Distributed Subtyping

Sébastien Baehni
I&C EPFL
Switzerland

João Barreto
INESC-ID/IST
Portugal

Rachid Guerraoui
I&C EPFL & CSAIL MIT
Switzerland & USA

ABSTRACT

One of the most frequent operations in object-oriented programs is the *instanceof* test, also called the *subtyping* test or the *type inclusion* test. This test determines if a given object is an instance of some type. Surprisingly, despite a lot of research on distributed object-oriented languages and systems, almost no work has been devoted to the implementation of this test in a distributed environment.

This paper presents the first algorithm to implement the *subtyping* test on an object received through the wire, without having to download the full code of the object type, nor to deserialize the object. We use a slicing technique that encodes a (multiple-subtyping) hierarchy using as little memory as the best known centralized implementation of the *subtyping* test. Our slicing technique is however different than centralized ones and allows for the dynamic addition of types without global reconfiguration.

We convey the practicality of our algorithm through performance measures obtained from a fully distributed implementation of our algorithm which we experiment on standard Java hierarchies. In particular, we show that we can perform a subtyping test between 3 and 12 times faster than the code downloading approach without hampering the time taken for deserialization. Moreover, we require the same subtyping time as a string-based approach while reducing the encoding length by a factor of 50.

1. INTRODUCTION

Testing if a type of an object is a subtype of another type is traditionally called a *subtyping* test, or sometimes a *type inclusion* test. Usually, object-oriented languages implement this test via native language keywords or specific methods. For instance, Java and C# use respectively the *instanceof* and *is* keywords, whereas Smalltalk uses the *isKindOf:* method.[1]

A common assumption made in object-oriented languages is that the code of the type of the object is available when performing a subtyping test. This assumption has been carried over in all object-oriented distributed systems we know of, including CORBA [12], JMS [5], MSMQ [9], and more generally, in any distributed Java [6] or Microsoft .NET [10] application we encountered.

Relying on the code of the type of an object to accomplish any subtyping test involving the object is problematic in distributed environments where objects are remotely exchanged. This is for instance the case in increasingly popular *event-based* [11, 2] (also called *publish/subscribe* [4]) systems. In such systems, whenever an object is received (in the form of an event), a type inclusion test might need to be performed. This is key for a node to throw away objects it is not interested in. If the received object is of an unknown type, its code has to be downloaded and the object has to be deserialized before the subtyping test can be performed. This is particularly cumbersome, especially if the probability that the received object is of interest to the node is low. Instead, it makes more sense to incorporate some (encoded) type information with the object and use this to perform the subtyping test without having to deserialize the object and thus without needing the code of the object.

In fact, several encoding schemes have been devised in the literature [14, 3, 7, 8, 1, 16, 17] to support efficient subtyping tests in centralized environments. These however cannot be ported to a decentralized setting as they typically require a global reconfiguration when new types are added. Whereas global reconfiguration might be considered reasonable in a centralized system where the addition of a new type would go through recompiling the type hierarchy anyway (and generating a new encoding), it is clearly unacceptable for long-lived distributed systems.

Efficiently encoding type hierarchies in a distributed context is not a trivial task. Ideally, the encoding should use a minimal representation to be sent with the object over the wire, just like in a centralized setting. To avoid reconfiguration, the encoding of the core type hierarchy, i.e., of the set of types present at the initialization of the system, should

[1] For the sake of presentation simplicity, and as in [16, 17], we do not make the distinction between a type, a class, an interface and a signature. These differences are irrelevant for the problem we address. Consequently, the terms multiple inheritance, multiple subtyping and multiple subclassing can be used interchangeably in our context. We also focus on languages with a nominal subtyping relation (e.g., Java, C#) and we simply identify a type with its name.

remain the same throughout the lifetime of the system even when new types are added at runtime.

This paper presents an algorithm for performing subtyping tests in a dynamic distributed environment. The algorithm, which we denote by *DST*, does not require downloading the full code of the object type, nor deserializing the object. It also supports multiple subtyping as well as the addition of new types at runtime without requiring any global reconfiguration. Subtyping tests can even be performed with objects of new dynamically added types against the original types of the core type hierarchy.

The idea underlying our encoding scheme consists in splitting the type hierarchy into smaller disjoint sequences of types called *slices* (we will explain in the paper how the decomposition into slices is actually performed). Each type is then assigned an identifier corresponding to its position in the slice (i.e., its position in the sequence). The identifier of a type t, together with the *intervals* of the identifiers of its super-types (simply called the intervals of t), represent the encoding of t. To test if a type t is a subtype of a type u, we simply check if the identifier of u is contained in the intervals of t. The addition of new types is then handled by extending an existing slice or adding a new one.

A distributed implementation of our *DST* algorithm (using Java 1.5) is available at *http://lpdwww.epfl.ch/baehni/dst.tgz*. The algorithm is provided in the form of a comprehensive set of Java APIs, together with a set of wrapper classes around the standard Java serialization APIs. Our performance measurements, conducted through standard Java hierarchies (1.5, 1.4 and 1.2), convey the fact that *DST* performs a subtyping test between 3 and 12 times faster than a standard code downloading approach without hampering the time taken to deserialize the object. Moreover, *DST* requires the same subtyping test time as a straightforward string-based encoding approach (that does not require global reconfiguration when new types are added), in which the type of an object is encoded via the name of the type together with the name of its super-types; with respect to this approach however, we reduce the encoding length by a factor of 50. We also show, for completeness, that *DST* is comparable, in terms of encoding length, to the best currently known centralized subtyping algorithm [16]. Yet, and as we pointed out, *DST* is designed for a dynamic distributed environment where new types can be added at runtime without global reconfiguration.

The rest of the paper is organized as follows. Section 2 overviews several alternatives to address the subtyping problem. Section 3 describes the encoding generated by the *DST* algorithm. Section 4 presents *DST*. Section 5 presents the key elements underlying our implementation of *DST*. Section 6 presents performance results. Section 7 summarizes the main contribution.

2. DESIGN ALTERNATIVES

Consider a process p_i, representing a physical node in a distributed system, receiving a given object O of type u. The subtyping problem consists for p_i to determine whether O is of (a subtype of) a type t. We say that p_i performs a *subtyping test* with O (or u) against t.

Clearly, the simplest way to perform the test is for p_i to use the native subtyping instruction of the language, e.g., *instanceof* and *is* keywords of respectively Java and C#. In other words, p_i would rely on the centralized implementation of the *subtyping test*. However, in a distributed system where O is received over the wire, to retrieve its type u, O has to be deserialized and the code of u (e.g., the bytecode in Java or IL in .NET) has to be loaded in memory. That is, p_i has to get the code of O and deserialize it. Hence, either the code of the type of O must be sent within O or downloaded afterwards. In any case, this means a clear waste of CPU and bandwidth if p_i turns out not to be interested in objects of type t.

A simple approach to prevent code downloading consists in representing the identifier of the type and all its super-types by a *string*. To construct this string identifier, we would for instance follow a top-down approach. For each level of the type hierarchy, we would add the identifier of the super-types of the object until the actual type of the object is reached. For instance, an encoding of type k in the hierarchy of Figure 1 would be "/abc/def/hi/k" (the circles in Figure 1 represent the types of the hierarchy). Each object O would transport the encoding of its type. When receiving O, process p_i simply checks if the string identifier of type t is contained in the string associated with O. This approach clearly reduces the amount of necessary information that has to be transferred in order to perform subtyping tests with respect to the code downloading solution. However, the size of the encoding and the time to perform the subtyping tests directly grow with the size of the type hierarchy.

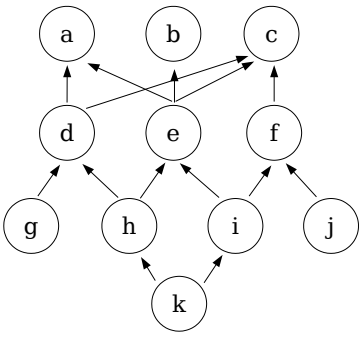

Figure 1: Example of a type hierarchy

There are indeed more efficient approaches to encoding type hierarchies [8, 7, 1, 16, 17, 13] but, as we will discuss below, these are centralized approaches that typically require global reconfigurations when new types are dynamically added to the system (which is usually not an issue in a centralized system). In fact, this reconfiguration is required even if no subtyping tests need to be performed against these new types.

With bit-vector encoding [8], the type hierarchy is embedded in a lattice of subsets of 1,...,k. The encoding of a type t is a vector of k bits (vec_t) [8]. A type i is a subtype of a type j if $vec_i \wedge vec_j = vec_j$. With range compression encoding [1], the type hierarchy is split into single inheritance trees and types are enumerated using a post-order transversal algorithm. The encoding of a type t consists of (1) its

identifier as well as (2) a set of intervals. The smallest, respectively highest, value of an interval of a type t contains the smallest, respectively highest, identifier of the subtypes of t while ensuring that only the subtypes of t are inside the considered interval. If it is not possible to encode all subtypes of t inside an interval, a new one is created. If the identifier of a type i is contained in the intervals of a type j then i is a subtype of j. With these approaches [8, 1], whenever a new type is added at runtime, the entire type hierarchy is reconstructed.

With Packed-Encoding [7] (PE), a hierarchy of N types is represented by a matrix with N lines and P columns called *buckets* (P is determined by the algorithm). The encoding of each type t is given by (1) the identifier of the bucket in which t is contained (p_t), (2) the identifier of t in this bucket (id_t) and, (3) an array in which the <index,value> pair corresponds respectively to the identifier of a bucket i and the identifier of the super-type of t into i (in order to support multiple subtyping, two super-types cannot be in the same bucket). Bit-Packed Encoding (BPE) enhances Packet-Encoding by permitting two or more buckets to be represented by a single byte. In both approaches, no global reconfiguration is needed. However, the number of buckets, and hence the size of the arrays, grows with the number of common ancestors in the type hierarchy.

