Static Inference of Transmission Data Sizes in Distributed Systems

Elvira Albert[1], Jesús Correas[1], Enrique Martin-Martin[1], Guillermo Román-Díez[2]

[1] DSIC, Complutense University of Madrid, Spain
[2] DLSIIS, Technical University of Madrid, Spain

Abstract. We present a static analysis to infer the amount of data that a distributed system may transmit. The different locations of a distributed system communicate and coordinate their actions by posting tasks among them. A task is posted by building a message with the task name and the data on which such task has to be executed. When the task completes, the result can be retrieved by means of another message from which the result of the computation can be obtained. Thus, the transmission data size of a distributed system mainly depends on the amount of messages posted among the locations of the system, and the sizes of the data transferred in the messages. Our static analysis has two main parts: (1) we over-approximate the sizes of the data at the program points where tasks are spawned and where the results are received, and (2) we over-approximate the total number of messages. Knowledge of the transmission data sizes is essential, among other things, to predict the bandwidth required to achieve a certain response time, or conversely, to estimate the response time for a given bandwidth. A prototype implementation in the SACO system demonstrates the accuracy and feasibility of the proposed analysis.

1 Introduction

Distributed systems are increasingly used in industrial processes and products, such as manufacturing plants, aircraft and vehicles. For example, many control systems are decentralized using a distributed architecture with different processing locations interconnected through buses or networks. The software in these systems typically consists of concurrent tasks which are statically allocated to specific locations for processing, and which exchange messages with other tasks at the same or at other locations to perform a collaborative work. A decentralized approach is often superior to traditional centralized control systems in performance, capability and robustness. Systems such as control systems are often critical: they have strict requirements with respect to timing, performance, and stability. A failure to meet these requirements may have catastrophic consequences. To verify that a given system is able to provide the required quality of control, an essential aspect is to accurately predict the communication traffic among its distributed components, i.e., the amount of data to be transmitted along any execution of the distributed system.

In order to estimate the transmission data sizes, we need to keep track of the amount of data transmitted in two ways: (1) by posting asynchronous tasks among the locations, this requires building a message in which the name of the task to execute and the data on which it executes are included; (2) by retrieving the results of executing the tasks, in our setting, we use future variables [8] to synchronize with the completion of a task and retrieve the result. This paper presents a static analysis to infer a safe over-approximation of the transmission data sizes required by both sources of communications in a distributed system. Our method infers three different pieces of information:

1. *Inference of distributed locations.* As locations can be dynamically created, in a first step, we need to find out the locations that compose the system and give them abstract names which will allow us to track communications among them during the analysis. This is formalized by means of points-to analysis [14,13], a typical analysis in pointer-based languages which infers the memory locations that a reference variable can point to. In our case, locations are referenced from reference variables, thus the use of points-to analysis.
2. *Inference of number of tasks spawned.* The second step is to infer an upper bound on the number of tasks spawned between each pair of distributed locations. This is a problem which can be solved by a generic cost analysis framework such as [3]. In particular, we need to use a *symbolic* cost model which allows us to annotate the caller and callee locations when a task is spawned in the program. In essence, if we find an instruction a!m(x) which spawns a task m at location a, the cost model symbolically counts $c(this, a, m) * 1$, i.e., it counts that 1 task executing m is spawned from the current location this at a. If the task is spawned within a loop that performs n iterations, the analysis will infer $c(this, a, m) * n$.
3. *Inference of data sizes.* Finally, we need to infer the sizes of the arguments in the task invocations. Typically, size analysis [7] infers upper bounds on the data sizes at the end of the program execution. Here, we are interested in inferring the sizes at the points in which tasks are spawned. In particular, given an instruction a!m(x), we aim at over-approximating the size of x when the program reaches the above instruction. If the above instruction can be executed several times, we aim at inferring the largest size of x, denoted $\alpha(x)$, in all executions of the instructions. Altogether, $c(this, a, m) * \alpha(x)$ is a safe over-approximation of the data size transmission due to such instruction. The analysis will infer such information for each pair of locations in the system that communicate, annotating also the task that was spawned.

We demonstrate the accuracy and feasibility of the presented cost analysis by implementing a prototype analyzer within the SACO system [2], a static analyzer for distributed concurrent programs. Preliminary experiments on some typical applications for distributed programs show the feasibility and accuracy of our analysis. The tool can be used on-line from a web interface available at `http://costa.ls.fi.upm.es/web/saco`.

The remaining of the paper is organized as follows. The next section will present the distribution model that we use to formalize the analysis. Sec. 3 defines the concrete notion of transmission data size that we then want to over-approximate by means of static analysis. Sec. 4 presents the static analysis that carries out the three steps mentioned above. Sec. 5 reports on preliminary experimental results and Sec. 6 concludes.

2 Distribution Model

We consider a distributed programming model with explicit locations and based on the actor-based paradigm [1]. Each location represents a processor with a procedure stack and an unordered queue of pending tasks. Initially all processors are idle. When an idle processor's task queue is not empty, some task is selected for execution. Besides accessing its own processor's global storage, each task can post tasks to the queues of any processor (*message passing*), including its own, and synchronize with the completion of tasks. This synchronization is done by means of *future variables* [8]. When a task completes or when it is awaiting for another task to terminate, its processor becomes idle again, chooses the next pending task, and so on. This distribution model captures the essence of the concurrency model of languages like X10 [12], Erlang [6], Scala [10] or ABS [11].

2.1 Syntax

Regarding data, the language contains basic types B (int, bool ...) and parametric data types D. Data types are declared by listing all the possible constructors C and their arguments, a syntax similar to functional languages like *Haskell*:

(Type variable)	N	$::= a, b, c \ldots$
(Basic type)	B	$::= \mathsf{int} \mid \mathsf{bool} \mid \mathsf{void} \mid \ldots$
(Data type declaration)	Dd	$::= \mathsf{data}\ D(N_1, \ldots, N_n) = C_1 \mid \ldots \mid C_k \quad (n \geq 0, k > 0)$
(Constructor)	C	$::= Co(N_1, \ldots, N_n) \quad (n \geq 0)$
(Ground type)	T	$::= B \mid D(T_1, \ldots, T_n) \quad (n \geq 0)$

Example 1 (Data types). We define integer lists and general binary trees as:

$\mathsf{data\ List} = \mathsf{Nil} \mid \mathsf{Cons}(\mathsf{int}, \mathsf{List})$

$\mathsf{data\ Tree}(a) = \mathsf{Leaf}(a) \mid \mathsf{Branch}(a, \mathsf{Tree}(a), \mathsf{Tree}(a))$

Using the previously declared constructors the list $l = [1, 2, 3]$ is defined as $\mathsf{l} = \mathsf{Cons(1,\ Cons(2,\ Cons(3,Nil)))}$, and the binary tree t with 2 at the root, 1 as left child and 3 as right child as $\mathsf{t} = \mathsf{Branch(2,\ Leaf(1),\ Leaf(3))}$

Apart from data type declarations, the language allows the definition of functions based on pattern matching as in functional languages—e.g. head, tail, length, etc. This syntax has been omitted for the sake of conciseness, as it does not play an important role for presenting the analysis.

