Efficient Distributed Subtyping Tests

Sébastien Baehni
I&C EPFL
Switzerland
sebastien.baehni@epfl.ch

João Barreto*
INESC-ID/IST
Portugal
joao.barreto@inesc-id.pt

Patrick Eugster†
Dept. of Computer Science
Purdue University, USA
p@cs.purdue.edu

Rachid Guerraoui
I&C EPFL
Switzerland
rachid.guerraoui@epfl.ch

ABSTRACT

Subtyping tests are essential in typed publish/subscribe infrastructures, especially when the underlying programming language supports subtype conformance, as in Java or C#. These tests are particularly challenging when the publish/-subscribe infrastructure is distributed, because processes have diverging views and new types may be added in a decentralized manner. Maybe surprisingly, subtyping tests for such distributed systems have been devoted only little attention so far; they are usually strongly intertwined with serialization and code transfer mechanisms.

This paper presents an efficient subtype testing method for event objects received through the wire, requiring neither the download of a full description of the types or classes of these objects nor their deserialization. We use a slicing technique that encodes a multiple subtyping hierarchy with as little memory as the best known centralized type encoding, but allows for the dynamic addition of event types without re-computing the encoding.

We convey the practicality of our approach through performance measures obtained with standard Java libraries in a publish/subscribe system. Our approach performs between 3 and 12 times faster than a code transfer approach without adding overhead to object deserialization, and requires the same testing time as a straightforward string-based type encoding while reducing the encoding length by a factor of 50.

Categories and Subject Descriptors

C2.4 [**Computer Communication Networks**]: Distributed Systems—*distributed applications*; D3.3 [**Programming Languages**]: Language Constructs and Features—*Data types and structures*

General Terms

Design, Languages

Keywords

Java, type, subtype, conformance, event

*Funded by FCT Grant SFRH/BD/13859 and MINEMA.
†Funded by NSF CAREER Grant 0644013.

DEBS '07, June 20–22, 2007 Toronto, Ontario, Canada

1. INTRODUCTION

At the heart of typed publish/subscribe-like interaction models with pass-by-value semantics lies the *subtyping test*, also called *type inclusion test*. In short, this test consists in verifying whether a given type t is a subtype of another type r. This elementary test, used to decide whether an object O (of a given type t) is an instance of a type r, occurs of course frequently at regular program execution, either implicitly as in Java [10] array assignments, upon dynamic dispatch in exception clauses, or explicitly as triggered by means of the `instanceof` operator (resp. `is` in C# [11] or `isKindOf` in Smalltalk [9]).

Significant seminal work on subtype tests has been centered around *encoding schemes* for assigning identifiers to types in a way using minimal *space* (sizes of identifiers, data structures reflecting subtyping relationships) and execution *time* (overhead of testing, construction of supporting data structures) [18, 3, 22, 12, 1, 23, 24].

In the dynamic distributed settings we consider in this paper, subtyping tests are crucial since *individual* processes might very well introduce new types *on the fly*, for example as subtypes of existing types. As pointed out in [19], type descriptions are not necessarily globally available due to dynamic joins, leaves, and updates of distributed components, and because there is no central authority that can be relied upon. Moreover, any communication of such information between remote hosts requires network resources and as such is subject to latency and failures. Consider the simple case of a process p_i sending an object O of type t to a process p_j expecting type r. It might be very well clear to p_i that t is a subtype of r, but p_j might never have heard of t, especially if p_i and p_j have not constantly been connected by a wired infrastructure (cf. [4]). The Java serialization mechanism for instance stipulates that a full description of the class C (e.g. its byte-code) of such an object O is available, or that an exception be raised if this is not the case. This assumption has been carried over to all object-oriented distributed systems we know of (e.g., CORBA [16], JMS [15], MSMQ [13],

.NET [14]). At first glance it seems straightforward to remedy the situation by putting a "code download" approach to work, where p_j asks p_i for C but C might also depend on further types/classes unknown to p_j. Transferring *everything* C depends upon leads to unnecessary communication, wasting network resources utterly and stalling p_j, p_i, as well as any processes depending on either of those.

With pass-by-value semantics in event-based interaction models à-la publish/subscribe, the problem becomes increasingly important as these entail the exchange of event objects by value at high rates between publishers and subscribers [21]. With subscriptions reflecting long-lasting remote interactions, it is likely that they witness the emergence of new (sub)types of event objects, especially when types are part of the subscription criteria [7]. In addition, certain processes might be relaying objects on behalf of others only, in which case these processes might not even require the full description of certain types. Besides, also with pass-by-reference semantics, as typical for client/server models, the addition of new (sub-)types at runtime is a realistic scenario [2], and the transfer of entire classes for the mere purpose of performing subtyping tests seems particularly onerous, since these classes will not even be instantiated. Ideally, in a strongly typed system, any object representation sent over the wire should encompass a type identification enabling global subtyping tests, which are *independent* from serialization and code transfer mechanisms.

Unfortunately, one cannot simply extend the centralized type encoding schemes mentioned above to dynamic distributed systems, since these require the set of types T pertaining to a given application to be entirely known when the encoding is determined. Furthermore, these assume complete descriptions of the types (typically byte-code representations) containing explicit information on super-types to be readily available in order to *reconfigure* the encoding. In a distributed system, reconfigurations become extremely costly as they require global agreement; just like the updating of processes with new types by uploading corresponding descriptions, they can be run as background tasks, but should not make a process or even entire application stall, even if new (sub)types are only seldomly added [2].

This paper addresses the problem of distributed subtype testing by presenting a novel *distributed type encoding* for efficient *distributed subtyping tests*, and a corresponding algorithm, called DST, designed for an environment with nominal multiple subtyping.[1] Our DST algorithm provides two important features:

- it *retains efficiency* of local subtype testing in time (no additional latency) and space (type identifiers are kept concise to spare network resources),
- it *permits extensibility* of the type encoding without forcing global reconfiguration.

The latter feature enables processes to perform subtyping tests without being compelled to download complete type descriptions when receiving objects, or to deserializing these objects.

Basically, we view the type hierarchy as disjoint sequences of types called *slices*. Each type is assigned an identifier corresponding to its position in a slice. The identifier of a type t, together with the *intervals* of the identifiers of its super-types (simply called the intervals of t) in the respective slices, represent the encoding of t. To test if a type t is a subtype of a type r, DST checks if the identifier of r is contained in the interval of t within r's slice. The addition of new types is then handled by extending an existing slice or by creating a new one (in certain specific cases). Roughly speaking, our scheme can be viewed as a combination of [23] and [22] with a fundamental difference: we order the *ancestors* of a type instead of its *descendants*. This is key to avoiding global reconfiguration with little memory for the encoding.