With PQ-Encoding [16] (PQE), the relative numbering (each type is encoded by an interval) is combined with the techniques used in PE or BPE. In PQE, the type hierarchy T is split into subsets of types, called *slices*. Each slice s_i contains the maximal number of types such that, for each type $t \in T$, the subtypes of t in s_i can be arranged in a contiguous interval. Consequently, the encoding of a type t in PQE consists of: (1) the slice identifier of t, (2) an identifier id_t of t, and (3) for each slice, an interval which smallest, respectively highest, values corresponding to the smallest, respectively highest, identifiers of the subtypes of t contained in the specific slice. Subtyping consists in testing if the identifier of i belongs to the interval of the slice of j. PQ-Encoding provides the best encoding length out of all centralized algorithms we know about. With PQE, it is however impossible to add new types at runtime without having to re-encode the entire type hierarchy.[2] With R&B encoding [13], *ranges and slices* are used for encoding in constant time, in an incremental way and with a small encoding length. The algorithm uses the range numbering technique of Schubert [14]. The algorithm supports the addition of new subtypes at runtime, assuming the algorithm of [7]. A complete renumbering is however sometimes needed, in which case the encoding of the type hierarchy needs to entirely change.

Our approach can be viewed as a combination of [16] and [7] with a fundamental difference: we order the ancestors of a type instead of its descendants. This is key to avoiding global configuration while using very little memory for the encoding. As we show through our experiments, our *DST* algorithm is comparable, in terms of performance, to the best known centralized subtyping algorithm [16]. Yet *DST* is designed for a dynamic distributed environment where new types can be added at runtime without global reconfiguration. As we pointed out in the introduction, *DST* allows for subtyping tests with objects of newly added types against the original types of the core hierarchy.

[2]The authors present in [17] a variant of the algorithm (BTS) that overcomes this difficulty in certain situations. There are however cases where the algorithm still has to re-encode again some parts of the type hierarchy.

3. TYPE ENCODING

This section describes the encoding we consider to represent type hierarchies and how this can be used in a distributed subtyping test. We will show in the next section how our *DST* algorithm generates such encoding.

3.1 Types and Subtypes

We first recall here some subtyping notations and definitions that are needed to describe our encoding scheme.

A *type hierarchy* is a partially ordered set $S = (T, \prec)$, where T is a set of types $\{t, u, ...\}$ and $\prec$ is the reflexive, transitive and anti-symmetric *subtyping* relation. Type u is said to be a *subtype* of a type t if $u \prec t$: t is in this case said to be a *super-type* of u. The hierarchy of types present at the initialization of the system is called the *core type hierarchy*, and is denoted by a unique identifier $cth(S)$. Every type that is later added to the system is attached to one core type hierarchy.

Figure 1 depicts a type hierarchy $S = (T, \prec)$, in which $T = \{a, b, c, d, e, f, g, h, i, j, k\}$; the arrows represent $\prec$ relations. For instance, d is a subtype of a, c and d while k is a subtype of a, b, c, d, e, f, h, i and k. As a consequence, a is a super-type of d and k, among others.

The following definitions, from [16], capture useful subtyping information (these are formally defined in Figure 2):

1. A type u is a *descendant* of a type t if $u \prec t$. We denote by D(t,S) the set of all the descendants of t in S. In Figure 1, we have for instance: D(a,S)=$\{a,d,e,g,h,i,k\}$.

2. A type u is an *ancestor* of a type t if $t \prec u$. We denote by A(t,S) the set of all the ancestors of t in S. In Figure 1, we have: A(g,S)=$\{a,c,d,g\}$.

3. A type u is a *child* of a type t if $t \neq u \wedge u \prec t$ and there is no type v ($v \neq u$ and $v \neq t$) that is both a subtype of t and a super-type of u. We denote by C(t,S) the set of all children of t in S. In Figure 1, C(a,S)=$\{d,e\}$.

4. A type u is a *parent* of a type t if t is a child of u. We denote by P(t,S) the set of all parents of t in S. In Figure 1, P(g,S)=$\{d\}$.

5. A *root* type u of a type hierarchy S is a type that does not have any parent. We denote by R(S) the set or root types of a type hierarchy S (a type hierarchy can have multiple *root types*). In Figure 1, R(S)=$\{a,b,c\}$.

6. The *level* L(t, S) of a type t in a type hierarchy S is the greatest level of its parents plus one. The level of the root types is zero. In Figure 1, L(g,S)=2.

Descendants: $\mathrm{D}(t,S=(T,\prec)) \stackrel{def}{=} \{u \in T \mid u \prec t\}$
Ancestors: $\mathrm{A}(t,S=(T,\prec)) \stackrel{def}{=} \{u \in T \mid t \prec u\}$
Children: $\mathrm{C}(t,S=(T,\prec)) \stackrel{def}{=} \{u \in T \mid u \prec t \text{ and } u \neq t \text{ and } (\nexists v \in T | v \neq u \text{ and } v \neq t \text{ and } u \prec v \prec t)\}$
Parents: $\mathrm{P}(t,S=(T,\prec)) \stackrel{def}{=} \{u \in T \mid t \prec u \text{ and } u \neq t \text{ and } (\nexists v \in T | v \neq u \text{ and } v \neq t \text{ and } t \prec v \prec u)\}$
Root Types: $\mathrm{R}(S=(T,\prec)) \stackrel{def}{=} \{t_i,...,t_n \in T \mid \{P(t_i,S),...,P(t_n,S)\} = \{\emptyset,...,\emptyset\}\}$

$$\textit{Level}: L(t,S=(T,\prec)) \stackrel{def}{=} \begin{cases} 0, & \text{if } t \in \mathrm{Root}(S); \\ max(L(p,S))+1 \mid p \in P(t,S), & \text{otherwise.} \end{cases}$$

Figure 2: Notations

3.2 Slicing

A *sub-hierarchy*, also called a *subtype hierarchy*, of a type hierarchy S, is a partially ordered subset $S_i = (T_i, \prec)$, where $T_i \subseteq T$. A *slice* s_i in a type hierarchy is a *sequence* of types of the hierarchy.[3] (Because a slice is a sequence, we will sometimes talk about the head and the tail of the slice.) A *slicing* of a type hierarchy $S = (T, \prec)$ is a set of slices such that: (1) each pair of slices is disjoint and (2) the union of all the slices is T. By extension, the root types of a slice s_i of a subtype hierarchy S_i are the root types of S_i.

For example, considering Figure 1, $S_0 = (\{f, i, j, k\}, \prec)$ and $S_1 = (\{a, c, d\}, \prec)$ are subtype hierarchies of S. A possible slicing of the type hierarchy $S = (T, \prec)$ is made of $s_0 = g; d; a; c; e; b; h$ and $s_1 = k; i; f; j$.

We now define the notion of *straight slice* which is key to our encoding scheme. A straight slice s_i of a type hierarchy $S = (T, \prec)$ is a sequence of types such that, for any type $t \in T$, all the ancestors of t are consecutive in the sequence s_i. A *straight slicing* is a slicing in which each slice is straight.

Table 1 describes a possible straight slicing of the type hierarchy of Figure 1. In this straight slicing, each straight slice contains one type only. The first column of Table 1 contains the different types of the type hierarchy, while the first row contains the different slices of the straight slicing. Each cell at position i,j ($i > 0$ and $j > 0$) contains the sequence of ancestors of the type at the head of row i for the slice at the head of column j.

As we will see later, the goal of our algorithm will be to generate a small number of slices, for this will be the secret to a frugal encoding. A slicing made of $s_0 = g; d; a; c; e; b; h$ and $s_1 = k; i; f; j$ is much more frugal (Figure 3 and Table 2) than that of Table 1 (Figure 1). In Table 2, the sequences of ancestors of any type are consecutive in each slice s_i. On the other hand, the following slicing of the type hierarchy of Figure 1 is not a straight one: $s_0 = a; b; c; d; e; f$, $s_1 = g; h; i; j; k$. We can clearly see here that the ancestors of type f are not consecutive in s_0 (i.e., c and f are not consecutive in s_0).

[3]Our terminology slightly differs from the one considered in [15, 16] in that we consider a slice to be a sequence instead of a set.

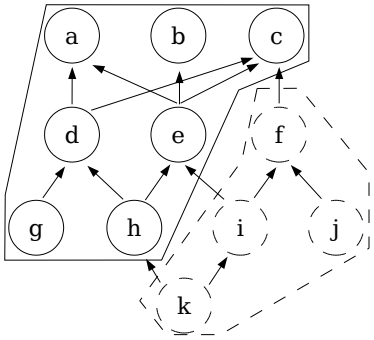

Figure 3: Two straight slices of the type hierarchy of Figure 1 (depicted by the dashed and plain polygons)

3.3 Encoding

Subtyping tests are not performed directly on the types themselves but on an encoded representation of them. We call the latter *the encoding of a type*. We use the straight slicing notion to encode types. The encoding of a type t of a type hierarchy S consists of:

1. The identifier $cth(S)$ of the core type hierarchy of S.
2. The type identifier $id_t \in \mathbb{Z}$ of t. This identifier corresponds to the position of t in the straight slice s_i to which t belongs and is denoted s_i^t. Identifier id_t is unique in s_i^t.
3. The identifier $id_{s_i} \in \mathbb{N}$ of the straight slice s_i to which t belongs. This identifier is denoted $id_{s_i}^t$.
4. For each straight slice s_i, the intervals $I_{s_i}^t$ corresponding to the smallest, respectively the highest, type identifier of the ancestors of type t in each straight slice s_i.

As a straight slicing ensures that the ancestors of any type t in a specific straight slice are consecutive, there is at most one interval $I_{s_i}^t$ for each slice s_i that corresponds to the union of the identifiers of the ancestors of the parents of t (we will see in Section 4.6 that this property might not be fulfilled when new types are added at runtime to the core type hierarchy $cth(S)$; we will however explain how we deal with this situation).