Regarding programs, the number of distributed locations needs not be known a priori (e.g., locations may be virtual). Syntactically, a location will therefore

```
1  main (List l, int s) {
2    x = newLoc;
3    y = newLoc;
4    z = newLoc;
5    x!extend(l,s);
6  }
7
8  int foo (int i) {
9    return i;
10 }
11 void extend (List l,int s) {
12   while(s > 0) {
13     Fut f= y!add(l,5);
14     await f?;
15     l = f!get;
16     z!process(l);
17     s = s − 1;
18   }
19 }
20 List add (List l, int e) {
21   List r = Cons(e,l);
22   return r;
23 }
24 void process (List le) {
25   while(le != Nil) {
26     Int h = head(le)
27     y!foo(h);
28     le = tail (le);
29   }
30 }
```

Fig. 1. Running Example

be similar to an *object* and can be dynamically created using the instruction newLoc. The program is composed by a set of methods finished with a return instruction $M ::= T\ m(\bar{T}\ \bar{x})\{s; \mathsf{return}\ x;\}$ where s takes the form:

$$s ::= s; s \mid x = e \mid x = f.\mathsf{get} \mid \mathsf{if}\ e\ \mathsf{then}\ s\ \mathsf{else}\ s \mid \mathsf{while}\ e\ \mathsf{do}\ s \mid b = \mathsf{newLoc} \mid f = b!m(\bar{x}) \mid \mathsf{await}\ f?$$

The notation $\bar{T}$ is used as a shorthand for $T_1, \ldots, T_n$, and similarly for other names. The special location identifier *this* denotes the current location. For the sake of generality, the syntax of expressions e is left open. The semantics of future variables f and concurrency instructions is explained below.

Example 2 (running example). Fig. 1 shows a method main which creates three distributed locations, x, y and z, and receives a list of integers, l, and one integer, s. In the example, we assume that x, y and z are global variables and thus accessible to all methods. Also, we have omitted return instructions in void tasks. Method main spawns task extend at location x in Line 5 (L5 for short) and sends data l and x (thus there is data transmission at this point). Method extend extends l with s new elements. To do this, it invokes method add at location y that extends the list with a new element (L13). The await instruction at L14 awaits for the termination of add. The result is retrieved using the get instruction at L15, where besides we assign the result to l. Within the loop of extend, tasks executing process are spawned at location z. The execution of process traverses the list in the while loop and invokes foo for each element in l. An important point to note is that, besides the data transmitted when asynchronous tasks are spawned, the instruction get also involves data transmission to retrieve the results.

2.2 Semantics

A *program state* has the form $loc_1 \| \ldots \| loc_n$, denoting the currently existing distributed locations. Each *location* is a term $loc(lid, tid, \mathcal{Q})$ where *lid* is the location identifier, *tid* is the identifier of the *active task* which holds the location's

$$\frac{t = tsk(tid, m, l, \langle x = \mathsf{newLoc}; s\rangle),\ \mathrm{fresh}(lid_1)\ ,\ l' = l[x \to lid_1]}{loc(lid, tid, \{t\} \cup \mathcal{Q}) \rightsquigarrow loc(lid, tid, \{tsk(tid, m, l', s)\} \cup \mathcal{Q}) \parallel loc(lid_1, \bot, \{\})} \quad (\textsc{newLoc})$$

$$\frac{l(x) = lid_1,\ \mathrm{fresh}(tid_1),\ l_1{=}buildLocals(\bar{z}, m_1),\ l' = l[f \to \langle tid_1, \bot, \bot\rangle]}{loc(lid, tid, \{tsk(tid, m, l, \langle f{=}x!m_1(\bar{z}); s\rangle)\} \cup \mathcal{Q}) \parallel loc(lid_1, \_, \mathcal{Q}') \rightsquigarrow loc(lid, tid, \{tsk(tid, m, l', s)\} \cup \mathcal{Q}) \parallel loc(lid_1, \_, \{tsk(tid_1, m_1, l_1, body(m_1)) \cup \mathcal{Q}'\})} \quad (\textsc{async})$$

$$\frac{l(x) = v, l_1(f) = \langle tid, \bot, \bot\rangle, l_1' = l_1[f \to \langle tid, true, \bot\rangle]}{loc(lid, tid, \{tsk(tid, m, l, \langle \mathsf{return}\ x\rangle)\} \cup \mathcal{Q}) \parallel loc(lid_1, \_, \{tsk(tid_1, \_, l_1, \_)\} \cup \mathcal{Q}_1) \rightsquigarrow loc(lid, \bot, \{tsk(tid, m, l, \epsilon(v))\} \cup \mathcal{Q}) \parallel loc(lid_1, \_, \{tsk(tid_1, \_, l_1', \_)\} \cup \mathcal{Q}_1)} \quad (\textsc{return})$$

$$\frac{t = tsk(tid, m, l, \langle \mathsf{await}\ f?; s\rangle), l(f) = \langle tid_1, true, \_\rangle}{loc(lid, tid, \{t\} \cup \mathcal{Q}) \rightsquigarrow loc(lid, tid, \{tsk(tid, m, l, s)\} \cup \mathcal{Q})} \quad (\textsc{await-t})$$

$$\frac{t = tsk(tid, m, l, \langle \mathsf{await}\ f?; s\rangle), l(f) = \langle tid_1, \bot, \bot\rangle}{loc(lid, tid, \{t\} \cup \mathcal{Q}) \rightsquigarrow loc(lid, \bot, \{tsk(tid, m, l, \langle \mathsf{await}\ f?; s\rangle)\} \cup \mathcal{Q})} \quad (\textsc{await-f})$$

$$\frac{l(f) = \langle tid_1, true, \bot\rangle, l' = l[x \to v, f \to \langle tid_1, true, v\rangle]}{loc(lid, tid, \{tsk(tid, m, l, \langle x = f.\mathsf{get}; s\rangle)\} \cup \mathcal{Q}) \parallel loc(lid_1, \_, \{tsk(tid_1, \_, l_1, \epsilon(v))\} \cup \mathcal{Q}_1) \rightsquigarrow loc(lid, tid, \{tsk(tid, m, l', s)\} \cup \mathcal{Q}) \parallel loc(lid_1, \_, \{tsk(tid_1, \_, l_1, \epsilon(v))\} \cup \mathcal{Q}_1)} \quad (\textsc{get-r})$$

$$\frac{l(f) = \langle tid_1, true, v\rangle, v \neq \bot, l' = l[x \to v]}{loc(lid, tid, \{tsk(tid, m, l, \langle x = f.\mathsf{get}; s\rangle)\} \cup \mathcal{Q}) \rightsquigarrow loc(lid, tid, \{tsk(tid, m, l', s)\} \cup \mathcal{Q})} \quad (\textsc{get-l})$$

$$\frac{select(\mathcal{Q}) = tid,\ t = tsk(tid, \_, \_, s) {\in} \mathcal{Q}, s \neq \epsilon(v)}{loc(lid, \bot, \mathcal{Q}) {\rightsquigarrow} loc(lid, tid, \mathcal{Q})} \quad (\textsc{select})$$

Fig. 2. (Summarized) Semantics for Distributed Execution

lock or $\bot$ if the lock is free, and $\mathcal{Q}$ is the set of tasks at the location. Only one task, which holds the location's *lock*, can be *active* (running) at this location. All other tasks are *pending*, waiting to be executed, or *finished*, if they terminated and released the lock. A *task* is a term $tsk(tid, m, l, s)$ where *tid* is a unique task identifier, m is the name of the method executing in the task, l is a mapping from local variables to their values and s is the sequence of instructions to be executed or $s = \epsilon(v)$ if the task has terminated with value v.

