# GOBLINS: un modello computazionale per la progettazione e lo sviluppo di sistemi multiagenti

Francesco Garelli e Carlo Ferrari
Dipartimento di Elettronica e Informatica
Università di Padova
via Gradenigo 6a - 35131 Padova
e-mail {pascal, carlo}@dei.unipd.it
tel 049-8277729 fax 049-8277699

## Abstract

*In questo articolo viene presentato un modello computazionale per lo studio e la progettazione di sistemi multiagenti derivato dalla teoria degli attori. Il modello proposto è affiancato da un'architettura software, basata su tecnologie ad oggetti e schemi XML, in grado di fornire un supporto per la sperimentazione e lo sviluppo.*

## 1 Introduzione

La possibilità di avere a disposizione reti di calcolatori a costo contenuto, ha contribuito a risvegliare l'interesse della comunità scientifica sulle problematiche relative al progetto e alla realizzazione di sistemi informatici distribuiti.

Il paradigma dei Sistemi Multiagente (MAS) fornisce un'adeguata risposta alle esigenza progettuali per questi sistemi. Un'architettura di questo tipo è composta da diverse entità che possiedono capacità decisionali autonome e indipendenti, nonostante le varie attività possano influenzarsi tra di loro. Nei MAS acquistano particolare rilievo quelle problematiche legate all'uso di protocolli di interazione. Si può quindi sia riconoscere un comportamento sociale del sistema nel suo complesso sia attribuire ad ogni agente una tipologia di comportamento sociale personale.

Negli anni '70 è stata proposta la teoria degli attori come modello computazionale per sistemi distributi [2]. Tale modello presenta caratteristiche comuni ai sistemi ad agenti, quali comunicazione asincrona attraverso lo scambio di messaggi e concorrenza, ed è quindi un interessante punto di partenza per uno studio teorico dei MAS. Un attore è costituito da due componenti:

1. una *mailbox*: una casella di posta dove vengono mantenuti i messaggi in attesa di essere elaborati. La consegna dei messaggi è garantita mentre non è garantito un tempo massimo per tale consegna.
2. un *behaviour*: una lista di metodi per elaborare i messaggi contenuti nella mailbox. Ogni metodo è associato ad un particolare tipo di messaggio.

Durante l'esecuzione un attore può creare nuovi attori o cambiare il proprio behaviour, cioè la procedura con cui vengono elaborati i messaggi.

In questo lavoro è stato esteso e modificato il modello ad attori per adattarlo ai recenti sviluppi nello studio dei sistemi distribuiti in particolare nell'ambito di applicazioni *real-time* [5].

## 2 Il modello computazionale

### 2.1 Messaggi

L'estensione alla teoria degli attori studiata utilizza lo stesso modello di comunicazione asincrona introducendo alcune forme ridotte di controllo. Su richiesta del mittente, il destinatario di un messaggio deve spedire alcune segnalazioni:

1. una segnalazione di acknowledge nell'istante in cui il messaggio sta per essere elaborato
2. una segnalazione di successo o fallimento rispettivamente quando il messaggio è stato elaborato con successo o quando il destinatorio non è stato in grado di elaborarlo. In ogni caso il destinatario non può sospendere il messaggio in attesa che una particolare condizione sia verificata: nel momento in cui lo preleva dalla mailbox deve elaborarlo o restituire la segnalazione di fallimento.
3. la restituzione di un messaggio di risposta.

## 2.2 Protocolli

Nel paradigma di programmazione ad attori, come in quello ad oggetti, l'attenzione del progettista è rivolta alla definizione dei componenti del sistema e solo in secondo piano alle interfacce pubbliche con cui i componenti comunicano; per questo motivo la interfacce vengono di solito studiate durante la progettazione degli attori.

Nei paradigmi ad agenti invece, il problema della comunicazione ha un ruolo centrale nella progettazione del sistema e spesso viene affrontato prima della definizione dei singoli componenti; quindi gli agenti interagiscono attraverso linguaggi e protocolli standard, realizzati in base al problema che si vuole affrontare e non in base ai servizi che il singolo agente deve fornire.

