DiTyCO: Concorrência e Mobilidade de Código

Álvaro Figueira   Fernando Silva   Luís Lopes   Vasco Vasconcelos

Technical Report Series: DCC-99-4

Departamento de Ciência de Computadores – Faculdade de Ciências

&

Laboratório de Inteligência Artificial e Ciência de Computadores

Universidade do Porto

Rua do Campo Alegre, 823 4150 Porto, Portugal

Tel: +351+2+6078830 – Fax: +351+2+6003654

http://www.ncc.up.pt/fcup/DCC/Pubs/treports.html

DiTyCO: Concorrência e Mobilidade de Código*

Álvaro Figueira[1] Fernando Silva[1] Luís Lopes[1] Vasco Vasconcelos[2]
arf@ncc.up.pt fds@ncc.up.pt lblopes@ncc.up.pt vv@di.fc.ul.pt

[1] *DCC-FC & LIACC, Universidade do Porto, Portugal*
[2] *DI-FCUL, Universidade de Lisboa, Portugal*

Resumo

O DiTyCO é um sistema capaz de realizar computações concorrentes, distribuídas e com mobilidade de código. O modelo de programação subjacente baseia-se num cálculo assíncrono de processos e é declarativo. Neste artigo descrevem-se os aspectos fundamentais da implementação do sistema exemplificados num servidor de *applets*.

1 Introdução

O desenvolvimento de software baseado na comunicação entre processos tem vindo a aumentar continuamente acompanhando o crescimento das redes de comunicação e da Internet. Neste contexto, um conceito fundamental é o de *mobilidade*, que consiste na capacidade de objectos ou computações mudarem a sua localização consoante a evolução de um sistema. A migração de objectos e computações é um factor de extrema importância para o acesso à informação num mundo em que a interligação e distribuição dos mais diversos recursos é cada vez maior.

O sistema que aqui se apresenta denomina-se DiTyCO (Distributed Typed Concurrent Objects) e constitui uma extensão da linguagem TyCO [4] que por sua vez implementa o cálculo com o mesmo nome [6]. A linguagem TyCO incorpora as noções de objectos, invocação assíncrona de métodos (mensagens) e definição de classes genéricas de processos. O modelo de programação é declarativo. A introdução de distribuição e mobilidade traduz-se ao nível da sintaxe do DiTyCO na adição de duas primitivas: **import** e **export** que declaram as interfaces iniciais dos agentes de um programa distribuído.

O modelo de programação do DiTyCO [5] mantém a distribuição e a mobilidade de código totalmente implícitas, sendo induzidas pelo âmbito léxico das variáveis de um programa. Inicialmente, uma variável exportada passa a estar acessível a qualquer agente de uma computação distribuída, bastando para isso ser importada no código de algum agente. A partir desse momento é possível realizar a comunicação entre dois agentes utilizando essa variável. A natureza da comunicação pode consistir na troca de mensagens assíncronas ou

*O primeiro autor é suportado pela bolsa PRAXIS/BD/13971/97.
Artigo apresentado no 1o. Encontro Português de Computação Móvel, Tomar, Novembro, 1999.

de objectos (caso em que existe mobilidade de código). A partir do momento em que há troca de informação entre agentes de uma computação torna-se possível enviar referências de novas variáveis criadas localmente nos agentes. Isto permite criar topologias de comunicação arbitrariamente complexas.

O modelo de programação do DiTyCO permite que a programação dos agentes de uma computação seja independente da sua localização na rede (*network transparent*). No entanto ao permitir especificar a interface inicial dos agentes (utilizando **import** e **export**), torna a programação *network aware*.

Ao nível da implementação, o DiTyCO utiliza o sistema desenvolvido para a linguagem TyCO que é constituído por um compilador, um assemblador de byte-code e um emulador. A implementação actual da linguagem TyCO é sequencial [3]. Com o DiTyCO a utilização de múltiplos threads na execução dos agentes de uma computação distribuída, introduz concorrência.

O facto do DiTyCO estar baseado num modelo formal (uma extensão do cálculo TyCO) permite a definição de uma semântica operacional adequada para a linguagem, bem como, a utilização de um sistema de tipos para a verificação de programas e detecção, parte estática parte dinâmica, de eventuais erros de protocolo.

No que se segue descreve-se o modelo de programação associado ao DiTyCO, a arquitectura do sistema e alguns detalhes da sua implementação. Finalmente o artigo termina com a apresentação de um exemplo e conclusões.