The execution of a program starts from a method m in an initial state S_0 with a single (initial) location with identifier 0 executing task 0 of the form $S_0{=}loc(0, 0, \{tsk(0, \mathsf{m}, l, body(\mathsf{m}))\})$. Here, l maps parameters to their initial val-

ues and local references to null (standard initialization), and *body*(m) refers to the sequence of instructions in the method m. The execution proceeds from the initial state S_0 by selecting *non-deterministically* one of the locations and applying the semantic rules depicted in Fig. 2. The treatment of sequential instructions is standard and thus omitted. The operational semantics $\rightsquigarrow$ is given in a rewriting-based style where at each step a subset of the state is rewritten according to the rules as follows. In NEWLOC, the active task *tid* at location *lid* creates a location lid_1 which is introduced to the state with a free lock. ASYNC spawns a new task (the initial state is created by *buildLocals*) with a fresh task identifier tid_1 which is added to the queue of location lid_1—the case $lid{=}lid_1$ is analogous, the new task tid_1 is simply added to the queue $\mathcal{Q}$ of *lid*. The future variable f allows synchronizing the execution of the current task with the completion of the created task, and retrieving its result. The association of the future variable to the task is stored in the local variables table $l'(f){=}\langle tid_1, \bot, \bot\rangle$: the future variable f is linked to task tid_1, the task has not terminated yet (first $\bot$ in the tuple), and the result of the invocation is not available yet (second $\bot$). The rule RETURN is used when a task *tid* executes a return instruction. The terminating task *tid* finishes the execution with value v (its sequence of instructions is set to $\epsilon(v)$) and the calling task tid_1 is notified that *tid* has terminated by setting to *true* the termination flag of the corresponding future variable—the case $lid{=}lid_1$ is analogous, but storing the termination flag in a task in queue $\mathcal{Q}$. In AWAIT-T, the future variable we are awaiting for points to a finished task (it has the termination flag set to *true* in the future variable f stored in the local variable table l) and await can be completed. Otherwise, AWAIT-F yields the lock so that any other task of the same location can take it. The rule GET-R retrieves the returning value from the task tid_1 linked to the future variable f, if the corresponding task has terminated and the value has not been retrieved before. If tid_1 has not terminated, it will wait for the value without yielding the lock. If the returning value has been retrieved from the remote object already, it is copied locally from the future variable f by means of GET-L. Finally, in rule SELECT an idle location takes a non-finished task to continue the execution—the function *select*($\mathcal{Q}$) non-deterministically returns a task identifier occurring in $\mathcal{Q}$.

Example 3 (semantics). The following sequence is the beginning of a trace of the program in Fig. 1 starting from main(Cons(1,Cons(2,Nil)),7). For the sake of conciseness we represent lists with square brackets—[1,2]—instead of constructors and we use l_e, l_a and l_p to denote initial local mappings, stressing only the important changes to them at each step.

$S_0 \equiv loc(0,0,\{tsk(0,\mathsf{main},l_m,\langle \mathsf{x} = \mathsf{newLoc};\ldots\rangle)\}) \rightsquigarrow^{\text{NEWLOC}\times 3}$
$S_3 \equiv loc(0,0,\{tsk(0,\mathsf{main},l_m[\mathsf{x}\mapsto 1,\mathsf{y}\mapsto 2,\mathsf{z}\mapsto 3],\langle \mathsf{x!extend(l,s)}\rangle)\}) \parallel loc(1,\bot,\{\})$
$\parallel loc(2,\bot,\{\}) \parallel loc(3,\bot,\{\}) \rightsquigarrow^{\text{ASYNC}}$
$S_4 \equiv loc(0,0,\ldots) \parallel loc(1,\bot,\{tsk(1,\mathsf{extend},l_e,\langle \mathsf{while}\ (s>0)\{\ldots\}\rangle)\})$
$\parallel loc(2,\bot,\{\}) \parallel loc(3,\bot,\{\}) \rightsquigarrow^{\text{SELECT}} S_5 \rightsquigarrow$
$S_6 \equiv loc(0,0,\ldots) \parallel loc(1,1,\{tsk(1,\mathsf{extend},l_e,\langle \mathsf{Fut\ f{=}y!add(l,5);}\ldots\rangle)\})$
$\parallel loc(2,\bot,\{\}) \parallel loc(3,\bot,\{\}) \rightsquigarrow^{\text{ASYNC}}$
$S_7 \equiv loc(0,0,\ldots) \parallel loc(1,1,\{tsk(1,\mathsf{extend},l_e[\mathsf{f}\mapsto\langle 2,\bot,\bot\rangle],\langle \mathsf{await\ f?;}\ldots\rangle)\})$
$\parallel loc(2,\bot,\{tsk(2,\mathsf{add},l_a,\langle \mathsf{List\ r = Cons(e,l);return\ r}\rangle)\}) \parallel loc(3,\bot,\{\}) \rightsquigarrow^{\text{SELECT}}$

$S_8 \equiv loc(0,0,\ldots)) \parallel loc(1,1,\{tsk(1,\mathsf{extend},l_e,\langle \mathsf{await\ f?;}\ldots\rangle)\})$
$\quad \parallel loc(2,2,\{tsk(2,\mathsf{add},l_a,\langle \mathsf{List\ r = Cons(e,l);return\ r}\rangle)\}) \parallel loc(3,\bot,\{\}) \rightsquigarrow S_9 \rightsquigarrow^{\text{RETURN}}$
$S_{10} \equiv loc(0,0,\ldots) \parallel loc(1,1,\{tsk(1,\mathsf{extend},l_e[\mathsf{f} \mapsto \langle 2,true,\bot\rangle],\langle \mathsf{await\ f?;}\ldots\rangle)\})$
$\quad \parallel loc(2,\bot,\{tsk(2,\mathsf{add},l_a,\epsilon([5,1,2]))\}) \parallel loc(3,\bot,\{\}) \rightsquigarrow^{\text{AWAIT-T+GET-R}}$
$S_{12} \equiv loc(0,0,\ldots) \parallel loc(2,\bot,\{tsk(2,\mathsf{add},l_a,\epsilon([5,1,2]))\}) \parallel loc(3,\bot,\{\}) \parallel loc(1,1,$
$\quad \{tsk(1,\mathsf{extend},l_e[\mathsf{f}\mapsto\langle 2,true,[5,1,2]\rangle,\mathsf{l}\mapsto[5,1,2]],\langle \mathsf{z!process(l);}\ldots\rangle)\}) \rightsquigarrow^{\text{ASYNC}}$
$S_{13} \equiv loc(0,0,\ldots) \parallel loc(2,\bot,\ldots) \parallel loc(3,\bot,\{tsk(3,\bot,l_p,body(\mathsf{process}))\})$
$\quad loc(1,1,\{tsk(1,\mathsf{extend},l_e,\langle \mathsf{s = s - 1;}\ldots\rangle)\})$