	$s_0 = a$	$s_1 = b$	$s_2 = c$	$s_3 = d$	$s_4 = e$	$s_5 = f$	$s_6 = g$	$s_7 = h$	$s_8 = i$	$s_9 = j$	$s_{10} = k$
a	a	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
b	$\emptyset$	b	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
c	$\emptyset$	$\emptyset$	c	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
d	a	$\emptyset$	c	d	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
e	a	b	c	$\emptyset$	e	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
f	$\emptyset$	$\emptyset$	c	$\emptyset$	$\emptyset$	f	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
g	a	$\emptyset$	c	$\emptyset$	$\emptyset$	$\emptyset$	g	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
h	a	b	c	d	e	$\emptyset$	$\emptyset$	h	$\emptyset$	$\emptyset$	$\emptyset$
i	a	b	c	$\emptyset$	c	f	$\emptyset$	$\emptyset$	i	$\emptyset$	$\emptyset$
j	$\emptyset$	$\emptyset$	c	$\emptyset$	$\emptyset$	f	$\emptyset$	$\emptyset$	$\emptyset$	j	$\emptyset$
k	a	b	c	d	e	f	$\emptyset$	h	i	$\emptyset$	k

Table 1: A straight slicing of the type hierarchy of Figure 1

	$s_0 = g; d; a; c; e; b; h$	$s_1 = k; i; f; j$
a	a	$\emptyset$
b	b	$\emptyset$
c	c	$\emptyset$
d	$d; a; c$	$\emptyset$
e	$a; c; e; b$	$\emptyset$
f	c	f
g	$g; d; a; c$	$\emptyset$
h	$d; a; c; e; b; h$	$\emptyset$
i	$a; c; e; b$	$i; f$
j	c	$f; j$
k	$d; a; c; e; b; h$	$k; i; f$

Table 2: A straight slicing of the type hierarchy of Figure 3

We describe in Table 3 the encoding of the straight slicing of Table 1 for the type hierarchy of Figure 1. For simplicity, we do not mention $cth(S)$ (which is the same for all types). The first column contains the different types of the type hierarchy while the first row depicts the straight slices of the straight slicing. The second column contains the identifier of the type together with its slice identifier. The other cells of Table 3 contain the intervals of the identifiers of the ancestors of the type at the head of the row for the straight slice at the head of the column. We finally present in Table 4 the encoding of the slicing of Table 2 for the type hierarchy of Figure 1. Again, for simplicity, we do not mention $cth(S)$ (which is the same for all types).

Considering Table 3, we use 13 bits to encode type a (without considering $cth(S)$). Indeed, $id_a = 0$ is encoded with 1 bit, $id^t_{s_i} = 0$ is encoded with 1 bit and $I^a_{s_i}$ can be encoded with 11 bits. The biggest encoding is for types i, j and k and equals 16 bits. To encode the entire type hierarchy, 162 bits are used (without taking into account the length of $cth(S)$). Considering Table 4, we simply used 6 bits to encode type g. Indeed, $id_g = 0$ is encoded with 1 bit, $id^g_{s_i} = 0$ is encoded with 1 bit and $I^g_{s_i} = [0, 3], \emptyset$ can be encoded with 4 bits (1 bit for 0, 2 bits for 3 and 1 for $\emptyset$). The maximal encoding length is for types e and i with a total of 10 bits. The encoding length of the entire type hierarchy is 86 bits (we will discuss in Section 5.5 how we can reduce the size of this encoding further).

3.4 Distributed Subtyping

We show now how our encoding can be used in a distributed fashion within a system of processes p_1, p_2,..,p_n,... The number of these processes is not bounded and processes can leave (e.g., crash) and join the system (e.g., recover) at any time. Applications running on different processes exchange typed objects.

To enable distributed subtyping tests, it is not necessary for a process to send, together with each object, the entire encoding of the type of the object. It is only necessary to send (1) the identifier $cth(S)$ (see Section 5) as well as (2) the intervals of the ancestors ($I^t_{s_i}$) of the type of the object (ordered according to their respective slice identifier). On the other hand, a process that needs to perform a subtyping test on a type does not need to maintain the entire encoding of this type. It only needs to know (1) $cth(S)$ as well as (2) the set containing, for each type t, the pairs $< id_t, id^t_{s_i} >$.

To test if a type u is a subtype of another type t (respectively belonging to type hierarchies S_u and S_t), we simply check the following property:

$$u \prec t \Leftrightarrow (id_t \in I^u_{s^t_i} \wedge cth(S_u) = cth(S_t)) \quad (1)$$

We discuss here few examples of subtyping tests considering first the the encoding of Table 3 and then the one of Table 4 (we omit $cth(S)$ in both cases as we consider only one hierarchy.).

With the encoding of Table 3, when a process p_1 receives, together with an object O_1, the encoding information $\emptyset$,[0,0],$\emptyset$,$\emptyset$,$\emptyset$,$\emptyset$,$\emptyset$,$\emptyset$,$\emptyset$,$\emptyset$, p_1 can test if O_1 is of type a. This is performed by checking if $id_a \in I^b_{s^a_i}$, i.e., $0 \in I^b_{s_0} = \emptyset$: O_1 is hence declared not to be of type a. When p_1 receives an object O_2 with [0,0],[0,0],[0,0],[0,0],[0,0],[0,0],$\emptyset$,[0,0],[0,0],$\emptyset$,[0,0], p_1 can test if the type of O_2 is a subtype of e by checking if $id_e \in I^k_{s^e_i}$ i.e., $0 \in I^k_{s_4} = [0, 0]$. Hence O_2 is declared subtype of e.

Consider now subtyping tests with the encoding of Table 4. For instance, if a process p_1 receives an object O_1 with [5,5],$\emptyset$, p_1 can test if O_1 is of type a by checking if $id_a \in I^b_{s^a_i}$,

	id_t, $id^t_{s_i}$	$s_0 = a$	$s_1 = b$	$s_2 = c$	$s_3 = d$	$s_4 = e$	$s_5 = f$	$s_6 = g$	$s_7 = h$	$s_8 = i$	$s_9 = j$	$s_{10} = k$
a	0,0	[0,0]	∅	∅	∅	∅	∅	∅	∅	∅	∅	∅
b	0,1	∅	[0,0]	∅	∅	∅	∅	∅	∅	∅	∅	∅
c	0,2	∅	∅	[0,0]	∅	∅	∅	∅	∅	∅	∅	∅
d	0,3	[0,0]	∅	[0,0]	[0,0]	∅	∅	∅	∅	∅	∅	∅
e	0,4	[0,0]	[0,0]	[0,0]	∅	[0,0]	∅	∅	∅	∅	∅	∅
f	0,5	∅	∅	[0,0]	∅	∅	[0,0]	∅	∅	∅	∅	∅
g	0,6	[0,0]	∅	[0,0]	∅	∅	∅	[0,0]	∅	∅	∅	∅
h	0,7	[0,0]	[0,0]	[0,0]	[0,0]	[0,0]	∅	∅	[0,0]	∅	∅	∅
i	0,8	[0,0]	[0,0]	[0,0]	∅	0	[0,0]	∅	∅	[0,0]	∅	∅
j	0,9	∅	∅	[0,0]	∅	∅	[0,0]	∅	∅	∅	[0,0]	∅
k	0,10	[0,0]	[0,0]	[0,0]	[0,0]	[0,0]	[0,0]	∅	[0,0]	[0,0]	∅	[0,0]

Table 3: The encoding of the type hierarchy of Figure 1 for the straight slicing of Table 1

	id_t, $id^t_{s_i}$	$s_0 = g; d; a; c; e; b; h$	$s_1 = k; i; f; j$
a	2,0	[2,2]	∅
b	5,0	[5,5]	∅
c	3,0	[3,3]	∅
d	1,0	[1,3]	∅
e	4,0	[2,5]	∅
f	2,1	[3,3]	[2,2]
g	0,0	[0,3]	∅
h	6,0	[1,6]	∅
i	1,1	[2,5]	[1,2]
j	3,1	[3,3]	[2,3]
k	0,1	[1,6]	[0,2]

Table 4: The encoding of the type hierarchy of Figure 1 for the straight slicing of Table 2

i.e., if $2 \in I^b_{s_0} = [5,5]$. Hence O_1 is declared not to be of type a. On the other hand, if p_1 receives an object O_2 with [1,6], [0,2], p_1 can test if the type of O_2 is a subtype of e by checking if $id_e \in I^k_{s^e_i}$ i.e., $4 \in I^k_{s_0} = [1,6]$. Hence O_2 is declared subtype of type e.

4. THE ALGORITHM

This section describes how we obtain a straight slicing of a type hierarchy through our *DST* algorithm. The straight slicing obtained is then used to generate the encoding of the types that are transferred with the objects exchanged in the distributed system (as we just explained at the end of the previous section).

In short, the goal of our *DST* algorithm is to create a minimal number of straight slices that include all types of the hierarchy. As we will explain, the idea is to start from the root types, put each within a singleton slice, which is inherently straight, and then add other types of the hierarchy to existing straight slices, as long as the addition leave the slices straight. If not, new straight slices are added. The challenge is to minimize the number of new straight slices that are created. To simplify our presentation, we first assume a static hierarchy and later discuss how to dynamically add new types.

Our algorithm is decomposed into five main phases (Figure 4) (1) the *bootstrapping*, (2) the *identification* of the *conflicting* straight slices of a type (which we explain below), (3) the *addition* of this type into each of its conflicting straight slices, (4) the *concatenation* of the conflicting straight slices in which a type has been added and (5) the *finalization*. (The pseudo-code of the overall algorithm is given in Figure 5):

Both the bootstrapping and the finalization are done once, respectively at the beginning of the algorithm and at the end of it, while the other phases are done for each type of the hierarchy.