An implementation of our DST algorithm (using Java 1.5) is available at `http://lpd.epfl.ch/baehni/dst.tgz`. The algorithm is made accessible to Java programmers (e.g., for use in middleware packages) in the form of a comprehensive set of Java APIs, together with a set of wrapper classes around the Java serialization classes.

We describe performance measures of our algorithm obtained with standard Java libraries in a publish/subscribe system. Our performance measurements, conducted through type hierarchies of the standard Java (1.5, 1.4, 1.3 and 1.2) class libraries, convey the fact that DST performs a subtyping test between 3 and 12 times faster than a standard *code downloading* approach without increasing the time taken to deserialize the object. Moreover, DST requires the same subtyping test time as a straightforward *string-based encoding* approach, in which the type of an object is naively encoded via the name of the type and recursively through the names of its super-types (e.g., an encoding of type k in Figure 1 could be `"./a/b/c./d/e/f./h/i./k"`). With respect to this approach, DST however reduces 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 [23]. Yet, as pointed out, DST is designed for dynamic distributed environments where new types can be added at run-time without requiring global reconfiguration.

Roadmap. Section 2 presents some basic elements needed to present our DST algorithm. Section 3 presents the algorithm. Section 4 describes the key elements underlying our Java implementation of DST and illustrates its use. Section 5 provides performance measurements. Section 6 summarizes related work whereas Section 7 draws some final conclusions.

2. BASIC ELEMENTS

This section introduces few definitions before describing the encoding we consider to represent type hierarchies in distributed settings.

2.1 Types and Subtypes

A *type hierarchy* is a partially ordered set $S = (T, \prec)$, where T is a set of types $\{r, t, ...\}$ and $\prec$ is the reflexive, transitive and anti-symmetric *subtyping* relation. Type t is said to be a *subtype* of a type r if $t \prec r$. In this case r is said to be a *super-type* of t. The hierarchy of types present at the bootstrapping of the system is called the *core type hierarchy*, and is denoted by a unique identifier $cth(S)$ (e.g., in a typed publish/subscribe system this includes the set of types in which subscribers can express interests). Every type that is later added to the system is attached to one

[1]For presentation simplicity and alignment with seminal work, we focus on (abstract) types without further distinguishing how they are declared (e.g., classes, interfaces).

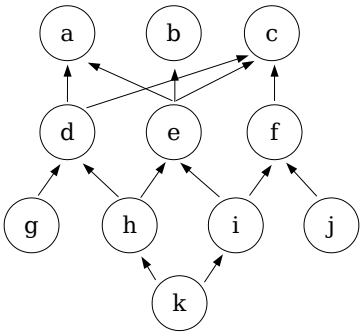

Figure 1: Example of a type hierarchy. Circles represent the types of the hierarchy while arrows represent *subtyping* relationships (e.g., ⓐ ← ⓔ means e is a subtype of a).

core type hierarchy. Figure 1 depicts a hierarchy $S = (T, \prec)$ with $T = \{a, b, c, d, e, f, g, h, i, j, k\}$. 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.

We reuse the following definitions from [23] (formally presented in Figure 2):

- 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: $D(a, S) = \{a, d, e, g, h, i, k\}$.
- 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: $A(g, S) = \{a, c, d, g\}$.
- 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, $C(a, S) = \{d, e\}$.
- 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, $P(g, S) = \{d\}$.
- 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, $R(S) = \{a, b, c\}$.
- 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, $L(g, S) = 2$.

2.2 Slicing

A *type sub-hierarchy*, or simply *sub-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, with identifier i, in a type hierarchy is a *sequence*[2] of types $a; b; c; ...$ of the hierarchy. (Because a slice is a sequence, we will sometimes talk about the head and the tail of the slice.) A concatenation of two slices s_i and s_j is written $s_i \oplus s_j$ ($\oplus$ is associative).

[2]Our definition differs here from the one in [22, 23] in that we consider a slice to be a sequence instead of a set.

Descendants	$D(t,S) \stackrel{def}{=} \{u \in types \mid u \prec t\}$
Ancestors	$A(t,S) \stackrel{def}{=} \{u \in T \mid t \prec u\}$
Children	$C(t,S) \stackrel{def}{=} \{u \in T \mid u \prec t \wedge u \neq t \wedge (\nexists v \in T \mid v \neq u \wedge v \neq t \wedge u \prec v \prec t)\}$
Parents	$P(t,S) \stackrel{def}{=} \{u \in T \mid t \prec u \wedge u \neq t \wedge (\nexists v \in T \mid v \neq u \wedge v \neq t \wedge t \prec v \prec u)\}$
Root types	$R(S) \stackrel{def}{=} \{t_1, ..., t_n \in T \mid \{P(t_1,S), ..., P(t_n,S)\} = \{\emptyset, ..., \emptyset\}\}$
Level	$L(t,S) \stackrel{def}{=} \begin{cases} 0 & t \in Root(S) \\ max_{p \in P(t,S)}(L(p,S)) + 1 & \text{otherwise} \end{cases}$

Figure 2: Formal notations for $S = (T, \prec)$.

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

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, all the definitions presented in Figure 2 also apply to slices. For instance, the root types of a slice s_i of a sub-hierarchy S_i are the root types of S_i. $S_0 = (\{f, i, j, k\}, \prec)$ and $S_1 = (\{a, c, d\}, \prec)$ in Figure 1 are subtype hierarchies of S. A possible slicing of the type hierarchy $S = (T, \prec)$ is given by $s_0 = g; d; a; c; e; b; h$ 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 within the sequence s_i are consecutive in 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 extreme case, each straight slice contains one type only. The leading column of Table 1 contains the different types of the hierarchy, while the heading row contains the different slices of the straight slicing. A cell at position (i, j) contains the sequence of ancestors of the type leading row i for the slice at the head of column j.

As we will see later, the goal of DST 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. 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, namely c and f, are not consecutive in s_0.

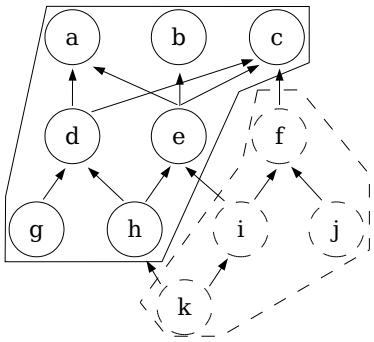

Figure 3: Two straight slices of the type hierarchy of Figure 1 (dashed and plain polygons respectively).

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

2.3 Encoding

We use the notion of straight slicing to encode types with a concise representation. 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 identifier $sid(t) \in \mathbb{N}$ of the straight slice to which t belongs.
3. The type identifier $id(t) \in \mathbb{Z}$ of t. This identifier corresponds to the position of t within its straight slice.
4. For each straight slice s_i, the interval $I_i(t)$ bounded by the smallest, respectively the largest, type identifier of the ancestors of type t in 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 $I_i(t)$ 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 3.5 that this property might be temporarily disabled when new types are added at runtime to the core type hierarchy $cth(S)$; we will however explain how we deal with this situation.

