer-to-peer system, as well as when comparing the algorithmic, network-independent behavior of different systems, it is useful to avoid actually communicating over a network, given the inherent

[43] This may be quite complicated, depending on the application. Unfortunately, it is impossible in general to do it automatically.

delays; instead, an in-memory simulation where communication is achieved by passing objects directly between the processes representing each simulated peer node can be much faster, thus allowing many test iterations. It also allows comparing the algorithms directly, without confounding factors inherent to network access, e.g., serialization performance.

On the other hand, testing in a real environment, with messages traveling over a physical network, can also provide invaluable information about the real-world behavior and performance of an application. Separately from the performance aspect, the truly parallel processes of nodes distributed on separate hosts, together with the vastly different latencies and bandwidth of a physical network, can often reveal bugs and anomalies that simulation within a single virtual machine may hide.

In addition to the various kinds of tests, we have also intended to allow transforming systems built with P2PTester from simulation prototypes to real applications with as little development effort as possible.

We chose to use the same approach we used for the relationship between test scenarios and tested applications, described above. A generic interface, `MessagePipe`, describes the functionality of a generic communications endpoint. Different implementations of this interface are provided, each using a different method for actually passing messages between peers; researchers may add other implementations, more suited to particular needs, which can be then used for any and all applications if needed. Each implementation provides to the testing framework a factory object, and the user may then choose, on a test-by-test basis, which network module to use, independently from the test scenario and the tested application.

The `MessagePipe` Interface: When deciding on an API intended to be used by many different applications, developed by different researchers, several choices must be made; some very different trade-offs between ease of use, simplicity, performance, and compatibility with network technologies are possible. In addition to the technical choices, different users may prefer, and different algorithms may be more amenable, to different models of communication.

We attempted to offer the maximum flexibility to the user with a two-pronged approach: We chose a low-level interface that expresses the simplest and most general communication model, designed to allow maximum optimization opportunities for the implementation. For ease of use, we provide *adapters*: classes that offer alternative, conceptually-simpler idioms built on the low-level API, allowing researchers to use the model most appropriate to their needs.

Our communication API is expressed by the `MessagePipe` interface.

A `MessagePipe` represents an asynchronous communication endpoint; each peer nodes receives a separate instance of it when instantiated. At creation time, the pipe is in the "closed" state. The owner can "open" the pipe for communication by registering a call-back handler to receive messages asynchronously. Message sending is also asynchronous: the `send` method will only confirm sending the message via a call-back method. Note that `MessagePipe` confirmations and errors are not guaranteed, and their semantics are best-effort: a confirmation means only that the pipe finished its work and did not detect an error; a message may be lost or discarded by the recipient; similarly, an error callback means that something unexpected went wrong, not that the message is certain not to have reached its destination.

A pipe can be "closed" at the application's choice;[44] it can be reopened and closed any number of times.

Each open `MessagePipe` has its own address. This is an abstract object, depending on the exact implementation of the network layer in use; as such, in only exposes very limited internal information, in an attempt to guarantee that the user application remains independent of its particularities.[45] The address is returned to the peer only when the pipe is opened—in fact, an endpoint does not have an address while in the "closed" state, and it may have different addresses if closed and re-opened.

Note that the `PeerAddress` encapsulates all the information needed to communicate with the *peer*, which may be different from the physical address of the *host* it runs on. For example, in a TCP/IP–based implementation of the messaging API, the `PeerAddress` will typically encapsulate *both* the IP address of the host machine *and* the TCP port the endpoint is listening on for messages. Applications are intended to use them as opaque objects.

Message content The messaging API does not constrain the type of messages transmitted. Each kind of application may simply send any kind of objects it requires, with no restriction other than it be serializable using Java's standard mechanism.

---

[44] The various implementations of the messaging API may also expose extra functionality to the test scenario; this may be used, for example, to allow the scenario to simulate changing network conditions.

[45] We attempted to use Java's generic type system to expose partially the type of addressing: Both the `MessagePipe` and `Peer` interfaces have a type parameter based on `PeerAddress`, allowing implementations to pick a refined kind of address they may work with. This may allow, for example, to write a P2P application that works only with IPv4 or IPv6 addresses. This system proved cumbersome, and will probably be refactored out.

```
public interface MessagePipe
    <AddressTYPE extends PeerAddress> {
  /** Callback for received messages. */
  public static interface Listener {
    /** Called when a message is received. */
    void messageReceived(
            Serializable message,
            Set<Watermark> marks
        );
  }

  /** Callback for send operations. */
  public static interface Callback {
    /** Called when the message was sent. */
    void messageSent();

    /** Called if an error was encountered. */
    void sendFailed(Exception cause);
  }

  /** Puts the pipe in a wait-for-connections
     state. */
  public AddressTYPE openPipe(
          MessagePipe.Listener listener
      );

  /** Closes the listening port. */
  public void closePipe();
}
```

Figure 3.4: The `MessagePipe` interface, abbreviated. (Declared exceptions and detailed comments have been omitted.)

In addition to the content, the MessagePipe API allows each message to be additionally accompanied by a set of `Watermarks`. This is intended to support the event logging and tagging system, described earlier. We decided to include the tags explicitly in the interface—rather than have the application itself attach it to the content of the message, if it needs to—for pragmatic reasons: This allows the framework to couple the messaging layer to the event-logging system, and log events related to sending and receiving messages, complete with their relevant tags, automatically.[46]

Several implementations of the `MessagePipe` interface are provided to the framework's user:

MessagePipeLocal provides simple and very fast in-process communication. It is intended to be used for tests where all peer nodes execute within the bounds of a single Java virtual machine. Messages are passed directly between the threads of various peers; no time is spent interacting with the machine's networking infrastructure, nor marshaling and unmarshaling objects.

MessagePipeSocket uses TCP/IP sockets for sending messages between peers. (An alternate implementation `MessagePipeChannel` is available, the main difference being that the latter uses the communication primitives of `java.nio`.) This class is necessary for running tests over networks; tests run using it will naturally reflect the performance of the underlying physical network. (However, it is possible to use this class also for in-process tests. In that case, messages will cross the entire network stack, which allows simulations closer to real conditions, but by not sending messages over the wire avoids being limited by the bandwidth and latency of the network.)

Wrappers A pair of "wrappers" are also provided. These take a `MessagePipe` instance as an argument, and implement the same interface by delegating all functionality to this wrapped instance; at the same time they perform an additional function: `MessagePipeTracer` will print a message each time a message is sent or received, and `MessagePipeLogWrapper` will automatically log `Events` using the event logging system described above. The former is intended for very simple debugging, while the latter can be used by the framework to log automatically all events related to messaging. The same pattern may be employed by users needing similar behaviors.[47]

[46] We may revise this decision for future versions of the framework.

[47] For example, such as "intercepting" wrapper may delay or drop messages to simulate network failures, display simulation activity in real time, or even to allow manual

The Tester and Test Scenarios

The last major component of the P2PTester framework is the tester itself: the programs that are executed by the tester's user, and which host and control the processes of the tested application and the test scenario.

P2PTester was designed from the start with the intent to allow the execution of tests on multiple machines. Single-machine tests are very important, too, especially in the first stages of application development: tests are simpler to run on a single machine, and the first stages of algorithm development usually need many low-scale, quick tests. However, since we considered execution on many machines very important, we decided to focus on that use case from the start; running tests on a single machine is only a special case of the distributed version.[48]

During a test run, from the point of view of the tester's user there are two types of processes running: The first is the tested P2P algorithm; there are as many P2P application nodes as needed, each of which is a separate process. The second is the test scenario, which is a single, separate process (though it may be threaded) which communicates with (indeed, controls) the application nodes.

In the most general case each such process runs in a different Java VM and communicates only through sockets with the others. However, it is possible to run several such processes in the same VM (and even sharing some objects), depending on the tester's configuration, trading some reliability and realism for efficiency. This is always done transparently (i.e., objects are shared only when they would have been exported through RMI) so that logically each process can be considered to run in its separate memory space. The intent is to run the several logical processes in separate JVMs (and separate physical machines) when we need precise timing measurements and in a single JVM when we are interested in evaluating algorithmic complexity (e.g., to measure only the number of messages exchanged) or verifying correctness of behavior. Usually, the number of messages exchanged is more important than how long it took to send them, since the later depends on variable network conditions, and the former on algorithm performance. However, for applications that take advantage of network locality (physical closeness between nodes) for caching and routing purposes, measuring behavior when communicating over a physical

---

inspection and control of messages *en route*. We envisaged adding each of these to P2PTester; they have been left for future development due to time limitations and the fact that they were not needed for this work.

[48] We did make a few single-machine optimizations, though, mentioned later.

network is important.[49]

Any test system contains two objects belonging to the tester: a `MasterTester` and at least one `SlaveTester`. There is always one `SlaveTester` for each physical machine that hosts the test run, and exactly one `MasterTester` for the entire test. The `SlaveTester` instantiates and hosts the processes for each P2P-application node, as directed by the `MasterTester`. The master hosts and controls the process of the test scenario, and coordinates the slave testers. The number of slaves and the distribution of application nodes (peers) on these slaves is hidden from the test scenario; the latter acts as if all the peers are directly accessible. Java RMI is used to connect the test scenario to the peers.

The `MasterTester` and `SlaveTester` objects are not stand-alone applications. In order to use them, a "helper" application is used; this application initializes the test environment by starting a master and/or a slave, connects them, and then starts a test run with a test scenario and an P2P application class chosen by the user.

We provide several such helper applications, which can be used immediately; a user with special requirements can modify or write an entirely different one to satisfy any particular test environment configuration.

Running distributed and local tests

For local tests (those tests that are meant to run on a single machine), the helper application starts a `MasterTester` and a `SlaveTester` in the same virtual machine and connects them directly (by passing each one references to the other). We have two versions: The first one uses a GUI with a menu system to allow the user to pick a test configuration (that is, which P2P application is tested, which test scenario is run, and what communication layer is used). Another starts the test directly, and changing it requires changing the source code of the set-up section; this is much simpler and quicker to use for incremental or automatic testing, both when using an IDE like Eclipse and when only a command-line is available. The last is also command-line-based, but it allows using command-line parameters to start a test, useful for scripting a set of test runs.

When running a distributed test, another factor is added: the master tester is started the same way as in the local case, but with a special flag set which instructs the helper application to wait for user input before starting the test run.

[49] It is also possible to estimate the effectiveness of such algorithms using a `MessagePipe` implementation that simulates various network conditions. We do not yet provide this, but it can be added by the user according to their needs.

On the other machines that participate in the test, a separate helper application is run. The latter starts a single `SlaveTester` and instructs it to connect to the (remote) master; the address of the master must be communicated through the command line. When slaves are started this way on all machines, the user (or a script) must tell the master to start the test, which is then run distributed on all machines.

In the cases described above, there is one Java VM running on each physical machine that participates in the test. There is another option; a separate JVM can be started for each separate process: the master, each slave, the test scenario, and each node of the tested P2P application runs in a separate JVM. This technique is useful when it is necessary to physically separate every process, for instance when the P2P application is still unstable and crashing nodes can stop the entire test run, or when peers cannot be reliably stopped from within the Java VM. (Rogue threads, for instance, might be refusing to stop. The separation of JVMs allows turning off the process directly by forcing its JVM to stop.) Other uses could include isolating memory-usage issues, or running applications that make use of `static` fields in an unsafe way. This method of execution, however useful, tends to be very slow, because of the added overhead of each JVM; because of this it is not active by default, but can be activated by users who need it. (Some JVMs have optimizations that avoid much of the VM launchin overhead after the first VM is launched, which is likely to reduce such performance problems.)

Customizing and extending the tester A researcher with a special configuration of the test environment might need to change the tester infrastructure to suit his needs. This is not a difficult task, but reading and understanding the code of the tester objects is required, which could take some time.

There are two main options for extending the tester: First, the "helper applications" that start, configure and connect the master and slave testers can be modified. This can be useful for complicated testing environments, e.g., Internet-distributed hosts or firewalls. In such cases a helper application might search for the testers available or open connections through firewalls and gateways. This is relatively easy to do, because the user does not need to know many details about the internal workings of the tester.

Second, the user can modify the tester itself. An example might be replacing the Java RMI-based connections between testers (and, thus, between the peer nodes and the test scenario) with another specialized messaging system, particular to their testing environment, which runs over a different communication channel than the one used by the tested application itself. This allows to minimize the influence of the tester communication overhead over measure-

ments of the P2P application behavior. (As an example, certain clusters built specifically for testing have two parallel network interfaces (or more), one of which could be used by the tested application and the other by the tester.)

Performance considerations and limitations

This section describes the limitations of the tester platform, including both those we encountered or those we anticipate.

Memory The tester, by itself, does not use a lot of memory. Depending on the JVM used (and the libraries loaded by default), the memory usage of the master and slave processes can be well below 32MB, including the JVM. Running each process in a separate JVM can quickly occupy a lot more, though, unless there are optimizations for sharing common parts of code, which are available in some newer JVM implementations.

On the other hand, each node of the tested P2P application can use up lots of memory. Especially in the cases we investigated—applications that index XML documents—it easily possible to fill up to a GB of memory with data and indexes, even with only a dozen logical nodes on every testing machine. Applications are usually built with the intent of running as a single-instance; using hundreds of megabytes of RAM, especially for something like a distributed XML database, is not unusual. It is easy to reach the limits of most machines with only a few instances of such an application.

The best solution, after increasing the memory as much as possible, is running tests on several machines, which is of course one of the reasons we developed the tester.

Processor Most P2P applications are not particularly processor-intensive. Except for some tasks, like hashing and local indexing, the algorithms depend on the network and the overlay structure for performance.

However, when running many nodes on a single machine tests can become CPU-bound. One particular case is when direct process-to-process communication is employed (i.e., messages are passed directly between nodes, as Java objects, instead of being sent over the network). This almost removes the communication cost, thus allowing a test scenario to be run at the limit of the available processing power. Modern machines with multiple processing cores and multi-threading can help, but will at best provide an order of magnitude increase in the number of threads supported before performance becomes a problem.

Remember that for CPU-bound tests, any timing measurements are not very relevant for the behavior of the application in real conditions. Analysis of

algorithmic complexity, based on the number of messages exchanged, is usually more interesting.

Hard-drive performance The performance of the drives used can easily become more important than that of the processor when running many peers on the same machine. For example, if several peers that run on a single machine are simultaneously indexing large XML documents, it is very likely that the hard-drive's throughput be saturated.

Another important issue here is the logging. Logs can easily become hard-drive limited, considering that all messages are recorded (and twice! once for the sender and once for the receiver, though sometimes this can be optimized-away). It is recommended that large logs be written to a different physical drive, or even to a fast RAID array, separate from the drives used by the tested P2P application.

(Of course, this is significant mostly for cases where precise timings are measured. If only number-of-messages are used for evaluating the application, the only concern is how long it takes to run a test.)

Another optimization, also useful for test runs where the user is interested in time of execution, not in analyzing the exchange of messages, is to disable the logging of messages altogether, and log only interesting events like asking/answering of a query, or starting/finishing an indexing operation.

When possible, it is usually best to strictly filter the events logged and use an in-memory log.

Networking Lastly, but the most important for our subject—testing P2P applications—are networking performance issues.

Certainly, most tested applications will use the network intensely. Some usage patterns—for example, a simple query on well-indexed data, with few results—will transfer only small amounts of data. In these cases latency is expected to be the most important factor for performance. However, this is not always true: First, test scenarios typically would run many operations in parallel, as fast as possible, which can easily overcharge a small testing network with many peers running on a single machine. Second, some operations, for instance indexing of XML documents, can create a temporary flood of information through the network.[50]

[50] In real-world usage, the distributed nature of such applications and human usage patterns cause such "floods" to be dispersed both temporarily and spatially. Except for very large-scale distributed tests, however, they are concentrated by the need to run the test on the machines available to the user, such as a local network or even a single machine.