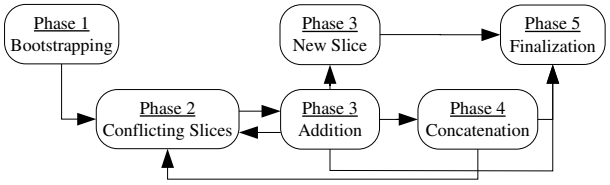

Figure 4: The different phases of *DST*

4.1 Bootstrapping

During the bootstrapping phase, each root type of the type hierarchy is added into a new distinct empty straight slice. The resulting slices, denoted by $s_0, s'_0, ...$, are initialized with the same slice identifier ($id_{s_0}, id_{s'_0}, ... = 0$) (these eventually get unique identifiers). For instance, if we consider the type hierarchy of Figure 1, the three root types, a, b and c, are put within three different straight slices: $s_0 = a, s'_0 = b$ and $s''_0 = c$.

4.2 Identification

After the bootstrapping phase, each type t of the type hierarchy S, starting from level 1 up to the highest level of S, is added to each of the straight slices of the set of *conflicting straight slices*, denoted by *confSlices(t)* (Figure 6). We determine this set as follows: we consider all the straight slices that contain at least one ancestor of t, and we select those that have the the maximum identifier id_{s_i}: the selected ones are the conflicting straight slices of t. There might be several of these (which explains why we consider a set *confSlices(t)*) because several straight slices might have the same identifier, in particular after the bootstrap.

```
1: {The set containing the different straight slices}
2: set sslices = {};

3: {Bootstrapping phase}
4: function boostrapping();
5:   for all t ∈ R(S) do
6:     new s_x; id_{s_x} = 0;
7:     add t in s_x;
8:     add s_x in sslices;
9:   end for
10: end

11: {Finalization phase}
12: function finalize()
13:   for all s_i ∈ sslices do
14:     if s_i contains only root types then
15:       append(s_i,s_0); // where id_{s_0} = 0
16:     end if
17:   end for
18:   for all s_i ∈ sslices do
19:     int id = 0;
20:     for all t in s_i do
21:       id_t = id;
22:       id^t_{s_i} = id_{s_i};
23:       id++;
24:     end for
25:   end for
26:   for all s_i ∈ sslices do
27:     for all t in s_i do
28:       compute I^t_{s_i};
29:     end for
30:   end for
31: end

32: {DST algorithm}
33: function encode
34:   boostrapping();
35:   for all level l > 0 of cth(S) do
36:     for all t at level l do
37:       boolean possibleToAdd = add(t);
38:       if possibleToAdd then
39:         array slices = [];
40:         for all s_i ∈ confSlices(t) do
41:           if t ∈ s_i then
42:             add s_i in slices;
43:           end if
44:         end for
45:         concat(t,slices);
46:       else
47:         new s_x; id_{s_x} = max(id_{T_x} ∈ sslices) + 1;
48:         add t in s_x;
49:         add s_x in sslices;
50:       end if
51:     end for
52:   end for
53:   finalize();
54: end
```

Figure 5: Encoding a type hierarchy

For instance, if we consider the type hierarchy of Figure 1, after the bootstrapping phase leading to the creation of straight slices $\{s_0, s'_0, s''_0\}$, *confSlices(d)* is $\{s_0, s''_0\}$. Indeed, both s_0 and s''_0 contain ancestors of d (respectively a and c) and their slice identifier (i.e., id_{s_0} and $id_{s''_0}$) is 0.

4.3 Addition

Once the set *confSlices(t)* has been computed, we add t into the straight slices s_i of *confSlices(t)*. The addition of t into a straight slice s_i of *confSlices(t)* is possible only if all the ancestors in s_i of any type u in S remain consecutive after the addition of t.

This condition implies that t can only be added at the head (resp. tail) of s_i (remember that the head/tail of a straight slice corresponds to the first/last element of the corresponding sequence of types). Indeed, if t is squeezed in between s_i, t will break the consecutivity between a type u and its ancestors in s_i and s_i will not remain straight.

Adding t at the head (resp. tail) of s_i implies that the head (resp. tail) of s_i must be an ancestor of t (otherwise t is not consecutive with its ancestors in s_i as s_i contains at least one ancestor of t, see Section 4.2). However, the fact that the head or the tail of s_i is an ancestor of t does not imply that all the ancestors of t are consecutive in s_i. For instance, consider the straight slice $s_f = b; d; e; c; a$ which corresponds to the output of the algorithm for the type hierarchy of Figure 7. Consider a type t which is a subtype of d and c. Even if the head and the tail of s_f are ancestors of t, it is not possible to add t at the head/tail of s_f, because t will not be consecutive with all its ancestors c, a, b, d as e is in between d and c. To test this condition, we check if all the parents of t are consecutive (i.e., if the set of consecutive ancestors of t in s_i contains all the parents of t).

At the end of this phase, t has been added into: (1) one straight slice, (2) no straight slice or (3) several straight slices. If t has been added into several straight slices of *confSlices(t)*, we proceed to the concatenation of the straight slices in which t has been added (see below).[4] On the other hand, if t was not added into any of the straight slices of *confSlices(t)* (because the straight slice does not remain straight), a new straight slice for t is created, its slice identifier is set to the maximum slice identifier of the straight slices in the current straight slicing plus one. Finally, in the case where t has been added into one straight slice only, the algorithm proceeds with the identification of the conflicting straight slices of a new type u (i.e., the second phase) and if all the types of the hierarchy have gone through the addition phase, proceeds to the finalization phase.

Consider for instance the type hierarchy of Figure 1 where the straight slicing up to type c corresponds to $\{s_0, s'_0, s''_0\} = \{a, b, c\}$ and *confSlices(d)* contains $s_0 = a$ and $s''_0 = c$ (as presented above). We can add d into s_0 and s''_0, as both a and c are ancestors of d. The resulting straight slices are $s_0 = d; a$ and $s''_0 = d; c$.

[4]Note that if a type t is added to multiple straight slices of *confSlices(t)*, the resulting straight slicing becomes non disjoint (as t belongs to more than one straight slice). This is, however, temporary, as the concatenation phase makes the slicing disjoint again.

```
1: {Retrieval of the consecutive ancestors of a type in a
   straight slice}
2: function getConsAncestors(t,s_i)
3:   set ancestors = {};
4:   int pos = 0;
5:   if head(s_i) = t then
6:     while ((pos < sizeof(s_i)) ∧ (u at position pos in s_i
       ∈ P(t,S))) do
7:       add u in ancestors;
8:       pos++;
9:     end while
10:  else
11:    pos = sizeof(s_i)-1;
12:    while ((pos ≥ 0) ∧ (u at position pos in s_i ∈
       P(t,S))) do
13:      add u in ancestors;
14:      pos–;
15:    end while
16:  end if
17:  return ancestors;
18: end

19: {Test if a slice is straight}
20: function isSliceStraight(t,s_i)
21:   if (t ∈ s_i ∧ P(t,s_i) ∈ getConsAncestors(t,s_i)) then
22:     return true;
23:   else
24:     return false;
25:   end if
26: end

27: {Addition phase}
28: function add(t)
29:   boolean result = false;
30:   for all s_i ∈ confSlices(t) do
31:     t_c new copy of t;
32:     if head(s_i) ∈ A(t,S) then
33:       add t_c to the head of s_i;
34:     else
35:       if tail(s_i) ∈ A(t,S) then
36:         add t_c to the tail of s_i;
37:       end if
38:     end if
39:     if isSliceStraight(t_c, s_i) then
40:       result = true;
41:     else
42:       remove t_c from s_i;
43:     end if
44:   end for
45:   return result;
46: end
```

Figure 6: Adding a new type t into *confSlices(t)*

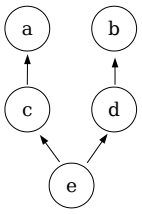

Figure 7: A subtype hierarchy that can be concatenated

4.4 Concatenation

We consider now the straight slices of *confSlices(t)* in which t has been added and we concatenate them, one by one (to reduce the total number of straight slices). The pseudo-code of this concatenation is outlined in Figure 8 and explained below.

We only concatenate two straight slices s_i and s_j if the ancestors of any type u in S remain consecutive after the concatenation. Therefore, the straight slices s_i and s_j are only concatenated at their head/tail and we denote this head/tail by ht (indeed, if s_j is squeezed inside s_i, this will break the consecutivity between a type u and its ancestors in s_i). If we consider the previous example where d was added into the straight slicing $\{s_0, s_0', s_0''\}$, the straight slices that are concatenated are $s_0 = d; a$ and $s_0'' = d; c$. In this example, we concatenate s_0 and s_0'' at their respective heads as this will not break the consecutivity of types a, b, c, d (as the only ancestor of a is a, of b is b, of c is c and d is consecutive with d, a and c).

To check that the concatenation of s_i with s_j at ht does not break any consecutivity between the ancestors of any type in s_i, s_j we make sure that: (1) ht is an ancestor of t and (2) for all types t that are part of the straight slicing (i.e., that have been added either in the bootstrapping or in the addition phase) and that have ancestors in both s_i and s_j that these ancestors are consecutive with ht.

If the conditions (1) and (2) are fulfilled, the concatenation of two straight slices s_i, s_j at ht is performed as follows (in the case where s_i, s_j do not contain only root types and t):

- If ht corresponds to the head (resp. tail) of s_i and to the tail (resp. head) of s_j, t is removed from s_j (remember that t was added in s_j during the previous phase) and we concatenate s_j with s_i (resp. s_i with s_j). For instance, if we have two straight slices $s_0 = t; u; v$ and $s_1 = w; x; t$, the concatenation of s_0 with s_1 is $w; x; t; u; v$.