From state S_0 to S_3 we create the three locations $\mathsf{x}(1)$, $\mathsf{y}(2)$ and $\mathsf{z}(3)$ applying rule NEWLOC. In S_3 a new task extend is spawned using rule ASYNC, that is placed in the queue of location 1. Since location 1 is idle but the queue contains the non-finished task 2 in S_4, it takes the lock (SELECT) and executes the first iteration of the loop. In S_6 and S_7 a new task add is spawned to location 2 and it takes the lock. Note that in S_7 the local mapping is extended to store that the future variable f is linked to task 2, which is not finished yet ($\bot$). Task 2 finishes immediately by assigning variable r and returning: it stores the final value [5,1,2] and notifies task 1 (RETURN). Since task 2 is finished in S_{10} the await and get instructions can proceed (rules AWAIT-T and GET-R resp.), yielding to S_{12}. Finally, task 2 spawns a new task process in location 3.

3 The Notion of Transmission Data Size

The *transmission data size* of a program execution is the total amount of data that is moved between locations. There are two situations that generate data movement between locations: a) when a task is invoked (in this case it sends a message to the destination location containing all the arguments); and b) when the returning value of a task invocation is retrieved (it sends a message containing that value). Therefore, only these two transitions of states will contribute to the transmission data size of a program execution. In order to define this notion we will consider that state transitions are decorated with transmission data size information: $S_1 \rightsquigarrow^{d}_{(lid_1,lid_2,m)} S_2$, meaning a transmission of d units of data from object lid_1 to lid_2 through m. Transitions that do not generate data transmission will be decorated as $S_1 \rightsquigarrow^{0}_{\epsilon} S_2$. Since we are considering an abstract representation of data by means of functional types, we will focus on *units of data* transmitted instead of bits, which depends on the actual implementation and is highly platform-dependent. Concretely, we assume that the cost of transmitting a basic value or a data type constructor is one unit of data. This size measure is known as *term size*. However, the static analysis we propose later would work also with any other mapping from data types to corresponding sizes (given by means of a function α such as the one below).

Definition 1 (term size). *The* term size *of value v—$\alpha(v)$—is defined as:*

$$\alpha(v) = \begin{cases} 1 + \sum_{i=1}^{n} \alpha(v_i) & \text{if } v = Co(v_1 \ldots v_n), \\ 1 & \text{otherwise.} \end{cases}$$

Example 4 (size measures). Considering the term size measure, the size of the list $\mathsf{l} = \mathsf{Cons(1,\ Cons(2,\ Cons(3,Nil)))}$ is $\alpha(\mathsf{l}) = 7$ (4 data constructors and 3 integers) and the size of the tree $\mathsf{t} = \mathsf{Branch(2,\ Leaf(1),\ Leaf(3))}$ is $\alpha(\mathsf{t}) = 6$ (3 constructors plus 3 integers).

Definition 2 (decorated step). *A step $S_1 \rightsquigarrow S_2$ using rule R from Fig. 2 is decorated as follows:*

- *If $R =$ ASYNC then the step is decorated as $S_1 \rightsquigarrow^{d}_{(lid,lid_1,m)} S_2$, where $d = \mathcal{I} + \sum_{z \in \overline{z}} \alpha(l(z))$, and m is the method invoked in the call. The constant $\mathcal{I}$ is the size of establishing the communication, and we add the size of all the arguments passed to the destination location. Note that a task invocation inside the same location ($lid = lid_1$) will not generate any transmission, so in these cases the decoration is $S_1 \rightsquigarrow^{0}_{\epsilon} S_2$.*
- *If $R =$ GET-R then the decorated step is $S_1 \rightsquigarrow^{d}_{(lid_2,lid_1,m)} S_2$, where $d = \mathcal{I} + \alpha(v)$, v corresponds to the returned value, and m is the method that returned v. As before, if $lid = lid_1$ then there is no transmission and the decoration is $S_1 \rightsquigarrow^{0}_{\epsilon} S_2$.*
- *If $R \in \{$NEWLOC, RETURN, AWAIT-T, AWAIT-F, GET-L, SELECT$\}$, then the step does not move any data, so it is decorated with an empty label: $S_1 \rightsquigarrow^{0}_{\epsilon} S_2$.*

Observe that rules AWAIT-T, AWAIT-F and GET-L use local variables only, and therefore do not perform any remote communication. Rule RETURN notifies the termination of a method to the caller location, although its cost is included in the size $\mathcal{I}$ for establishing the communication included in rule ASYNC.

Definition 3 (transmission data size of a trace). *Given a decorated trace $\mathcal{T} \equiv S_0 \rightsquigarrow^{d_1}_{o_1} S_1 \rightsquigarrow^{d_2}_{o_2} \ldots \rightsquigarrow^{d_n}_{o_n} S_n$, the transmission data size of $\mathcal{T}$—trans($\mathcal{T}$)—is defined as:*

$$trans(\mathcal{T}) = \sum_{i=1}^{n} d_i$$

Example 5 (transmission data size). The decorated trace from Ex. 3 is:

$$\mathcal{T}_d \equiv S_0 \rightsquigarrow^{0}_{\epsilon} S_1 \rightsquigarrow^{0}_{\epsilon} S_2 \rightsquigarrow^{0}_{\epsilon} S_3 \rightsquigarrow^{\mathcal{I}+6}_{(0,1,\mathsf{extend})} S_4 \rightsquigarrow^{0}_{\epsilon} S_5 \rightsquigarrow^{0}_{\epsilon} S_6 \rightsquigarrow^{\mathcal{I}+6}_{(1,2,\mathsf{add})} S_7 \rightsquigarrow^{0}_{\epsilon} S_8 \rightsquigarrow^{0}_{\epsilon} S_9 \rightsquigarrow^{0}_{\epsilon} S_{10} \rightsquigarrow^{0}_{\epsilon} S_{11} \rightsquigarrow^{\mathcal{I}+7}_{(2,1,\mathsf{add})} S_{12} \rightsquigarrow^{\mathcal{I}+7}_{(1,3,\mathsf{process})} S_{13}$$