2 Modelo de Programação

A linguagem TyCO baseia-se num cálculo assíncrono de processos com o mesmo nome [6], uma forma do cálculo π assíncrono [1]. As abstracções fundamentais da linguagem são invocações assíncronas de métodos e objectos:

$x!l[\tilde{v}]$	invocação assíncrona de método
$x?\{l_1(\tilde{x}_1) = P_1, \ldots, l_n(\tilde{x}_n) = P_n\}$	objecto

Nesta sintaxe, x é uma variável e $\tilde{x}_1, \ldots, \tilde{x}_n$ sequências de variáveis. Num dado contexto num programa uma variável está associada a um valor v que pode ser uma constante primitiva (e.g., booleanos ou inteiros) ou um canal a de comunicação. $\tilde{v}$ é uma sequência de valores. $l, l_1, \ldots, l_n$ são identificadores constantes de métodos. Os nomes nas sequências $l_1, \ldots, l_n$, e também em $\tilde{x}_i$, são todos distintos. O termo $a!l_i[\tilde{v}]$ denota invocação do método l_i com argumentos $\tilde{v}$ num objecto localizado no canal a, por exemplo, $a?\{l_1(\tilde{x}_1) = P_1, \ldots, l_n(\tilde{x}_n) = P_n\}$; o resultado é o corpo P_i do método l_i onde as variáveis de $\tilde{x}_i$ são substituídas pelos valores $\tilde{v}$. Estas abstracções são ainda complementadas pelos seguintes constructores usuais em linguagens concorrentes:

$P_1 \mid P_2$	composição concorrente
new x P	criação de novas variáveis
def D **in** P	definição de processos genéricos
$X[\tilde{v}]$	instanciação

Nos constructores acima D representa uma sequência de definições, possivelmente recursivas, de processos genéricos: $X_1(\tilde{x}_1) = P_1$ **and** ... **and** $X_n(\tilde{x}_n) = P_n$. Os X_i são variáveis sobre processos genéricos. Uma instanciação $X[\tilde{v}]$ de um processo genérico $X(\tilde{x}) = P$ é outra forma de redução a qual resulta no processo P onde as variáveis $\tilde{x}$ são substituídas pelos valores $\tilde{v}$.

No que se segue pressupomos que o âmbito de uma variável introduzida com o constructor **new** estende-se o mais para a direita possível num termo. O identificador *val* denota o nome do método para objectos com um único método e cujo nome não é relevante, permitindo assim abreviar a sintaxe da invocação.

$$\begin{array}{lll} a![\tilde{v}] & \text{abrevia} & a!\mathit{val}[\tilde{v}] \\ a?(\tilde{x}) = P & \text{abrevia} & a?\{\mathit{val}(\tilde{x}) = P\} \end{array}$$

A extensão da linguagem descrita para o DiTyCO é muito simples necessitando apenas da introdução de dois novos constructores.

export variável **in** programa
import variável **from** agente **in** programa

export cria uma variável local e torna-a pública, permitindo que possa ser importada de outros agentes. **import** torna uma variável pública remota parte integrante do contexto local de um agente e da computação que aí decorre.

Exemplo. Um servidor de applets disponibiliza um conjunto de k applets diferentes, para downloading, através de k métodos de um objecto. Uma instância de AppletServer é criada na variável local appletserver, a qual é exportada para a rede. O código do applet j é dado pelo termo P_j e é seleccionado mediante a invocação do método applet_j.

O código para o servidor no agente borneo é da forma:

```
export appletserver in
def AppletServer (self) =
   self ? {
      applet1(p) = p?(x)=P1 | AppletServer[self],
       ...
      appletk(p) = p?(x)=Pk | AppletServer[self]}
in AppletServer[appletserver]
```

Um cliente deste serviço, no agente sumatra, importa a variável appletserver onde está localizado o servidor de applets no agente borneo. Em seguida cria uma nova variável local p, onde o código do applet que vai receber irá ser colocado. Finalmente invoca o método no servidor que dá acesso ao applet pretendido e em paralelo envia uma mensagem de arranque ao applet com um argumento v.

```
import appletserver from borneo in new p appletserver!appletj[p] | p![v]
```

Devido ao âmbito léxico de appletserver, a mensagem appletserver!applet_j[p] é enviada para o agente borneo onde reduz com o servidor de applets. O resultado da redução é a execução em borneo do processo p?(x)=P_j. Uma vez que o âmbito de p é o agente sumatra o código do objecto aí colocado ({*val*(x)=P_j}) migra para o canal associado à variável p do cliente. Finalmente, uma vez que o cliente tinha já enviado a mensagem p![v] para arrancar com o applet, o termo {v/x}P_j é executado no agente sumatra.