- If ht corresponds to the head (resp. the tail) of both s_i and s_j, t is removed from s_j, and we concatenate *reverse(s_j)* with s_i (resp. s_i with *reverse(s_j)*), where *reverse(s_j)* corresponds to the reversed sequence of s_j (e.g., *reverse($a; b; c; d$)* corresponds to $d; c; b; a$). For instance, if we have two straight slices $s_0 = t; u; v$ and $s_1 = t; w; x$, the concatenation of s_0 with s_1 is $x; w; t; u; v$.

```
1: {Test if it is possible to concatenate two straight slices}
2: function testConcat(u, v, types)
3:   for all k ∈ types do
4:     if (u ∉ A(k,s_i) ∨ v ∉ A(k,s_j)) then
5:       return false;
6:     end if
7:   end for
8:   return true;
9: end

10: {We concatenate different straight slices together}
11: function concat(t,slices)
12:   while sizeof(slices) ≠ 1 do
13:     s_i = slice[0];s_j = slice[1];
14:     array types = [];
15:     for all s_i ∈ sslices do
16:       for all u ∈ s_i do
17:         if ((A(u,S) ∩ (s_i ∪ s_j)) ≠ ∅) then
18:           add u in types;
19:         end if
20:       end for
21:     end for
22:     if (head(s_i) ∈ A(t,S) ∧ head(s_j) ∈ A(t,S)) then
23:       if testConcat(head(s_i),head(s_j),types) then
24:         remove t in s_j; s_i = (reverse(s_j);s_i);
25:       else
26:         id_{s_j} = max(id_{s_x} ∈ sslices) + 1;
27:         remove s_j from slices;
28:       end if
29:     else
30:       if (tail(s_i) ∈ A(t,S) ∧ tail(s_j) ∈ A(t,S)) then
31:         if testConcat(tail(s_i),tail(s_j),types) then
32:           remove t in s_j; s_i = (s_i;reverse(s_j));
33:         else
34:           id_{s_j} = max(id_{s_x} ∈ sslices) + 1;
35:           remove s_j from slices;
36:         end if
37:       else
38:         if (head(s_i) ∈ A(t,S) ∧ tail(s_j) ∈ A(t,S))
            then
39:           if testConcat(head(s_i),tail(s_j),types) then
40:             remove t in s_j; s_i = (s_j;s_i);
41:           else
42:             id_{s_j} = max(id_{s_x} ∈ sslices) + 1;
43:             remove s_j from slices;
44:           end if
45:         else
46:           if (tail(s_i) ∈ A(t,S) ∧ head(s_j) ∈ A(t,S))
              then
47:             if testConcat(tail(s_i),head(s_j),types)
                then
48:               remove t in s_j; s_i = (s_i;s_j);
49:             else
50:               id_{s_j} = max(id_{s_x} ∈ sslices) + 1;
51:               remove s_j from slices;
52:             end if
53:           end if
54:         end if
55:       end if
56:     end if
57:   end while
58: end
```

Figure 8: Concatenating straight slices

In the case where both straight slices contain only root types and t, the concatenation is achieved as follows: (1) t is removed from s_j and (2) if t is the head (resp. the tail) of s_i, the root types of s_j that have another child than t in the type hierarchy S are concatenated at the tail (resp. at the head) of s_i. For example, if we have two straight slices $s_0 = t; u; v$ and $s_1 = t; w; x$ (in which u, v, w, x are root types and w has another child than t) the concatenation of s_0 with s_1 is $t; u; v; x; w$.

At the end of the concatenation, the slice identifier of the concatenated straight slice is set to the lowest slice identifier of the straight slices that have been concatenated.

If it is not possible to concatenate two straight slices s_i and s_j, a unique slice identifier is set for s_j. This is important because s_j might be a straight slice containing only a root type. Remember that after the bootstrapping, all straight slices are assigned the identifier $id_{s_i} = 0$.

To illustrate the concatenation of two straight slices, consider the straight slices $s_0 = d; a$ and $s_0'' = d; c$ which correspond to the output of the algorithm at the end of the addition of d into *confSlices(d)* (as presented above). The concatenation of s_0, s_0'' is possible and the new concatenated slice is $s_0 = d; a; c$. It is then possible to add e in both s_0, s_0' as both type c and type b are ancestors of e. Hence, at the end of the addition of e, we end up with: $s_0 = d; a; c; e$ and $s_0' = e; b$. It is furthermore possible to concatenate s_0 and s_0' as the concatenation lets the ancestors of a, b, c, d be consecutive in the new straight slice. Indeed, the ancestors of a, b, c are respectively a, b, c, and d remains consecutive with d, a, c while e is consecutive with a, b, c, e. The new concatenated straight slice s_0 corresponds to $d; a; c; e; b$. If now we consider adding f, we compute *confSlices(f)* $= s_0$. It is not possible to add f in s_0 as neither the head of s_0 (i.e., d) nor the tail of s_0 (i.e., b) are ancestors of f. Thus a new straight slice is created: $s_1 = f$.

4.5 Finalization

We first check that there is no straight slice containing only root types and which slice identifier is equal to 0 (this case might happen if the type hierarchy contains root types that do not have any descendants). If such straight slices $s_u, s_v, ..., s_n$ exist, they are appended at the tail of the first straight slice s_i that does not contain only root types. If s_i does not exist (i.e., the type hierarchy contains only root types), the straight slices $s_v, ..., s_n$ are appended at the end of s_u. In this case, appending straight slices at the end of another straight slice does not break any consecutivity between the ancestors of any type t of the type hierarchy, as the straight slices that are appended contain only types that do not have any descendants.

Then, we construct the encoding of each type of the hierarchy, i.e., for each type t we assign: (1) its identifier id_t, (2) its slice identifier $id_{s_i}^t$ and (3) for each slice s_i, its interval of ancestors $I_{s_i}^t$.

To assign an identifier to a type in a straight slice s_i, we simply parse the sequence s_i starting at its head up to its tail and we assign a unique identifier id_t to each item t of s_i incrementally. The slice identifier of a type t corresponds to

the slice identifier of the straight slice t belongs to. Finally, the computation of $I^t_{s_i}$ consists in the union of the identifiers of the ancestors of the parents of t.

For example, if the type f was the last type of the type hierarchy of Figure 1, the encoding of the resulting straight slicing $= \{s_0, s_1\}$ where $s_0 = d; a; c; e; b$ and $s_1 = f$ would be the one of Figure 5. For instance, the identifier of f is 0 (because f is at the first position in s_1), its slice identifier is 1 and the interval of its ancestors in s_0 is [2] and in s_1 is [0].

	id_t, $id^t_{s_i}$	$s_0 = d; a; c; e; b$	$s_1 = f$
a	1,0	[1]	$\emptyset$
b	4,0	[4]	$\emptyset$
c	2,0	[2]	$\emptyset$
d	0,0	[0,2]	$\emptyset$
e	3,0	[1,4]	$\emptyset$
f	0,1	[2]	[0]

Table 5: The encoding of the type hierarchy of Figure 1 up to type f

4.6 Addition of New Types at Runtime

So far, we assumed that a process can only perform subtyping tests against types in the core type hierarchy *cth*(S). Modulo one modification we explain now, *DST* supports the addition of new types at runtime and subtyping tests against those new types. The addition procedure is exactly the same as before except that we do not concatenate the straight slices anymore. This is justified by the fact that, when successful, a concatenation always implies changing the slice identifier of at least one straight slice. Since we want to preserve the encoding of the types that belong to such straight slices, we cannot change the slice identifier. In practice, this means that we lost the optimization that gives us the concatenation (in reducing the total number of straight slices).

If it is possible to add t into a straight slice $s_j \in$ *confSlices(t)*, then its slice identifier is id_{s_j} and its $I^t_{s_i}$ is determined by the union of the intervals of the parents of t for all s_i in S and, if $s_i = s_j$, also with the type identifier of t. The type identifier of t is the highest type identifier of s_j plus one, respectively to the lowest type identifier of s_j minus one, depending if t is added at the tail, respectively the head of s_j. If it is not possible to add t into an existing straight slice, then a new straight slice is created (and its slice identifier is set to the highest slice identifier plus one).

To illustrate the idea, consider the type hierarchy of Figure 9. In that case, the newly added type l is both a subtype of g and k. The conflicting slice of l with the highest identifier is s_1 and, consequently, the new slicing of the type hierarchy is made of s_0 (which remains the same) and $s_1 = l; k; i; f; j$.

It is not possible to concatenate s_0 and s_1 for that would not preserve the encoding of one of the original straight slices. Remember that we want to preclude global reconfigurations. The encoding of l is thus the following: id_l = -1, $id^l_{s_1} = 1$, $I^l_{s_0} = I^g_{s_0} \cup I^k_{s_0}$ = [0,3] $\cup$ [1,6] = [0,6], $I^l_{s_1} = I^g_{s_1} \cup I^k_{s_1} \cup id_l$ $= \emptyset \cup [0, 2] \cup \{-1\}$ = [-1,2].

We also add to the type identifier a unique identifier based on the process name. This is necessary as, concurrently, another process may create another new type u which shares the same parents with t, and hence is assigned the same identifier as t. In this case, if a process performs a subtyping test against t, the subtyping tests would succeed even if the process receives an object of type u (whereas u is not a subtype of t). We prevent this using local information based on the process name.

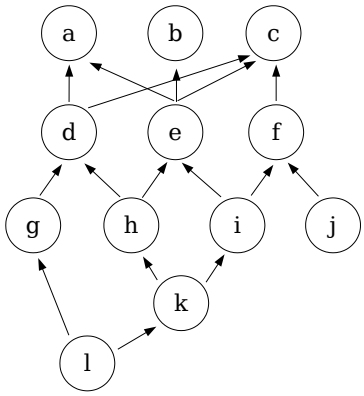

Figure 9: Addition of a new type l to the type hierarchy of Figure 1

Through the modification we just presented, it is not only possible to test if new added types are subtypes of types of the core type hierarchy *cth*(S), but also to perform subtyping tests against those new types. When a new type t is added into the system, we do not change the encoding of the types of *cth*(S), while still ensuring that the identifier of t is locally unique.

5. IMPLEMENTATION

We implemented a set of (40) Java 1.5 classes (1) to encode a type hierarchy according to *DST*, (2) to serialize Java objects into a standard output stream, along with their type encoding information and/or with their bytecode, (3) to deserialize such objects from a standard input stream as well as (4) to exchange encoded objects using a type-based publish/subscribe scheme (which illustrates the possibilities of distributed subtyping). The complete source code and APIs can be downloaded at
http://lpdwww.epfl.ch/baehni/dst.tgz and an *ant* file is available in the archive.