Il modello proposto segue questo approccio attraverso il concetto di *protocollo*. Il contenuto di un messaggio è organizzato in una struttura di dati primitivi (interi, stringhe...), un *template*; attraverso un template si realizza un controllo sintattico sui dati scambiati analogo al controllo fornito dalla definizione di funzione nei linguaggi procedurali. Inoltre un insieme di templates rivolti alla risoluzione di uno stesso problema sono raccolti in un *protocollo.* Per realizzare un protocollo è necessario un linguaggio in grado di descrivere strutture di dati. Di fatto qualsiasi linguaggio che permette di definire nuovi tipi è in grado di assolvere a questa funzione; nell'implementazione sviluppata in questo lavoro è stato utilizzato XMLSCHEMA.

Come esempio consideriamo la gestione del segnale orario: vogliamo implementare un protocollo per ricevere l'ora esatta da un agente. Abbiamo bisogno di due templates, uno con la richiesta dell'ora ed uno con la corrispondente risposta. Nella richiesta specifichiamo lo Stato da cui viene spedito il messaggio, per permettere al goblin di considerare il fuso orario. Nella risposta restituiamo il valore temporale suddividendolo nei tre campi numerici, ore, minuti e secondi. Di seguito è riportato un documento XMLSCHEMA che descrive i due templates.

```
<schema name="timelet">
 <element name="query">
 <type>
  <attribute name="country" type="string"/>
 </type>
 </element>
 <element name="time">
 <type>
  <attribute name="hour" type="integer"/>
  <attribute name="minute" type="integer"/>
  <attribute name="hour" type="integer"/>
 </type>
 </element>
</schema>
```

## 2.3 Goblins

Gli agenti che realizzano la computazione nel sistema vengono chiamati *goblins*. Un goblin estende il concetto di attore prevedendo un supporto esplicito per i protocolli. Ad ogni protocollo utilizzato nella ricezione di messaggi corrisponde una mailbox diversa. Questa scelta ha motivazioni formali e pratiche. Ogni mailbox può essere pensata infatti come un diverso sensore nel modello di agente proposto in letteratura: ogni mailbox riceve informazioni omogenee raccolte in un unico protocollo e rivolte alla gestione di un particolare problema.

Inoltre un goblin definisce un diverso metodo di elaborazione per ogni template: quando riceve un messaggio deve discriminarne il tipo ed invocare il corretto metodo di elaborazione. Di conseguenza mantenere messaggi con protocolli diversi in mailboxes diverse semplifica la procedura di discriminazione e in generale l'architettura dell'agente.

Un'altra differenza fondamentale tra il comportamento di un goblin e quello di un attore riguarda il ciclo di vita. Un attore normalmente viene sospeso quando le sue mailboxes sono vuote. Un goblin invece viene attivato periodicamente da un segnale temporale esterno con una frequenza che dipende dal tipo di agente e dalle sue funzioni. In questo modo il goblin può eseguire azioni sull'ambiente anche quando non viene sollecitato. Ad ogni risveglio, cioè ad ogni segnale temporale, il goblin

1. inserisce i messaggi provenienti dal sistema di trasporto nella mailbox associata al protocollo del messaggio
2. preleva ed elabora un messaggio da ogni mailbox
3. esegue ulteriori azioni sull'ambiente

Questo modello permette di coniugare un comportamento reattivo (risposte immediate agli stimoli dell'ambiente) con una pianificazione dell'azione nel tempo. Il modello prevede uno stato locale associato ad ogni interlocutore: il goblin mantiene cioè una rappresentazione degli agenti da cui riceve messaggi.

In figura 1 è schematizzata la struttura di un goblin. Per semplicità è rappresentato un unico protocollo e quindi un'unica mailbox; il protocollo contiene tre templates a cui sono associati i tre metodi di elaborazione $b_1$, $b_2$ e $b_3$. L'agente interagisce con tre goblins $G_1$, $G_2$e $G_3$ a cui corrispondono rispettivamente i tre stati locali $s_1$, $s_2$ e $s_3$.