Table 3 presents the encoding of the slicing of Figure 2 for the type hierarchy of Figure 3. For simplicity, we do not mention $cth(S)$ (which is the same for all types). The leading column again contains the different types of the type hierarchy while the heading row depicts the straight slices of the straight slicing. The first (non-leading) column contains the identifier of the type together with its slice identifier. The other cells (i, j) contain the intervals of the identifiers of the ancestors of the type leading row i for the straight slice at the head of column j.

Using an adequate encoding of type and slice identifiers (described in Section 4.2), we may represent each type in the hierarchy very efficiently, with a small number of bits.

	$id(t),\ I_i(t)$	$s_0 = g; d; a; c; e; b; h$	$s_1 = k; i; f; j$
a	2,0	[2,2]	$\emptyset$
b	5,0	[5,5]	$\emptyset$
c	3,0	[3,3]	$\emptyset$
d	1,0	[1,3]	$\emptyset$
e	4,0	[2,5]	$\emptyset$
f	2,1	[3,3]	[2,2]
g	0,0	[0,3]	$\emptyset$
h	6,0	[1,6]	$\emptyset$
i	1,1	[2,5]	[1,2]
j	3,1	[3,3]	[2,3]
k	0,1	[1,6]	[0,2]

Table 3: The encoding of the hierarchy of Figure 1 for the straight slicing of Table 2.

2.4 Distributed Subtyping Test

We show now how our encoding can be used in a distributed fashion within a system of processes $\Pi = \{p_1, p_1, ...\}$ exchanging objects, where processes can leave (e.g., crash) and join the system (e.g., recover) at any time.

To enable distributed subtyping tests, it is not necessary for processes to send, together with each serialized object, the entire encoding of the type of the object. Only (1) the identifier $cth(S)$ (see Section 4) as well as (2) the set of intervals of the ancestors $I_i(t)$ of the type of the object (ordered according to their respective slice identifiers i) are required. The information contained in (1) and (2) together with the serialized form of the object fully qualifies the type of the object. 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 $\langle id(t), sid(t)\rangle$.

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:

$$u \prec t \Leftrightarrow (id(t) \in I_{sid(t)}(u) \wedge cth(S_u) = cth(S_t))$$

Consider the encoding of Table 3 (we omit $cth(S)$ as we consider only one hierarchy). If a process p_1 receives an object O_1^b with the encoding information $\{[5,5], \emptyset\}$, then p_1 can test if the object O_1^b is of type a by verifying whether $id(a) \in I_{sid(a)}(b)$, i.e., if $2 \in I_0(b) = [5,5]$. Visibly, O_1^b is not of (a subtype of) type a. On the other hand, if p_1 receives an object O_2^k with $\{[1,6],[0,2]\}$, p_1 can test if O_2^k is of a subtype of e by checking if $id(e) \in I_{sid(e)}(k)$ i.e., $4 \in I_0(k) = [1,6]$. This yields that O_2^k is indeed of type e.

3. THE DST ALGORITHM

This section describes how we obtain a straight slicing of a type hierarchy through our DST algorithm. The straight slicing is then used to generate the encoding of the types that are transferred along with objects sent over the wire.

3.1 Overview

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 type hierarchy and later discuss how to dynamically add new types.

Our algorithm is composed of three main phases (see Figure 4) (1) the *initialization*, (2) the incremental *straight slices creation* and (3) the *finalization*. The creation phase is in turn made of several steps, that are carried out for each type $t \notin R(S)$: (I) the *identification* of the *conflicting* straight slices of a type t, (II) the *addition* of t into each of its conflicting straight slices and (III) the *concatenation* of the conflicting straight slices in which t has been added.

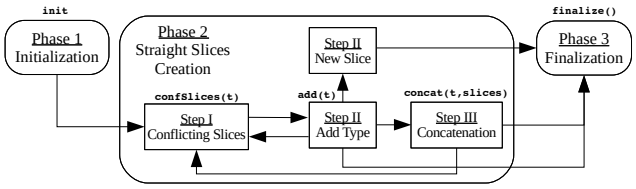

Figure 4: The different phases of DST.

3.2 Initialization – Phase 1

During the initialization phase (lines 1–4, Figure 1), each root type of the type hierarchy is added into a new distinct empty straight slice. For instance, if we consider the type hierarchy of Figure 1, the three root types, a, b and c, are put in three different initial straight slices: $s_0 = a, s_1 = b$ and $s_2 = c$.

3.3 Creation – Phase 2

This phase is performed for each type t of the type hierarchy S when executing ENCODE() (lines 21–30, Figure 1), starting with level 1 up to the highest level of S. This phase can be decomposed in three main steps we detail hereafter.

Identification. During this step, a type t is added to each of the straight slices of the set of *conflicting straight slices* (lines 5–13, Figure 1 and lines 16–23, Figure 2). This set consists in all the straight slices that contain at least one ancestor of t.

For instance, if we consider the type hierarchy of Figure 1, after initialization leading to the creation of straight slices $\{s_0, s_1, s_2\}$, CONFSLICES(d) is $\{s_0, s_2\}$.

Addition. Once the set of conflicting slices $confSlices$ of a type t has been computed, we add t into each straight slice s of $confSlices$ (lines 16–23, Figure 2). The addition of t into a straight slice s is possible only if all the ancestors in s of any type u in S remain consecutive. This condition implies that t can only be added at the head (resp. tail) of s (otherwise, t may break the consecutivity between a type u and its ancestors in s).

Adding t at the head (resp. tail) of s implies that the head (resp. tail) of s must be an ancestor of t (otherwise t is not consecutive with its ancestors in s as s contains at least one ancestor of t, see Section 3.3). However, the fact that the head or the tail of s is an ancestor of t does not imply that all the ancestors of t are consecutive in s. For instance,

```
1: INIT
2:   sslices ← ∅   {ordered set containing the straight slices}
3:   for all t ∈ R(S) do
4:     sslices ← sslices ∪ {t}                {one type per slice}

5: function CONFSLICES(t)   {The conflicting slices of a type}
6:   slices ← ∅                              {Set of conflicting slice}
7:   mcs ← ⊥        {Conflicting slice with non-root type(s)}
8:   for all s_i ∈ sslices do
9:     if s_i ⊆ R(S) ∧ A(t, s_i) ≠ ∅ then
10:      slices ← slices ∪ s_i
11:    else if A(t, s_i) ≠ ∅ ∧ id_{s_i} > id_{mcs} then
12:      msc ← s_i
13:  return slices ∪ {msc}

14: procedure FINALIZE()                       {finalization}
15:   for all s_i ∈ sslices by increasing i do
16:     if s_i ⊆ R(S) then
17:       s_j ← s_j ⊕ s_i | j = min(k : s_k ∈ sslices)   {k = 0 if
          none exists}
18:   for all s_i ∈ sslices and t ∈ s_i do
19:     sid(t) ← i
20:     I_i(t) ← [k, l] | k=first, l=last pos of super-types of t in s_i

21: procedure ENCODE()                         {main DST algorithm}
22:   for all level l of cth(S) in ascending order do
23:     for all t at level l do
24:       confSlices ←CONFSLICES(t)
25:       ADD(t,confSlices)
26:       if ∃s ∈ sslices | t ∈ s then        {addition succeeded}
27:         CONCAT(t, {s | s ∈ confSlices ∧ t ∈ s})
28:       else
29:         sslices ← sslices ∪ {t}
30:   FINALIZE()
```