In the following, we first illustrate, with code examples, how the *DST* APIs can be used by Java application programmers. We then describe the general design underlying our implementation of *DST* and the associated serialization services. In particular, we give some details of the most relevant classes and packages.

5.1 APIs

To serialize and deserialize objects, the programmer creates a new instance of the basic class *DSCTH* to construct and encode the type hierarchy, specified by an array of classes that includes the leaves of the hierarchy. This is done both at the sender and receiver sides.

```
Class[] leaves = {Dummy.class};
DSCTH dscth = new DSCTH();
dscth.constructTypeHierarchy(leaves);
dscth.encode();
```

Once the type hierarchy is encoded, a process can serialize an object [5] in order, for instance, to send it through a TCP socket, *sdrSocket*. We illustrate this in the following through a *Dummy* object (see Figure 10).

```
Dummy obj = new Dummy();
DSSerializer serial = new DSSerializer(dscth);
OutputStream outStrm = sdrSocket.getOutputStream();
serial.serialize(obj, outStrm);
outStrm.flush();
```

At the other end of the socket, a receiver process may then obtain the object, encapsulated in a *DSObject*, through the wire (e.g., *rcvSocket*).

```
DSSerializer serial = new DSSerializer(dscth);
InputStream inStrm = rcvSocket.getInputStream();
DSObject newEvent = serial.deSerialize(inStrm);
```

It is then possible, out of the received instance of *DSObject*, to perform subtyping test over the encapsulated object:

```
newEvent.getType().
    isSubTypeOf(dscth.getGenericType("Dummy"));
```

The *isSubTypeOf()* method returns true or false depending on whether the object encapsulated in the *DSObject* instance is of type the one given as a parameter.

A more sophisticated type-based publish/subscribe can also be performed using our APIs as follows. The publisher executes the following in order to serialize an object (instead of the previous *serialize()* method above):

```
serial.serialize(serFormatTypeObj, obj, outStrm);
```

A subscriber might express interests in certain types, such as type b (see Figure 10) by executing the following:

```
dscth.addInterest(dscth.getGenericType("b"));
```

A subscriber that receives an object deserializes it by performing the following:

```
Object newEvent = subSerial.deSerialize(inStrm);
```

Subtyping tests against the interests of the subscriber are performed directly during the deserialization phase. If these tests are successful, the serialized object is returned; otherwise the *deSerialize()* method returns `null`.

[5]The object must implement the *Serializable* interface.

5.2 Design Overview

Our implementation is structured into several main packages: (1) *algorithms*, (2) *serialization*, (3) *utils* and (4) *exceptions*. Figure 11 depicts the dependencies between these packages. The *serialization* package uses the *algorithms* package to serialize new Java objects. The *exceptions* and *utils* packages are used by all the others. We designed our APIs in a generic way, i.e, using Java interfaces in order not to be tied to a specific implementation (of the algorithm, or of the serialization mechanism). We overview now the packages, together with their classes.

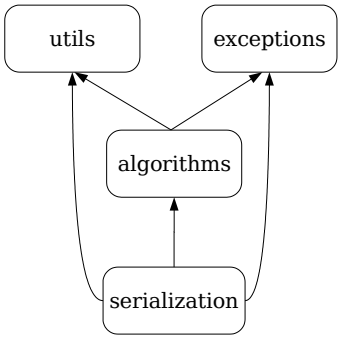

Figure 11: Dependencies between the packages

5.3 Algorithms Package

This is the main package of our implementation. It contains the classes used to create a type hierarchy, to encode it and to perform subtyping tests using these types.

GenericCTH. This is the main class of this package and is a super-class of *StringCTH*, *DSCTH* and *AssemblyCTH*, as presented in Figure 12.

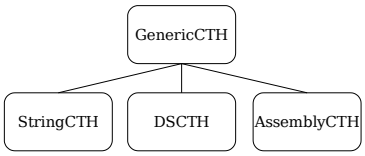

Figure 12: *GenericCTH* inheritance tree

The *GenericCTH* abstract class encodes the core type hierarchy. This is achieved through the *constructTypeHierarchy()* method. The parameter of this method is an array of *Class* which represents the leaves of the hierarchy. During the creation process, a set of *Level*s is created. Each *Level* contains an array of *Class* corresponding to the classes at the specified level of the type hierarchy. *GenericCTH* also exports method *getGenericType()*. This method is redefined in the subclasses and returns a *GenericType* (see Figure 13). As we will see below, the different classes redefining *GenericCTH* are responsible for encoding the type hierarchy. *GenericCTH* also defines the identifier $cth(S)$ used during the subtyping tests. This identifier corresponds to the 128 bits hashcode of the concatenation of the name of the root types of the hierarchy S.

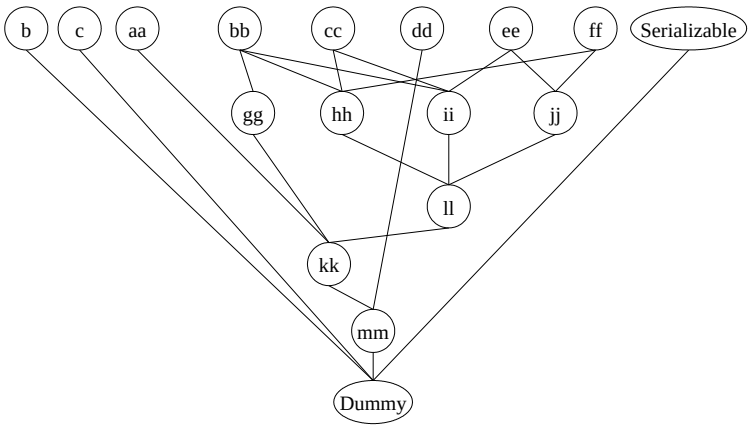

Figure 10: Type hierarchy of the Dummy class used in our performance tests

It is possible (for a subscriber) to express interest in specific classes by giving them as a parameter of the *addInterest()* method of *GenericCTH*. Hence, when the deserialization happens, the object to deserialize is only returned in case its type was subscribed to, as described in Section 5.4.

The two following classes are provided for performance comparisons.

StringCTH. This class defines the *encode()* method of *GenericCTH*, which implements the string encoding alternative against which we compared our protocol. It also redefines the *getGenericType()*, returning an instance of a *StringType*. This instance can then be used to check whether a *StringType* is a subtype of another *StringType* or not.

AssemblyCTH. This class represents the code downloading alternative and has an empty implementation. Recall that serialization and deserialization using this alternative does not require a core type hierarchy to be encoded, since it relies directly on the native core type hierarchy of Java.

DSCTH. This class implements our *DST* algorithm, as presented in Section 4. It handles several data structures, such as *DSEncodingType* and *DSSlice*, explained below. The main phases described in Section 4 are implemented: the *bootstrapping()* method; the *finalization()* method; the *getConfSlices()* method, used to get the conflicting straight slices; the *merge()* method, used to concatenate the different straight slices; and the main encoding method, *encode()*.

Throughout the encoding, *DSCTH* uses *DSEncodingType* structures to represent a type in $cth(S)$. However, at the end of the encoding, only objects of *DSType* (explained in the following) are manipulated. Straight slices are represented and manipulated through *DSSlice* objects, which we describe below.

DSEncodingType. This is the most complex data structure used in our implementation. It basically represents an item of a modified double linked list. The double linked list is used for implementing the sequence of types in a straight slice.

An instance of a *DSEncodingType* stores the set of its parents, as well as the straight slice (*DSSlice*) it belongs to. This is used to implement the different methods for (1) retrieving the ancestors of a type, (2) retrieving the ancestor intervals, (2) retrieving the straight slice to which the type belongs, and (3) printing its encoding.

DSSlice. A *DSSlice* object represents a straight slice. It contains the different types that have been added to this straight slice and hence can provide useful information: the head/tail of the straight slice and the root types.

GenericType. This class represents a generic type containing the necessary information for the subtyping tests. Again, this class is abstract and redefined by *StringType* and *DSType* which we describe below.

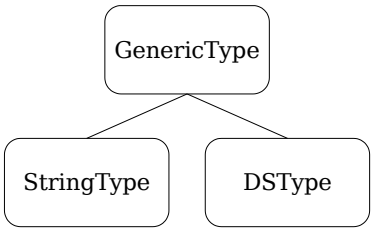

Figure 13: *GenericType* inheritance tree

StringType. This class represents the generic type used when the string representation is considered for comparison purposes. It implements the *isSubTypeOf()* method to compare the ancestors of one *StringType* with the others, supplied as parameters. The ancestors are stored in a string, as explained in Section 2.

DSType. This class represents a type according to our *DST* algorithm. The class stores the $cth(S)$, the identifier of the type, as well as a *Parents* data structure (see below). Similarly to *StringType*, the class implements the *isSubTypeOf()* method. This is simply done by checking if the identifier of the type is contained in the *Parents* structure of the *DSType*

given as a parameter.

Parents. This class stores the intervals of the parents of a type for all the different straight slices. An interval is basically an array of pairs of integers. During the serialization process, such integer pairs are converted into shorter bit arrays using the integer encoding technique described in Section 5.5.

GenericObject. A generic object represents the data structure manipulated by the serialization process. The standard deserialization returns a *GenericObject* instead of the traditional typed-object. A generic object, in its initial form, only stores the *thid*(S) and a byte array representing the serialized object. Its subclasses are responsible for storing the ancestors of the type of the object.

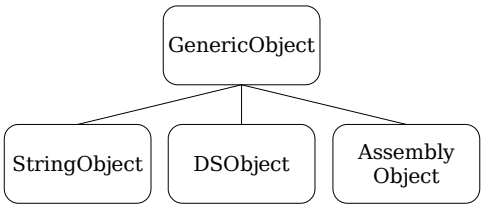

Figure 14: *GenericObject* inheritance tree

AssemblyObject. An *AssemblyObject* contains simply the Java Object it represents.