## 2.4 Switchers

I costrutti che abbiamo introdotto permettono di realizzare una comunicazione asimmetrica (esistono nella comunicazione due ruoli separati, il mittente ed il destinatario)

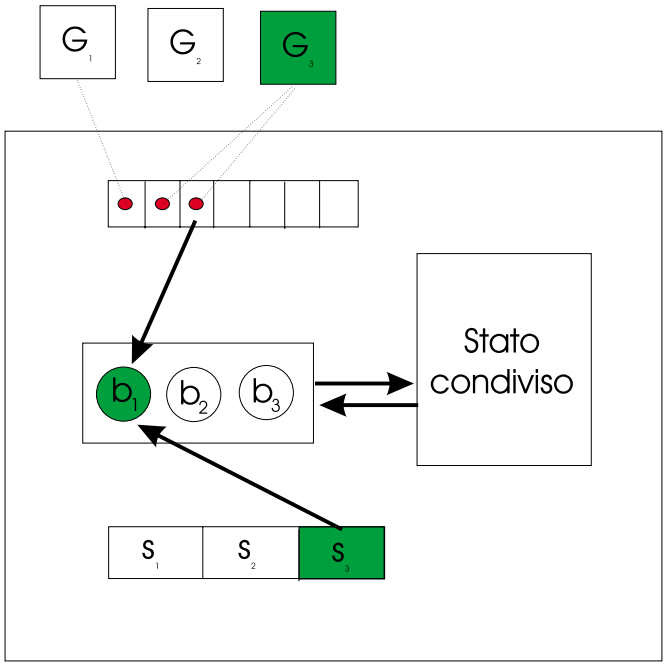


**Figura 1. Rappresentazione di un goblin**

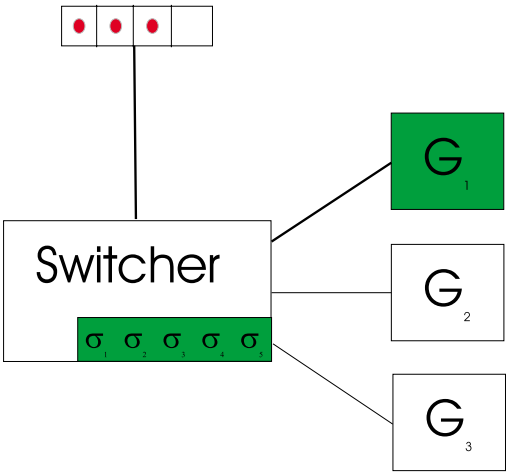


**Figura 2. Pipeling switcher**

da molti agenti ad un unico agente, una comunicazione cioè molti ad uno: un goblin può spedire messaggi a più goblins contemporaneamente (*unicast*) ma non è in grado di spedire un messaggio ad un unico indirizzo ed ottenere che tale messaggio venga recapitato a molti goblins (*multicast*). Per realizzare comunicazioni multicast abbiamo bisogno di uno strumento in grado di ricevere in ingresso un messaggio e spedirlo in uscita a più destinatari.

Uno *switcher* è un componente in grado di instradare messaggi provenienti da un insieme di goblins verso un altro insieme di goblins. La funzione di instradamento dipende dal tipo di messaggio, dal mittente e dallo stato interno dello switcher.

In base al tipo di routing possiamo distinguere diverse categorie di switcher.

### 2.4.1 Copy switcher

Un *copy switcher* inoltra un messaggio in ingresso a tutti o a parte dei destinatari in parallelo. Può essere utilizzato per fornire servizi a più utenti contemporaneamente, come avviene nei sistemi di streaming video. Il fatto che lo stesso messaggio venga elaborato contemporaneamente da più goblins crea una situazione di indeterminismo se il mittente del messaggio ha richiesto una qualche forma di risposta. Infatti a fronte di un'unica risposta da restituire al mittente vi sono le risposte di diversi destinatari; in questa situazione lo switcher deve riassumere le diverse risposte in un unico sunto. Esistono diverse strategie in base alla appli-

cazione che si vuole realizzare: è possibile restituire una segnalazione di successo quando almeno un destinatario ha restituito una segnalazione di successo o al contrario quando tutti i destinatari lo hanno fatto. Inoltre se i destinatari restituiscono oltre a segnalazioni anche messaggi di risposta è necessario costruire dei filtri in grado di elaborare il contenuto delle risposte. In generale quando il sistema prevede messaggi di risposta è preferibile utilizzare altre forme di switchers.