3 Arquitectura do DiTyCO

A arquitectura do DiTyCO está organizada em três camadas: agentes, máquinas e redes. Os agentes são as entidades que executam os programas. As máquinas correspondem a nós-IP e podem albergar um número arbitrário de agentes em execução concorrente ou paralela. Por último, a rede compõe-se de múltiplos nós DiTyCO interligados numa topologia-IP estática. Trocas de código e mobilidade de código ocorrem ao nível dos agentes, e a este nível a topologia é dinâmica. A figura 1 ilustra esta organização do DiTyCO em camadas.

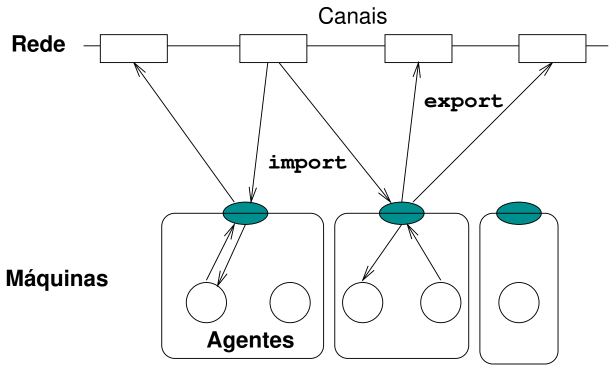

Figura 1: A arquitectura do DiTyCO.

3.1 A Rede

Esta camada fornece os meios para que determinadas variáveis possam ser tornadas públicas e portanto acessíveis por qualquer agente que saiba da sua existência. A informação sobre os agentes activos e as variáveis públicas é mantida em duas tabelas geridas por um servidor a que chamamos SNS (Site Name Server). Cada entrada da tabela dos agentes inclui um lexema que identifica o nome do agente, o número IP da máquina onde está em execução e a sua localização (um número natural único que identifica o agente dentro do nó IP). Por seu lado, as entradas na tabelas de variáveis públicas incluem um lexema que identifica a variável, o identificador da variável (um número natural único que identifica a variável dentro do nó IP) e o lexema que identifica o agente que exportou a variável. A título de exemplo, aquando da criação de um agente borneo, com identificador loc1 no nó ip1 é adicionado à tabela de agentes um tuplo (borneo,ip1,loc1). Uma instrução **export** x executada no agente borneo tem como efeito adicionar o tuplo (x,var2,borneo), admitindo que var2 é o identificador da variável x.

Apenas as variáveis declaradas públicas através do constructor **export** são registadas no SNS. Quando um agente, por exemplo sumatra, executa uma instrução de partilha de uma variável pública, **import** x **from** borneo, pergunta primeiro ao SNS pelo endereço de x:borneo na rede.

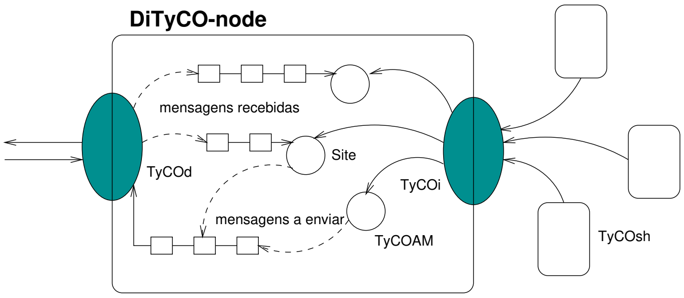

Figura 2: Um nó DiTyCO.

Caso exista uma entrada para esse endereço no SNS, o agente sumatra recebe a descrição completa da variável pública x do agente borneo, ficando assim com a informação necessária para poder comunicar através dessa variável. No agente sumatra, o endereço da variável x ficará representado pelo tuplo (ip1,loc1,var2).

Na versão actual do sistema, o SNS está centralizado e é, à partida, conhecido por todos os agentes. O serviço está implementado com sockets TCP/IP. Cada agente pode ser um cliente do servidor SNS para, por exemplo, efectuar as operações **import/export**. Os pedidos de um agente ao servidor são feitos através de um thread, local a cada nó IP, dedicado ao processamento de comunicação.