StringObject. A *StringObject* contains the necessary information to perform subtyping tests over an object with a string representation. Basically in addition to the *thid*(S) and the data, it stores also a string containing the ancestors of the type of the object. Moreover this class redefines the *getType()* method in returning a *StringType*.

DSObject. The *DSObject* class extends the *GenericObject* one for supporting the *DST* algorithm. It stores (with the *thid*(S) and the data) the integer intervals of its ancestors. Moreover it redefines the *getType()* method to return a *DSType* that can be used for the subtyping tests (via the *isSubTypeOf()* method).

5.4 Serialization Package

This package contains the classes needed for serializing and deserializing a Java object, for each of the approaches we consider: (1) *AssemblySerializer* (code downloading approach), (2) *StringSerializer* (string representation approach) and (3) *DSSerializer*. The first two approaches, as we pointed out earlier, are used in our performance comparisons.

All serializers inherit from the *Serializer* class (see Figure 15). This abstract class implements the basic *serialize()*, and *deSerialize()* methods together with the specific *deSerialize()* method used for performing type-based publish/subscribe interactions.

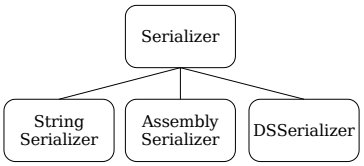

Figure 15: *Serializer* inheritance tree

```
// Basic (de)serialization methods
boolean serialize(Serializable s,
                  OutputStream outStream);
GenericObject deSerialize(InputStream inStream);

// Type-based (de)serialization methods
boolean serialize(byte serializationFormat,
                  Serializable s,
                  OutputStream outStream);
Object deSerialize(InputStream inStream);
```

The first method serializes an object that provides the standard *java.io.Serializable* interface to an output stream of bytes, such as the one provided by the *getOutputStream()* method of the standard *java.net.Socket* class. The second method is then called by a process at the other endpoint of such a byte stream to deserialize the object into a *GenericObject*. The two last methods are used for the implementation of the type-based publish/subscribe: the third method serializes an object according to a specific serialization format presented below while the last method returns (a) the deserialized object in case its type is a subtype of at least one of the expressed types of interest, or (b) a null reference otherwise.

All methods call in turn specialized methods that are responsible for the serialization and the deserialization of type encodings (*serializeType()* and *deSerializeType()*, respectively). The actual serialization and deserialization of objects rely on the standard Java support for object serialization.

Serialization Format Options. The output generated by the *serialize()* method of our type-based publish/subscribe APIs follows one of three possible serialization format options, supplied as an argument (i.e., *serializationFormat*):

- *serFormatTypeObjClasses.* In this format, the serialization of an object includes its type encoding (except for the case of the code downloading approach), the serialized object data and the corresponding class code (serialized by a binary copy of the contents of the corresponding class file). This is the most complete format, which enables the receiver process to perform the subtyping test and, in case of interest in the object's type, has immediate access to the serialized object data and its class code.

  Should the class code be loaded (if the received object is of interest) via our custom class loader (implemented by the the class *ByteArrayClassLoader*) the object is then deserialized and, hence, returned to the

application via a call to the *deserialize()* method. In case the class loader of the receiver process requires the code of other classes to load the object's class, a request for the code of each such class is sent back to a *ClassCodeProvider* server thread running at the sender process.[6] Each such request is triggered whenever the class loader at the receiver process, during the object's class loading process, tries to load a class whose code is not locally available; in response, the *ClassCodeProvider* provides the class loader with the requested code.

- *serFormatTypeObj.* In case the receiver process already has the object's class code loaded (or locally available for loading), serializing and sending its class code is unnecessary. As an alternative, this format excludes the class code from the serialization output, thus reducing its length. In this case, the receiver process simply tries to deserialize the serialized object data, which will trigger the loading of the required classes. Just as in the case of the previous format, should any class code be unavailable (including the code of the object's class), it is obtained by a request to the *ClassCodeProvider* at the sender process.

  It is important to notice that removing the class code from the serialization output does not require any addition of other information to allow the class loader to determine the class to be loaded. The serialized object that results from the standard Java object serialization services already includes an identifier of the corresponding class.

- *serFormatOnlyType.* The third format is only applicable to the string representation and *DST* approaches. It is suitable for situations where the type of the serialized object fails, with a high frequency, the subtyping test against the types of interest of the receiver process. In case of a failed subtyping test (performed using only the type encoding), the actual deserialization of the object's data is not performed, and hence such information is not necessary.

  This format replaces the serialized object data by an *object identifier*, generated by the sender process and unique to that process, hence reducing the serialization output length. Additionally, the serialized object data is stored in an *ObjectStore* server thread at the sender process. In case the receiver process happens to be interested in obtaining the object, it requests it from the *ObjectStore* using the received object identifier.

The serialization format option is encoded as a prefix of the serialization output; this prefix is then analyzed by the *deserialize()* method prior to the remaining deserialization steps. Therefore, the serialization format employed for each call to *serialize()* may vary.

[6]Namely, the code of super-classes, member classes or classes referenced within the object's class may be needed if it is not available at the receiving process. An alternative would be to include the code of all these classes in the serialization output, introducing a possibly significant overhead but preventing the occurrence of class code requests from the receiver process.

AssemblySerializer. The code downloading approach does not add any additional type encoding to the serialized object, since it relies exclusively on the standard subtyping support of Java. Accordingly, the *serializeType()* and *deSerializeType()* of *AssemblySerializer* are empty methods.

StringSerializer. The *serializeType()* and *deSerializeType()* methods of the *StringSerializer* class respectively write and read a string identifying the set of ancestors of the object (as well as the $cth(S)$). Subtyping is performed by a string look-up of the string identifiers of the types of interest (expressed in *StringCTH*) in the deserialized string.

DSSerializer Class. The *serializeType()* and *deSerializeType()* methods of the *DSSerializer* class respectively encode and decode the $cth(S)$ and the list of the intervals of the ancestors of the type of the object. The interval values and their respective slice identifiers are encoded as described in Section 5.5.

5.5 Bit-Encoding of Serialized Integers

As explained in Section 4, the encoding of a type t consists of: (1) the intervals of the ancestors of type t (each interval being represented by a slice identifier and containing the type identifiers of the highest and the lowest ancestor of t in that straight slice), (2) the slice identifier, (3) the type identifier and (4) $cth(S)$. We chose to implement the encoding of an identifier using an array of bits representing the absolute value of the identifier preceded by a bit of sign (our implementation supports negative identifiers).

To deal with the variable size of the encoding (e.g., "1" is not represented with the same number of bits than "3"), an initial mark (a 0-bit) and a final mark (a sequence of n 1-bits) are appended to the identifier, which is encoded as groups of n bits. Whenever one such group of n bits is identical to the final mark, it is repeated in the actual encoding. Therefore, final marks can be distinguished by a unit of n 1-bits that is followed by a 0-bit or by the end of the array.

We optimized the encoding the intervals of the ancestors. We encode the second value of the interval (representing the highest type identifier of the ancestors in the interval) as the relative offset to the first value. With the exception of intervals whose lowest type identifier is negative and whose highest type identifier is positive, the relative value is smaller than the absolute value of the highest type identifier. Therefore, the resulting encoding length of the interval is reduced. One especially advantageous situation is the encoding of singleton intervals, where the relative value is zero, hence the interval becomes encoded in a particularly efficient manner.

Implementing the encoding using the above technique does not impose any assumption on the size of the straight slices and thus allows the algorithm to dynamically add new types to its straight slices. Furthermore, it has the desirable advantage of encoding small identifiers with a small number of bits and hence suits perfectly *DST*.

Upon an experimental evaluation under the conditions described in Section 6, the optimal unit size was found to be $n = 1$. The results of Section 6 use such unit size.

6. PERFORMANCE

This section presents some performance results of our Java implementation of *DST* and compares them to the main alternatives mentioned in Section 2: string, code downloading and CPQE, an optimized version of the centralized PQE algorithm. The interest of the string approach is its simplicity and the very fact that it does not require global reconfiguration. CPQE is interesting because it is the most efficient centralized approach we know of. The code downloading approach is interesting because it is the only way to perform subtyping tests in current distributed systems.

All measurements were obtained using an *Intel Pentium 4* 2.66 GHz computer with 1GB RAM, running Java virtual machine version 1.5.0-b64 on a *Fedora Core 2* (kernel 2.6.11) operating system. All the presented values are averaged over 10000 measurements.

We considered the type hierarchies of Java 1.5 (around 12500 classes), Java 1.4.2 (8900), Java 1.3.1_15 (4500) and Java 1.2.2 (4500) as core type hierarchies. More precisely, we considered all Java 1.5 classes, 96% of the Java 1.4.2 classes, 78% of the Java 1.3.1 classes and 99% of the 1.2.2 Java classes- The rest of the classes are not compatible with our Java 1.5 implementation. To have fair comparison with CPQE, all measurements assumed the existence of a unique type hierarchy.

6.1 Performance of Type Encoding

An average time of 0.691 ms is taken by *DST* to initially encode a type hierarchy. This is for instance substantially higher than the time taken with a string approach (0.063 ms). This is explained by the increased complexity of our algorithm. Since the encoding of a core type hierarchy occurs only once, at the initialization of the system, we consider a delay of less than a millisecond to be very acceptable.

Table 6 depicts the encoding length per type, in bits, averaged over the total number of types for each Java type hierarchy. We also distinguish for *DST* and the string approach, the number of bits that (1) must be sent along with each object of that type and (2) must be maintained in order to perform subtyping tests against such type. The results from [16] do not allow us to make this discrimination in the case of CPQE.