From S_3 to S_4 it sends a message ($\mathcal{I}$) from location 0 to 1 containing the arguments of the call: l=Cons(1,Cons(2,Nil)) and s=7, where $\alpha(l) = 5$ and $\alpha(7) = 1$. Similarly, from S_6 to S_7 it sends a message from location 1 to 2 with the arguments l and 5 for task add. In State S_9 it executes a return instruction, that notifies the termination to the caller, but its size is already considered in the call (S_6). The returning value from the call to add is actually received from the caller at S_{12}, by means of a message from location 2 to 1 with the returning value $\mathsf{r} = \mathsf{Cons(5,Cons(1,Cons(2,Nil)))}$, $\alpha(r) = 7$. Finally, the invocation of task process in state S_{12} sends a message from location 1 to 3 containing the argument $\mathsf{l} =$

Cons(5,Cons(1,Cons(2,Nil))), of size 7. Considering this decorated trace, the total transmission data size is:

$$trans(\mathcal{T}_d) = (\mathcal{I}+6)+(\mathcal{I}+6)+(\mathcal{I}+7)+(\mathcal{I}+7) = 4{*}\mathcal{I}+26$$

In other words, the transmission data size is $4{*}\mathcal{I}$ units of data for creating 4 messages, and 26 units of data for the transmission of values.

The transmission data size of a trace takes into account all the invocation and returning messages, independently of the location involved. In our setting we have several locations that can be executing in different machines or CPUs, so it is interesting to limit transmission data size to some locations. We define a *restriction* operator over traces to consider only data-moving steps between certain locations.

Definition 4 (trace restriction). *Given a decorated trace $\mathcal{T}$, two location identifiers, l_1 and l_2, a method m, the trace restriction $\mathcal{T}|_{l_1 \xrightarrow{m} l_2}$ is defined as:*

$$\mathcal{T}|_{l_1 \xrightarrow{m} l_2} = \{S_{i-1} \rightsquigarrow^{d_i}_{(l_1,l_2,m)} S_i \mid S_{i-1} \rightsquigarrow^{d_i}_{(l_1,l_2,m)} S_i \in \mathcal{T}\}$$

4 Automatic Inference of Transmission Data Sizes

The analysis has three main parts which are introduced in the following sections: Sec. 4.1 is encharged of inferring the locations in the distributed system and using them to define the *cost centers* on which the cost analysis is based; Sec. 4.2 infers upper bounds on the number of tasks spawned along any execution of the program; Sec. 4.3 over-approximates the sizes of the data transmitted when spawning asynchronous calls and when retrieving their results.

4.1 Inference of Distributed Locations

Since locations can be dynamically created, we need an analysis that abstracts them into a *finite* abstract representation, and that tells us which (abstract) location a reference variable is pointing-to. *Points-to* analysis [14,13,15] solves this problem. It infers the set of memory locations that a reference variable can *point-to*. Different abstractions can be used and our method is parametric on the chosen abstraction. Any points-to analysis that provides the following information with more or less accurate precision can be used (our implementation uses [13]): (1) $\mathcal{O}$, the set of abstract locations; (2) a function $pt(pp, v)$ that, for a given program point pp and variable v, returns the set of abstract locations in $\mathcal{O}$ to which v may point.

Example 6 (distributed locations). Consider the main method shown in Fig. 1 which creates three locations x, y and z at L2, L3 and L4, and which are abstracted, respectively, as o_x, o_y and o_z. By using the points-to analysis we obtain the following set of objects created along the execution of main, $\mathcal{O} = \{o_x, o_y, o_z\}$.

Besides, the points-to analysis can infer information for the local variables at the level of program point, that is, $pt(L11, \mathsf{this}) = \{o_x\}$, $pt(L13, \mathsf{y}) = \{o_y\}$, $pt(L16, \mathsf{z}) = \{o_z\}$, $pt(L20, \mathsf{this}) = \{o_y\}$, $pt(L24, \mathsf{this}) = \{o_z\}$, $pt(L26, \mathsf{y}) = \{o_y\}$ or $pt(L8, \mathsf{this}) = \{o_y\}$.

The distributed locations that the points-to analysis infers are used to define the *cost centers* [3] that the resource analysis will use. The notion of cost center is used to attribute the cost of each instruction to the location that executes it. In the above example, we have three locations which lead to three cost centers, $c(o_x)$, $c(o_y)$ and $c(o_z)$.

4.2 Inference of number of tasks spawned

Our analysis builds upon well-established work on cost analysis [9,16,3]. Such analyses are based on a generic notion of resource which can be instantiated to measure different metrics such as number of executed instructions, amount of memory created, number of calls to methods, etc. In particular, the *cost model* is used to determine the type of resource we are measuring. Traditionally, a cost model is a function $\mathcal{M} : \mathit{Instr} \to \mathbb{N}$ which, for each instruction in the program, returns a natural number which represents its cost. As examples of cost models we could have: for counting the number of instructions executed by a program, the cost model counts one unit for any instruction, i.e., $\mathcal{M}^i(ins) = 1$; for counting the number of calls, we can use $\mathcal{M}^c(ins) = 1$ if $ins \equiv x!m(\_)$; and 0 otherwise. When the analysis uses cost centers, the cost model additionally defines to which cost center the cost must be attributed. For instance, when counting number of instructions, we have that $\mathcal{M}(i) = \sum_{o \in pt(pp,this)} c(o) * 1$, where pp is the program point of instruction i, i.e., the instruction is accumulated in all locations that it can be executed (this is given by the locations to which the this reference can point).

In what follows, we use the cost analyzer as a black box in the following way. Given a method $m(\bar{x})$ and a cost model, the cost analyzer gives us an *upper bound* for the total cost (for the resource specified in the cost model) of executing m of the form $\mathcal{U}_m(\bar{x}) = \sum_{i=1}^{n} cc_i * C_i$, where cc_i is a cost center and C_i is a cost expression that bounds the cost of the computation carried out by the cost center cc_i. If one is interested in studying the computation performed by one particular cost center cc_j, we simply replace all cc_i with $i \neq j$ by 0 and cc_j by 1. In order to obtain the cost expression C_i, the cost analyzer needs to over-approximate the number of iterations that loops perform, and infer the maximum sizes of data. For the sake of this paper, we do not need to go into the technical details of this process. To infer an upper bound on the number of tasks spawned by the program, we simply have to define a *number of tasks cost model* and use the cost analyzer as a black box.

Definition 5 (number of tasks cost model). *Given an instruction ins at program point pp, we define the number of tasks cost model, $\mathcal{M}^t(ins)$ as a function which returns $c(o_1, o_2, m)$ if $ins \equiv f{=}y!m(\_) \wedge o_1 \in pt(pp, this) \wedge o_2 \in pt(pp, y) \wedge o_1 \neq o_2$, and 0 otherwise.*

The main feature of the above cost model is that we use an extended form of cost centers which are triples of the form $c(o_1, o_2, m)$, where o_1 is the object that is executing, o_2 is the object responsible for executing the call, and m is the name of the invoked method. These cost centers are symbolic expressions that will be part of the upper bound computed by the analyzer. Let us see an example.