Figure 1: Encoding a type hierarchy.

consider a straight slice $s_0 = b;d;e;c;a$ which corresponds to the output of the algorithm for the type hierarchy of Figure 5. Consider a type f which is a subtype of d and c. Even if the head and the tail of s_0 are ancestors of f, it is not possible to add f at the head/tail of s_0, because f will not be consecutive with all its ancestors c, a, b, d as e is in between d and c. To avoid this, the algorithm checks if all the parents of t are consecutive.

At the end of this phase, t has been added into either (a) one straight slice, (b) no straight slice or (c) several straight slices. In the case of (a) the algorithm proceeds with the identification of the conflicting straight slices of a new type u and if all the types of the hierarchy have gone through the addition step, proceeds to the finalization phase. In the case of (b), i.e., t was not added into any of the straight slices (because the slicing would not remain straight), a new straight slice is created for t. Finally, in the case of (c) we proceed to the concatenation of the straight slices in which t has been added. Note that concatenation will also take care of the temporary situation where the resulting straight slices become non-disjoint, if it occurs.

Consider the hierarchy of Figure 1 where the straight slicing up to type c corresponds to $\{s_0, s_1, s_2\} = \{a, b, c\}$ and CONFSLICES(d) returns $s_0 = a$ and $s_2 = c$. We can add d into s_0 and s_2, as both a and c are ancestors of d. The resulting straight slices are $s_0 = d;a$ and $s_2 = d;c$.

Concatenation. We consider now the conflicting straight slices of t in which t has been added and we try to concatenate them, one by one.

We only concatenate two straight slices s_i and s_j if the ancestors of any type u in S remain consecutive after the

```
1: function getConsAncestors(t,s)        {retrieve consec. ancestors of type in straight slice}
2:   ancestors ← ∅
3:   if head(s) = t then
4:     pos ← 0
5:     while pos < sizeof(s) ∧ u at position pos in s ∈ P(t,S) do
6:       ancestors ← ancestors ∪ {u}
7:       pos ← pos + 1
8:   else
9:     pos ←sizeof(s)-1
10:    while pos ≥ 0 ∧ u at position pos in s ∈ P(t,S) do
11:      ancestors ← ancestors ∪ {u}
12:      pos ← pos − 1
13:  return ancestors

14: predicate isStraightSlice(t,s)   {test if a slice is straight}
15:   return (t ∈ s ∧ P(t,s) = getConsAncestors(t,s))

16: procedure add(t,confSlices)      {add new type if possible}
17:   for all s ∈ confSlices do
18:     if head(s) ∈ A(t,S) then
19:       s ← t ⊕ s
20:     else if tail(s) ∈ A(t,S) then
21:       s ← s ⊕ t
22:     if not isStraightSlice(t, s) then
23:       s ← s\{t}
```

Figure 2: Adding a new type t into the set of its conflicting slices $confSlices$.

concatenation. Therefore, the straight slices s_i and s_j (see Figure 3) are only concatenated at their head/tail and we denote this head/tail by *ht*. If we consider the previous example where d was added into the straight slicing $\{s_0, s_1, s_2\}$, the straight slices that are concatenated are $s_0 = d; a$ and $s_2 = d; c$. In this case, we concatenate s_0 and s_2 at their respective heads as this does 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 t, part of the straight slicing (i.e., that have been added either at initialization or in the addition phase), we make sure that (1) *ht* is an ancestor of t and (2) that ancestors of t in s_i and s_j are consecutive with *ht*.

If conditions (1) and (2) above are fulfilled, the concatenation of two straight slices s_i, s_j at *ht* is performed as follows (in the case where s_i and 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, see lines 26–29 and lines 30-33 of Figure 3 resp.). 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 $\lfloor s_j \rceil$ with s_i (resp. s_i with $\lfloor s_j \rceil$, see lines 22–25 and lines 34–37 resp.), where $\lfloor s_j \rceil$ corresponds to the reversed sequence of s_j (e.g., $\lfloor a; b; c; d \rceil$ 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 $x; w; t; u; v$.

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 (see lines 15–21, Figure 3). 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 $t; u; v; x; w$.

If it is not possible to concatenate two straight slices s_i and s_j, t is removed from s_j and 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.

To illustrate the concatenation of two straight slices, consider the straight slices $s_0 = d; a$ and $s_2 = d; c$ which correspond to the output of the algorithm at the end of the addition of d into its set of conflicting straight slices (as presented above). The concatenation of s_0 with s_2 is possible and the new concatenated slice is $s = d; a; c$. It is then possible to add e in both s_0, s_2 as both c and 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 $s_1 = e; b$. It is furthermore possible to concatenate s_0 and s_1 as the concatenation allows the ancestors of a, b, c, d to 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: $s_1 = f$.

```
1: predicate concatenable(u, v, s_i, s_j, types)
2:   return (∀ k ∈ types : u ∈ A(k, s_i) ∧ v ∈ A(k, s_j))

3: procedure concat(t,slices)   {modifying a slice s in slices modifies also s in sslices}
4:   while sizeof(slices) ≠ 1 do
5:     s_i ← slices[0]
6:     s_j ← slices[1]
7:     types ← ∅
8:     for all s_k ∈ sslices do
9:       for all u ∈ s_k do
10:        if A(u,S ∈ s_i ∧ A(u,S ∈ s_j) then
11:          types ← types ∪ {u}
12:    types ← types ∪ {t}
13:    s_j ← s_j\{t}
14:    slices ← slices\{s_j}
15:    if (s_i\{t} ∪ s_j) ⊆ R(S) then
16:      tmp ← (first u ∈ s_j | C(u, S)\{t} ≠ ∅)
17:      if (head(s_i)=t then
18:        s_i ← s_i ⊕ s_j ⊕ tmp
19:      else
20:        s_i ← tmp ⊕ s_j ⊕ s_i
21:      sslices ← sslices\{s_j}
22:    else if (head(s_i) ∈ A(t,S) ∧ head(s_j) ∈ A(t,S)) then
23:      if concatenable(head(s_i),head(s_j),s_i, s_j,types) then
24:        s_i ← ⌊s_j⌉ ⊕ s_i
25:        sslices ← sslices\{s_j}
26:    else if (head(s_i) ∈ A(t,S) ∧ tail(s_j) ∈ A(t,S)) then
27:      if concatenable(head(s_i),tail(s_j),s_i, s_j,types) then
28:        s_i ← s_j ⊕ s_i
29:        sslices ← sslices\{s_j}
30:    else if (tail(s_i) ∈ A(t,S) ∧ head(s_j) ∈ A(t,S)) then
31:      if concatenable(tail(s_i),head(s_j),s_i, s_j,types) then
32:        s_i ← s_i ⊕ s_j
33:        sslices ← sslices\{s_j}
34:    else if (tail(s_i) ∈ A(t,S) ∧ tail(s_j) ∈ A(t,S)) then
35:      if concatenable(tail(s_i),tail(s_j),s_i, s_j,types) then
36:        s_i ← s_i ⊕ ⌊s_j⌉
37:        sslices ← sslices\{s_j}
```