3.2 As Máquinas

Cada máquina ou endereço IP suporta um nó DiTyCO, constituído por um número arbitrário de agentes e dois demónios de comunicação, um para comunicação entre nós DiTyCO (TyCOd) e outro para comunicação com o utilizador (TyCOi). A figura 2 ilustra esta arquitectura.

Um nó DiTyCO é um processo Unix. As suas componentes: agentes, TyCOd e TyCOi, são threads e partilham os recursos (e.g., memória, descriptores) atribuídos ao processo. Os agentes executam programas DiTyCO e constituem a unidade básica de computação do modelo. Um novo agente é criado para cada programa submetido para execução e a sua existência expira quando o programa termina. Um agente corresponde a um emulador da máquina abstracta do DiTyCO. A submissão de novos programas para execução num dado agente faz-se através da interface com o utilizador gerida pelo TyCOi.

O TyCOd é responsável por todas as comunicações entre agentes. Essas comunicações podem ser locais, se os agentes estão no mesmo nó DiTyCO, ou remotas no caso de não estarem no mesmo nó. As comunicações locais são optimizadas através de memória partilhada. As comunicações remotas são executadas via TCP/IP e envolvem quatro passos distintos, transparentes para o programador:

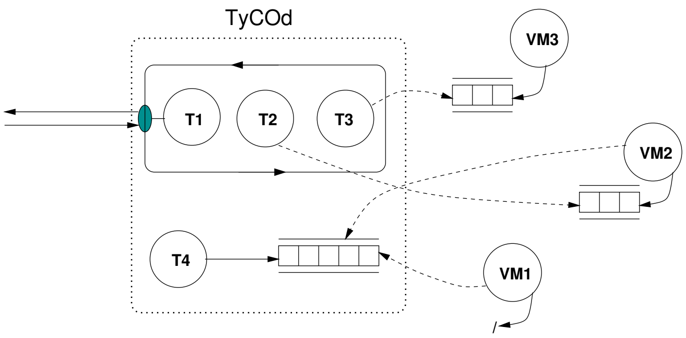

Figura 3: O TyCOd.

1. o agente inicia a comunicação colocando uma mensagem na fila do TyCOd local;
2. o TyCOd local comunica com o TyCOd do nó DiTyCO remoto;
3. o TyCOd remoto coloca a mensagem na fila do agente receptor;
4. o agente receptor processa a mensagem.

O TyCOd é implementado por dois conjuntos de threads: threads-emissores e threads-receptores. Na versão actual existe apenas um thread-emissor, que faz a gestão da fila de mensagens provenientes de agentes locais e que se destinam a outros agentes. Essa fila é implementada em memória partilhada e com recurso a *locks* para evitar acesso concorrente por parte do TyCOd (a consumir as mensagens) e dos agentes (a produzir mensagens). Os threads-receptores fazem a recepção de mensagens externas, provenientes de agentes remotos, e encaminham-nas para a fila de mensagens do agente a que se destina a mensagem. Na figura 3 os threads-receptores são os threads 1, 2 e 3; e o thread-emissor é o thread 4.

A opção de implementar o TyCOd com múltiplos threads deve-se a questões de eficiência. A recepção de mensagens externas é um processo demorado que exige o estabelecimento de uma conexão TCP (e portanto um procedimento que envolve 3 mensagens). Para minimizar este efeito, tem-se vários threads pré-criados que, à vez, aceitam conexões correspondentes ao estabelecimento de uma comunicação entre agentes. Uma vez estabelecida uma ligação, o atendimento é delegado para um dos threads livres.

O uso de vários threads não sobrecarrega demasiadamente o sistema porque quando não há mensagens para receber/distribuir, só um dos threads está em funcionamento e todos os outros estão adormecidos. Este sistema funciona com threads pré-criados e um *lock* em torno do `accept`. O código para um thread receptor é:

```
enquanto verdade faz
   adquire lock
   aceita conexão
   liberta lock
   recebe mensagem para um endereço de rede (ip,loc,var)
   fecha conexão
   traduz (ip,loc,var) num endereço no heap
   para todas as variáveis na mensagem
      traduz variáveis com prefixo (ip,loc) em endereços no heap
   coloca a mensagem na fila do agente em loc
```