Another, even more important, issue is linked to other network-related resources besides bandwidth and latency. Peer-to-peer applications commonly exhibit a very particular communication pattern: first, they open a relatively large number of connections (dozens to around a hundred) which draw their "neighbors" in the logical overlay of the network. These connections occasionally change as nodes and data are added and removed, but their order of magnitude is mostly constant. This order of magnitude is usually proportional to a slow-growing sub-linear function of the number of peers in the network, for instance a logarithm or the square root. Second, during active operation (e.g., querying and indexing) the peers need to communicate with many other nodes, often exchanging a few messages before finding the address of another, and so on.

An unexpected problem caused by these communication patterns is that it is very easy to simply exhaust the ports available, or at least severely confuse the OS. There are 2^{16} ports for TCP/IP, but only not all of these can be used for opening random connections. When many nodes are run on a single connection[51] it is surprisingly easy to reach a point where connections cannot be opened reliably. Experimentally, we have encountered this situation from between 50 to 200 tested nodes on a machine, much quicker than we had expected.[52]

We commonly use direct (not over-the-network) communications when running large tests on a single machine, which avoids this problem and also makes tests much quicker (an order of magnitude, or more, is not unusual). However that technique is not available on multi-machine tests. We believe some optimizations are necessary to layers when scaling (complex) test scenarios above a few hundred application nodes.

Test scale Related to the above problem is the fact that peers running on a different machine than the `MasterTester` must be connected to the test scenario through the network. We use RMI over sockets for this, as it is a well-optimized system, powerful, well-known and relatively easy to implement.

It has a problem, though: it seems to be not very robust in larger-scale (over one hundred nodes on one machine), intense scenarios, with many peers working in parallel. We believe socket-deprivation is one of the reasons, combined

[51] Note that two ports are used for each logical "link": one belongs to the sender and one to the receiver.

[52] We have struggled at one point diagnosing inexplicable failures to open connections, which were traced to what appeared to be bugs in the `native` code parts of the JVM used that dealt with opening connections. We are not overly fond of the memory.

perhaps with synchronization bugs. Sun's implementation of RMI also has an annoying habit of succumbing to strange bugs when the network set-up is less than ideal. For instance, we have repeatedly observed it breaking when a DNS server was unavailable or misconfigured, despite the fact that only numeric IP addresses were used.

On the other hand, readers must surely have noted that the distributed testing environment we present is a very traditional client-server, centralized system. Certainly, such a system will fall behind the tested application in scaling ability. It may seem counter-intuitive, but we consider that this is preferable: the alternative would have been to build a complex, smart P2P system to handle tests of... other P2P systems. That solution might scale a bit better, but there would still be a difference between the scalability of the tester and of the tested applications (otherwise, we would not need a tester), at the added cost of complexity and the risk of hard-to-discover interactions between the two parallel P2P networks.[53]

We decided that simplicity is the best bet, and using brute force will work best at the beginning. We intend to use a relatively small number of testing machines, hundreds at most, and to test only up to a few thousand nodes. This should be enough for a huge part of developing a P2P application, on one hand, and more than this is impossible or very difficult to do in a generic way. However, an advantage of the simple design is that someone who needs more can extend and adapt the master/slave design to something more appropriate to their needs.

An extension that we anticipate we will use, and which might be generic enough to be useful to others, is running the equivalent of a `MasterTester` on each participating machine, or one for every few slaves, each master running a copy of the "test scenario" process. These parallel test scenarios can then communicate and inter-link the networks they each command. This is actually possible without any changes to the tester: a user needs only write a test scenario that "cheats" and communicates directly with its "clones", without specific support from the framework.

Other resources A less specific problem we encountered when using the tester is that P2P applications (or rather, their developers) often assume that the applications are run individually, one per machine. This can lead to diffi-

[53] That said, there are situations where this might be useful. The test infrastructure can consist of two parallel networks (using two network adapters for each machines), for example. The tester framework could use one of these while the tested application uses the other. Though we experimented briefly with such a set-up, a P2P implementation of the tester is left for future work.

cult conflicts when trying to run tens of copies simultaneously. This can be something as simple as sharing configuration files (and conflicts at start-up), to more complex problems.

One important example is that one application we tested used a stand-alone relational database as a backend. It also assumed the database was installed and well-configured, which can be difficult or impossible with a dozen or more instances running in parallel. A very well-designed P2P application would have the option of using any of several similar databases, including a built-in module like SQL-Lite. However, because of differences and incompatibilities between different database engines, we expect a lot of work will be needed in cases like this. Even when the problem of sharing these resources is solved, they always leave the problem of performance limitations, too.

3.6 Use for XQ2P

XQ2P was constructed using several components provided by P2PTester. Specifically:

Communication: All messaging between peers happens using per-peer instances of the `MessagePipe` primitive. When used in tests distributed over several hosts, the default socket-based implementation is used; in local tests the in-memory message-passing implementation is preferred for speed reasons. Note that this was done for simplicity reasons, as XQ2P is intended initially as a research tool; a more practical application would perhaps benefit from a streaming approach to transferring index lists, including optimizations like those proposed by [58], and using message passing only for overlay-maintenance and routing operations.

Tracing and Logging: The number of messages exchanged by XQ2P for indexing and querying tend to be easily analyzed statically. Nevertheless we took advantage of the tracing and logging features provided by P2PTester to monitor peer loading state and the *size* of exchanged transfer, as well as for debugging purposes during development.

Key-based routing: XQ2P uses a key-based routing module as a fundamental primitive on which it implements its higher-level indexing and querying functions. A simplified version of Chord provided by P2PTester is used by default. In addition, a ready-made module that simulates a KBR overlay is also useful for running quick local tests of the higher-level functions (simulating the

```
interface XQueryDB extends Peer {
  public void publish(File document);

  public Sequence evaluate(String xquery);
}
```

Figure 3.5: A simplified version of the XQueryDB interface, implemented by XQ2P nodes. Exception handling and event tagging have been omitted for brevity.

lower-level layer eliminates the large number of messages needed to establish the overlay; this is transparent to the higher layers).

Note that we do not use the provided DHT modules directly: The basic operation of the indexes (see the following chapters) resemble a DHT in the sense that they associate values to keys; however, the exact semantics of `put` operations are slightly more complex, as peers need to merge indexes associated by several nodes with a single key.

Test Scenarios: Test scenarios were used extensively during XQ2P development for debugging and evaluation purposes. P2PTester allows precisely specifying behavior—such as which peer publishes what document or initiating queries from any of the peer nodes and checking or comparing results, sequentially or in parallel—and re-executing such scenarios with perfect reproducibility.[54] In addition, such tests can be run repeatedly while changing the lower-level modules, such as the network layer or the KBR implementation, without any modification to XQ2P itself.

Interface: The interface exposed by XQ2P is very simple, as most of its complexity is contained in the query language. It resembles the one in Figure 3.5. Adding more features, such as indexing a local document without publishing it, removing a document from the index, or various access controls, can of course be expressed by derived interfaces; being focused on indexing and query evaluation, we ignored such features in this work.

Note that it is relatively easy to write adapters around XQ2P that respond to other query languages; an adapt or needs only translate the query to XQuery

[54] Injecting any desired amount of randomness is also easy.

syntax. Such adapters would allow comparing XQ2P with other engines using the same test scenarios. For example, tree-pattern queries are simple to express in XQuery; most reuse subsets of XQuery syntax, making translation unnecessary.

Chapter 4

Distributed XQuery Processing with Structural Indexing

In the previous chapter we presented the general architecture of XQ2P, and explained the functioning of its XQuery processing kernel on local data. As a distributed database, XQ2P needs to be able to answer queries using data published by peer nodes as well. We adopted a simple approach for solving this problem:

First, nodes that participate in the distributed database organize themselves into an overlay network with (efficient) key-based routing.

Second, peer nodes that wish to make a local document available to the distributed database will publish an index of that document throughout the overlay, using a DHT-like protocol deployed on top of the KBR layer.

Finally, nodes that wish to execute a query over the shared data use the distributed index to locate relevant documents and contact the peer nodes that hold them; either the entire located documents or parts of them are downloaded, and the local XQuery processor is used to perform the query on the retrieved data.

In this chapter we describe the use of a structural index—i.e., one based on the types and relationships between XML elements—to execute this strategy. In the next chapter we apply the same approach to a value-based index.

The contents of this chapter are as follows: The overlay network used by XQ2P is presented in the first section, below. Structural join algorithms and the particular case of TwigStack, used by XQ2P, are then presented in §4.2; the index used is described in §4.3.

Though we use TwigStack to enable XQuery evaluation on the distributed database, the algorithm was originally proposed for local evaluation, a fact XQ2P takes advantage of. Accordingly, §4.4 introduces the TwigStack-based XQ2P operators and describes their use for local query evaluation, and finally §4.5 extends this to execution over distributed data.

4.1 The XQ2P Overlay Network

XQ2P is a *peer-to-peer* distributed XML database. As such, the first and most basic task of the system is to organize the participating node in a overlay network. This layer provides key-based routing functionality, which is fundamental to the indexing and querying functions built on it.

In principle, almost any KBR protocol would be suitable for our use. We implemented a simplified variant of the Chord system to serve as a KBR layer. Its basic features are as follows:

Communication XQ2P nodes use the `MessagePipe` primitive provided by the P2PTester framework. The exact implementation can be chosen depending on the intended usage, the default being socket-based.

Node identity Each node participating to the XQ2P network has an associated identifier. Its only purpose is to distinguish the node from other nodes on the same overlay; this has certain implications:

- A node does not have an identifier while not connected to a network;
- As long as it is connected to the network, the node's identifier does not change;
- No two nodes connected to the same overlay may have the same identifier.

The address of a `MessagePipe` matches these conditions, and for this reason we use it as its identity. Note that the exact content of the address is not important for XQ2P; it is used as an opaque object, the only operation needed being comparison for equality. Thus, XQ2P can use any of the `MessagePipe` implementations without modification.

Observe that a node does not have identity when not connected to a network, and that if node disconnects and then re-connects to the same network it may well have a different address (and identity). In other words, a node's identity does not persist while it is disconnected.

Keyspace We use the usual 160-bit circular key space of Chord; node IDs are obtained by hashing the serialization of the node's address with the SHA-1 algorithm. Note that the objects used for IDs and keys are an implementation of a very limited interface; they are opaque to the upper logic layer, which only performs comparisons for equality with them. This allows replacing the KBR layer with a different one transparently, without any modification to the rest of the code.

Overlay structure As in basic Chord, each node maintains connections to its successor, its predecessor and "fingers" to nodes owning keys at distances $2^i,\ i = 1..160$ from its ID.

Routing The only operation this layer offers is key-based routing: Given a key (from the higher logic layers), it finds and returns the address of the peer owning that key. This operation requires $O(log_2 N)$ message exchanges, where N is the number of peers connected to the overlay. (The message length is very short; message payloads consist only of an ID or a peer address, plus a tag for the message type. It is latency rather than bandwidth that limits performance at this level.) The signature of the operation is simply:

```
ID find_owner(ID key, Watermark... marks);
```

where the `marks` argument is used for the optional event-tagging feature (see page 65).

Differences from Chord Because we use XQ2P as a research platform, we simplified the KBR layer as much as possible. We did not implement the parts of the Chord protocol related to checking connectivity periodically and maintaining the overlay. Instead, a connecting or disconnecting node will explicitly notify the (already connected) peers that it might need to update their fingers table. This is intended to reduce network chatter during testing. However, note that the layers above are ignorant of these details, and a more full-featured KBR layer can be substituted transparently.

For a more detailed description of the Chord protocol itself, consult §2.3, in particular page 26.

4.2 Structural Join Algorithms

Document-centric XML processing tools usually need to process entire documents (and collections of documents); their focus is efficient traversal of entire trees. In contrast, use of XML in data-centric environments tends to require

querying and returning relatively small parts of very large collections of XML data. In such cases, techniques that traverse the entire document tree to locate relevant data are too slow, no matter their efficiency. Indexing is necessary to rapidly identify and locate only those parts of an XML connection that are necessary for answering a query.

Traditional indexing structures and algorithms developed for relational databases can be usually applied to solving *value predicates* in typed XML data, and text-search indexes and algorithms are similarly applicable to some queries on XML content.

Efficiently answering *structural queries* on XML, however, requires more specific techniques. By "structural queries" we mean such queries that select tuples of XML nodes that have specific types and structural relationships in relation to the XML trees—e.g., parent/child, ancestor/descendant—rather than through specifying conditions on the content they hold. XML querying languages like XPath and XQuery rely heavily on such structural relationships.

XQ2P implements structural indexing based on node numbering and the TwigStack algorithm to perform structural queries. Consult Chapter 6 for other examples of structural indexes and algorithms that apply to them.

The TwigStack Algorithm

XQ2P uses an implementation of TwigStack to implement pertinent-document location within the distributed document collection, and to accelerate query execution over local documents.[1] In this section we summarily describe the functioning of the algorithm; see [12] for a more detailed description and analysis.

TwigStack uses an index based on a numbering scheme where each element is assigned a pair if integers, `leftpos` and `rightpos`, such that $\texttt{leftpos}_1 < \texttt{leftpos}_2 \leq \texttt{rightpos}_2 < \texttt{rightpos}_1$ whenever node 1 is an ancestor of node 2. Documents are indexed by constructing, for each element type,[2] the list (document ID, `leftpos`, `rightpos`, `level`) of occurrences of that element type in the collection, ordered by document ID and `leftpos`; `level` is the

[1] Many other algorithms using the holistic join principle have been published since TwigStack, several of which are mentioned in *Structural Join Algorithms*)(see page 83). We chose TwigStack for being well-known, and partly a demonstration of integrating a complex optimization in XQ2P. Adapting further improvements and more algorithms is a promising direction for future work.

[2] TwigStack, as defined in [12], also indexes individual words in text nodes, allowing execution of some text value predicates simultaneously with the structural query. We omitted this feature in XQ2P for now.

depth of the element occurrence in its tree, with the `level` of the root element equal to zero.

TwigStack solves *twig-pattern* queries. Twig pattern queries are trees with nodes corresponding to node tests. An answer for a twig pattern is the set of all assignments of document nodes of correct type to each query node, such that the parent-child relationships between nodes in the twig pattern correspond to parent-child or ancestor-descendant relationships between assigned nodes. (In common graphic representations of twig patterns, parent-child edges are represented by single lines and ancestor-descendant edges by double lines.)

TwigStack accumulates results using a tree of stacks, one stack for each node in the twig pattern. Each stack holds tuples of a node (more exactly, the positional representation of a node) and an integer, representing a pointer to some position within that stack's parent stack; the stacking algorithm (below) ensures this pointer indicates the highest element on the parent stack that is an ancestor of this node, and that a node on a stack is a descendant of all nodes below it in the same stack. This makes the tree of stacks a compact encoding of partial query results. (Root stacks, i.e., stacks corresponding to the root node in the twig pattern, do not need the integer pointer.)

A stream of node positional identifiers is used for each twig pattern query node, each stream referring to nodes of the same type as that referenced by its query node. Note that the streams are ordered by `leftpos`,[3] which translates to document order in XML.

The algorithm advances all streams until the elements pointed to by each stream are in the relationship demanded by twig pattern; this state can be easily determined by testing inequalities between the positional identifiers at the head of each stream. Consider the case of a two-level twig pattern where the streams are not in such a state: then at least one of the child streams points to an element before the root element in document order, or comes after the closing of the root element's end-tag; in the former case, the child stream can be advanced until it reaches the first element with a greater `leftpos` than the root element's, *without needing to examine intermediary elements*; similarly, in the latter case the root element must be advanced until its `rightpos` is greater than *all* the child elements'. Repeatedly advancing the streams[4] in this way TwigStack rapidly skips over portions of the index that cannot contribute to results.

When a correct position for a pair streams is found, the element pointed to

[3] In multi-document instances of the algorithm, by (document ID, `leftpos`).

[4] Because the "targets" values for skipping are known, an appropriate index can use, e.g., binary search to advance without examining each element.

by the child stream is either a descendant of the element on top of that stream's stack, or comes after it in the document. (Remember streams are sorted in document order.) In the former case, it is pushed onto its stack, with the integer pointing to the parent stack's top. In the latter case, none of the results accumulated on the lower stacks may form a valid answer with any future elements; if the stream belongs to leaf query node, then results accumulated on the stacks that depend on the streams' top are outputted,[5] and the top element is lifted from the stack; if not, nodes are lifted from parent stacks successively until another correct pair of streams is found.