Example 7 (number of tasks). For the code in Fig. 1, cost analysis infers that the number of iterations of the loop in extend (at L12) is bounded by the expression $\mathsf{nat}(s)$, where $\mathsf{nat}(e)$ returns e if $e > 0$ and 0 otherwise. Since the size of l is increased within the loop at L12, the maximum number of iterations for the loop at L25 is produced in the last call to process. Recall that l represents the term size of the list l (see Def. 1), and it counts 2 units for each element in the list. Therefore, each iteration of the loop at L25 increments the term size of the list in 2 units and, consequently, the last call to process is done with a list of size $l+2*s$. The loop in process (L25) traverses the list received as argument consuming 2 size units per iteration. Therefore, the expression $(l + 2 * s)/2 = l/2 + s$ bounds the number of iterations of such loop. As process is called $\mathsf{nat}(s)$ times, $\mathsf{nat}(s) * \mathsf{nat}(l/2 + s)$ bounds the number of times that the body of the loop at L25 is executed. Then, by applying the number of tasks cost model we obtain the following expression that bounds the number of tasks spawned:

$$\begin{aligned}\mathcal{U}^t_{\mathsf{extend}}(l, s) = {} & c(o_x, o_y, \mathsf{add}) * \mathsf{nat}(s) + \\ & c(o_x, o_z, \mathsf{process}) * \mathsf{nat}(s) + \\ & c(o_z, o_y, \mathsf{foo}) * (\mathsf{nat}(s) * \mathsf{nat}(l/2+s))\end{aligned}$$

From the upper bounds on the tasks spawned, we can obtain a range of useful information: (1) If we are interested in the number of communications for the whole program, we just replace all expressions $c(o_1, o_2, m)$ by 1. (2) Replacing all cost centers of the form $c(o, \_, \_)/c(\_, o, \_)$ by 1 for the object o and the remaining ones by 0, we obtain an upper-bound on the number of tasks spawned from/in o. We use, respectively, $\mathcal{U}_m|_{o\rightarrow}$ and $\mathcal{U}_m|_{\rightarrow o}$ to refer to the UB on the outgoing/incoming tasks. (3) Replacing $c(o_1, o_2, \_)$ by 1 for selected objects and the remaining ones by 0, we can see the tasks spawned by o_1 in o_2, denoted by $\mathcal{U}_m|_{o1\rightarrow o_2}$. (4) If we are interested in a particular method p, we can replace $c(\_, \_, p)$ by 1 and the rest by 0, we use $\mathcal{U}_m|_{\xrightarrow{p}}$ to denote it.

Example 8 (number of tasks restriction). Given the upper bound of Ex. 7, the number of tasks spawned from o_x to o_y is captured by replacing $c(o_x, o_y, \_)$ (the method is not relevant) by 1 and the rest by 0. Then, we obtain the expression $\mathcal{U}^t_{\mathrm{extend}}|_{o_x\rightarrow o_y} = \mathsf{nat}(s)$, which shows that we have one task for each iteration of the loop at L13. We can also obtain an upper bound on the number of tasks from o_z to o_y, $\mathcal{U}^t_{\mathrm{extend}}|_{o_z\rightarrow o_y} = \mathsf{nat}(s) * \mathsf{nat}(l/2+s)$. The number of tasks spawned using method foo are captured by $\mathcal{U}^t_{\mathrm{extend}}|_{\xrightarrow{\mathsf{process}}} = \mathsf{nat}(s)$.

4.3 Inference of amount of transmitted data

Our goal now is to infer, not only the number of tasks spawned, but also the sizes of the arguments in the task invocation and of the returned values. Formally, this

is done by extending the previous cost model to include data sizes as well. We rely on two auxiliary functions. Given a variable x at a certain program point, function $\alpha(x)$ returns the term size of this variable at this point, as defined in Sec. 3. Besides, after spawning a task, we are interested in knowing whether the result of executing the task is retrieved, and in such case we accumulate the size of the return value. This information is computed by a *may-happen-in-parallel* analysis [5] which allows us to know to which task a future variable is associated. Thus, we can assume the existence of a function $hasGet(pp)$ which returns if the result of the task spawned at program point pp is retrieved by a get instruction. Now, we define a new cost model that counts the sizes of the data transferred in each communication by relying on the two functions above.

Definition 6 (data sizes cost model). *Given a program point pp we define the cost model $\mathcal{M}^d(ins)$ as a function which returns $sc(ins)$ if $pp : ins \equiv r = y!m(\overline{x}) \wedge o_1 \neq o_2 \wedge o_1 \in pt(pp, this) \wedge o_2 \in pt(pp, y)$, and 0, otherwise; where*

$$sc(ins) = \begin{cases} c(o_1,o_2,m)*(\mathcal{I} + \sum_{x_i \in \overline{x}} \mathsf{nat}(\alpha(x_i))) + c(o_2,o_1,m)*(\mathcal{I} + \mathsf{nat}(\alpha(r))) & \textit{if } hasGet(pp) \\ c(o_1,o_2,m)*(\mathcal{I} + \sum_{x_i \in \overline{x}} \mathsf{nat}(\alpha(x_i))) & \textit{otherwise} \end{cases}$$

Observe that the above cost model extends the one in Def. 5 as it extends the number of tasks cost model with the sizes of the data transmitted. Intuitively, as any call always transfers its input arguments, their size is always included (second case). However, the size of the returned information is only included when there exists a get instruction that retrieves this information (first case). In each case, we include the size for sending the messages $\mathcal{I}$. Note that the cost centers reflect the direction of the transmission, $c(o_1, o_2, m)$ corresponds to a transmission from o_1 to o_2 through a call to m, whereas $c(o_2, o_1, m)$ corresponds to the information returned by o_2 in response to a call to m spawned by o_1. If needed, call and return cost centers can be distinguished by marking the method name, e.g., m for calls and m^r for returns. As already mentioned, nat denotes the positive value of an expression. We wrap the size of each argument using nat because this way the analyzer treats them as an expression whose cost we want to maximize (the technical details of the maximization operation can be found in [4]). Therefore, the upper bound inferred by the analyzer using this cost model already provides the overall information (i.e., number of tasks spawned and maximum size of the data transmitted).