Por sua vez, o thread emissor executa o seguinte ciclo:

```
enquanto verdade faz
   acorda quando a fila de mensagens a enviar deixa de ser vazia
   adquire lock da fila de mensagens
   para todas as mensagens na fila
      para todas as variáveis na mensagem
         traduz variável no seu endereço na rede
      envia mensagem
   liberta lock da fila de mensagens
   volta a adormecer
```

Em geral as mensagens transmitidas no DiTyCO são pequenas, mas há casos em que o seu tamanho pode ser considerável. Por exemplo quando um agente envia código para outro, não é à partida conhecido o tamanho da mensagem, mas há a possibilidade de ser superior ao tamanho máximo permitido pelo protocolo TCP/IP. Para precaver esta situação o TyCOd fragmenta a mensagem se necessário.

De modo simétrico se a mensagem vier fragmentada, os fragmentos são empilhados (note-se que no TCP não pode haver troca de ordem de mensagens) e a mensagem original é formada antes de ser passada ao agente.

O TyCOi é um thread lançado pelo nó DiTyCO que realiza a comunicação inicial com o utilizador. Em particular cria threads para novos agentes e regista-os na tabela de agentes do SNS. A partir do momento em que o agente é criado a sua interacção com o utilizador (input/output) é feita directamente, sem a intervenção do TyCOi.

4 Os Agentes

O agente constitui a unidade básica de execução do DiTyCO e corresponde a um emulador para o DiTyCO com um programa associado. O nome de um agente identifica o "lugar" onde um programa vai ser executado. Este nome deve ser único na rede e tal é verificado pelo SNS quando um utilizador lança o agente para a execução de um programa.

O Emulador para os agentes do DiTyCO corresponde a uma extensão do emulador para a linguagem TyCO [2]. Cada emulador mantém o seu estado numa estrutura de dados onde constam os seus registos, apontadores para o heap, zona de código e pilhas auxiliares. O emulador do DiTyCO acrescenta a esta arquitectura: duas filas para mensagens a enviar/receber através do TyCOd; duas novas instruções – **import** e **export** e uma porta que liga o emulador à shell do utilizador para realização de *input/output*.

As instruções **import** e **export** foram implementadas no emulador com a seguinte semântica. A instrução **export** x executada num agente borneo desencadeia a seguinte sequência de operações:

1. cria um novo canal associado à variável local x;
2. gera um índice único j para a variável x;
3. envia ao SNS o tuplo (x,j,borneo).

Por outro lado, a instrução **import** x **from** borneo executa as operações:

1. contacta o SNS pedindo informações sobre (x,borneo);
2. recebe um tuplo (ip,loc,var) que é o endereço na rede de x;
3. coloca o endereço de rede da variável pública x na variável local x.

A exportação implícita de variáveis ocorre quando uma variável local aparece no corpo de uma mensagem ou um objecto que deverá migrar para um agente remoto. No exemplo seguinte:

import x **from** sumatra **new** y x?()=y![]

y é uma variável local que ocorre livre no corpo do objecto x?()=y![] que deve migrar para o canal associado com a variável pública x de sumatra. Quando o objecto migra para sumatra a variável y é implicitamente exportada. Para cada variável local exportada nestas condições gera-se um endereço de rede que segue na mensagem para o agente remoto. Simultaneamente adiciona-se uma entrada na tabela de exports do nó DiTyCO local que mapeia o endereço de rede da variável no seu endereço físico no heap do agente local.

Quando ocorre migração de objectos é necessário identificar que parte do byte-code do agente local corresponde à implementação do objecto, isto é, a tabela de métodos e o código para os métodos. Actualmente o compilador gera byte-code que preserva a estrutura sintática dos programas originais facilitando assim a identificação do código a migrar.

A redução no DiTyCO envolve alterações na implementação de algumas instruções da máquina virtual desenvolvida para o TyCO. Estas alterações derivam do facto de a redução no DiTyCO depender do âmbito léxico dos canais envolvidos. Para implementar interacções remotas, como envio de mensagens ou migração de código, a máquina virtual tem de oferecer uma interface para a camada de software responsável pela troca de mensagens entre agentes. Como vimos, interacções remotas acontecem quando o emulador encontra uma mensagem $a\,!\,l[\tilde{v}]$, ou um objecto $a?M$, em que o canal a não é local. Outra possibilidade decorre de instanciações $X[\tilde{v}]$ em que a abstracção X é definida num agente remoto.