The precise sequence of operations ensures that all possible correct assignments are considered and generated, which ensures correctness. The mechanics of advancing the streams depending on positional identifiers efficiently skips over nodes that do not participate in results, which ensures speed. Finally, the stack-encoding minimizes memory use for partial results.

4.3 Document Indexing

All documents in a node's collection are indexed at loading time. We implement an index based on the popular technique of pre/post structural labeling, because it is useful in several different stages by several different algorithms during query processing (described on page 37). Note however that XQ2P was designed such that it is easy to add other indexes at the same time.

Conceptually, our index is built as follows:

First, all attribute and element nodes are traversed in document order. A counter, initially zero, is incremented each time the traversal passes from a node to another. Each node is assigned a *start* label, equal to the value of counter when the node is encountered, and a *end* label, equal to the value of the counter when the subtree rooted at that node has been traversed. Each node is also assigned a *level* label, equal to the depth of the node in the tree. This kind of labeling is commonly called *region encoding*.[6]

Observe that the labeling ensures that, for each node:

- the start label is always lower than the start label of any of its descendants;
- the end label is greater than the end label of its descendants;

[5] If the query contains parent/child edges, not all nodes accumulated on the stacks necessarily form a result: the stacks guarantee only ancestor/descendant relationships, so the `level` of nodes needs to be considered.

[6] There are many different possible region encodings, depending on exactly which nodes are counted and when the counter is incremented.

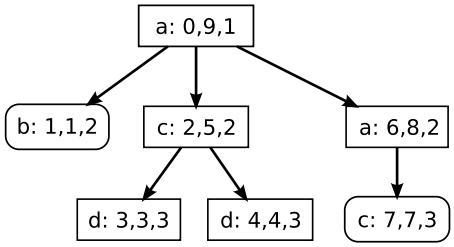


Figure 4.1: Region encoding of a simple tree

```
a: (0,9,1); (6,8,2)
@b: (1,1,2)
c: (2,5,2)
@c: (7,7,3)
d: (3,3,3); (4,4,3)
```

Figure 4.2: The index lists generated for the document tree in Figure 4.1. Note that attributes are listed separately from elements that have the same name.

- the start label is greater than the end label of any preceding node;
- the start label is greater than the end label of any preceding node;
- the level is exactly one less than that of its children, exactly one more than its parent's, and equal to that of its siblings.

These properties are very useful for a implementing so-called "structural join algorithms"; we describe these in more detail later.

Then, for each kind of node thus traversed, we construct the list of (start, end, level) triplets assigned to nodes of that kind. In the preceding, two nodes are of the same kind if they are of the same node type (i.e., they are either both element nodes or both attribute nodes) and they have the same QName.

A document's index is the map having as keys the node kinds and the list of nodes of that kind, in document order (i.e., sorted by the "start" label). The index of each document is kept in memory, associated with that document.

Publishing documents

As a *distributed* XML database, XQ2P must allow the user to query documents owned by any node of the network, not just those on his or her own node. Nodes must publish information about the documents they own, to allow other nodes to find them. This information is called the index, and we call the process of informing the network about documents "publishing" those documents. (See §3.2.)

XQ2P's architecture allows using many different kinds of indexes, and even using several kinds of indexes at the same time. For this work, we implemented an index that is an extension of the index described above for a peer-to-peer context. Because of their similarity, we refer to both as "index"; when disambiguation is necessary, we will refer to the indexing of a document as the "document index", to the collection of indexes of documents owned by a peer as the "local index", and to the index of all documents in the network as the "distributed index".

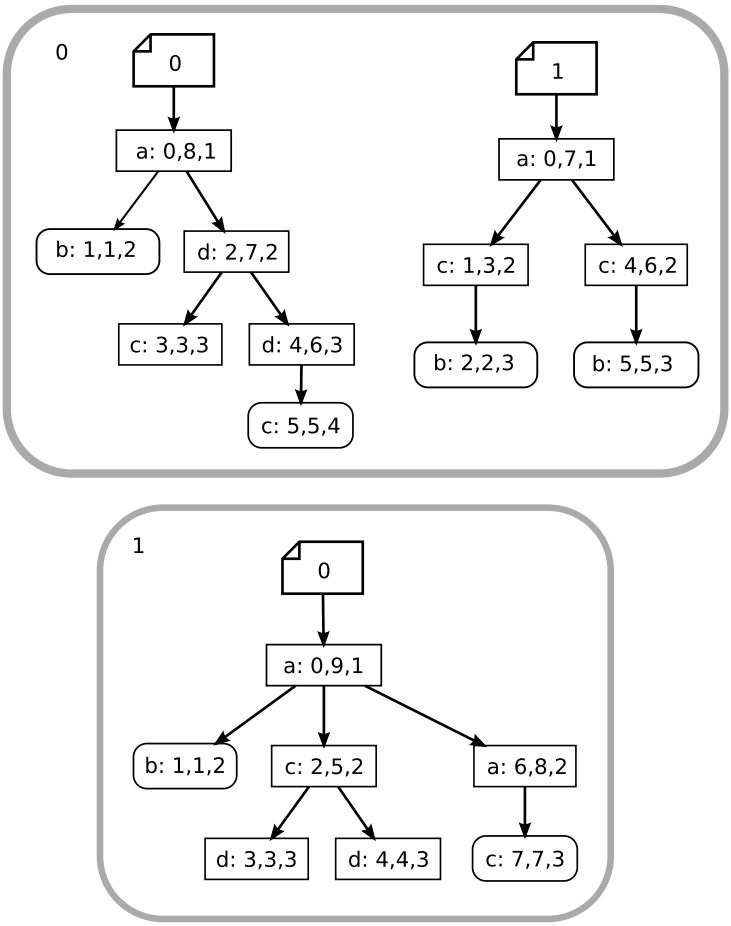
Logically, the distributed index is a straight-forward extension of the local version to the case of multiple documents owned by multiple peers. Recall that local index represented each indexed node by a (start, end, level) triplet. The distributed version adds to this the identifier of document that contains the node, and the identifier of the peer that owns that document.[7]

Thus, the logical structure of the distributed index is a map from node kinds to the list of 5-tuples interpreted as the structural identifiers of nodes of that kind. Observe that the structural properties of these tuples are the same as those of triplets in the single document index as long as the peer ID and document ID are identical (when they are not, the nodes are part of different documents, thus they have no relationship).

The lists of 5-tuples are sorted first by peer ID, then by document ID, and finally by the start label. (Because the document indexes are already sorted by start label, the distributed index is assembled with an insertion-sort on the (peer ID, document ID) sort key.) Incidentally, the order given by (peer ID, document ID) is used for the extension of document order to nodes between two documents (see page 17).

This index is distributed through the overlay network similarly to a DHT: Each time a document is published, the publisher node will do the equivalent of a `put` operation for each node kind. The node kind is the key, while the list of 5-tuples is the value associated with that key. The KBR key is determined by hashing the UTF-8 representation of the QName for elements, and the QName

[7] Recall that documents are assigned an identifier that is unique only among documents owned by the same peer.




```
 a: (0:0:0,8,1); (0:1:0,7,1); (1:0:0,9,1); (1:0:6,8,2)
@b: (0:0:1,1,2); (0:1:2,2,3); (0:1:5,5,3); (1:0:1,1,2)
 c: (0:0:3,3,3); (0:1:1,3,2); (0:1:4,6,2); (1:0:2,5,2)
@c: (0:0:5,5,4); (1:0:7,7,3)
 d: (0:0:4,6,3); (1:0:3,3,3); (1:0:4,4,3)
```

Figure 4.3: Region encoding for a two-peer, three-document collection. The rounded gray boxes represent peers; peer IDs are in the top-left corner (short consecutive numbers are used here for brevity). Boxes with folded corners represent document nodes and their IDs.

prefaced with an "@" character for attributes. (Recall that "@" cannot be the first character in a QName.)

Recall that in a usual DHT multiple values associated with a single key are simply kept in a list. In our case, for a particular QName, the value is a list of 5-tuples, which must be distributed for each document published that contains nodes of that kind. (We occasionally call such a list a "posting list", similarly to [4].) But in a peer-to-peer network it is in general unpredictable when each peer will publish documents. Because we want the distributed index to be ordered by (peer ID, document ID), using the simple DHT semantic of appending to a list will lead to posting lists being appended in the order of posting. For this reason, we use slightly modified semantics: the peer that holds the posting lists for a QName will insert newly received posting lists (from newly published documents) in the necessary position to maintain the sorting, rather than just appending them at the end.

The description above is of the logical structure of the index, which is the form used to describe the querying algorithms, later in this document. However, for efficiency reasons in the actual implementation peers do not publish or hold a flat (sorted) list of 5-tuples. Instead, the piece of distributed index held for each QName is represented as a (sorted) list of (node ID, local index) pairs, where the local index is also a (sorted) list of (document ID, document index) pairs; the document index is simply the list of triplets described in the previous section. This avoids duplicating the peer and document ids for each indexed document node, and also allows the constructing the distributed index from the document indexes by simple inclusion. For example, the compact form of the list for `c` elements in Figure 4.3 would be:

$$\{0 : [0 : (3, 3, 3)], [1 : (1, 3, 2), (4, 6, 2)]\}, \{1 : [0 : (2, 5, 2)]\}.$$

Modification of indexed documents

Other than publishing newly added documents, the only modification to the distributed index we support explicitly is document removal. If a peer wants to cease to offer access to a published document, it must issue a `drop` request for every posting list it has published for that document. This requires the same number of messages as the initial publishing. (However, note that these request need only mention the QName, the peer ID and the document ID; since the actual list of triplets does not need to be sent, the messages are much shorter.) This step is not strictly necessary: the node can simply refuse to service query requests for that document; the querying algorithms must be able to handle this case anyway, to deal with nodes that drop from the overlay network, e.g., due to broken connectivity.

There is no explicit support for *changed* documents, although a peer can achieve this by `dropping` the published document then publishing the changed version. Efficient support for updates of this kind of structural indexes is complex, and interesting future work.

4.4 The TwigStack operators

We implement the TwigStack algorithm in an eponymous class. A TwigStack object is constructed from a tree of input streams (represented as a list of streams with a second list of integer pointers to the parent). It has a single `evaluate` method, which returns an encoded form (based on the stack encoding of the algorithm) of the results, i.e., correct assignments of nodes for each node in the stream query. Note that the algorithm itself does not depend on the type of nodes encoded by the input streams; we only implement indexes of element of specific types in XQ2P, but this is possible to extend later, for example to include composite streams that allow two kinds of elements, or streams of words in text nodes and attribute values.

Note that TwigStack is not an operator class in the XQ2P sense, as it does not generate `Sequences` of items. Instead, operators are constructed based on the class and embedded into a modified query plan.

Query plan transformation

In order to use TwigStack-based processing, the operator tree (basically, the executable query plan) generated by XQ2P after standard static analysis is transformed into an analogous operator tree that substitutes TwigStack-based operators to the default axis-navigation ones.

There are two TwigStack-based operators: the first replaces stand-alone XPath expressions; the second replaces FLWOR expressions. The separate FLWOR operator is used because iterating over tuples can be done more efficiently using the stack-encoded results of TwigStack directly rather than over instantiated sequences of results.

The transformation step visits the of the original query plan to discover tree patterns. Stand-alone path expressions are transformed in three phases:

Discovery: If the expression currently examined can begin a tree pattern that may be optimized, begin the construction of a new tree-pattern builder. Currently, only calls to `fn:document` or to `fn:collection` are such expressions. (If a tree-pattern was already in construction, a stack is used to keep track of it until the new tree pattern is finished.)

Assembly: When visiting subexpressions, there are several possibilities: If the subexpression can continue the currently constructed query plan, a new query node is added to the latter. Currently only `descendant::`, `descendant-or-self::`, `child::` and `attribute::` axis steps with element/attribute name tests can participate in a tree pattern.

If axes have predicates, their subexpression are added to a "predicate list" of the current (last added) node of the tree pattern.

Otherwise, the expression is a continuation of the XPath that cannot be performed using TwigStack; it is analyzed as a separate expression, and the resulting sub-tree of operators is set as the *output transformer operator* of the current query plan. Examples of expressions that cannot form parts of the twig pattern query are function calls, arithmetic expressions, or non-descending axis steps.

Operator substitution: When the visit of subexpressions of a tree-pattern root expression is done, the tree pattern is finished. The query nodes are used to construct a new TwigStack instance, which together with the predicate list and the output receiver, if any, is used to construct a new operator (i.e., a sequence-producing) instance.

Execution: When the `evaluate` method of the new operator is called, it first obtains the index streams from the dynamic environment and passes them to the TwigStack instance for execution. It then iterates over result tuples and executes any predicates in the predicate list on the appropriate nodes; tuples that fail a predicate test are skipped. It then generates the sequence of nodes corresponding to the output node. If there is an output transformer operator, the sequence is passed to the latter and the result is returned; otherwise, the sequence is returned as generated.

FLWOR expressions : If a tree-pattern root expression is discovered as the root expression assigned to a `for` or `let` clause of a for expression, the process above is slightly modified: The variable that was assigned is added to a list of twig-pattern–bound variables; in the assembly phase, path expressions beginning with accesses to a variable that is in the list of twig-pattern–bound variables become eligible to continue the same twig pattern rather than beginning a new one, and variables that are assigned the result of such expressions are also added to the bound-variables list. The twig pattern will have several output nodes (one per variable), each of which might have an output transformer expression.

Twig patterns discovered in FLWOR expression clauses cause the substitution of the entire FLWOR expression, not just of individual variable assignments. When the replacement operator is executed, the tuples generated by the FLWOR expression are generated as needed from the compact representation returned by the TwigStack expression.

Note that not all `for` and `let` clauses of a FLWOR expression are necessarily bound to a twig-pattern output node; those that are assigned expressions that are not expressible in the tree-pattern will be computed by the original operator subtree.

Query formulation

XQ2P requires no syntactical changes for distributed queries. Queries that must be executed over the distributed database use the same XQuery dialect as the local ones. The only user-visible modifications are to the data-source functions:

The `fn:collection` function is augmented: it will also accept the "`distributed`" collection. This behaves similarly to the "`local`" collection described on page 96, but refers to all documents in published on the network. The "`remote`" collection is also provided, which holds the difference of the "`distributed`" and "`local`" collection, i.e., all documents published on the network *except* those published by the node running the query.

The `fn:doc` function is extended to accept references to documents on other nodes. These are identified by URL following the generic syntax of [5], with the following elements:

Scheme name This is always the string `"xq2p"`.

Host part This depends on the specific KBR routing layer in use, as it identifies the node within the network that the document is retrieved from. For Chord, it is the case-insensitive hexadecimal representation of the node's identifier.[8]

[8] It is of course possible to also allow node names, to offer URL persistence with regards to node ID changes, but we have not yet implemented this feature.

Document path This corresponds to the document name used by the publishing node. It is in the form of an absolute path, i.e., it is always separated from the host part by a forward slash.[9]

`xq2p:` URIs contain no username, password, no port, query string or fragment identifiers. As an example, a typical XQ2P URI might look like `xq2p:0123456789ABCDEF/some/document.xml`, referring to the document labeled "/some/document.xml" published by the node with Chord ID "`0x`0123456789abcdef".[10] The `fn:doc-available` function is extended similarly.

Even in the distributed environment, the two data-source functions retain their local capabilities. It is thus possible to write a query that references at the same time local documents and distributed ones (the later including the former). See Figure 4.4 for an example.