Example 9 (data sizes cost model). Let us see the application of the cost model to the calls at L16, L13 and L26. At L16 we have the instruction z!process(l). As the program does not retrieve any information from process(l), the function $hasGet(L16)$ returns false, and thus we only include the calling data. Then, using the points-to information in Ex. 6, the application of $\mathcal{M}^d$ at L16 returns: $\mathcal{M}^d(\mathsf{z!process(l)}) = c(o_x, o_z, \mathsf{process}) * \mathcal{I} + \mathsf{nat}(\alpha(l))$. As l is a data structure and it is modified within the loop, $\alpha(l)$, returns the term size of l. Observe that the expression captures, not only the objects and the method involved in the

call within the cost center, but also the amount of data transferred in the call, $\mathsf{nat}(\alpha(l))$. The application of $\mathcal{M}^d$ to the call at L13, $\mathsf{f} = \mathsf{y!add(l,5)}$, returns the expression:

$$\begin{aligned}\mathcal{M}^d(\mathsf{f{=}y!add(l_0{,}5)}) = {} & c(o_x, o_y, \mathsf{add}) * (\mathcal{I} + \mathsf{nat}(\alpha(l_0)) + \mathsf{nat}(\alpha(5)))+ \\ & c(o_y, o_x, \mathsf{add}) * (\mathcal{I} + \mathsf{nat}(\alpha(f)))\end{aligned}$$

In this case, at L15 we have a get for the call at L13, so $hasGet(L13) = true$. Note that we use l_0 to refer to the value of l at the beginning of the loop and l to refer to the value of the list after calling add. The application of $\alpha(5)$ returns 1, as it is a basic type (counting as one constructor). The call at L27 returns the expression $c(o_y, o_z, \mathsf{foo}) * (\mathcal{I} + \mathsf{nat}(\alpha(h)))$.

As we have explained above, the size of a data structure might depend on the input arguments that in turn can be modified along the program execution. Consequently, if we are in a loop, for the same program point, the amount of data transferred in one call can be different for each iteration of the loop. Soundness of the cost analysis ensures that it provides the worst possible size in such case. Technically, it is done by maximizing [4] the expressions inside nat within their calling context.

Example 10 (data sizes upper bound). Once the cost model is applied to all instructions in the program, we obtain a set of recursive equations which define the transmission data sizes within the locations in the program. After solving such equations using [4], we obtain the following expression which defines the transmission data sizes of any execution starting from extend, denoted by $\mathcal{U}^d_{\mathsf{extend}}$:

$$\begin{aligned}\mathcal{U}^d_{\mathsf{extend}}(l, s) = {} & c(o_x, o_y, \mathsf{add}) * \mathsf{nat}(s) * (\mathcal{I} + \mathsf{nat}(l + s * 2 - 2) + 1)+ & \textcircled{1} \\ & c(o_y, o_x, \mathsf{add}) * \mathsf{nat}(s) * (\mathcal{I} + \mathsf{nat}(l + s * 2))+ & \textcircled{2} \\ & c(o_x, o_z, \mathsf{process}) * \mathsf{nat}(s) * (\mathcal{I} + \mathsf{nat}(l + s * 2))+ & \textcircled{3} \\ & c(o_z, o_y, \mathsf{foo}) * (\mathsf{nat}(s) * \mathsf{nat}(l/2{+}s)) * (\mathcal{I} + 1) & \textcircled{4}\end{aligned}$$

The expression at ① includes the transmission from o_x to o_y. The worst case size of the list at this point is $\mathsf{nat}(l+s*2-2)$, this is because initially the list has size $\mathsf{nat}(l)$ and at each iteration of the loop, the size is increased in method add by two elements: Cons and an integer value. As the loop performs s iterations, in the last invocation to add it has length $l + (s-1) * 2$. This size is assumed for all loop iterations (worst case size), hence we infer that the maximum data size transmitted from o_x to o_y is $\mathsf{nat}(s) * (\mathcal{I} + \mathsf{nat}(l + s * 2 - 2) + 1)$, the 1 is due to the second argument of the call (an integer). At ②, o_x receives from o_y the same list, but including the last element, that is $\mathsf{nat}(l + s * 2)$. The same list is obtained at ③. In ④, the cost is constant in all iterations (1 integer).

As already mentioned in Sec. 4.2, the fact that cost centers are symbolic expressions allows us to extract different pieces of information regarding the amount of data transferred between the different abstract locations involved in the communications. With $\mathcal{U}^d$ we can infer, not only an upper-bound on the total amount of data transferred along the program execution, but also the size of the data transferred between two objects, or the incoming/outgoing data sent/received by a particular object.

				Nodes			Methods			Pairs		
Benchmark	loc	$\#_c$	T	$\%^n_M$	$\%^n_m$	$\%^n_a$	$\%^m_M$	$\%^m_m$	$\%^m_a$	$\%^p_M$	$\%^p_m$	$\%^p_a$
BBuffer	200	17	829	25.7	0.6	16.3	43.9	0.1	6.2	7.3	0.0	0.7
MailServer	119	13	693	30.0	4.4	15.4	27.3	0.5	10.0	8.7	0.0	0.6
Chat	302	10	171	40.5	7.5	20.0	12.7	0.1	3.0	9.6	0.0	1.1
DistHT	146	9	1204	48.0	3.0	18.7	40.7	0.3	10.0	8.0	0.0	0.9
BookShop	366	10	3327	58.7	3.9	23.9	23.6	0.1	8.3	29.5	0.0	1.5
PeerToPeer	263	19	62575	27.7	0.1	15.6	20.6	0.1	5.8	5.9	0.0	0.5

Table 1. Experimental results (times in ms)

Example 11 (data sizes restriction). From $\mathcal{U}^d_{\mathsf{extend}}(l,s)$, using the cost centers as we have explained in Ex. 8, we can extract different types of information about the data transferred. For instance, we can bound the size of the outgoing data from location x:

$$\mathcal{U}^d_{\mathsf{extend}}(l,s)|_{o_x\rightarrow} = \mathsf{nat}(s)*(\mathcal{I}+\mathsf{nat}(l+s*2-2)+1)+\mathsf{nat}(s)*(\mathcal{I}+\mathsf{nat}(l+s*2))$$

Or the incoming data sizes for the location y:

$$\mathcal{U}^d_{\mathsf{extend}}(l,s)|_{\rightarrow o_y} = \mathsf{nat}(s)*(\mathcal{I}+\mathsf{nat}(l+s*2-2)+1)+(\mathsf{nat}(s)*\mathsf{nat}(l/2{+}s))*(\mathcal{I}+1)$$

Theorem 1 (soundness). *Let P be a program and l_1, l_2 location identifiers. Let $\mathcal{O}$ be the object names computed by a points-to analysis of P. Let o_1, o_2 be the abstractions of l_1, l_2 in $\mathcal{O}$. Then, given a trace $\mathcal{T}$ from P with arguments $\overline{x}$ we have that*

$$trans(\mathcal{T}|_{l_1 \xrightarrow{m} l_2}) \leq \mathcal{U}^d_P(\overline{x})|_{o_1 \xrightarrow{m} o_2}.$$

5 Experimental Results

We have implemented our analysis in SACO [2] and applied it to some typical examples of distributed systems: BBuffer, a bounded-buffer for communicating several producers and consumers; MailServer, a client-server distributed system; Chat, a chat application; DistHT, a distributed hash table; BookShop, a web shop client-server application; and PeerToPeer, a peer-to-peer network with a set of interconnected peers. Experiments have been performed on an Intel Core i7 at 3.4GHz with 8GB of RAM, running Ubuntu 12.04.