As instruções `redobj`, `redmsg` da máquina sequêncial devem ser modificadas da seguinte forma. O emulador tem de verificar se o endereço do canal onde a redução vai ocorrer é local ou composto. O primeiro caso, a redução num canal local, é semelhante à redução na máquina sequêncial. No caso de uma redução num canal remoto, com endereço composto, a mensagem ou objecto correspondente têm de ser enviados pela rede até ao agente remoto onde a redução será local.

```
di_redobj(a,M):
   se a=(ip,loc,var) então
      extrai byte-code de M
      traduz variaveis locais em fv(M)
      envia para agente loc em ip
   senão
      redobj(a,M)
```

```
di_redmsg(a,l,v1, ... ,vk):
   se a=(ip,loc,var) então
      traduz variaveis locais em v
      envia para agente loc em ip
   senão
      redmsg(a,l,v1, ... ,vk)
```

De forma similar, a instrução `instof`, utilizada na instanciação de classes de processos, deve verificar se o seu argumento, uma label para um bloco de byte-code, corresponde a byte-code local ou de um agente remoto. No primeiro caso a instanciação decorre como na máquina sequêncial. Se o byte-code não fôr local então a máquina virtual terá de enviar uma mensagem ao agente remoto pedindo o código para a definição de X e as variáveis livres que possa conter. Esta operação é síncrona. Quando o código chega ao agente local a instanciação prossegue como na máquina sequêncial.

```
di_instof(X,v1, ... ,vk):
   se X=(ip,loc,byte) então
      pede codigo de X
   instof(X,v1, ... ,vk)
```

5 O Servidor de Applets Revisitado

Aqui assumimos que temos dois agentes, de nome borneo e sumatra, com localizações e endereços IP, respectivamente, (borneo,loc1,ip1) e (sumatra,loc4,ip2). Os agentes interagem através da variável pública appletserver que é exportada explicitamente no início do programa que corre em borneo. Segundo o exemplo da secção 2 temos o seguinte código em cada nó:

borneo: **export** appletserver **in** ... AppletServer[appletserver]

sumatra: **import** appletserver **from** borneo **in new** p appletserver!$applet_j$[p] | p![v]

O constructor **export** cria uma nova variável local appletserver, a localização do servidor de applets, no heap do agente borneo. Em seguida, é introduzida na tabela de exports do agente uma entrada para o endereço da variável num índice único (por exemplo, 3) que irá servir de identificador da variável daqui para a frente. Finalmente borneo envia uma mensagem ao TyCOd local pedindo a disponibilização do canal appletserver para a restante rede. O resultado desta operação traduz-se na inserção dos tuplos (borneo,loc1,ip1) e (appletserver,var3,borneo) nas tabelas de agentes e canais exportados no SNS.

Entretanto, no agente sumatra, o emulador processa a instrução **import**. Primeiro envia uma mensagem ao TyCOd local, o qual por sua vez questiona o SNS pelo endereço de rede de uma variável appletserver disponibilizada pelo agente borneo. O endereço de rede (ip1,loc1,var3) é retornado ao agente sumatra (a operação **import** é síncrona) e é ligado a uma variável local appletserver. Ainda em sumatra, o emulador executa a próxima instrução criando uma nova variável local p.

Em seguida o emulador encontra uma mensagem destinada ao canal appletserver cujo âmbito léxico o liga a borneo. Neste caso a variável appletserver contém um endereço composto, de rede, (ip1,loc1,var3). Assim, a mensagem deve ser enviada pela rede até ao agente cujo identificador é loc1 no endereço ip1. Todos os argumentos da mensagem locais a sumatra devem ser traduzidos de apontadores para o heap local para endereços de rede e para cada um deles uma entrada deve ser inserida na tabela de exports de sumatra, por forma a preservar o mapeamento entre os endereços na rede das variáveis locais e o seus endereços no heap.

Neste caso p está ligada a um canal local pelo que um inteiro único (por exemplo, 2) é escolhido como índice de p na tabela de exports. Na mensagem enviada a borneo, p é escrita em formato de rede. Desta forma teríamos appletserver!$applet_j$[p] re-escrito como (ip1,loc1,var3)!$applet_j$[(ip2,loc4,var2)].

Entretanto o emulador em borneo criou uma instância da classe AppletServer. Esta instância é um objecto que é colocado numa fila no heap associada a appletserver.