Stages of processing

XQ2P performs query execution in several stages. In short:

- First, the query operator plan is examined to identify data sources. The result is the set of twig patterns that assemble the query, and the set of *node sources* these patterns are sourced from.
- Second, each enabled query transform is executed. The result is an optimized query plan; in particular, TwigStack operators replace the unoptimized operators where possible, and the list of required indexes is constructed.
- Third, a dynamic environment is constructed for the query; in particular, the dynamic environment is augmented with the references to the node sources and indexes needed by the query. (To recap, documents required by unoptimized tree patterns sourced from `fn:doc()` and `fn:collection('local')` are retrieved from cache, and indexes are attached to the TwigStack operators; constructed node–patterns are left

[9]The exact syntax of a document name is in principle free-form, but XQ2P nodes currently use a format based on Unix file path syntax.

[10]The additions to the `fn:doc` function are intended mainly for debugging purposes. It is expected that user queries will in general access the distributed database via `fn:collection`.

```
<html><body> {
for $artist:=fn:collection('local')/favorites/song
   /@artist
let $albums:=fn:collection('distributed')/music/
   artists/*[@name=$artist]//album/@title
order by $artist, $album
return
  <h3>Albums of {$artist}:</h3>
  {
    for $album:=$albums
    return <p>{$album}</p>
  }
} </body></html>
```

Figure 4.4: Example use of `fn:collection`.
This query might find albums from the entire distributed database that are created by an artist who created at least one song that the owner of the executing node liked. The result is an HTML document.

alone, to be assembled during query plan execution; unidentified data sources, e.g., the initial value of the context node, raise dynamic errors.[11])

- Finally, the final query plan is executed, by passing the `DynamicContext` instance to the `evaluate` method of the query plan root operator. The result is either the `Sequence` containing the result, or a dynamic error. The third step might be re-run during this stage, in cases where the data source functions have non-literal arguments.

The execution of distributed queries follows the same sequence. However, the third stage is modified to support the additions to the `doc` and `collection` functions, as follows.

[11]Observe that although technically the execution of the query plan has not started, this step belongs to the execution rather than analysis stage of query processing, because it depends on the actual data (specifically, the presence thereof); thus, only dynamic errors can be raised.

The `fn:doc` and `fn:doc-available` functions

The modifications to `fn:doc` are the least extensive. If URI refers to a document published by the node executing the query, the function behaves as if having received the `local:` URI to that document.

Otherwise, the node identified by the host part of the URI is contacted and the document is retrieved, indexed and cached, then used just as a local document would be.[12] If any of these steps fail (e.g., if there is no node with the given identifier, or it exists but it has not published a document with the given name, or a connection error occurs), `err:FODC0005` is raised.

`fn:doc-available` is similarly extended. Note that to satisfy the requirements of this function, any call to `fn:doc-available` will cause the document to be downloaded and indexed, even if it is not actually accessed via `fn:doc`.

The `fn:collection` function

The `fn:collection` function is treated differently: when called with the `'local'`[13] argument, the query plan is simply given reference to the (indexed and cached) set of local documents. A different approach is needed for the distributed case, because simply downloading every shared document would defeat the purpose of a peer-to-peer network.

Instead, we attempt to retrieve only the documents that contain data relevant to the query. Recall that the efficiency of the TwigStack algorithms described above is due to their skipping over nodes that do not participate in the result as soon as possible. We exploit this same property to skip over (remote) document nodes that do not participate in the query. Since TwigStack algorithms are needed, thus implicitly the optimized tree patterns, queries where `fn:collection('distributed')`[14] appears outside of an optimized tree pattern are rejected with `err:FODC0002`, "Error retrieving resource."

A call with a non-literal argument will raise `err:FODC0003`, "Function sta-

---

[12] Note that any document is loaded at most once. Thus, the function is stable, even if the URI is constructed dynamically.

[13] Note that the argument to `fn:collection` should technically be interpreted as an URI, resolved against the base URI from the static context if relative. For simplicity XQ2P accepts `'local'`, `'distributed'` and `'remote'` (which are relative URIs) regardless of the base URI; this may be interpreted as mapping *all* URIs ending in the three strings to one of the three collections.

[14] This is the default collection, so that `fn:collection()`, i.e., a call without argument, is synonymous and refers to the entire indexed collection.

bility not defined".[15]

4.5 Indexed document retrieval

XQ2P uses the same execution engine for local and distributed queries; this means that the actual execution of the query requires having direct access to the needed documents, where "needed" means documents whose nodes contribute to the results. In the P2P case, we are concerned primarily with queries where only a small subset of shared documents contribute to the results, in which case we can use the following method:

- First, each optimized query pattern sourced at the `'distributed'` collection is transformed into a pattern that retrieves the document URIs; the parts of the query that refer to non-document nodes is transformed to a predicate on those document nodes. For example,

  ```
  fn:collection()//a[b]/c
  ```

  becomes the equivalent of

  ```
  fn:document-uri(
      fn:collection()/document-node()[//a[b]/c]
  ).
  ```

  Note that the "predicate" will rarely consist of the entire query: it is only the sub-part of the query that can be executed using the index and the TwigStack algorithms available. Any call to `fn:collection` the results of which are not filtered with an optimized query pattern raises `err:FODC0002`. We sometimes call this derived query a "filtering query", because it filters out from the collection documents that do not participate in results.

[15] Given the way indexed documents are retrieved, described in the next section, it would be quite complex to implement the function stably without this requirement. XQ2P does not yet provide an option to disable function stability.

(The actual requirements for stability is that all required documents be discoverable in one pass, as described in the following section. This is technically possible for any statically-computable string, not only literals, and even for dynamically-computed strings where the resulting pattern is strictly more restrictive than another statically-computable pattern in the query. We leave such refinement for future work.)

- Second, this derived query is executed using only the distributed index. Observe that any remaining part of the original query is a predicate in this derived query, and it contains only relationships that the TwigStack algorithm can determine from the index, without accessing the documents, because this is how the optimized query patters are selected in the analysis phase. The result of this step is the list of documents that may contribute to the query result.[16]

- Third, the documents identified in the second step are downloaded and temporarily[17] merged to the local collection. Optionally, this stage may raise `err:FODC002` if the result list of documents is too large;[18] this can happen if the indexes used are not selective enough.

- Finally, the original optimized query plan is executed on the set of resulting documents. Since every document is downloaded locally, this uses the local query engine, and the process is identical to executing a local query.

Filtering query generation As mentioned above, the first step of querying is determining which documents may contain results. Distributed documents can only be accessed by queries that do so via the optimized query patterns (described earlier); we need only transform the pattern from one that specifies a node to one that specifies a document containing such a node.

Since we use the same indexes for document and node retrieval, this transformation is straightforward,[19] every pattern `P` becoming the equivalent of

```
collection()/document-node()[P].
```

Note that we do not construct and execute a new query for this step; instead, the "pattern operators" are substituted in a transformed version of the operator tree in memory, which is then executed immediately before the root query expression; if the step is successful, the dynamic query environment (specifically, the sequences returned by the `fn:doc` and `fn:collection` func-

[16] Calls to `fn:collection('remote')` are treated the same, except that documents belonging to the local peer are simply filtered from the list.

[17] Currently XQ2P retains the documents only as long as the query is executed, but a caching mechanism could be added at this point.

[18] The exact limit is configurable by the user.

[19] There are some more complex transformations that may be more efficient, but we leave these for future work.

tions) is augmented with the retrieved documents. The local query plan is simply run with this dynamic environment.

Chapter 5

Processing Time Series Efficiently with Value Indexing

To further validate XQ2P as an *extensible* platform for research on efficient techniques for XQuery evaluation, we also experimented with value indexing by extending to allow efficient execution of operations on time-series data. The content in this chapter focuses on the modifications to XQ2P; for more details and references on the general problem of time-series processing, including comparisons with a non-distributed XML database, we invite the reader to consult [16].

Time series have been used in many application domains, including such distinct areas as economy and finance, weather and climate, or transport control. In our prototype extension to XQ2P, we focused on efficiently computing large time-series of financial data. A generic construct for processing continuous streams of data based on windows has been proposed in [10] and adopted by XQuery 3.0[80]. As window-based queries are particularly well suited to processing time-series data, we decided to implement limited support[1] for XQuery 3.0[2] syntax for this task. The relative ease of adding the new constructs further validates the suitability of XQ2P as a research platform for

[1]We remind the reader that XQ2P is based on XQuery 1.0. In order to support processing of time-series data we added support for the windowed-for constructs of XQuery 3.0, but not for the entire set of features added in the latter version of the standard.

[2]Earlier working drafts referred to the version of XQuery succeeding 1.0 as 1.1.

XQuery.

5.1 The Time Series Model

The model we adopted for time-series data is derived from the RoSeS project[88]. The model is composed of a vector space of time-series equipped with operators that map one or several time-series to one, analogous to relational operators. Also included are time-domain aggregate operators that change the time unit of a series.

A time-series is defined as a (potentially unlimited) vector of values. The vector is associated with a calendar that associates each value with a point in time. The sampling granularity is specific to each time-series (e.g., a value each second, hour, day). Values can in principle be of any XPath/XQuery type; for the specific needs of application—where values stock valuations—we use floating-point numbers. Regardless of the domain of specified values of each time-series, we also define two distinct null values, the empty or non-existent value, denoted "`!`", meaning a value is known not to exist for the given time, and the unknown value, denoted "`?`". Scalar operators on a time-series' values are extended to act suitably on null operands when applied to time-series: Operations between the unknown value and any other value result in the unknown value, i.e., $\forall x, x \circ \texttt{?} = \texttt{?}$. Other operations are extending depending on semantics; in our case we extend the $(+, \times)$ operations on floating point values to operate pairwise on time-series, forming a linear vector space such that for any x except `?`, $x + \texttt{!} = x$ and $x \times \texttt{!} = \texttt{!}$ (i.e., adding a non-existing value to any value does not change the latter, and the product of a non-existing value does not exist for any value).[3]

The calendar starts at the a given time, corresponding to the first available value, and ends with the last available value. When operating on time-series whose calendars overlap only partially, these are padded to cover the union time domain using `!` by default. An explicit operation may be used to pad with `?` or with another value, such as numeric zero.

Time-series Operators

We derive operators specialized for time-series from relational algebra. The following definitions denote (t, v) an entry of a time-series having the value v time t:

[3] Such semantics resemble the behavior of `null`, zero, or `NaN` (not-a-number) values in many programming languages.

Projection: $\text{PROJ}_f\, un(S) = \{[t, v] | [t, val] \in S \wedge v = fun(val)\}$. In other words, projection applies the scalar function $func$ to each value in a time-series, and maps each time point to the result in a new time-series.

Selection: $\text{SEL}_{pred}(S) = \{[t, v] | [t, val] \in S \wedge v = pred(val)\}$, where $pred(val) = val$ if val satisfies the predicate, and `!` otherwise. In other words, the result is a same-domain time-series where values that do not satisfy the predicate are replaced with the non-existing value.[4]

Union: $S_1 \cup S_2 = \{[t, v] | [t, val] \in S_1 \vee [t, val] \in S_2\}$.

Intersection: $S_1 \cup S_2 = \{[t, v] | [t, val] \in S_1 \wedge [t, val] \in S_2\}$.

K-ary Join: $\text{JOIN}_{fun}(S_1 \ldots S_k) = \{[t, v] | [t, v_1] \in S_1 \wedge \cdots \wedge [t, v_k] \in S_k \wedge m = fun(v_1, \ldots, v_k)\}$. This operation perform a join on the time attribute of k time-series using the same calendar, and applies a matching function to the k-tuple of values for each time point.

The above operations are represented with `let` and `where` operations in XQuery.

Window operator: $\text{WIN}_{fun}(S, w) = \{[t, v] | val = fun([t-1, v_1], \ldots, [t-w, v_w])\}$ such that $[t - i, v_i]$ designate the ith value previous to t in series S, except that if $t - i$ is negative (i.e., if it would point to a time before the series' calendar's beginning), it is set to val_0 (i.e., the first value in the series).

Window operations are expressed in XQuery by `for window` clauses, using XQuery functions for mapping windows to values.

5.2 Stock Selection and Strategy Evaluation

Technical analysis is a technique used in finance that considers prices and volumes as temporal signals and analyzes such signals to detect indicators, patterns and events, in an attempt to predict future prices and guide investment.

Several window-based operations are used often in such analysis, including for example the Moving Average (MAVG) and the Relative Strength Index (RSI). MAVG_w computes the classical moving average series of a series S with

[4] Selection can be seen as a special-case of projection, using a function that switches between the identity function and the constant-`!` function depending on the result of applying the predicate.

a sliding window of fixed size w. Let $V = \text{MAVG}_w(S)$. The value $V[t]$ is defined by:

$$V[t] = \sum_{i=1..w} \frac{S[t]}{w} = \frac{(w-1) \times V[t-1] - S[t-w] + S[t]}{w};$$

A common variation is the exponential moving average, where value $[t-i]$ is modulated by weight $(1-\alpha)^i$ in the sum above. Many other indicators with similar structure exist.[43] The common feature we remark upon is that such operations generate a result time-series based on the same calendar as the source time-series.

More complex operators can be obtained by combining logical, vectorial, and windowing operators. For example, Moving Average Convergence/Divergence (MACD) is one of the simplest indicators used by investors. A usual base formula for MACD is the difference between a stock's 26-day and 12-day moving averages; a 9-day moving average of MACD is then computed, acting as a signal for buying and selling at zero-crossing. The following expression computes non-empty values as signals for buy decisions using the formula in this paragraph:

$$BUY = \text{SEL}_{>0}(\text{MAVG}_9(\text{MAVG}_1\, 2(S) - \text{MAVG}_2\, 6(S))).$$

In a financial application, such complex queries with statistical operators are executed on very long input series. As an example, a year of stock quotes at minute resolution would contain over 180 thousand entries; at 15 second resolution this stretches to 734300 entries for one year. Typical queries may run over intervals of decades, and often such a query must be applied to dozens or hundred of stocks (i.e., separate time series) at once. In consequence, the ability to run complex queries with high efficiency is very important.

5.3 Time-Series in XQuery

We adopt a very simple XML schema for representing time-series: a `timeseries` element contains a list of `time` and `value` elements, alternating between the two. In our financial-application tests, `time` elements are interpreted as `xs:dates`, and values as `xs:double` for stock prices. Such a simple representation allows us to write optimized operators for manipulating time-series with relative ease. Data in different formats can be easily transformed to this schema; for example, representations with implicit time, i.e., series of values associated with a starting time and a sampling granularity can

```
declare function local:mavg($ts as ts:document,
    $i as xs:integer) as ts:document {
  <ts:document> {
    for sliding window $w in $ts//ts:value
      start at $s when fn:true()
      only end at $e when $e-$s eq $i-1
    return <ts:timeseries>
      {(data($w/preceding-sibling::ts:date))[$i]}
      <ts:value>{avg(data($w))}</ts:value>
    </ts:timeseries>
  } </ts:document>
};
```

Figure 5.1: The MAVG operator implemented as an XQuery 3.0 function.

be transformed by iterating over the values and computing the time moment for each.

Based on this schema we developed XQuery 3.0 functions that use `for window` clauses to implement the operators defined above. Refer to Figure 5.1 for an example showing an XQuery implementation of the MAVG operator. Such a declaration can be used in any XQuery 3.0 processor. In XQ2P however we augment the execution environment with functions that achieve the same effects but are implemented in Java; their code can be hand-optimized to take advantage of the known schema and achieve much higher performance in the inner loops than the standard XQuery operations. (Recall that XQ2P lacks general support for XML Schema, and thus cannot yet perform such optimizations automatically.)

Using the provided operators one can compose more complex operators and strategies in using a declaratively. For example, the MACD-based strategy used as an example at the end of the previous section could be implemented as in Figure 5.2.