Java Version	*DST*		String		CPQE
	(1)	(2)	(1)	(2)	(1)+(2)
1.2.2	12.4	4.5	432.6	29.1	10
1.3.1_15	12.3	4.5	434.3	29.8	18
1.4.2	12.2	4.3	437.4	30.6	-
1.5	12.2	4.5	510.0	35.3	-

Table 6: Average number of bits for the encoding

The encoding length of *DST* outperforms the string approach by a factor of more than 35 for the type information sent with the objects, and 2.5 to 7.8 for the information maintained by processes that need to perform subtyping tests. In fact, the difference is bigger if one considers that the name of the classes have more than one letter (which is typically the case in most applications). More importantly, *DST* is comparable, in terms of encoding length, to CPQE. This might be explained by the fact that the number of ancestor intervals needed to encode a type is typically one, which is a consequence of the low average number of straight slices per type hierarchy (1.019 straight slices).

6.2 Performance of (De)Serialization

We also measured the performance of our implementation when serializing/deserializing an object of the *Dummy* class (see Figure 10) in order to transfer it among different processes.[7] The resulting serialized byte array contains the (previously encoded) type information of the object (in the case of *DST* and the string approach), the bytecode of the object, as well as the object itself.

The time taken to serialize the *Dummy* object is presented in Figure 16 for *DST*, the string and the code downloading approaches. Clearly *DST* outperforms the string approach but does not perform as well as the code downloading approach by 82.76 μs. This overhead is due to the time to serialize the encoding of the type.

The serialized object was then sent by a process p_i, over a local TCP socket to a process p_j that, upon a positive subtyping test with the object's type against any of the types received by p_j, completed the deserialization to obtain the object. During the deserialization, four distinct situations may happen with respect to (1) the types received by p_j and (2) the availability of the bytecode of the object. We depict all the cases in Figure 16.

As expected, *DST* and the string approach are much better when the subtyping test fails. *DST* outperforms the code downloading approach by factors of 3 and 12, depending, respectively, on whether the class of the object is already loaded or not in the Java virtual machine of p_j. On the other hand, if the subtyping test succeeds, the overhead induced by *DST* does not hamper the complete deserialization process.

It is surprising to notice that both *DST* and the string approaches are comparable (we could have expected *DST* to be better). This result can be explained by the fact that the algorithm used to (de)serialize the type information of the object in a length-efficient way, in *DST*, is quite complex. If the deserialization time is more important than the length of the encoding, we can then use plain bytes for encoding the type information and consequently outperform the string approach by a factor of 4. Regarding the time taken to actually perform the subtyping test, Figure 16 depicts the fact that this delay is negligible (around 10μs with every approach) with respect to the time taken for the deserialization.

Finally, Table 7 conveys the information about the worst encoding length that is achieved for the specific Java version.

This encoding corresponds, in Java 1.5, to *java.awt.dnd.DnDEventMulticaster*. This type hierarchy has 18 straight slices. In this case, our algorithm still performs

[7]It was not possible to experiment the serialization and deserialization on all Java classes, for they do not all implement the *Serializable* interface.

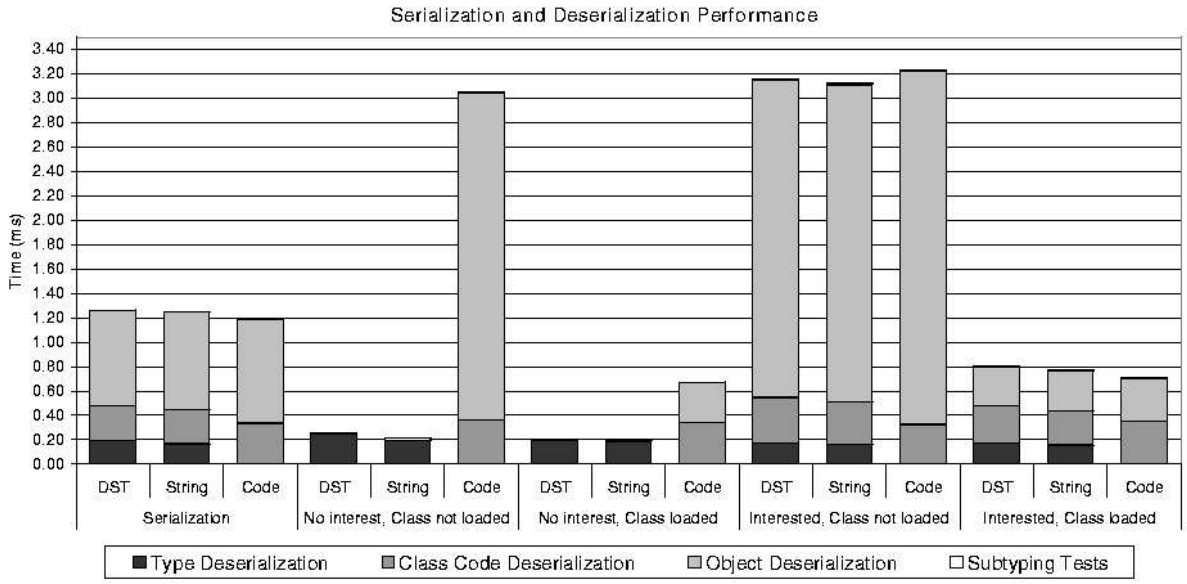

Figure 16: Serialization and deserialization time of a *Dummy* object

Java Version	*DST*		String	
	Sender	Receiver	Sender	Receiver
1.2.2	277	15	1432	1432
1.3.1_15	277	15	1432	1432
1.4.2	294	15	1432	1432
1.5	294	15	2494	2494

Table 7: Maximum number of bits to encode type information, as held by receiver processes and sent by sender processes

between 3 and 6 times better than the string-based approach.

7. CONCLUDING REMARKS

This paper presents an algorithm to perform subtyping tests in a dynamic distributed environment where types can be added at runtime. Our algorithm encodes a multiple subtyping hierarchy in a memory-efficient manner and can perform subtyping tests over each type of the hierarchy without downloading its code nor having to deserialize objects of that type. We avoid reconfiguration when new types are added and ensure that the encoding of the core type hierarchy, i.e., of the set of types present at the initialization of the system, remains the same throughout the lifetime of the system. The performance measures we obtain convey the practicality of our algorithm in terms of the size of the messages exchanged in the network as well as the time taken to out subtyping tests. We show that the performance of our algorithm is comparable to the best currently known centralized subtyping algorithm [16], which requires however reconfiguration if new types are dynamically added.

An important challenge of the design of our algorithm was to enable subtyping tests against types that are added dynamically, without global reconfiguration. To handle the tricky situation where processes concurrently add new types, we had to adopt a scheme where elements from a process name are added to a type identifier. This form of encoding is not ideal if different processes successively add new types one after the other, and would end up not being efficient in this case (more specifically, if a type t is a subtype of two types belonging to the same slice s_i, which do not share any common root type, t would have multiple intervals $I^x_{s_i}$ for one slice s_i). This issue seems inherent to handling the dynamicity of a distributed environment in a concurrent manner, and it would be interesting to prove a fundamental lower bound on the memory required to handle concurrent additions.

8. REFERENCES

[1] R. Agrawal, A. Borgida, and H. V. Jagadish. Efficient Management of Transitive Relationships in Large Data and Knowledge Bases. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*, pages 253–262, may 1989.

[2] A. B. Arulanthu, C. O'Ryan, D. C. Schmidt, M. Kircher, and J. Parsons. The Design and Performance of a Scalable ORB Architecture for CORBA Asynchronous Messaging. In *Proceedings of the 2nd IFIP/ACM Middleware Conference*, april 2000.

[3] N. H. Cohen. Type-Extension Tests can be Performed in Constant Time. *ACM Transactions on Programming Languages and Systems*, 13:626–629, 1991.

[4] P. Eugster, P. Felber, R. Guerraoui, and A.-M. Kermarrec. The Many Faces of Publish/Subscribe. *ACM Computing Surveys*, 35(2):114–131, june 2003.

[5] S. M. Inc. *Java Message Service - Specification, version 1.1.* http://java.sun.com/products/jms/docs.html, 2005.

[6] *Java 1.5 Language Specification*. http://java.sun.com/j2se/1.5.0/docs/index.html, 2005.

[7] A. Krall, J. Vitek, and R. N. Horspool. Efficient Type Inclusion Tests. In *Proceedings of the 12th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 142–157, october 1997.

[8] A. Krall, J. Vitek, and R. N. Horspool. Near Optimal Hierarchical Encoding of Types. In *Proceedings of the 11th European Conference on Object-Oriented Programming*, pages 128–145, june 1997.

[9] Microsoft. *Microsoft Message Queuing*. http://www.microsoft.com/windows2000/technologies/communications/msmq, 2005.

[10] *.NET Framework Reference Documentation*. http://www.microsoft.com/net/, 2005.

[11] B. Oki, M. Pfluegl, A. Siegel, and D. Skeen. The Information Bus - An Architecture for Extensible Distributed Systems. In *Proceedings of the 14th ACM Symposium on Operating System Principles (SOSP '93)*, pages 58–68, december 1993.

[12] OMG. *The Common Object Request Broker: Architecture and Specification*, 2001.

[13] K. Palacz and J. Vitek. Java Subtype Tests in Real-Time. In *Proceedings of the 17th European Conference on Object-Oriented Programming*, july 2003.

[14] L. K. Schubert, M. A. Papalaskaris, and J. Taugher. Determining Type, Part, Colour, and Time Relationships. *Computer*, 16 (special issue on Knowledge Representation):53–60, october 1983.

[15] J. Vitek, R. Horspool, and A. Krall. Efficient Type Inclusion Tests. In *Proceedings of the 12th ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 142–157, 1997.

[16] Y. Zibin and J. Gil. Efficient Subtyping Tests with PQ-Encoding. In *Proceedings of the 16th ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, october 2001.

[17] Y. Zibin and J. Gil. Fast Algorithm for Creating Space Efficient Dispatching Tables with Application to Multi-Dispatching. In *Proceedings of the 17th ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 142–160, november 2002.