### 2.4.2 Pipeling switcher

Un *pipeling switcher* inoltra un messaggio ad un solo destinatario. Se il messaggio viene rifiutato lo switcher può proporlo ad un altro destinatario o può notificare il rifiuto al mittente. Dato che in un certo istante un messaggio è elaborato da un solo goblin la risposta ad un messaggio è determinata in modo univoco. Questo fatto non comporta la mancanza di concorrenza; infatti dopo che lo switcher ha spedito un messaggio ad un destinatario può iniziare ad instradare il successivo messaggio senza attendere la conclusione dell'elaborazione del primo.

## 3 L'ambiente GDE

L'architettura basata su goblins è stata sperimentata nel corso di questo lavoro grazie alla realizzazione di un ambiente di sviluppo denominato GDE (Goblins' Development Environment). Questo ambiente è costituito da tre componenti indipendenti:

1. un compilatore *schemac*, in grado di generare i protocolli di comunicazione tra gli agenti
2. una libreria, in grado di fornire la struttura interna di un goblin
3. un name service goblin, utilizzato per ottenere canali di comunicazioni con altri goblins appartenenti al sistema

### 3.1 I protocolli di comunicazione

Il primo passo nella creazione di un sistema di goblins è la definizione dei protocolli utilizzati. La descrizione dei templates di cui sono costituiti i protocolli avviene attraverso il linguaggio XMLSCHEMA, un linguaggio di mark-up utilizzato per la definizione di linguaggi XML. Il compilatore *schemac* elabora lo schema dei templates e genera codice sorgente per la gestione del protocollo in due formati:

1. formato nativo: i templates vengono codificati attraverso i costrutti del linguaggio scelto per l'implementazione degli agenti. L'attuale prototipo genera per ogni template una classe C++ le cui istanze sono i contenuti dei messaggi. Questo approccio permette di definire i protocolli attraverso un linguaggio astratto quale XMLSCHEMA e poi utilizzarli con il linguaggio più adatto alla applicazione sviluppata.
2. formato indipendente: la comunicazione tra goblins sviluppati con linguaggi diversi avviene attraverso un formato intermedio, una codifica binaria del linguaggio XML [7] chiamata XMLBinary. Tale codifica, ottenuta associando valori numeri ai tags XML, evita l'overhead per la gestione di un formato testuale e nello stesso tempo permette di convertire (in caso di necessità) un messaggio in una forma comprensibile ad un essere umano o ad un visualizzatore XML.

Il compilatore genera il codice sorgente per la conversione trasparente tra i due formati. Tale conversione richiede una serializzazione degli oggetti C++ in stream XMLBinary.

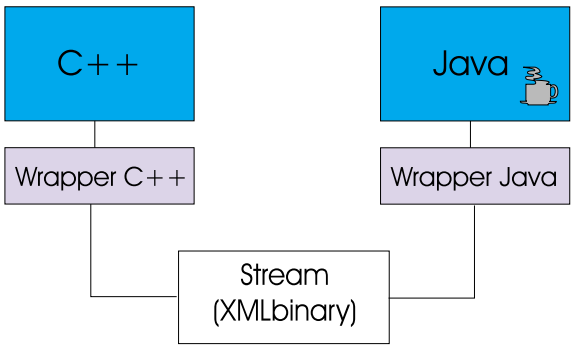


**Figura 3. Trasferimento di messaggi tra goblins implementati con strumenti di sviluppo diversi**

I messaggi vengono trasferiti in formato XMLBinary attraverso sistemi di trasmissioni diversi, in base ai requisiti dell'applicazione. L'ambiente fornisce attualmente tre diversi streams:

1. *gtcpstream*: basata sul protocollo TCP/IP permette il trasferimento di messaggi tra hosts diversi connessi in rete IP.
2. *gpipestream*: basata su pipes UNIX permette la comunicazione tra processi in esecuzione sulla stessa macchina.
3. *gmemstream*: permette la comunicazione tra goblins all'interno dello stesso processo. Goblins realizzati all'interno dello stesso processo utilizzano lo stesso formato nativo per codificare i messaggi; di conseguenza non è necessario convertire i messaggi in XMLBinary.

Generalmente questi streams sono sufficienti per la maggior parte delle applicazioni. In alcune situazioni particolari il programmatore può sviluppare un proprio stream per dialogare con dispositivi hardware o per interfacciarsi con moduli software esterni.

Un'ulteriore funzione dello *schemac* è la generazione delle interfacce dei protocolli: il compilatore genera solo la dichiarazione dei metodi usati nell'elaborazione dei messaggi; la codifica effettiva dei metodi viene invece lasciata al programmatore durante la realizzazione degli agenti.