5.4 Time-Series over a DHT

Because individual time-series can be very large—tens of GBs is not unusual—relying on a single peer to handle even a single time-series in a peer-to-peer application can easily lead to overloading. For this reason we

```
let $doc := doc("lvmh-quotes-ts.xml")/document
let $mavg12 := ts:mavg($doc, 12)
let $mavg26 := ts:mavg($doc, 26)
let $sub := ts:msub($mavg12, $mavg26)
let $macd := ts:mavg($sub, 9)
return <ts:document> {
  for $ts in $macd//ts:value
  return <ts:timeseries>
    {$ts/preceding-sibling::ts:date}
    <ts:value>
      {if ($ts > 0) then "buy" else "sell"}
    </ts:value>
  </ts:timeseries>
} </ts:document>
```

Figure 5.2: The MACD-based strategy described on page 104. The optimized functions implementing time-series operators are provided in the dynamic environment under the provisional namespace `http://www.prism.uvsq.fr/dim/ts/`, bound by default to the prefix `ts`. `ts:msub` is the linear subtraction operator, computing the pairwise difference of two timeseries' values at each point in time. Although XQ2P does not have generic XML Schema support, the same namespace is used to detect elements representing timeseries, triggering timeseries-specific behavior.

implemented a horizontal-partitioning scheme for time-series data, spreading responsibility for a time-series using the DHT.

When a peer node wishes to share a time-series document, the following steps are performed: First, the time-series is split into a sequence of segments of fixed size, e.g., 1024 entries; when needed, the last segment is padded with `?` values. A fixed amount of overlap is introduced between segments (e.g., 128 values at each end); this allows performing many windowing operations on a segment without needing to access adjacent segments. The time-series is then published over the DHT; see Figure 5.3.

To publish a time-series, two types of key-value pairs are stored in the DHT. First, the identifier of the time-series[5] is associated with metadata that declares

[5]Currently, time-series are simply identified by an URI denoting the document they were extracted from. A single time-series can be stored in one document.

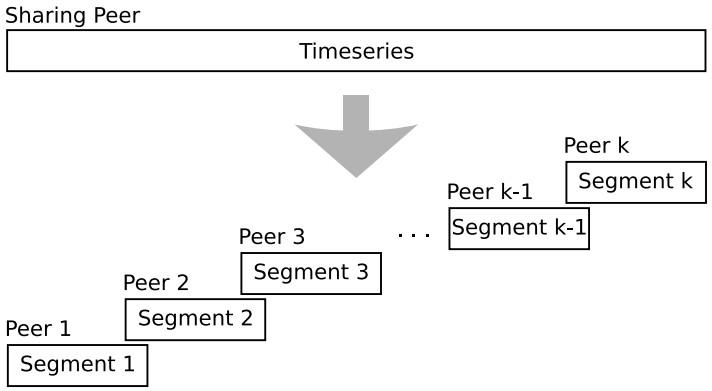


Figure 5.3: Splitting a time-series in segments over a DHT. Note that the numbering of peer nodes and the placement of the segments in the diagram are sequential only for readability and to make obvious the overlap of successive segments. In practice the key-hashing function of the DHT will distribute the segments randomly throughout the overlay.

its length (this could be extended with further information), and a list of cache entries that will be discussed later.

Second, a tuple of the time-series identifier plus a segment index is used as key to store each segment. Peers that need to access a time-series will first contact the node holding the metadata to obtain the length of the time-series and thus to determine how it is segmented (this step also implicitly determines the existence of the time-series). Then it can contact the peer nodes holding the segments it is interested in. Note that using both the time-series ID and the segment index as a key causes segments to be randomly distributed throughout the overlay network due to the uniform-hashing used on keys; this provides a load-balancing for storage resources.

In cases where one or a few time-series, or even only some segments, are very popular—i.e., they are accessed by many peers—this scheme could cause disproportionate network load on the nodes holding the popular segments. We add a segment-specific cooperative caching scheme to address the issue: peer nodes that access and download a segment also cache it for a certain period of time, and they announce this fact by entering their ID together with the index of the segment in a cache entry held together with the time-series metadata;

when segments are evicted from cache the entry is removed.[6]

With the cache in place, the retrieval algorithm is altered: whenever a peer wishes to retrieve a particular segment, it searches the cache entries for peers that hold that segment, and asks one of these at random. If this fails, the peer will fall-back to the procedure described above and contacts the "canonical" owner of the segment. (Canonical peers also register themselves in the cache, so that the random selection distributes load equally to them even for cached segments; the fall-back is usually necessary only in the short time after publishing, before the canonical segment owners have time to update the cache entries.) The fact that peers cache segments they use leads to more popular segments being more widely cached; picking between caching peers at random ensures network-load–balancing between them.

5.5 Distributed Computation

The reader might have noticed that the program in Figure 5.2 computes several transformations of the initial time-series by applying MAVG operators to it. This situation is quite common. Observe that, although the elementary operations of a moving-average computations are relatively simple, applying such an operator to a large time-series can be very expensive simply because of the length of the timeseries (requiring many iterations of the elementary function).

However, executing a windowed operation can be executed in parallel on slices of a time-series; the elementary operations applied to each window do not depend of the results on other windows, except indirectly through the overlapping elements in the source data.

In the previous section, we mentioned that time-series are split in *overlapping* segments. The overlap allows computing the results of a windowing operation on a time-series in parallel, as long as the window length is shorter than the overlap.[7]

---

[6]There are many improvements that could be applied to this caching scheme; for example, a peer can usually anticipate which segments it will cache and announce this to the metadata-storing peer in the same message with which it requests a time-series' metadata, to reduce the number of message exchanges. In the interest of simplicity, and because the large volume of time-series lowers the relative cost of individual messages, we did not implement such optimizations.

[7]Observe that, given a segment of a time-series, operations with window size w can only be performed starting at the wth element of the segment; the first w elements of the result require data from the previous segment.

We take advantage of this feature: when a peer requires, either as a complete query or as an intermediary result, a time-series derived via a windowing function (of sufficiently small window size) from an original time-series, it will initially proceed as above—i.e., it locates peers holding the segments it needs of the original time-series—but will then request the *modified* time-series. Thus a peer that holds segment s of time-series S can be asked for segment s of the derived series, e.g., $MAVG(S, w)$. It will compute this segment using the locally-stored input segment and reply directly the computed results.

By issuing such a request in parallel for all segments of a derived time-series it needs, a peer can in effect cause it to be computed in parallel; the number of computing peers increases together with the number of segments, which can make operations on long time-series occur in time comparable to those on segment-length series.[8]

The cache system described above can be extended to explicitly support derived queries: peers that compute a derived query can simply cache the results.[9]

[8]Communication costs increase, however, linearly with the number of segments.

[9]Peers that do this could send a cache entry to the metadata node *of the original* time-series (S in the previous example), keyed by the segment index *together* with the expression used for the derived time-series. Since a node that needs a time-series first retrieves its metadata, it can determine which derived series are cached by another peer node, and can choose to contact those retrieve those rather only nodes with the source time series, and to prefer peers without cached results for derived segments it can't find in the cache. However such detailed cache control is beyond the scope of this work.

Chapter 6

Related Work

Situated at the confluence of three large domains—databases, XML processing, and peer-to-peer networks—our work on XQ2P touches many established areas of research. It would be impossible to include here a review of all or even a major fraction of all related contributions. Instead, we focus only on those that are conceptually closest to this work.

The area of databases—and, in particular, relational databases—being the oldest and best established, we refrain from exploring it in any detail; instead, we will mention only publications that are directly connected to this paper. The interested reader might consult such works as [29] for a thorough treatment of the subject. Similarly, we keep silent on document-oriented work on XML, focusing on its use in peer-to-peer databases.

The focus of this work is *querying*, i.e., identifying, locating and retrieving some subset of the total data available. Accordingly, in §6.1 we identify and discuss the *Data Models*, the kinds of data a data storage system can hold and, most importantly, the *Query Models* they offer, which allow the user to specify *what* data is needed. In §6.2 we compare the main *Retrieval and Indexing Schemes* that allow a storage engine to efficiently retrieve the queried data. Structural indexes, corresponding to the scheme used by XQ2P, are described in more detail in §6.3. §6.4 presents protocols for key-based routing and distributed hash tables, of which the Chord variant used by XQ2P is an particular case. Lastly, §6.5 presents some other approaches to testing and simulating peer-to-peer networks.

6.1 Data and Querying Models

The least sophisticated data model might be called the *"labeled blob"*: in this model, the storage system is not aware of the content of the data it stores except

via limited metadata, such as a label or a key.

The blob model allows only a very restricted query model, retrieving the block of bytes associated with a given retrieval key; this is commonly called *key lookup*. Slightly more complex variations are possible: in some cases, more than one kind of metadata may be used, such as alternate labels, version, last modification date, or blob size may be used in querying, and the user might be allowed to demand only a sub-part (e.g., a set of byte ranges) of a blob.

Even such a simple model is tremendously useful. All DHT systems fall in this category, and file-sharing applications[57, 21] are examples of very successful peer-to-peer applications using it.

Slightly more advanced is the *filesystem model*, where data is stored mainly as opaque blobs, but where the labels or retrieval keys have structure of which the storage system is aware. Such systems allow the more sophisticated querying model of asking for the blobs whose labels satisfy more complex conditions than just equality with a user-supplied value. Filesystems allow such querying models as identifying (and retrieving) groups of files by a single key (e.g., directories), retrieving files via key substrings, keys with wildcards, or even regular expressions on the key. Common features of such systems is that they are hierarchical, i.e., groups of files (folders) have a key (or name), and the files they contain are prefixed by that key; almost always folders are implemented as special kinds of files that the storage system can read. Indeed, such systems are commonly implemented on top of a simpler labeled-blob layer. Most distributed filesystems follow this model; see [34] for a survey.

We label *text search engines* those storage systems that can interpret the data they contain as strings of symbols. This allows a query-by-content model, where the user can request files that contain certain strings. This might mean asking for data that contains one or several substrings, similarity searches or even regular expressions.[1] The most common query model added by text engines is that of *keyword* searching; often some type of *aggregate* query is also available, e.g., counting occurrences of a keyword. Such systems are usually said to be *unstructured data* systems: Although strictly speaking the data is structured as strings of symbols, most of its structure and meaning (e.g., sentences in the case of document storage) is not apparent to the storage engine itself. See [64] for a proposal of a peer-to-peer web search engine that includes

[1]In common parlance the term *full-text engine* refers to more advanced systems for querying human-readable text documents; the term usually implies such language-specific features as case-insensitivity, word-splitting or stemming, and similarity or phonetic searching. Conceptually, though, systems that do not handle text as such, e.g., a protein database, could be classified in this same category. The only difference is how human users interpret the symbols.

a version of the PageRank[54] algorithm.

Systems that allow to express the structure of data and query using such structure—as well as the values—are said to use *structured data model*.

In the *relational data model* information is structured as relations, usually represented as tables or rows of tuples; databases using this model are termed *relational databases*. The model allows querying with languages based on relational algebra, the most popular of which is SQL and its many variants. Such languages have powerful expressivity, including such concepts as data types, selection with complex predicates, projections, joins and aggregation. PIER[56, 37, 38, 33] is a peer-to-peer query processor, based on a DHT, which uses a relational model. PeerDB[53] and AmbientDB[8] are other examples.

The *object graph data model* represents entities as labeled nodes of a graph and relationships between them as labeled oriented arcs between these nodes. Primitive values such as numbers or text strings are usually represented as special valued nodes attached to the entity they describe. Systems with a graph data model tend to use query models based on *paths*. A path is a sequence of relationships that describes how to reach from a node to another by following relationship arcs (paths are usually defined to apply on collections of nodes, e.g., going from an initial collection of nodes to that of nodes reachable from them via a certain arc); filtering constructs (e.g., predicates) and set and sequence operators complete these languages. One notable example of such a data model is RDF, the Resource Description Framework. Edutella[52] is a system that indexes RDF schemas and queries them in it specific query language. [69] proposes a system for routing queries on RDF/S fragments over a DHT.

Trees are an important special case of graph; accordingly, the *tree data model*, which represents data as labeled nodes connected by ancestor/descendant relationships. (The exact semantic is often implicit in the type of nodes, or expressed in a separate ontology.) Many-to-many relationships are usually not expressed through structure but by data. Tree data models also tend to rely heavily on paths.

XML is the most popular example of a tree data model. In the *XML* model, data is structured as labeled trees, with untyped (text) or typed (e.g., numeric) values as leaves. XML has become very popular, especially in distributed contexts and over the Web, among other reasons because it is well suited to express data with high heterogeneity and widely variable structure; especially relevant to us is that its tree structure allows query languages that are expressive, powerful and often more intuitive than a more strictly structured model like SQL, while remaining much simpler than most object-graph–based models.[2]

[2]Concepts like grouping, ownership, and composition are often easier to express

The canonical language for querying XML is *XPath*. As its name suggests, it is a path-based language: its fundamental operation is navigating the ancestor-descendant relationships between XML nodes. *XQuery* is a superset of XPath; it uses XPath expression syntax to refer to parts of an XML document, and supplements it with "FLWOR expressions" (the acronym stands for the keywords for, let, where, order by, return) to perform SQL-like joins, as well as other constructions.

Several projects address the problem of querying XML databases using XPath, or, more often, specific subsets of it. XQ2P attempts to support the entire XQuery language, optimizing as much of it as possible, in a top-down manner.

6.2 Indexing and Retrieval Schemes

Most practical queries are intended to retrieve a small amount of data—the results—from what is, in general, a large database. To achieve high (or even acceptable) performance, databases almost universally use indexes and carefully designed schemes of accessing these indexes to answer queries by examining as little as possible of the data they hold.

The problem is manifests itself even more acutely in the case of distributed data management systems: because the data is widely spread throughout the network, the cost of accessing unnecessary parts of it are much higher. Appropriate querying schemes vary, of course, depending on the data model and the network/distribution system.

The earliest peer-to-peer systems, such as Napster,[87] eschewed the issue by relying on a *centralized server* to index and perform queries, using their peer-to-peer features only after the necessary data (i.e., the node owning it) was located. Various reasons, not all of them of a technical nature, pressured developers towards less centralized solutions; that said, the semi-centralized model continue to exist in such systems as the extremely popular BitTorrent[57] system, where "trackers" coordinate "swarms" of peer nodes, and the earlier eDonkey[35] network.

The first fully distributed systems, like Gnutella,[62] relied on *"flooding"* a query through the unstructured connections between the peers: basically, a query is sent to all peers a node knows about, who either answer if they have results or propagate the query further to their own peers. Even for the simple

---

and especially visualize in terms of ancestor/descendant branches than in terms of tables.

data model of these networks (queries consisted mostly of keyword-search on file names), such an approach was considered insufficiently efficient.[17]

The introduction of *key-based routing* over overlay networks, and the distributed hash tables they made possible, was a major advancement. Such networks allow executing key lookups efficiently; in addition, such systems provided features like fault tolerance, load balancing and reliability. Several of them, including Chord,[71] Pastry[63] and CAN,[71] are described in more detail on page 119.

DHT networks provide a restricted model for data representation and querying. Due to their simplicity and efficiently, however, they have been used as building blocks for more complex storage systems.[3] Many higher level systems store indexes and metadata in a DHT sub-layer and provide more advanced services based on them.

As an example, a distributed text-search engine might store an inverted index in a DHT, using words or terms as keys and the names of documents containing those words as values.[44, 6] A keyword query can be solved by first retrieving the set of document names containing the desired word (intersecting the sets if more than one keyword is searched for), then retrieving only those documents. (Other local text indexing techniques can be similarly adapted to DHTs.)

Systems with a structured data model also use indexes stored on an overlay network to solve queries expressed in their high-level models. In general, all such systems use a variant of the following two-step technique: First, the index is queried to determine the set of or peer nodes containing data that is potentially relevant to the result. Then, the original query is forwarded to the peers found in the first step, which compute partial results locally and return them to the original node. In a few cases, for example existence or counting queries, the first step might be sufficient to determine the final result, and the second step is not executed. Note that in most cases the index cannot determine with certitude that a document is relevant for a particular query; in general the index is queried conservatively, which leads to some number of false positives. Since false positives cause unnecessary connections and communication, an index's selectivity for the kind of queries supported by a system is very important for efficiency.

Many DHT-based indexing schemes have been proposed for XML indexing. We concentrate on the following approaches, listed by the choice for key/-value:

[3] In fact, several have been developed specifically for this purpose.