Figure 3: Concatenating straight slices.

3.4 Finalization – Phase 3

We first check that there is no straight slice containing only root types, which might happen if the type hierarchy contains root types that do not have any descendants. If such straight slices $s_u, s_v, ...$ exist, they are appended at the tail of the first straight slice s_i that does not contain only root types. If there is no such s_i (i.e., the type hierarchy contains only root types), $s_v, ...$ 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 parent (lines 17–19, Figure 1).

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 $sid(t)$ and (3) for each slice s_i, its interval of ancestors $I_i(t)$ (lines 20–23, Figure 1). To assign an identifier to a type in a straight slice s_i, we 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 $I_i(t)$ 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 Table 4. 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 intervals of its ancestors are $[2, 2]$ and $[0, 0]$ in s_0 and s_1 respectively.

	$id()$, $sid(t)1$	$s_0 = d; a; c; e; b$	$s_1 = f$
a	1,0	[1,1]	$\emptyset$
b	4,0	[4,4]	$\emptyset$
c	2,0	[2,2]	$\emptyset$
d	0,0	[0,2]	$\emptyset$
e	3,0	[1,4]	$\emptyset$
f	0,1	[2,2]	[0,0]

Table 4: Encoding of the type hierarchy of Figure 1 up to type f.

3.5 Addition of New Types at Runtime

So far, a new type t is simply added at the head or tail of one of its conflicting slices s and only if s remains straight (lines 2–13, Figure 4). If it is not possible to add t into one of its conflicting slices, a new slice is created and t is added to it (lines 14–16, Figure 4). We do not concatenate anymore the slices together. 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 must retain the slice identifiers.

Once t is added into a slice s_i, then the slice identifier of t is id_{s_i} and the set of $I_j(t)$ is determined by the union of the intervals of the ancestors of t for all s_j in S. The type identifier of t is the highest type identifier of s_i plus one, respectively the lowest type identifier of s_i minus one, depending if t is added at the tail, respectively the head of s_i (lines 17–31, Figure 4).

To illustrate the idea, consider the type hierarchy of Figure 7. 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. The encoding of l is thus the following: $id(l)$ = -1, $sid(l) = 1$, $I_0(l) = I_0(g) \cup I_0(k) = [0,3] \cup [1,6] = [0,6]$, $I_1(l) = I_1(g) \cup I_1(k) \cup I_1(l) = \emptyset \cup [0, 2] \cup \{-1\} = [-1,2]$.

Note that in specific cases, the resulting $I_j(t)$ can be disjoint. This is especially true when the newly added type is a subtype of two types of a slice s_j whose ancestors are not consecutive in s_j. This might happen due to a previous concatenation. For instance, let us take the type hierarchy presented in Figure 6 (which corresponds to the type hierarchy presented in Figure 5 in which a new type f, subtype of type c and d is added). The output of our DST algorithm gives the following slice $s_0 = a; c; e; d; b$. We can clearly see if a new type f, subtype of c and d, is added to the type hierarchy of Figure 5, the ancestors of f in s_0 are not consecutive as e is not an ancestor of f. Consequently, $I_0(f) = I_0(c) \cup I_0(d) = [0,1] \cup [3,4]$.

This means that the number of range queries to perform a subtyping test depends, when a new type t is added at runtime, on the number of non-consecutive parents of t in the slice of these parents. However, the greater the number of parents of a new type t in a slice s_i, the more likely the union of the intervals of the ancestors of t in s_i will be non-disjoint.

```
1:  procedure addNewType(t)                                   {At runtime}
2:    confSlices ← confSlices(t)
3:    newSlice ← ⊥
4:    for all s ∈ confSlices do
5:      if head(s) ∈ A(t,S) then
6:        s ← t ⊕ s
7:      else if tail(s) ∈ A(t,S) then
8:        s ← s ⊕ t
9:      if not isStraightSlice(t, s) then
10:       s ← s\{t}
11:     else
12:       newSlice ← s
13:       break
14:   if newSlice = ⊥ then
15:     newSlice ← {t}
16:     sslices ← sslices ∪ newSlice
17:   for all s_i ∈ sslices do
18:     if t ∈ s_i then
19:       sid(t) ← i
20:     k ← ⊥
21:     l ← ⊥
22:     for all u ∈ s_i do
23:       if u ∈ A(t, S) ∧ k = ⊥ then
24:         k ← u
25:         l ← u
26:       else if u ∈ A(t, S) ∧ k ≠ ⊥ then
27:         l ← u
28:       else if u ∉ A(t, S) ∧ k ≠ ⊥ ∧ l ≠ ⊥ then
29:         I_i(t) ← I_i(t) ∪ [k, l]
30:         k ← ⊥
31:         l ← ⊥
```

Figure 4: Addition of a new type at runtime.

3.6 Limitations

The modifications of DST needed to add new types at runtime may affect both the encoding length (due to the local information of the processes) and the subtyping tests time (due to increased encoding length and the fact that

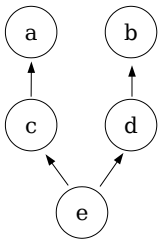

Figure 5: Concatenable subtype hierarchy.

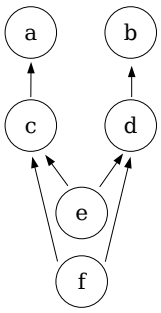

Figure 6: Ancestors of the new added type f are not consecutive in s_0.