### 3.2 Libreria di supporto

Il secondo componente del GDE è una libreria di classi e metodi con servizi utili alla creazione e alla gestione dei goblins. Il prototipo sviluppato, in linguaggio C++, fornisce due classi *inbox* e *outbox* rispettivamente per la ricezione e la spedizione dei messaggi. Fornisce inoltre una classe *goblin* con servizi di temporizzazione e concorrenza utilizzati dagli agenti.

La definizione di un goblin avviene attraverso il costrutto dell'eredità multipla, derivando dalla classe *goblin* i servizi di multithreading e dalle interfacce dei protocolli, generate dallo *schemac*, le definizione dei metodi per l'elaborazione dei messaggi. Tali metodi vanno poi ridefiniti per fornire al sistema una risposta adatta ai messaggi ricevuti. Infine è necessario definire una struttura o una classe per rappresentare gli interlocutori. Di seguito è riportato un semplice listato dimostrativo: il goblin implementato gestisce il protocollo *gnslet* (un protocollo standard per la risoluzione dei nomi definito nell'ambiente) ereditando l'interfaccia *gnslet::interface* e ridefinendo i corrispondenti metodi astratti *resolve*, *publish* e *group*; la rappresentazione degli interlocutori avviene attraverso la struttura *Sender.*

```
—— Parte del file header del goblin ——
struct Sender {
   uint32_t ipaddress;
};
class ResolveGoblin: public goblin, public
gnslet::interface<Sender>  {
public:
      ResolveGoblin(vector <wstring> _forwarders);
      ~ResolveGoblin();
protected:
  vector <wstring> forwarders;
  virtual gresult init(gstream& stream, Sender& sender);
  virtual gresult resolve(resolve_t& content, Sender&
sender);
  virtual gresult publish(publish_t& content, Sender&
sender);
  virtual gresult group(group_t& content, Sender&
sender);
 .....
—— Parte del file sorgente ——
ResolveGoblin::ResolveGoblin(vector <wstring>
_forwarders):
   forwarders(_forwarders) {
  gstream::server* server = new
gtcpstream::server(5277);
  int boxhandle = addinbox(new inbox<gnslet,
Sender>(*this));
  addserver(boxhandle, server);
}
ResolveGoblin::~ResolveGoblin(){
}
gresult ResolveGoblin::init(gstream& stream, Sender&
sender) {
  sender.ipaddress = stream.ipaddress();
  return ok;
}
.....
```

### 3.3 Name service

Il terzo componente del GDE è il *name service goblin* (NSG), un agente che converte l'identificatore di una mailbox (definito dal nome di un goblin, il suo dominio e un protocollo di comunicazione) nell'identificatore di uno stream fisico (TCP, PIPE…) su cui possono essere trasmessi i messaggi. Ogni mailbox ha un proprio name service, assegnato in modo dinamico, che organizza gli indirizzi all'interno di gruppi, in una struttura gerarchica analoga ad un file system. Un gruppo viene descritto con una sintassi analoga al formato URL utilizzato in INTERNET, cioè dall'estremità alla radice della gerarchia utilizzando come separatore il punto. Ad esempio un goblin utilizzato per la conversione di un segnale vocale in testo può avere una mailbox in un gruppo *audio*, contenuto a sua volta in un gruppo *devices*; il dominio corrispondente è *audio.devices.*

La conversione da identificatore di una mailbox ad identificatore di uno stream avviene attraverso una query con template *resolve*, uno schema parametrico nel protocollo *gnslet* . La richiesta può avere i seguenti esiti:

1. La query ha successo e viene restituita una lista di streams su cui è possibile aprire la connessione.
2. La query non ha successo e viene restituita una lista di NSG a cui si può riproporre la richiesta.
3. La query non ha successo e non vengono restituiti indirizzi di NSG alternativi da utilizzare. In questo caso il dominio dell'indirizzo da risolvere viene analizzato alla ricerca di un host INTERNET contenuto in esso. Se l'host viene trovato il procedimento viene ripetuto utilizzando il NSG dell'host.