Tag Name/Paths [27] discusses using tag names as keys to a DHT, with values set to the list of paths that tag can be reached from in each document where it occurs. For example, given `<a><b><c/></b></a>`, key `c` would denote a list containing `docid/a/b/c` (with docid a unique identifier for the indexed document). The system can easily identify documents answering linear ancestor/descendant queries, and with some processing of the path list supports embedded wildcards. The method has low specificity for twig queries: it can only identify documents containing each embedded path, but not those containing the paths at the same time. There is no direct support for other axes (e.g., ancestor or sibling). [27] suggests using attaching other information ("summaries" in the paper's terminology) to each node, e.g., the list of all paths descending from the node, but they do not discuss the rapid escalation of such lists needed to support large parts of XPath. Indexing of text content is mentioned but not elaborated upon.

Path Suffixes/Path In [70] the authors list every path occurring in a document. They store the path, together with a document ID, using as keys all its suffix subpaths; for example, path `a/b/c` would be stored with keys {a/b/c}, {b/c}, and {c}. P-Grid is used as the underlying routing primitive, which uses an order-preserving hash to construct a trie. The system attempts to answer all linear descendant paths, including uses of wildcards: the query path is divided in segments containing only the child axis (i.e., without `//`) separated by descendant axis calls, after replacing wildcards with descendant axis paths. The index is queried for the longest child-axis-only segment; if it is the last segment, the search trivially terminates here; however, if the segment is not last, the receiving peer must flood the subtree rooted at itself to filter the results, a costly operation. [70] proposes a caching system to reduce the need for flooding. Like [27], the index has no specificity for twig queries.

Document Signatures/Document URIs *psi*X[60] proposes a method of summarizing XML documents: For each document *psi*X builds its *structural summary graph*;[4] a signature is constructed from the SSG based on irreducible polynomials over a finite field: each edge in the SSG of the document collection is assigned a distinct irreducible polynomial, and the document signature is obtained by multiplying the polynomials of the edges it contains. A similar scheme is used to assign signatures to twig patterns; algebraic operations on

[4] A directed graph containing all distinct node labels as vertices, linked by arcs if and only if a parent-child relationship exists between two nodes of those names in the summarized document.

the signature polynomials can be used to determine if a document signature is compatible with a query. The authors construct an tree-based structure called an *H-index* on top of a Chord DHT to store these signatures; the document URIs are used as values. The authors of [60] further extend this with value indexes, summarizing values in a manner similar to Bloom fields. Note that the *psi*X system is specific only to the presence of pairwise parent-child relationships: it can tell that a document contains, e.g., a/b and b/c edges, but it cannot test for $a/b/c$.

Paths/Document Fragments XP2P[9] proposes a system where documents are split into fragments, which are pushed into a DHT (Chord) using the paths of the fragment in the original document as key; the value is the XML fragment (a sub-tree of the original document), together with the path of its parent fragment and the paths of its children fragments (if they exist). Note that, despite using Chord, XP2P uses a *path fingerprinting* technique rather than hashing to generate keys; this puts into question the validity of the usual guarantees Chord provides.[5] The work focuses on executing linear path queries over such fragmented documents rather than on specific algorithms for fragmentation.

Tag Name/Structural Identifiers Such systems use region encoding to generate structural identifiers for each occurrence of a tag in a document; the list of positions for each tag is recorded in the DHT, using the tag name as key. Systems based on region encoding aim to take advantage of several efficient structural join algorithms[30, 20] to match twig pattern queries against documents; such algorithms can provide very high specificity when used to filter documents for relevance against a large fraction of structural queries. XQ2P uses this index and a variant of TwigStack for query resolution. KadoP [3, 4] uses this same model; it refers to the list of structural identifiers as a "posting list". KadoP also uses posting lists for individual words in documents, allowing the index to solve keyword queries against text content; it uses a conjunctive XML tree pattern query language, and also supports semantic annotations and querying. [58] presents the DPP structure, which horizontally partitions long posting lists and distributes them among peers to increase load balancing and parallelism. Several types of structural indexes developed for non-distributed databases, many of which are adaptable to this scheme, are collected in the following section.

[5]The possibility of using an overlay not dependent on uniform hashing is not discussed by the authors.

A slightly different approach is proposed in [24]: Instead of a DHT, the hybrid peer-to-peer system MediaPeer[25] is used to index path-sets in a trie (which uses tag names as symbols) managed by super-peers. Nodes add to the index every path existing in their shared documents. A particularity of this system is that, instead of first locating relevant peers and then querying them directly, individual path queries are routed through the trie to peers, which then respond with relevant data to the node that originated the queries. Tree-pattern queries are decomposed into individual path queries, and the final result is assembled by intersecting the partial results for each path. An optional value-indexing service may be used to asynchronously index values (the text content of XML nodes), again using the trie structure.

6.3 Structural Indexes for XML

Several attempts have been made to map the structure of the XML documents into relational structure. A basic approach would be to create a relation (or class, for object-oriented databases) for each kind of element; a different schema is needed for each XML type; tree structure is represented as child-to-parent pointers, implemented by foreign keys between relations. [67] is an example of this approach in SQL, as is [2] for OODBs. Such techniques are difficult or impossible to apply for semi-structured XML and documents the schema of which is not known in advance.

The *model-mapping* approach represents directly the tree model of the stored XML documents. In principle, a single schema would represent all types of elements, from all types of documents; the type of each element is represented as an attribute rather than a separate relation. [26] presents several variants of using a single relation to represent all parent-child edges in the XML tree; structural relationships are queried by joins.

Such approaches encounter serious limitations when applied outside of a restricted set of query types. For example, queries on structure other than parent/child are difficult to express. Path and tree-pattern queries require one structural join for each step, which can generate large amounts of unused intermediary results; about information the query optimizers of most relational database engines have little opportunity of ordering joins optimally in complex queries. In addition, document-ordering constraints of such languages as XPath require complex and costly ordering steps.

More efficient indexing techniques make use of *node numbering* schemes to express tree structure instead of explicitly encoding parent-tree edges. Such schemes in general construct and inverted list for each node type (e.g., element name) holding one or several numbers reflecting the position of element open-

and end-tags or its level in the tree (i.e., the distance from the root node); many variations are possible, differing on such features as what kind of relationships can be determined via joins or efficient support for updates. This type of index allow expressing containment queries via a single join using *inequalities* between the numbering of elements, rather than eq-joining element ids to determine parentship.

[84] compares implementations of this approach in XML-specific engines and using a RDBMS. XRel[83] is another encoding that takes advantage of usual RDBMS indexes and join algorithms for XPath processing using numbering. [20] proposes structural joins taking advantage of B+-indexes of region-encoded nodes. [32] introduces the *staircase join* as an addition to a RDBMS kernel that takes advantage of tree structure to perform structural joins more efficiently. [42] proposes a new stack-based algorithm to break twig-queries into a set of binary join components.

A problem with early structural join techniques is that they generate a large amount of intermediary results; such methods consider each pair of structural relationships separately then merge the results; in the case of relations that occur often in documents, large intermediary lists are generated only to be filtered later.

[12] proposes PathStack and TwigStack, as well as a specific indexing structure called XB-tree and corresponding improved TwigStackXB algorithms; the _Stack family consists of *holistic* join algorithms which avoid producing large intermediary results by joining all pairs of relations in parallel rather than sequentially. A large number of improved variants of holistic join techniques have been proposed subsequently: [40, 39] introduced the XR-tree index and the TSGeneric+ algorithm; [47] improves handling of parent-child edges with TwigStackList; [47] and [48] introduce the *extended Dewey* encoding and the TJ-Fast algorithm for accessing only leaf elements; Twig^{2}Stack[19], TwigList[59], PathStack¬[41] are other improvements of holistic algorithms.

6.4 KBR and DHT protocols

This section describes several of the existing DHT protocols. A simplified variant of the first we present, Chord, is used by XQ2P; see page 26 and page 82.

Chord: One of the first DHT protocols proposed, Chord[71] organizes its overlay using keys of 2^m bits, organized as a circle: keys range from 0 to $2^m - 1$; the normal succession relation, i.e., 1 succeeds 0, 2 succeeds 1, ..., is extended with the relation 0 succeeds $2^m - 1$ to complete the circle. In practice the SHA-1 algorithm is used to generate keys, which limits m to a generous 160, but the

algorithm is specified in general for any sufficiently large m. In principle the hashing function can be changed, the algorithm's properties depending only on the hash function being cryptographically secure (non-invertible) and its distribution of values uniformly.

Nodes are assigned ids by hashing their network address, and values are assigned keys by hashing their descriptor (e.g., file name), using the same hash function. Because the hash function is uniform, both node ids and keys are distributed uniformly around the key circle. The hash function is used to implement *consistent hashing*, which ensures that adding or removing one hashing slot does not greatly change the mapping of keys to slots: Chord defines `successor(k)` to be the first node whose identifier succeeds `k` in the circle,[6] Each key `k` is owned by `successor(k)`; the uniform hashing ensures that ownership of keys is fair, i.e., on average nodes own a similar number of keys, and use of consistent hashing means that a peer joining or leaving the network does not require the reassignment of any keys other than those held by a few peers; on average, $O(K/N)$ keys change hands for each node change, where N is the total number of nodes and K the total number of keys.[7]

Each node knows the address of its successor in the ring; this invariant is sufficient to find the location of any key, by simple linear search along the successor chain. However, this would be very slow. Chord requires each node to maintain a "finger table",[8] containing the addresses of successor$((n + 2^i) \bmod 2^m)$ for $i = 0..(m - 1)$; there are at most m such addresses held, i.e., the number of overlay connections is bounded by $O(\log N)$. Using always the farthest "finger" that precedes the key needs when searching at least halves the space of remaining nodes, which ensures that finding the owner of a key is $O(\log N)$.

Pastry: Another one of the originally proposed DHT algorithms is Pastry[63]. Similarly to Chord, it uses a hashtable with a circular keyspace, however with 128-bit keys. Node IDs are assigned randomly and uniformly; this ensures that nodes that are close in ID values are diverse geographically.

Pastry is capable of using a scalar *routing metric*, an indication of some

[6] Usually, the smallest k' such that $k' >= k$, except near the origin, around $2^m - 1$ and 0.

[7] In the case of a leaving peer, only the responsibility for its keys needs be transferred; in case of a joining peer, it needs only take responsibility for, on average, half the keys owned by its predecessor.

[8] As long as at least one other peer is known, a Chord node can always reestablish the correct table. It is enough to check the addresses in the table periodically to maintain connectivity.

sense of "closeness" between nodes in the physical network's geography. The metric is not fixed and can be supplied by an external program; examples of useful metrics are network latency, number of IP-routing hops, or available bandwidth between the nodes.

The 128-bit keys are logically represented as a string of digits of b bits, i.e., represented in base 2^b. For example, with a typical value of $b = 4$ the keys are represented as 16-digit numbers in base 16. From each peer's point of view, this allows partitioning the keys into levels: level 1 consists of all keys that share a one-digit prefix with that peer's ID, level 2 of all keys with a shared two-digit prefix, and so on (level 0 contains all keys that have a different first digit, i.e., those sharing a zero-length prefix with the node's ID).

Each node maintains three tables of addresses. The leaf node list consists L nodes, the $L/2$ closest peers by ID in each direction on the key circle. The neighborhood list consists of the M closest nodes in terms of the routing metric. The routing table contains, for each key level, and for each possible digit at that level except that in the node's own ID, the address of the closest known peer (in routing metric terms) that has the corresponding prefix, resulting in $2^b - 1$ contacts per level, with the number of levels proportional to $\log_2 N/b$; L and M are usually of the order of 2^b.

A packet can be routed to any address in the keyspace; the peer whose ID is closest to the given key will receive it. When routing towards a key, a peer first examines its leaf node list (and itself), and routes directly to the destination if a match is found. If that fails, the peer will search its routing table for a node that shares a longer prefix with the key than it itself does (the longest common prefix is preferred if several such nodes are found). If there are no contacts with a longer prefix, the node will choose a contact with the same prefix length but with an ID numerically closer to the key. (There is always such a node, otherwise the routing node is the destination and routing would have ended at the first step.)

The neighborhood list is not directly used during routing; instead, it is used to maintain the routing table.

Similarly to Chord, on average Pastry maintains $O(\log N)$ routing entries and requires $O(\log N)$ steps to route a message. However, the use of a routing metric allows Pastry to exploit network locality, and each step is potentially less costly.

Tapestry: Similarly to Pastry, Tapestry[85] uses prefix routing with links prioritized by a routing metric like latency or network locality. Like Chord, it uses SHA-1 to generate a 160-min identifier space; the keys are represented as 40-digit hexadecimal values, thus dividing the keyspace into "levels" like Pas-

try does. An interesting feature of the protocol is that it explicitly uses application IDs to allow several applications to share an overlay, which increases efficiency.[9]

Kademlia: Described in [50], Kademlia uses 128-bit IDs and keys; keys are generated via hashing and node IDs are random. The protocol defines a logical distance calculation: between any two IDs, it is defined to be the Exclusive Or (XOR) of their IDs, interpreted as an unsigned integer.

Each Kademlia node maintains 128 lists of nodes (i.e., one per bit of ID); list n will contain peers whose IDs match the node's first $n-1$ bits and a different nth bit. Notice that with respect to the XOR metric defined above, all peers held in list n are *further* away than any peer in list $n+1$ and closer than any peer in list $n-1$. Kademlia literature refers to the lists as k-buckets; k is a predefined constant (e.g., 10 or 20) and represents the maximum length of such a list. Note that for progressively smaller distances there are progressively fewer candidate nodes, due to the fixed length of IDs; this means that each node will have very good knowledge of its neighborhood, i.e., the k-buckets with large k will be exhaustive.

Node membership in each list is very dynamic; each node encountered via normal network operations is considered for inclusion in the list, according to certain heuristics. Kademlia favors long-lived nodes; when a new candidate is available for a full list, it will only replace an old node if it has stopped responding. Note that the fine-grained distance *within* a k-bucket is *not* a factor in the heuristics.

When searching for a peer with a certain ID, a Kademlia node will query a certain number (3 is a common value) of those nodes it knows that are closest to the searched ID; the target nodes will respond with their closest k nodes, and the querying node will update its k-buckets (keeping the best ones, i.e., closest to the desired key, that respond to messages) and re-iterate. Values are stored at the k closest nodes to their hashed keys to provide redundancy; nodes that store values explore the network periodically and replicate the values, to compensate for disappearing nodes.

Searching for values is done in a similar manner: peers with IDs closer than the searching node are successively interrogated, thus progressively diminishing the distance to a node that would hold the value's key. Kademlia also supports *accelerated lookups*; this is done by extending the routing tables beyond single bits, adding k-buckets for groups of bits, which function similar to Pastry's "digits".

[9] This is possible for other DHT networks, too, but defining the feature in the protocol lowers the barrier for cooperation between applications.

CAN: Unlike the examples above, the *Content Addressable Network*, or CAN[61], uses a virtual multi-dimensional toroidal Cartesian coordinate space to organize its overlay network. Given a number of dimensions d, d numerical coordinates can be used to specify a point in this coordinate space. The coordinate space is partitioned among peers dynamically by *splitting*: on joining the network, a node picks a random point in coordinate space and attempts to take ownership of the block of space ("zone") surrounding that point. To do so, it communicates with the node currently owning the zone that contains the chosen point; an attempt is made to split the zone in two halves along one dimension,[10] and ownership of one of the new sub-zones is passed to the joining node. If the join attempt fails for any reason, the joining node will simply pick another random point and attempt to join again.

Each connected node maintains a routing table holding the coordinates of the zones adjacent to its own and the IP addresses of those zones' owners (the node's neighbors). In CAN, messages are routed to points in the coordinate space; the points' coordinates function like the keys in circular-space DHTs, and are similarly obtained by hashing. To route a message towards a point, each node simply forwards the message to its neighbor whose coordinates are closest to the target point; when the message reaches the node that owns the zone (volume) which contains the target point, it has reached its destination; in case of failure of the closest node, the procedure simply falls back to the next-closest neighbor.

For a given d-dimensional space with N nodes, the average routing path length is of the order $O(dn^{1/d})$, and each node maintains a routing table of size $O(d)$.[11] The exact value of d can be chosen by each application, achieving different balances between routing state and average path length. In addition to varying the number of dimensions, CAN also allows the use of multiple,

[10]The dimension picked cycles between dimensions, to maximize the ease of merging zones in case of nodes leaving the network, e.g., in a three-dimensional space splits happen along the x coordinate, then y, then z, then x again.