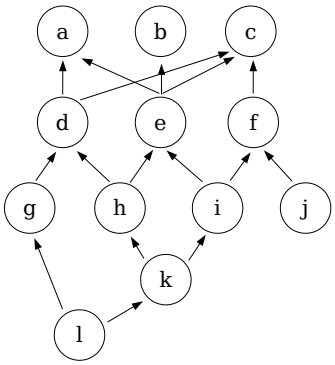

Figure 7: Addition of a type l to the type hierarchy of Figure 1.

several intervals of ancestors might be disjoint). This results from the incremental flavor of DST, and is more pronounced in other incremental algorithms (for instance in [17] adding new types at runtime implies creating new buckets as well as an exclusion list). Avoiding any degradation would require predicting the future.

To avoid cumulated effects of many types added at runtime, our DST algorithm could be used together with a strong-consistency protocol run in the background to update the encoding of the core type hierarchy with respect to the newly added types. In between such updates our DST algorithm can be used to ensure the liveness of the systems.

4. PUTTING DST TO WORK

This section outlines the implementation and use of our type encoding.

4.1 Implementation

Rather than extending/modifyinig a given Java virtual machine, we have implemented a set of 40 Java classes to encode type hierarchies according to DST and perform serialization and deserialization accordingly. The APIs are designed in a generic way. Programmers can easily define their own type encoding/subtyping tests by implementing prescribed interfaces. Our packages currently implements three flavors: besides the DST scheme presented previously, we provide implementations of the two naive approaches outlined in Section 1, i.e., the code transfer and string-based encoding approaches. These will be compared in terms of efficiency to our DST approach in Section 5. The complete source code and APIs can be downloaded at `http://lpd.epfl.ch/baehni/dst.tgz`.

4.2 Bit-Encoding of Integers

As explained in Section 2, the encoding of a type t includes integer values, namely: the intervals of the ancestors of 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 slice); the slice identifier; and the type identifier. We chose to encode identifiers 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.

An optimization has been implemented for encoding the intervals of the ancestors. It consists of encoding 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 19 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 slices and thus allows the algorithm to dynamically add new types to its 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 5, the optimal unit size was found to be $n = 1$. The results of Section 5 use such a unit size.

4.3 Illustration

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 representing the leaves of the hierarchy. Consider the case of a class `Sample`:

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

Once the type hierarchy is encoded, a process can serialize an object, e.g. to send it over a TCP connection, as follows:

```
Sample obj = new Sample();
DSSerializer ser = new DSSerializer(dscth);
OutputStream oStrm = sdrSk.getOutputStream();
ser.serialize(obj, oStrm);
```

Note that to this end the object must only implement the interface `java.io.Serializable`. This makes it possible to easily send proxies for Java RMI over the wire.

A receiver process at the other end of the connection may then obtain the object as a `DSObject` as follows:

```
DSSerializer ser = new DSSerializer(dscth);
InputStream iStrm = rcvSk.getInputStream();
DSObject dObj = ser.deSerializeObject(iStrm);
```

It is then possible, via the received instance of `DSObject`, to perform a subtyping test over the encapsulated object:

```
dscth.forName("Sample").isInstance(dObj);
```

The `isInstance()` method returns true or false depending on whether the object encapsulated in the `DSObject` instance is of the type whose name is passed to `forName()`.

5. PERFORMANCE

To illustrate how DST retains efficiency of centralized approaches this section compares our Java implementation of DST (1) in a *local* setting with CPQE (an optimized version of the PQE algorithm – the most efficient centralized approach we know of) and (2) in a *distributed* setting with the code download approach (a common way to perform subtyping tests in current distributed systems; henceforth termed CD), and the string-based encoding (SE, a simple way to perform subtyping tests in a way avoiding global reconfiguration).

5.1 Local Setting

In the local setting, we measured both the overhead of type encoding and the efficiency of (de)serialization of a received object according to specific cases.

Configuration. All measurements in this setting were obtained using an *Intel Pentium 4* 2.66GHz 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.

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

Table 5 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 SE 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 that type. The results from [23] do not allow us to make this discrimination in the case of CPQE. The encoding length of DST outperforms SE by a factor of more than 35 for the information sent with the objects, and 2.5 to 7.8 for the information maintained by processes that need to perform subtyping tests. 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).

Java version	DST (1)	DST (2)	SE (1)	SE (2)	CPQE (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 5: Average number of bits for the encoding.

Java version	DST Sender	DST Receiver	SE Sender	SE 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 6: Maximum number of bits to encode type information, as held by receiver processes and sent by sender processes.

Efficiency of (de)serialization. We also benchmarked our implementation when serializing/deserializing an object of a `Sample` event class. Each instance of that class contained an instance of class `ServerManagerImpl` in package `com.sun.corba.se.impl.activation`. The resulting serialized byte array contains the (previously encoded) type information of the object (in the case of DST and SE), the byte-code of the class of the object, as well as the object itself. The time taken to serialize an object is presented in Figure 8 for DST, SE, and CD. Clearly DST and SE take a similar amount of time for serializing an object but do not perform as well as CD in this case by $82.76\mu s$. This overhead is due to the time to serialize the type encoding.

The object is then sent by a process p_i over a local TCP socket to a process p_j, which, upon a positive subtyping test with the object's type completes the deserialization. During the deserialization, four distinct situations may arise according to the possible permutations of two conditions, namely (1) the types received by p_j and (2) the availability of the byte-code of the object. We depict all the cases in Figure 8. As expected, DST and SE are much better when the subtyping test fails. DST outperforms CD 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 observe that both DST and SE are comparable. This result can be explained by the fact that the algorithm used to (de)serialize the type information of the object in a size-efficient way, in DST, is quite complex. If the (de)serialization time is more important than the length of the encoding, it is possible to use plain bytes for the encoding instead of bits. In this case, the deserialization time of the encoded type information is 4 times shorter with our DST algorithm than with SE. Moreover, even if the encoding of our algorithm uses bytes instead of bits, it still outperforms the SE approach by a factor of 5 in terms of encoding length. Regarding the time taken to actually perform the subtyping test, Figure 8 depicts the fact that this delay is negligible (around $10\mu s$ with every approach) with respect to the time taken for the deserialization.

Table 6 presents the worst encoding length that is achieved

for individual Java versions. This corresponds to the Java 1.5 type `java.awt.dnd.DnDEventMulticaster`. This type hierarchy has 18 straight slices. In this case, our algorithm still performs between 3 and 6 times better than SE.

5.2 Distributed Setting

We measured and compared the average number of incoming objects a process p_i can handle in the case it is using (1) DST, (2) SE, or (3) CD. We focus here on a typed publish/-subscribe scenario, in which objects representing *events* are passed by value between processes. Publish/subscribe applications are a particularly interesting study case, due to their wide adoption in large scale applications, and because they require high throughput and thus push existing technologies to their limits [21].