Questo tipo di architettura permette una risoluzione degli indirizzi attraverso una struttura gerarchica simile ai servizi di DNS INTERNET ([8]) e rappresentata in figura 4. La richiesta *resolve* viene sempre eseguita direttamente dall'agente interessato senza la mediazione di altri goblins; in questo modo il NSG che riceve la query può distinguere l'agente, ad esempio attraverso l'indirizzo IP, e ridurre i servizi forniti o gli indirizzi restituiti.

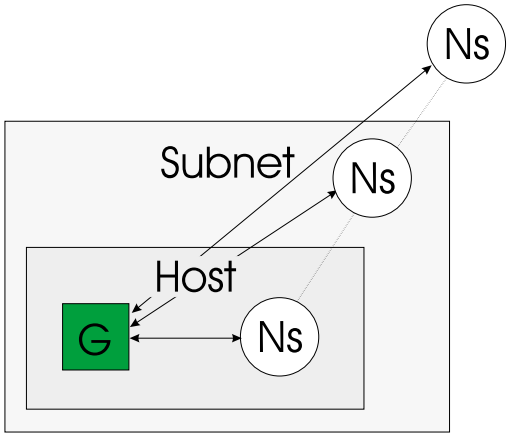


**Figura 4. Name service**

I *name services* forniscono infine servizi per la registrazione delle mailboxes e per la creazione di nuovi gruppi.

## 4 Sviluppi futuri

Il prototipo sviluppato fornisce i principali costrutti definiti nell'architettura. In futuro il GDE potrà essere esteso con un supporto completo per linguaggi ad oggetti diversi dal C++. Successivamente si approfondirà la potenzialità della *reflection* fornita dal linguaggio XMLSCHEMA: il linguaggio utilizzato per descrivere i protocolli ha la stessa struttura dei protocolli. Di conseguenza è possibile spedire attraverso i normali streams la descrizione di un protocollo. In questo modo è possibile interagire con agenti di cui non si conosce a priori il protocollo.

Un ulteriore problema da approfondire è relativo alla gestione del ciclo di vita degli agenti e riguarda il comportamento di un goblin nel momento in cui viene disattivato o per manutenzione della macchina su cui si trova o per conservare risorse utili. Poichè i goblins sono oggetti attivi è auspicabile che il loro stato interno venga conservato tra diverse esecuzioni. Una possibile soluzione prevede di congelare lo stato interno in uno stream XML ed è inoltre uno strumento utile per lo sviluppo di tecniche di migrazione, comuni in molti sistemi ad agenti.

## Riferimenti bibliografici

[1] F. Garelli. *Un'architettura basata su attori per la progettazione e lo sviluppo di sistemi Multiagente.* Tesi di laurea, dipartimento di Elettonica ed Informatica, università di Padova, Aprile 2000.

[2] G. Agha. *Actors: A model of concurrent computation in distributed systems*

[3] J. Pugh. *Actors - The stage is set*. Sigplan notices 19:61-65, March 1984

[4] G. Agha. *An overview of actor languages*. Sigplan notices 21: 58-67, October 1986

[5] A. Burns. *Real time systems and programming languages*. Addison-Wesley, 1996

[6] G. Weiss. *Multiagent Systems*. The MIT Press: Cambridge, 1999

[7] E. Armstrong. *The* XML *tutorial*. 9 Aug 1999

[8] A. Tanenbaum, *Computer Network*. Prentice Hall, 1996

[9] A. Mason, C. Talcott. *Actor languages. Their syntax, semantics, translation, and equivalence*. Theoretical Computer Science, 220:409-467, 1999

[10] P. O'Hare, N. Jennings. *Foundations of Distributed Artificial Intelligence.* Wiley-Interscience Publication, New York, 1996

[11] M. Luck, M. D'Inverno, M. Fischer. *Foundations of Multi-Agent systems: techniques, tools and theory.* The Knowledge Engineering Reviews, 13(3):297-302, 1998

[12] R. Aylett, F. Brazier, N. Jennings, M. Luck, H. Nwana, C. Preist. *Agent systems and applications*. The Knowledge Engineering Reviews, 13(3):303-308, 1998

[13] H. Nwana, D. Ndumu. *A perspective on software agents research*. The Knowledge Engineering Reviews, 14(2):125-142, 1999

[14] J. Ferber. *Multiagent systems. An introduction to distributed artificial intelligence*. Addison-Wesley, 1999