[11]Although CAN seems fundamentally different from the circle-key approaches described above, the differences may be deceiving. The key-circle in fact forms a one-dimensional torus; Chord routing with only successor nodes or Pastry routing using only $L = 1$ leaf node tables is analogous to $d = 1$ CAN routing. Multiple-dimension CAN is analogous to "folding" the Chord circle; where Chord skips large portions of its one dimension, CAN "goes around" through its separate dimensions. Pastry splitting its key in 2^b-sized levels is analogous to partitioning a point's coordinates in dimensions, albeit the resulting logical "zones" and the path taken by a routed message are slightly different. CAN has the same scaling properties as the circle-key protocols if d is scales like $(\log_2 n)/2$.

parallel coordinate spaces, called *realities*; each node owns a different zone in each reality, and the contents of the DHT are replicated in each reality, thus increasing data availability. There exist also variants where zones are shared between more than one peer (called *zone overloading*), and the possibility to use several hash functions to map each key onto several different points.[12]

P-Grid: Distinct from the other entries in this section, technically P-Grid[1, 66] is not quite a DHT. We describe it here because it is conceptually very close, and despite its differences it can replace a DHT in most applications. P-Grid uses prefix-routing over an overlay topologically similar to Pastry: Keys are treated as binary strings; Peers are responsible for a partition of the key space based on key prefixes, referred to as "paths"; Paths associated with peers are of equal length, thus building a balanced binary tree with peers at leaf nodes; Peers maintain routing tables in levels, with exponentially increasing distance in key space from themselves (i.e., a peer with path "000" will keep in its routing table links to peers with paths matching with "1*", "01*" and "001*"). P-Grid nodes hold several alternative nodes at each level in their routing table and pick randomly among them during routing, which allows it to achieve a logarithmic expected search cost even though the data distribution over the tree may be imbalanced.

The main characteristic of P-Grid that distinguishes it from pure DHTs is that it uses an order-preserving hash function. The initial publication defines a mapping for strings in the following manner: Using a database of sample strings, it constructs a balanced trie: the database is partitioned recursively into equal sections, until each partition is smaller than a threshold. When searching for a given string, first the string is "searched" for in the trie, appending "0" to the key for left turns and "1" for right turns. The resulting binary string is then searched for in the overlay network. If the distribution of the sample strings resembles that of the actual data, the distribution of values in the network will be balanced.[13]

---

[12]Technically, these variations are also possible for the other DHT protocols, and some have been explored. It would even be possible to combine several protocols, using for example a Chord reality and a CAN reality in the same application.

[13]Because the distribution of strings—or other hashed data—in different databases will not be similar in general, the keys generated via this procedure are application-specific. However, a single P-Grid overlay can be easily used by several applications, as long as each uses its own sample-data trie (i.e., based on its own data distribution), they all have the same maximum key-string length, and the protocol can discriminate between requests from individual applications.

Using this type of order-preserving hashing allows P-Grid to efficiently search for ranges of data: values that are close by (e.g., succeeding values) will be either handled by a peer with the same path, or by peers with "successive" paths; the routing layer efficiently supports finding peers with lexicographically successive paths via the longest applicable "prefix levels" (in fact, it can do so both ways, i.e., one can also search for lexicographically preceding paths).

6.5 Testers and Simulators for P2P Systems

Several projects attempt to assist researchers of peer-to-peer applications to test their results. Below we present some of them, summarizing their goals, design choices and features.

PlanetSim

PlanetSim[28] is a simulation framework for overlay networks and services. It presents a layered and modular architecture, intended to allow a developer to choose the lever on which they need to work: creating and testing new overlay algorithms (e.g. Chord or Pastry), or creating and testing new services (e.g. distributed hash table, multicast, distributed object location and routing) on top of such overlays. PlanetSim is developed at the Architecture and Telematic Services Research Group at Universitat Rovira i Virgili in Tarragona.

PlanetSim also attempts to allow easy transition from simulation code to experimentation code running in the Internet. This is done by hiding the underlay network in wrapper code that takes care of network communication; this way the same code can be run unchanged in the simulator and in network test-beds such as PlanetLab. A common API for structured overlays is used by the distributed services. This enables transparency of the running environment (either the simulator or the network) for the running services.

PlanetSim has been developed in the Java language. It enables running scalable simulations in reasonable time on a single machine. It offers two pre-implemented overlays (Chord and Symphony) and a variety of services, including multicast and DHT, to serve both as example implementations and to test the framework itself.

Simplified, PlanetSim's architecture consists of three layers. At the top is the application layer; it is here that the various (tested) services are implemented. It communicates via a common API with the middle layer, that of the overlay network, which handles basic routing functionality. In the case of PlanetSim, this module offers the sole service of KBR (key-based routing). The common API between the application level and the routing (overlay) level is a

small set of interfaces, designed to allow "plugging" different routing modules (Chord and Symphony are provided), and to allow the application to use them transparently. The API allows the application level to demand basic routing-related operations (send, receive, broadcast) and to give some guidance to the routing layer.

At the bottom is the physical network layer. However, the routing layer does not communicate directly with the network; instead, a simulator module is interposed between the two. The simulator module hides the network layer from the routing layer; it also administers and controls the simulation.

The overlay level of a running node passes its messages to the simulator module. The simulator passes the message to the destination node, as the network would have done. This allows the simulator to intercept messages, to record the state of the network (it is, in fact, possible to serialize an entire network that reached a stable state; the serialized network can be loaded and be subjected to several different scenarios), to simulate some network effects (delay or failures, for instance), and even to step through a simulation. The simulator dictates the overall life cycle of the framework by calling the appropriate methods of the nodes, starting and stopping nodes as required by the simulation.

In current simulations, the network layer is emulated by the simulator module: messages are passed directly to the destination nodes by the simulator, after any processing is over. The main initial efforts went to optimizing this section, to allow as efficient as possible simulation of large-scale networks. The system allows in theory writing network modules that are used for actually passing messages through a physical network, for example a TCP/IP or a UDP-based module; to our knowledge, however, such a module is not included in the distribution.

The simulator can produce graphs of the overlay structure during a large simulation.

Though no generic tools for analyzing a network's performance are included, the simulator module is versatile enough to provide many useful measurements of network behavior.

When we began this work, PlanetSim did not yet feature a true distributed simulator. Any simulation would take place on a single physical machine. The framework is well optimized; this allows reasonably large simulations, but places limits on the scale of a simulation. In addition, running all nodes on a single machine, and using in-memory message passing to emulate a network layer cannot accurately reproduce real-life situations; this reduces drastically the ability to evaluate and compare the real behavior of equivalent networks. More recent versions provide wrapper code for the network communication layer that allow existing code to run in network testbeds such as PlanetLab,

but we have not reviewed this.

A more subtle restriction of PlanetSim is its focus on key based routing as a base layer for all services. This is appropriate for current applications, most if not all of which are indeed based on the KBR concept. However, this means that the framework may not be helpful to develop any applications with a different routing primitives. Extending the framework to support such developments is likely to be a daunting task due to the implicit assumptions built into it.

Overlay Weaver

Overlay Weaver[68] is an overlay construction toolkit developed at the National Institute of Advanced Industrial Science and Technology in Japan. It features a similar decomposition in layers and modules to that of PlanetSim. However, there is considerable more flexibility in the OW framework, mostly due to the more fine-grained decomposition in modules separated by well-defined interfaces.

The toolkit provides implementations for each module (in some cases, several), and thus it enables overlay designers to implement a structured overlay algorithm with a relatively low amount of code and improve it rapidly by iterative testing on a single computer. Because the modules are designed to be pluggable, it is possible to make realistic comparisons between new and existing overlay algorithms. The framework is designed such that it allows running the same applications and algorithms on the provided emulator (thus hosting thousands of virtual nodes on a single machine) and, in addition, on a real network.

As in the PlanetSim system, all applications are based on a key-based routing layer. This layer (split into several modules to allow adding of new implementations) exposes a stable KBR API to what OW calls the services layer. The latter corresponds to the application layer in PlanetSim; it provides higher-level P2P primitives, such as distributed hash table, to any higher-level application, using the routing layer for networking. Assuming there are well-defined interfaces towards the higher tiers, it is possible to mix and match different routing protocols (based on Chord or Kademlia, for example) with the same implementation of a service (for instance, DHT) and compare their behavior.

Above these services lay any high-level, user-interfacing application. Again, because the tier one services provide each stable APIs, it is possible to exchange different implementations for each module.

Several services (low-level functionality modules) are offered by the framework, usable by a higher-level application. For example, the Messaging service handles point-to-point message passing over the underlying network, and the

Directory service abstracts storage. Both services are pluggable, with several implementations available for each (TCP, UDP and direct in-memory inter-thread for message passing and Berkley DB or in-memory hash-table for storage). It is intended that the applications will use the provided modules directly (as the focus of the framework is on high-level development), but the developers can nevertheless implement their own modules if needed.

There are several example applications available on the highest level to serve as examples for developers. Among them is an IP multicast router and, notably, shells that allow command-line user-interaction (and scripting) to the DHT and multicast service, which are useful for testing purposes.

The OW toolkit provides several features destined to help developers to test their applications and improve them through iterative testing. The most important is the Distributed Environment Emulator. The DEE can host many nodes on a single machine (on the order thousands) and control them through *scenarios*. A simple scenario generator can be used to build scenarios, or alternatively the scenarios can be written by hand. It is also possible to collect a trace by running an existing application and translate this into a scenario. There is also a distributed mode: several emulators, running on separate machines, are connected to form a single emulator in cooperation. Then the emulator is used, the application uses the emulator's messaging service (by plugging a different network module) instead of the normal TCP or UDP messaging. This improves emulation performance and allows easier monitoring and evaluation of the applications' behaviors, but naturally reduces the realism of the simulation.

Besides a simple connectivity graph generator and message counter, there were no generic measurement tools included when we evaluated Overlay Weaver.

Evaluations of deployed P2P networks

Aside from the above approaches, there have been several well coordinated attempts to measure the behavior of real, working peer-to-peer networks. Such studies[45] are generally restricted to file-sharing networks, which, as mentioned above, often use unsophisticated algorithms. The researchers had no way of influencing the networks, being strictly limited to observation. Despite these limitations, the research generated some very useful results. By monitoring the activity of real users at the scale of the entire Internet, and for long stretches of time, the researchers were able to gather many behavior statistics. For example, time spent connected, rate of failure, network performance (meaning the distribution of local performance characteristics of each peer's connection rather than global averages), number of queries attempted,

success rate of said queries, number of links established, or a network's degree of connectivity. While particular to the individual networks measured, such information may be very useful for extrapolating generic real-life Internet-scale network conditions, which can be applied to simulation of network conditions in simulators.

Chapter 7

Conclusion

Ever-increasing amounts of information is exchanged today between an ever-increasing number of network end-points; web services, connected mobile devices and even individual network-enabled applications on each device are only going to multiply. In this context, data sharing technologies are naturally an important area of research.

Peer-to-peer networking, particularly structured peer-to-peer networks, provide some very successful solutions to problems of scalability and self organization in large networks. At the same time, the almost ubiquity of XML for data representation significantly lowered barriers to interoperability and facilitates the development of very heterogeneous systems. The lack of widespread common language for querying distributed XML information, however, is still a difficulty. Keyword search and application-specific query interfaces are still the norm for distributed systems.

We believe that widespread support of XQuery is a desirable solution. In this work we have demonstrated that such an approach is feasible, and present a system that exemplifies it, and furthermore that can be improved with relative ease.

Query language: In our work we took the approach opposite to the usual for related research: rather than developing an indexing scheme or retrieval algorithm and then defining a query language adapted to its abilities, we instead began with the intention to support a specific language, and attempt to solve the problem of designing a system that can do this.

We picked XQuery as the target language for two main reasons: First, because of XQuery's power and expressivity, which makes it very well suited to supporting practically every kind of querying needs. And second, XQuery is already the standard language for *non-distributed* XML databases; its sub-

set XPath is also very widely implemented as an embedded language in non-database applications that must nevertheless manipulate XML data, such as web browsers.

Extensible platform: We designed XQ2P with an intense focus on developing a platform that is as simple as possible[1] to extend and experiment with. XQuery is a complex language, and attempting to support all or even most of its features is daunting task. Most research focuses on efficient implementations for only a restricted subset of these features, which makes it very difficult to compare and combine the results.

Starting with a platform that already contains adequate implementations of all features would allow researchers to concentrate only on the parts they are focused on. In this way, much development effort is avoided, and incremental improvement in separate features are more easily combined and compared.

We followed these ideas throughout this entire work: The core XQuery processing kernel of XQ2P is constructed with simplicity in mind, to allow it to be understood—and modified—with ease; we avoided optimizations wherever they conflicted with clarity or they introduced entanglement between features; each feature of the language was implemented by separate modules, allowing facile removal, replacement and addition of features. We approached the peer-to-peer side of the problem in the same manner: P2PTester, the platform upon which XQ2P is built, encourages incremental development by providing common interfaces, ready-built modules for lower-level peer-to-peer networking layers, and a generic distributed testing infrastructure. Development not related to XQ2P or XQuery in general is also just as well supported.

Structural index–based XQuery evaluation: We demonstrate the validity of our approach by adapting the TwigStack holistic join algorithm and the node numbering scheme it uses for indexing for KBR-based indexing and peer-to-peer querying as an XQ2P module. Our distributed indexing and querying model is shared by some related projects, such as KadoP (see §6.2), but to our knowledge this is the first peer-to-peer system that attempts to support all of XQuery's features. It must be remarked that *not* all queries are supported with *efficiency*. There are certainly situations where a structural index in general, and the one we use in particular, is not selective enough. This does not invalidate our focus on complete XQuery support, however: Firstly, it is easy to express queries that *no* system can solve quickly; as a trivial example, a query that demands all the data must necessarily access all available data. Secondly, XQ2P

[1] But not simpler!

makes it easy to add new indexes and retrieval techniques (see below), to extend the fraction of queries that can be optimized; it is certainly easier to improve it than to construct a better system from scratch. Finally, the many features of XQuery allow complex manipulation of the data that the index system can retrieve, increasing the number of queries that can be answered; slow support for some queries is still preferable to none at all.

Integration of value indexes: We also implement a novel value-based index and horizontal partitioning technique for time-series data as another module on top of XQ2P. The approach demonstrates XQ2P's support for application-specific optimizations: operators and user functions optimized for time-series operations are included, which allow high performance and in some cases distributed evaluation. The time-series processing module is completed by a caching system. We apply these techniques for large volumes of stock-price data, demonstrating that the XML model can be successfully applied for data-intensive applications.

Perspectives: Our target of complete XQuery support and the complexity of that language present many promising avenues for future contributions. Perhaps the most natural of these would be the addition of some of the improved structural join algorithms mentioned in §6.3. We also left out of our implementation support for mixed keyword and structural queries that holistic join algorithms in general can provide; adding this and other text indexing methods has potential for much better selectivity. On a more abstract level, we believe it very interesting to study how several indexes can be employed at the same time, i.e., to execute different parts of a query using different indexes; analogous research for relational databases should provide some interesting directions in this respect.

A technique that we did not have time to explore but that XQ2P's modular structure should support exemplarily is employing parallel overlays to support multiple indexing methods: for example, an overlay based on an order-preserving hash or a tree structure in addition to the uniform-hash one we chose should allow both range queries and structural queries to be optimized at the same time.

With regards to the XQuery processing kernel itself, we would like to see many of the techniques for expression reordering and query transformation, such as [18, 73, 51, 11, 31], applied as a preprocessing step. Besides providing increased efficiency for local evaluation, such methods also have the potential to benefit distributed query processing: Simplifying queries and reduction of redundant subexpressions can reduce the number of index accesses. Reordering

and factorization can merge twig patterns; holistic algorithms usually perform better with one large twig pattern than on many smaller ones. Some query rewriting techniques, for example expressing backward axes (i.e., `ancestor::`) with forward ones (`descendant::`) can allow an index to optimize more of the query.