A mensagem (ip1,loc1,var3)!$applet_j$[(ip2,loc4,var2)] proveniente de sumatra chega a ip1 e é recebida pelo TyCOd local. Este procura na mensagem endereços compostos com prefixo (ip1,loc1) que identifica canais originários do agente borneo. Em seguida o TyCOd usa a tabela de exports para traduzir estes endereços compostos em endereços no heap local. O endereço é seleccionado usando o sufixo do endereço composto como índice (por exemplo, 3 para (ip1,loc1,var3)). A mensagem é depois entregue a borneo e é eventualmente processada.

O emulador em borneo recebe a mensagem e reduz localmente com o objecto em appletserver produzindo um novo objecto $\mathsf{p?(x)} = \mathsf{P}_j$ onde a variável p está ligada ao endereço composto (ip2,loc4,var2) associado com sumatra.

O caso agora é dual. O emulador em borneo ao verificar que p é um endereço composto associado com sumatra, envia o byte-code para o objecto $\mathsf{p?(x)} = \mathsf{P}_j$ pela rede. O byte-code necessário, a tabela de métodos e o código para os métodos do objecto, é extraído do programa e colocado numa zona de memória para ser enviado pela rede para a localização loc4 em ip2. As variáveis livres dos métodos têm também de ser enviadas devidamente traduzidas em endereços compostos caso o seu âmbito léxico assim o exija. O código do applet é enviado pela rede para o cliente em sumatra.

Quando o código chega ao TyCOd do sumatra o endereço composto (ip2,loc4,var2) da variável p é traduzido no seu endereço local no heap de sumatra. O objecto é entregue a esse agente e reduz com a mensagem p![v] previamente enviada pelo cliente para invocar o applet. O resultado é a execução do código do applet j, P_j com o parâmetro x ligado ao argumento v.

6 Conclusões e Trabalho Futuro

O DiTyCO é um ambiente para execução de computações distribuídas permitindo concorrência e mobilidade de código, apoiado num cálculo de processos assíncrono. O modelo de programação induzido pelo cálculo com distribuição e mobilidade de código é muito simples e permite ao utilizador abstrair-se da arquitectura distribuída subjacente.

O sistema está na fase terminal de implementação de uma primeira versão, após o que estamos interessados em efectuar uma avaliação de performance preliminar. O próximo passo será expandir o actual sistema de tipos estático para um sistema híbrido no qual os programas que correm nos agentes são tipificados estaticamente. A tipificação dinâmica é necessária quando existe interacção entre agentes em variáveis exportadas. As variáveis enviadas numa tal comunicação serão acompanhadas do seus respectivos tipos, que serão verificados à chegada ao agente remoto.

Agradecimentos

Este trabalho foi parcialmente suportado pela Fundação para a Ciência e Tecnologia através dos projectos Dolphin (PRAXIS/2/2.1/TIT/1577/95) e DiCoMo (PRAXIS/P/EEI/12059/98).

Referências

[1] K. Honda and M. Tokoro. An Object Calculus for Asynchronous Communication. In *European Conference on Object-Oriented Programming (ECOOP'91)*, volume 512 of *LNCS*, pages 141–162. Springer-Verlag, 1991.

[2] L. Lopes, F. Silva, and V. Vasconcelos. A Virtual Machine for the TyCO Process Calculus. In *Principles and Practice of Declarative Programming (PPDP'99)*, volume 1702 of *LNCS*, pages 244–260. Springer-Verlag, September 1999.

[3] L. Lopes and V. Vasconcelos. TyCOAM - The Definition. Technical Report DCC-97-1, DCC-FC & LIACC, Universidade do Porto, May 1997.

[4] V. Vasconcelos and R. Bastos. Core-TyCO - The Language Definition. Technical Report TR-98-3, DI / FCUL, March 1998.

[5] V. Vasconcelos, L. Lopes, and F. Silva. Distribution and Mobility with Lexical Scoping in Process Calculi. In *Workshop on High Level Programming Languages (HLCL'98)*, volume 16(3) of *ENTCS*, pages 19–34. Elsevier Science, 1998.

[6] V. Vasconcelos and M. Tokoro. A Typing System for a Calculus of Objects. In *International Symposium on Object Technologies for Advanced Software (ISOTAS'93)*, volume 742 of *LNCS*, pages 460–474. Springer-Verlag, November 1993.