Configuration. The measurements conducted in the distributed setting were obtained using 5 *Intel Pentium 4* 3.0GHz with 1 GB, running Java virtual machine version 1.5.0_06-b05 on *Red Hat Enterprise Linux WS* (kernel 2.4.21-27.EL) operating systems on a 100Mbit/s LAN. All the presented values are averaged over 30 measurements.

In these experiments one process acting as "event broker" received events containing instances of either `Sample` (the same type used in the previous tests) or `NewSample` (`NewSample` has all the ancestor types of `Sample`, without being a subtyping of `Sample`) serialized either with the DST serializer, the SE serializer or only the native Java serializer (CD approach). We assumed that the code of the `Sample` and `NewSample` classes were available locally. In this setting the server considered the events containing `Sample` objects as "relevant" (i.e., of interest for the process), whereas events containing `NewSample` objects were "parasite" events (i.e., only being forwarded on behalf of other processes) [**?**]. Each of the remaining 4 computers sent 200'000 events with objects of 192 bytes each to the broker.

Throughput. We now analyze the number of events a process can handle with respect to the approach we use. When receiving an event the following piece of code is performed at the server side (`in` is an input stream from a socket, and `gcth` is an instance of class `GenericCTH` representing $cth(S)$):

```
GenericObject go = ser.deSerializeObject(in);
GenericClass gc =
   (GenericClass)gcth.forName("Sample");
if (gc.isInstance(go)) {
   Object obj = go.getObject();
}
```

First, the broker deserialized an event by means of the method `deSerializeObject()`. In the DST and SE approaches this means deserializing only the encoded type information whereas in the CD approach this means deserializing the object contained in the event. Then a subtyping test is performed against the type of the conveyed object (i.e., `isInstance()`) to see if it is of interest. Finally, if the object is of interest, it is retrieved from the `GenericObject` (through `getObject()`). In the DST and SE approaches this means deserializing the object (with the native Java APIs) whereas with CD this implies just returning the previously deserialized object.

Figure 9 presents the average number of events per second a broker can process according to the percentage of relevant events received and the serialization approach (i.e., DST, SE, or CD). Not surprisingly, processing an object of interest or a parasite one does not make any difference with respect to the CD approach – the conveyed object is deserialized in any case. As expected the DST approach is always better than the SE. This can be explained by the fact that when receiving many events, the time taken to deserialize the type encoding information of the SE approach takes much more time than in a local setting.

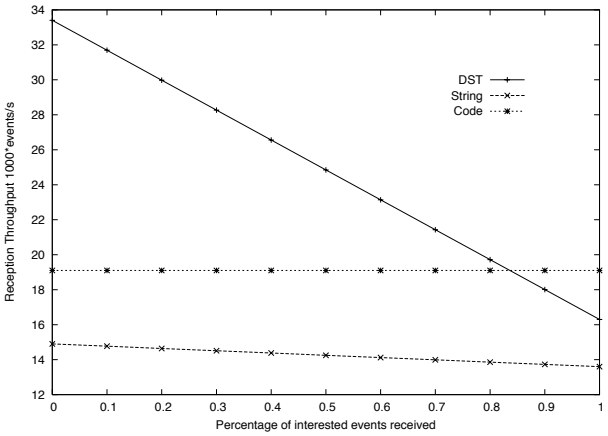

Figure 9: Average number of relevant and parasite events processed for the different approaches.

What is interesting to notice is that the overhead induced by our DST approach in the case of an object of interest (i.e., deserializing the type encoding information) becomes a problem with respect to the CD approach only when a process receives more than 83% events of interest. Indeed in the case where the process receives only parasite events, our DST approach can process 75% more events than the CD approach. However, if the process is interested in all the objects it receives, our DST approach processes 17% less events than the CD approach. Finally, note that we assume in the measurements that the code of the actual objects was available locally. If this was not the case, our approach would perform even better than CD in the processing of parasite events (as no download of code is necessary), without of course mentioning cases in which the CD approach stalls if types with complex dependencies are downloaded.

6. RELATED WORK

Several authors have proposed efficient schemes for encoding type hierarchies [12, 22, 1, 23, 24, 17], but, as we discuss below, these approaches typically require global reconfigurations of the encoding when new types are dynamically added to the system which is usually not an issue in a centralized setting.

With *bit vector encoding* [12], a type hierarchy (e.g. Figure 1) is embedded in a lattice of subsets of $\{1, ..., k\}$. The encoding of a type t is a vector of k bits (vec_t). A type t is a subtype of a type r if $vec_t \wedge vec_r = vec_r$. With *range compression encoding* [1], the type hierarchy is split into single inheritance trees and types are enumerated using a post-order traversal algorithm. The encoding of a type t consists of (1) its identifier as well as (2) a set of intervals. The

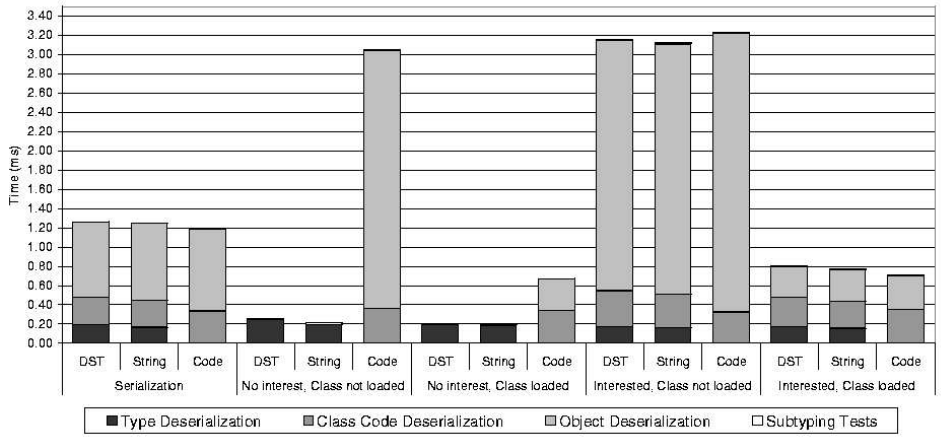

Figure 8: Serialization and deserialization time of a `Sample` object.

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 t is contained in the intervals of a type r then t is a subtype of r. With these approaches [12, 1], whenever a new type is added at run-time, the entire type hierarchy encoding is re-computed.