List of Figures

Bibliography

[1] K. Aberer, P. Cudre-Mauroux, A. Datta, Z. Despotovic, M. Hauswirth, M. Punceva, and R. Schmidt: *P-Grid: a Self-Organizing Structured P2P System*. SIGMOD Record, 32(3), 2003

[2] S. Abiteboul, S. Cluet, V. Christophides, T. Milo, G. Moerkotte, and J. Simeon: *Querying Documents in Object Databases*. Technical report, INRIA 1996

[3] S. Abiteboul, I. Manolescu, and N. Preda: *Constructing and querying peer-to-peer warehouses of XML resources*. ICDE 2005

[4] S. Abiteboul, I. Manolescu, N. Polyzotis, N. Preda, and C. Sun: *XML processing in DHT networks*. ICDE 2008

[5] T. Berners-Lee, R.T. Fielding, and L. Masinter: *Uniform Resource Identifier (URI): Generic Syntax*. 2005, `http://tools.ietf.org/html/rfc3986`

[6] M. Bender, S. Michel, P. Triantafillou, G. Weikum, and C. Zimmer: *Minerva: Collaborative P2P search*. In proc. of the 31st International Conference on Very Large Data Bases 2005

[7] R. Bolze, F. Cappello, E. Caron, M. Daydé, F. Desprez, E. Jeannot, Y. Jégou, S. Lanteri, J. Leduc, N. Melab, G. Mornet, R. Namyst, P. Primet, B. Quetier, O. Richard, E.-G. Talbi, and I. Touche. *Grid'5000: A large scale and highly reconfigurable experimental grid testbed*, International Journal of High Performance Computing Applications, 20(4):481–494, 2006

[8] P. Boncz and C. Treijtel: *AmbientDB: relational query processing in a P2P network*. In proc. of the International Workshop on Databases, Information Systems and Peer-to-Peer Computing, 2003

[9] A. Bonifati, U. Matrangolo, A. Cuzzocrea, and M. Jain: *XPath lookup queries in P2P networks*. WIDM 2004

[10] I. Botan, P.M. Fischer, D. Florescu, D. Kossmann, T. Kraska, and R. Tamosevicius: *Extending XQuery with Window Functions*. In VLDB 2007, pp.75–86

[11] M. Brantner, C-C. Kanne, and G. Moerkotte: *Let a Single FLWOR Bloom (to improve XQuery plan generation).* In XSym Workshop, 2007

[12] N. Bruno, N. Koudas, D. Srivastava: *Holistic twig joins: optimal XML pattern matching*. SIGMOD 2002

[13] B. Butnaru, F. Drăgan, G. Gardarin, I. Manolescu, B. Nguyen, R. Pop, N. Preda, and L. Yeh: *P2PTester: a tool for measuring P2P platform performance*. Demonstration at BDA 2006

[14] B. Butnaru: *Architecture de test pour les systèmes Pair à Pair*, Rapport de stage de Master, Université de Versailles Saint-Quentin-en-Yvelines, 2006

[15] B. Butnaru, F. Drăgan, G. Gardarin, I. Manolescu, B. Nguyen, R. Pop, N. Preda, and L. Yeh: *P2PTester: A tool for measuring P2P platform performance*. In proc. of ICDE, pp.1501–1502, 2007

[16] B. Butnaru, B. Nguyen, G. Gardarin, and L. Yeh: *XQ2P: Efficient XQuery P2P Time Series Processing*. BDA 2009

[17] Y. Chawathe, S. Ratnasamy, L. Breslau, M. Lanham, and S. Shenker: *Making Gnutella-like p2p systems scalable*. In proc. of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, pp.407–418

[18] D. Che, K. Aberer, and M.T. Ozsu: *Query optimization in XML structured-document databases*. VLDB J., 15(3):263–289, 2006

[19] S. Chen, H.G. Li, J. Tatemura, W.P. Hsiung, D. Agrawal, and K.S. Candan: *Twig^{2}Stack: Bottom-up processing of generalized-tree-pattern queries over XML documents*. VLDB 2006

[20] S-Y. Chien, Z. Vagena, D. Zhang, V. Tsotras, and C. Zaniolo: *Efficient structural joins on indexed XML documents*. In VLDB, pages 263–274, 2002

[21] I. Clarke, O. Sandberg, B. Wiley, and T.W. Hong: *Freenet: A distributed anonymous information storage and retrieval system*. Lecture Notes in Computer Science, Volume 2009/2001, pp.46–66, 2001

[22] F. Dabek, B. Zhao, P. Druschel, J. Kubiatowicz, and I. Stoica: *Towards a Common API for Structured Peer-to-Peer Overlays*. In proc. of the 2nd International Workshop on Peer-to-Peer Systems, 2003

[23] R. Dingledine, N. Mathewson, and P. Syverson. *Tor: The second-generation onion router*. In proc. of the 13th USENIX Security Symposium, 2004

[24] F. Drăgan, G. Gardarin, and L. Yeh: *Routing XQuery in a p2p network using adaptable trie-indexes*. IADIS 2005

[25] F. Drăgan, G. Gardarin, and L. Yeh: *MediaPeer: A safe, scalable p2p architecture for xml query processing*. In DEXA Workshops, pp.368–373, 2005

[26] D. Florescu and D. Kossmann: *A performance evaluation of alternative mapping schemes for storing XML data in a relational database*. Technical report, INRIA 1999

[27] L. Galanis, Y. Wang, S.R. Jeffrey, and D.J. DeWitt: *Locating data sources in large distributed systems*. VLDB 2003

[28] P. García, C. Pairot, R. Mondéjar, J. Pujol, H. Tejedor, and R. Rallo: *PlanetSim: A New Overlay Network Simulation Framework*, Lecture Notes in Computer Science, Volume 3437. Software Engineering and Middleware, pp.123–137, March 2005

[29] G. Gardarin and P. Valduriez: *Relational databases and knowledge bases*. Addison-Wesley Publishing Company, 1990

[30] G. Gottlob, C. Koch, and R. Pichler: *Efficient algorithm for processing XPath queries*. In proc. of VLDB 2002

[31] M. Grinev and S. Kuznetsov: *Towards an Exhaustive Set of Rewriting Rules for XQuery Optimization: BizQuery Experience*. In proc. ADBIS 2002

[32] T. Grust, M. van Keulen, and J. Teubner: *Staircase Join: Teach a Relational DBMS to Watch its (Axis) Steps*. VLDB 2003

[33] M. Harren, J.M. Hellerstein, R. Huebsch, B.T. Loo, S. Shenker, and I. Stoica: *Complex Queries in DHT-based Peer-to-Peer Networks*. IPTPS, March 2002

[34] R. Hasan, Z. Anwar, W. Yurcik, L. Brumbaugh, and R. Campbell: *A Survey of Peer-to-Peer Storage Techniques for Distributed File Systems* ITCC'05, vol. 2, pp.205–213, 2005

[35] O. Heckmann and A. Bock: *The eDonkey 2000 Protocol*. Technical Report KOM-TR-08-2002, Multimedia Communications Lab, Darmstadt University of Technology, December 2002.

[36] J. Hidders, P. Michiels, J. Siméon, and R. Vercammen: *How to Recognise Different Kinds of Tree Patterns From Quite a Long Way Away*. In proc. PLAN-X pp.14–24, 2007

[37] R. Huebsch, B. Chun, J.M. Hellerstein, B.T. Loo, P. Maniatis, T. Roscoe, S. Shenker, I. Stoica, and A.R. Yumerefendi: *The Architecture of PIER: an Internet-Scale Query Processor*. CIDR 2005

[38] R. Huebsch, J.M. Hellerstein, N. Lanham, B.T. Loo, S. Shenker, I. Stoica: *Querying the Internet with PIER*. VLDB 2003

[39] H. Jiang, H. Lu, W. Wang, and B.C. Ooi: *XR-tree: Indexing XML data for efficient structural joins*. ICDE 2003

[40] H. Jiang, W. Wang, H. Lu, and J.X. Yu: *Holistic twig joins on indexed XML documents*. In proc. VLDB, pp.273–284, 2003

[41] E. Jiao, T.W. Ling, and C-Y. Chan: *PathStack¬: A holistic path join algorithm for path query with not-predicates on XML data*. DASFAA 2005

[42] S. Al-Khalifa, H.V. Jagadish, J.M. Patel, Y. Wu, N. Koudas, and D. Srivastava: *Structural joins: A primitive for efficient XML query pattern matching*. ICDE 2002

[43] A. Khan and V. Zuberi: *Stock Investing for Everyone*. John Wiley & Sons, 1999

[44] J. Li, B.T. Loo, J. Hellerstein, F. Kaashoek, D.R. Karger, and R. Morris: *On the feasibility of peer-to-peer web indexing and search*. In proc. of the 2nd International Workshop on Peer-to-Peer Systems 2003

[45] J. Li, J. Stribling, T.M. Gil, R. Morris, and M.F. Kaashoek: *Comparing the performance of distributed hash tables under churn*. In proc. of the 3rd International Workshop on Peer-to-Peer Systems 2004

[46] D. Liben-Nowell, H. Balakrishnan, and D. Karger: *Analysis of the Evolution of Peer-to-Peer Systems*. ACM Conf. on Principles of Distributed Computing, July 2002

[47] J. Lu, T. Chen, and T.W. Ling: *Efficient processing of xml twig patterns with parent child edges: a look-ahead approach.* CIKM, pp.533–542, 2004

[48] J. Lu, T.W. Ling, C.Y. Chan, and T. Chen: *From region encoding to extended Dewey: On efficient processing of XML twig pattern matching*. In proc. VLDB, pp.193–204, 2005

[49] R. Mahajan, M. Castro, and A. Rowstron: *Controlling the Cost of Reliability in Peer-to-peer Overlays*, IPTPS 2003

[50] P. Maymounkov and D. Mazières: *Kademlia: A peer-to-peer information system based on the XOR metric*. In proc. of IPTPS 2002, pp.53–65

[51] P. Michiels: *XQuery optimization*. VLDB Workshop 2003

[52] W. Nejdl, B. Wolf, C. Qu, S. Decker, M. Sintek, A. Naeve, M. Nilsson, M. Palmer, and T. Risch: *Edutella: A P2P networking infrastructure based on RDF*. In proc. of the 12th International Conference on World Wide Web. 2003

[53] B. Ooi, Y. Shu, and K-L. Tan. *Relational data sharing in peer-based data management systems*. SIGMOD Record, 23(3), 2003

[54] L. Page, S. Brin, R. Motwani, and T. Winograd: *The PageRank citation ranking: Bringing order to the web.* Technical report, Stanford Digital Library Technologies Project, 1998

[55] V. Papadimos, D. Maier, and K. Tufte: *Distributed Query Processing and Catalogs for Peer-to-Peer Systems*. CIDR 2003

[56] *The PIER project*. `http://pier.cs.berkeley.edu/`

[57] J. Pouwelse, P. Garbacki, D. Epema, and H. Sips: *The BitTorrent P2P File-sharing System: Measurements and Analysis*. IPTPS 2005

[58] N. Preda: *Efficient web resource management in structured peer-to-peer networks*. PhD thesis, Université de Paris XI, 2008

[59] Lu Qin, Jeffrey Xu Yu, and Bolin Ding: *TwigList: Make Twig Pattern Matching Fast*. DASFAA 2006

[60] P. Rao and B. Moon: *Locating XML Documents in a Peer-to-Peer Network using Distributed Hash Tables*. IEEE Transactions on Knowledge and Data Engineering, 21(12):1737–1752, December 2009

[61] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker: *A scalable content addressable network*. In proc. ACM SIGCOMM 2001

[62] M. Ripeanu: *Peer-to-peer architecture case study: Gnutella network*. Technical report, University of Chicago, 2001.

[63] A. Rowstron and P. Druschel: *Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems*. In proc. of the 18th IFIP/ACM Int. Conf. Distributed Systems Platforms, pp.329–350, 2001

[64] K. Sankaralingam, S. Sethumadhavan, and J. Browne: *Distributed PageRank for P2P systems*. In proc. of the Twelfth International Symposium on High Performance Distributed Computing, June 2003

[65] Ş. Săroiu, P.K. Gummadi, S.D. Gribble: *A Measurement Study of Peer-to-Peer File Sharing Systems*. In proc. Multimedia Computing and Networking 2002

[66] R. Schmidt: *The P-Grid System—Overview*. `http://www.p-grid.org/implementation/`

[67] J.G. Shanmugasundaram, K.H. Tufte, C. Zhang, D.J. DeWitt, and J. Naughton: *Relational Databases for Querying XML Documents: Limitations and Opportunities*. VLDB 1999

[68] K. Shudo: *Overlay Weaver*, `http://sourceforge.net/projects/overlayweaver`

[69] L. Sidirourgos, G. Kokkinidis, and T. Dalamagas: *Efficient Query Routing in RDF/S schema-based P2P*. HDMS 2005

[70] G. Skobeltsyn, M. Hauswirth, and K. Aberer: *Efficient processing of XPath queries with structured overlay networks*. Lecture Notes in Computer Science, 2005, n3761, pages 1243-1260

[71] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan: *Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications*. In proc. ACM SIGCOMM 2001

[72] The Tor Project. `https://www.torproject.org/`

[73] A.M. Weiner, C. Mathis, and T. Härder: *Rules for Query Rewrite in Native XML Databases*. In proc. EDBT DataX Workshop, pp.21–26, 2008.

[74] World Wide Web Consortium: *Extensible Markup Language (XML) 1.1 (Second Edition)*. W3C Recommendation. `http://www.w3.org/TR/xml11/`

[75] World Wide Web Consortium: *Namespaces in XML 1.1 (Second Edition)*. W3C Recommendation. `http://www.w3.org/TR/xml-names11/`

[76] World Wide Web Consortium: *XML Path Language (XPath) Version 2.0 (Second Edition)*. W3C Recommendation, 14 December 2010. http://www.w3.org/TR/xpath20/

[77] World Wide Web Consortium: *XQuery 1.0: An XML Query Language (Second Edition)*. W3C Recommendation. http://www.w3.org/TR/xquery/

[78] World Wide Web Consortium: *XQuery 1.0 and XPath 2.0 Functions and Operators (Second Edition)*. W3C Recommendation, 14 December 2010. http://www.w3.org/TR/xpath-functions/

[79] World Wide Web Consortium: *XQuery 1.0 and XPath 2.0 Data Model (XDM) (Second Edition)*. W3C Recommendation, 14 December 2010. http://www.w3.org/TR/xpath-datamodel/

[80] World Wide Web Consortium: *XQuery 3.0: An XML Query Language (Second Edition)*. W3C Working Draft. http://www.w3.org/TR/xquery-30/

[81] World Wide Web Consortium: *The XML Query TestSuite*. www.w3.org/XML/Query/test-suite/

[82] World Wide Web Consortium: *XML Syntax for XQuery 1.0 (XQueryX) (Second Edition)*. W3C Recommendation, 14 December 2010. http://www.w3.org/TR/xqueryx/

[83] M. Yoshikawa, T. Amagasa, T. Shimura, and S. Uemura: *XRel: A Path-Based Approach to Storage and Retrieval of XML Documents using Relational Databases*. ACM Transactions on Internet Technology, 1(1):110–141, August 2001

[84] C. Zhang, J.F. Naughton, D.J. DeWitt, Q. Luo, and G.M. Lohman: *On Supporting Containment Queries in Relational Database Management Systems*. SIGMOD 2001

[85] B.Y. Zhao, J. Kubiatowicz, and A.D. Joseph: *Tapestry: a fault-tolerant wide-area application infrastructure*. Computer Communication Review, 32(1):81, 2002

[86] There is no official description of the *Direct Connect protocol*. A short description of its properties is provided by Wikipedia at http://en.wikipedia.org/wiki/Direct_Connect_(file_sharing)

[87] *LLC Napster*. http://www.napster.com

[88] RoSeS — *Really Open and Simple Web Syndication*. http://www-bd.lip6.fr/roses/doku.php