We have applied our analysis and evaluated the upper bound expressions for different combinations of concrete input values so as to obtain some quantitative information about the analysis. Table 1 summarizes the results obtained. Columns Benchmark and loc show, resp. the name and the number of program lines of the benchmark. Column $\#_c$ displays the number of locations identified by the analysis. Column T shows the time to perform the inference of the transmission data sizes. We have studied the transmission data sizes among each pair of locations identified by the points-to analysis. We have studied data transmission from three points of view: (1) from a location with the rest of the program, (2) from a method, and (3) among pairs of locations. In case (1), we try to

identify potential bottlenecks in the communication, i.e., those locations that produce/consume most of the data in the benchmark. Also, we want to observe locations that do not have much communication. In the former, such locations should have a fast communication channel, while in the latter we can still have a good response time with slower bandwidth conditions. Columns $\%^n_M$, $\%^n_m$, $\%^n_a$ show, respectively, the percentage of the location that accumulates more traffic (incoming + outgoing) w.r.t. the total traffic in the system, for the location with less traffic, and the average for the traffic of all locations. Similarly, columns $\%^p_M$, $\%^p_m$, $\%^p_a$ show, for case (3), which is the percentage of the total traffic transmitted by the pair of locations that have more traffic, by the pair with less traffic and the average between the traffic of all pairs, respectively. Finally, regarding case (2), columns under Methods show similar information but taking into account the task that performs the communication, i.e., the percentage of the traffic transmitted by the task that transmits more (resp., less) amount of data, $\%^m_M$ (resp., $\%^m_m$), and the average of the transmissions performed by each task ($\%^m_a$).

We can observe in the table that our analysis is performed in a reasonable time. One important issue is that we only have to perform the analysis once, and the information can be extracted later by evaluating the upper bound with different parameters and focusing in the communications of interest. In the columns for the locations, we can see that all benchmarks are relatively well distributed. The average of the data transmitted per location is under 25% for all benchmarks. BookShop is the benchmark which could have a communication bottleneck as it accumulates in a single location 58.7% of the total traffic. Regarding methods, it is interesting to see that for all benchmarks no method accumulates more than 45% of the total traffic. Moreover, the table shows that in all benchmarks there is at least one method that requires less than 0.5%, in most cases this method (or methods) is an object constructor. Regarding pairs of locations, in all benchmarks there is at least one pair of locations that do not communicate, $\%^p_m = 0$ for all benchmarks. This is an expected result, as it is quite often to have pairs of locations which do not communicate in a distributed program. Our experiments thus confirm that transmission among pairs of locations is relatively well distributed, as in most benchmarks, except for BookShop, the pair with highest traffic requires less than 10% of the total traffic.

6 Conclusions

We have presented a static analysis to soundly approximate the amount of data transmitted among the locations of a distributed system. This is an important contribution to be able to infer the response times of distributed components. In particular, if one knows the bandwidth conditions among each pair of locations, we can infer the time required to transmit the data and to retrieve the result. This time should be added to the time required to carry out the computation at each location, which is an orthogonal issue. Conversely, we can use our analysis to establish the bandwidth conditions required to ensure a certain response time. Technically, our analysis is formalized by defining a new cost model which

captures only the data transmission aspect of the application. This cost model can be plugged into a generic cost analyzer for distributed systems, that directly returns an upper bound on the transmission data sizes, without requiring any modification to the other components of the cost analyzer.

Acknowledgments. This work was funded partially by the EU project FP7-ICT-610582 ENVISAGE: Engineering Virtualized Services (http://www.envisage-project.eu) and by the Spanish projects TIN2008-05624 and TIN2012-38137.

References

1. Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, MA, USA, 1986.
2. E. Albert, P. Arenas, A. Flores-Montoya, S. Genaim, M. Gómez-Zamalloa, E. Martin-Martin, G. Puebla, and G. Román-Díez. SACO: Static Analyzer for Concurrent Objects. In *Proc. of TACAS'14*, volume 8413 of *LNCS*, pages 562–567. Springer, 2014.
3. E. Albert, P. Arenas, S. Genaim, M. Gómez-Zamalloa, and G. Puebla. Cost Analysis of Concurrent OO programs. In *Proc. of APLAS'11*, volume 7078 of *LNCS*, pages 238–254. Springer, December 2011.
4. E. Albert, P. Arenas, S. Genaim, and G. Puebla. Closed-Form Upper Bounds in Static Cost Analysis. *Journal of Automated Reasoning*, 46(2):161–203, 2011.
5. E. Albert, A. Flores-Montoya, and S. Genaim. Analysis of May-Happen-in-Parallel in Concurrent Objects. In *Proc. of FORTE'12*, volume 7273 of *LNCS*, pages 35–51. Springer, 2012.
6. J. Armstrong, R. Virding, C. Wistrom, and M. Williams. *Concurrent Programming in Erlang*. Prentice Hall, 1996.
7. P. Cousot and N. Halbwachs. Automatic Discovery of Linear Restraints Among Variables of a Program. In *POPL*. ACM Press, 1978.
8. F. S. de Boer, D. Clarke, and E. B. Johnsen. A Complete Guide to the Future. In *Proc. of ESOP'07*, volume 4421 of *LNCS*, pages 316–330. Springer, 2007.
9. S. Gulwani, K. K. Mehra, and T. M. Chilimbi. Speed: Precise and Efficient Static Estimation of Program Computational Complexity. In *Proc. of POPL'09*, pages 127–139. ACM, 2009.
10. P. Haller and M. Odersky. Scala actors: Unifying thread-based and event-based programming. *Theor. Comput. Sci.*, 410(2-3):202–220, 2009.
11. E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A Core Language for Abstract Behavioral Specification. In *Proc. of FMCO'10 (Revised Papers)*, volume 6957 of *LNCS*, pages 142–164. Springer, 2012.
12. Jonathan K. Lee and Jens Palsberg. Featherweight x10: a core calculus for async-finish parallelism. *SIGPLAN Not.*, 45(5):25–36, 2010. 1693459.
13. A. Milanova, A. Rountev, and B. G. Ryder. Parameterized Object Sensitivity for Points-to Analysis for Java. *ACM Trans. Softw. Eng. Methodol.*, 14:1–41, 2005.
14. M. Shapiro and S. Horwitz. Fast and Accurate Flow-Insensitive Points-To Analysis. In *Proc. of POPL'97*, pages 1–14, Paris, France, January 1997. ACM.
15. M. Sridharan and R. Bodík. Refinement-based context-sensitive points-to analysis for Java. In *PLDI*, pages 387–400, 2006.
16. F. Zuleger, S. Gulwani, M. Sinn, and H. Veith. Bound analysis of imperative programs with the size-change abstraction. In *SAS*, volume 6887 of *LNCS*, pages 280–297. Springer, 2011.