With *packed encoding* (PE) [22], a hierarchy of N types is represented by a matrix with N lines and P (determined by the algorithm) columns called *buckets*. The encoding of a 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 $\langle index, value\rangle$ pair corresponds respectively to the identifier of a bucket i and the identifier of the super-type of t in i (in order to support multiple subtyping, two super-types can not 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 upon addition of new types. 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* (PQE) [23], named after *PQ-trees*, 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 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 such a way that their identifiers represent a contiguous interval. If, due to the addition of a new type v into s_i, the identifiers of the subtypes of any type t do not remain consecutive in s_i, a new slice s_j is created for v. Consequently, the encoding of a type t in PQE consists of: (1) the slice identifier of t, (2) the type identifier of t in its slice, and (3) for each slice s_i, an interval whose first and last values correspond to the smallest, respectively highest, identifiers of the subtypes of t contained in s_i. Testing if a type t is a subtype of a type r consists in checking if the identifier of t is part of the interval of the subtypes of r in the slice of t. PQE provides the best encoding length out of all centralized algorithms we know about, yet does not support the addition of new types at run-time without re-encoding the entire hierarchy.[3]

R&B encoding [17] uses *ranges* and *slices* for encoding a type hierarchy in constant time and in an incremental way. The algorithm uses the range numbering technique of Schubert [18] and supports the addition of new subtypes at runtime, assuming the algorithm of [22]. No global reconfiguration is thus needed, but the size of the encoding grows with the number of common ancestors in the type hierarchy.

Perfect hashing [20, 8] can be used to map each identifier of an ancestor of a type (i.e., a value in the hash table) to a unique identifier in a hash table [6] (i.e., a key in the hash table). The subtyping test simply consists in checking if the value mapped to the hash of the identifier of the ancestor in the hash table against which a subtyping test is performed corresponds to the identifier of the ancestor. New types can be added at runtime as each type has its own hash table containing $\langle key, value\rangle$ pairs for each of its ancestors. This implies that the size of the encoding grows with the number of the ancestors of a type in the considered type hierarchy.

Acute [19, 5] adds mechanisms for distributed programming to OCamL, including support for distributed subtype testing. The "encoding" of types in Acute is based on hash functions applied to (abstract) type descriptions. The goal is to retain a fine grained type information to reflect invariants of types, and not to associate unique, minimal, identifiers with types. Unlike the above-mentioned approaches and the one described in this paper, the encoding does not focus on nominal subtyping relationships, but supports structural conformance, in order to deal also with versioning. The hash functions as well as the resulting encoding and its efficiency are not detailed.

7. SUMMARY

We devised an efficient subtyping test algorithm, DST, to boost the performance of dynamic and distributed typed publish/subscribe infrastructures. Our algorithm (1) retains the efficiency of local subtype testing but (2) avoids global reconfiguration upon addition of new types. The algorithm

[3]The authors present in [24] a variant of the algorithm (BTS) that overcomes this difficulty in certain situations. In many cases however, the algorithm still has to re-encode parts of the type hierarchy.

encodes a multiple subtyping hierarchy in a memory-efficient manner and can perform subtyping tests against each type of the hierarchy without downloading its code nor having to deserialize objects of that type.

As we pointed out, our DST algorithm can be viewed as a combination of [23] and [22] with a fundamental difference: we order the *ancestors* of a type instead of its *descendants*. This is key to avoiding global reconfiguration while using very little memory for the encoding. We ensure that the encoding of the core type hierarchy (the set of types present at the bootstrapping of the system) remains the same throughout the entire lifetime of the system. Unlike the approach outlined in [5], we rely on the Java convention that types are unique (by their name), and do not deal with versioning. Possibilities for extending and updating existing types or classes are given by the ability to introduce new subtypes or subclasses at runtime.

As we show through our experiments, our DST algorithm is comparable, in terms of performance, to the best known centralized subtyping algorithm, namely PQE [23], which requires however re-computing the encoding if new types are dynamically added. In particular, we show that when deployed in a distributed publish/subscribe system, our DST algorithm enables significant performance improvements by allowing up to 30% more events to be handled than with current approaches.

Acknowledgements

The authors would like to thank Jan Vitek for invaluable feedback.

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 International Conference on Management of Data*, pages 253–262, 1989.

[2] S. Ajmani, B. Liskov, and L. Shrira. Modular Software Upgrades for Distributed Systems. In *Proceedings of the 20th European Conference on Object-Oriented Programming*, pages 452–476, 2006.

[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] J. Dedecker, T. V. Cutsem, S. Mostinckx, T. D'Hondt, and W. D. Meuter. Ambient-Oriented Programming in AmbientTalk. In *Proceedings of the 20th European Conference on Object-Oriented Programming*, pages 230–254, 2006.

[5] P.-M. Denielou and J. Leifer. Abstraction Preservation and Subtyping in Distributed Languages. In *Proceedings of the 11th ACM International Conference on Functional Programming (ICFP '06)*, pages 286–297, 2006.

[6] R. Ducournau. Le hachage parfait fait-il un parfait test de sous-typage? In *Proceedings of the 2006 Conférence des Languages et Modèles à Objets*, pages 71–86, 2006.

[7] L. Fiege, M. Mezini, G. Mühl, and P. Buchmann. Engineering Event-Based Systems with Scopes. In *Proceedings of the 16th European Conference on Object-Oriented Programming (ECOOP 2002)*, pages 309–333, 2002.

[8] M. L. Fredman, J. Komlós, and E. Szemerédi. Storing a Sparse Table with $O(1)$ Worst Case Access Time. *Journal of the ACM*, 31(3):58–544, 1984.

[9] A. Goldberg and A. Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.

[10] J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java 1.5 Language Specification*, third edition. `http://java.sun.com/j2se/1.5.0/docs/index.html`, 2005.

[11] A. Hejlsberg and S. Wiltamuth. *C# Language Specification*. Microsoft Press, 2001.

[12] 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, 1997.

[13] Microsoft. *Microsoft Message Queuing*, 2005.

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

[15] Sun Microsystems Inc. *Java Message Service - Specification, version 1.1*. `http://java.sun.com/products/jms/docs.html`, 2005.

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

[17] K. Palacz and J. Vitek. Java Subtype Tests in Real-Time. In *Proceedings of the 17th European Conference on Object-Oriented Programming*, pages 378–404, 2003.

[18] L. K. Schubert, M. A. Papalaskaris, and J. Taugher. Determining Type, Part, Colour, and Time Relationships. *IEEE Computer*, 16:53–60, 1983.

[19] P. Sewell, J. Leifer, K. Wansbrough, F. Z. Nardelli, M. Allen-Williams, P. Habouzit, and V. Vafeiadis. Acute: High-level Programming Language Design for Distributed Computation. In *Proceedings of the 10th ACM International Conference on Functional Programming*, pages 15–26, 2005.

[20] R. Sprugnoli. Perfect Hashing Functions: A Single Probe Retrieving Method for Static Sets. *Communications of the ACM*, 20(11):841–850, 1977.

[21] R. Strom and J. Auerbach. The Optimistic Readers Transformation. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 275–301, 2001.

[22] J. Vitek, R. N. 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.

[23] 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*, pages 819–856, 2001.

[24] 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, 2002.