INAF-OAA Gruppo Strumentazione Infrarossa

Progetto Giano
Interfacciamento con il programma gianoMotors: il *server* couatl

E.Giani, C.Baffa

Rapporto Interno di Arcetri N° 6-2010
Versione 1.07, Firenze Novembre 2010

Sommario

Questo documento descrive l'applicazione server couatl *sviluppata per includere in modo semplice ed efficiente il programma di controllo dei motori* gianoMotors *all'interno del sistema gestito dell'applicazione middle-ware* gbridge-balor.

Il nome di questa applicazione è stato ripreso da quello di una razza di mostri del gioco di ruolo "Dungeons & Dragons".

Leggendo la descrizione di queste figure fantastiche, abbiamo intravisto molte similitudini nel comportamento della nostra applicazione e di queste Creature Mostruose. Scherzosamente, l'abbiamo così battezzata.

"Couatls speak Celestial, Common, and Draconic, and also have the power of telepathy. A couatl uses its detect thoughts ability on any creature that arouses its suspicions. Since it is highly intelligent, a couatl usually casts spells from a distance before closing.

At will, it detect chaos, detect evil, detect good, detect law, detect thoughts."[5]

Figura 1: *Aspetto di un couatl*

Capitolo 1

Il sistema di controllo dei motori

Il programma di movimentazione gianoMotors, attualmente utilizzato in laboratorio per posizionare le ottiche dello strumento Giano, è un'applicazione sviluppata in linguaggio Tcl/Tk e fornisce una propria interfaccia utente (GUI).
Lo scopo di questo lavoro consiste nella realizzazione di un'interfaccia software che consenta di utilizzare, da remoto ed in automatico parte delle funzionalità dell'applicazione gianoMotors, più specificatamente quelle relative al solo controllo delle movimentazioni. Quando questa modalità è attiva, l'operatore non potrà interagire direttamente con la GUI del programma, ma solo attraverso comandi specifici, inviati dalla GUI di controllo complessivo di Giano, che passano attraverso l'applicazione couatl la cui descrizione è oggetto di questo rapporto.
Operando in questo modo il programma gianoMotors funziona come una e vera applicazione *server* nei confronti di una qualunque altra applicazione che sia in grado di comunicare con essa, inviando i relativi comandi opportunamente formattati.
Quando gianoMotors lavora seconda questa modalità, diremo che gianoMotors opera in modalità *demone*. Quando invece sarà usato come applicazione di laboratorio, attraverso l'interazione diretta dell'utente con la GUI, diremo che gianoMotors lavora in modalità *laboratorio* o *normale*.
Come illustrato nel documento [1], la funzionalità di *demone* puó essere ottenuta in modo relativamente semplice appoggiandosi alla possibilità di inviare al programma gianoMotors, attraverso una *shell* `wish`, stringhe ASCII con in comandi per la gestione delle ottiche.
L'interfaccia grafica attraverso la quale l'osservatore può interagire direttamente con il sistema dei motori delle ottiche di Giano, sarà invece implementata all'interno della GUI generale di controllo dello strumento.

Capitolo 2

Le applicazioni di controllo dei motori di Giano

Traendo spunto da quanto descritto nell'articolo [2], abbiamo sviluppato un'applicazione C che comunica da una parte con il programma Tcl/Tk gianoMotors, dall'altra con il sistema di controllo di Giano.
L'idea alla base di questa implementazione consiste nell'eseguire il `fork` di un'istanza di `wish` come processo figlio del programma C e comunicare con esso attraverso l'uso di due *pipe*: una aperta in scrittura, l'altra in lettura. In questo modo la shell `wish` rimane un processo separato e lavora come *server* per i comandi `Tcl/Tk` relativi al programma gianoMotors: i comandi sono inviati tramite *pipe* al processo `wish` che li esegue, sfruttando la struttura interpretata del linguaggio in uso.
Da quanto detto sopra emerge come sia possibile utilizzare il programma gianoMotors avendo come sorgente dei comandi il software *middleware* gbridge-balor. Quest'ultimo, per come è stato progettato, prevede la possibilità di gestire le operazione di I/O da sorgenti diverse, tra cui anche *pipe* e quindi è il candidato "perfetto" per interfacciarsi direttamente al programma gianoMotors attraverso questi canali di comunicazione.
Invece di implementare direttamente nel codice gbridge-balor la parte relativa alla comunicazione con gianoMotors, abbiamo deciso di sviluppare una nuova applicazione, denominata couatl, che funzioni da intermediario per le richieste inviate dal programma di interfaccia al sistema di controllo dei motori.
Sebbene questa struttura possa apparire, a prima vista ridondante, alla base di questa scelta vi è la scelta di lasciare aperta la possibilità che, in futuro, gianoMotors venga sostituito da un'applicazione *server* sviluppata in C, linguaggio che riteniamo molto piú idoneo e performante nella gestione di basso livello dell'*hardware*.
In quest'ottica la scrittura del programma couatl semplifica notevolmente l'integrazione dell'eventuale nuovo codice: la parte di comunicazione con lo strato *middle-ware* sarebbe già sviluppata, implementata e testata; sarebbero pre-

senti e disponibili tutti gli agganci necessari su cui innestare la parte di basso livello responsabile della comunicazione con i controller dei motori.
L'applicazione couatl lavora a tutti gli effetti come estensione di gbridge-balor. Costituisce uno strato *middle-ware* tra la parte di alto livello e quella di controllo dei motori: riceve i comandi di movimento, li esamina e li traduce poi in richieste di esecuzione di procedure Tcl/Tk, inviando al nuovo programma `wish`, avviato come descritto nei paragrafi precedenti, stringhe ASCII opportunamente formattate.
In modo analogo all'applicazione gbridge-balor, couatl si comporta come applicazione *server* nei confronti del software *middle-ware*, ma fruisce dei servizi offerti da gianoMotors per soddisfare le richieste dell'osservatore, mostrando quindi, allo stesso momento, il comportamento di una tipica applicazione *client*.

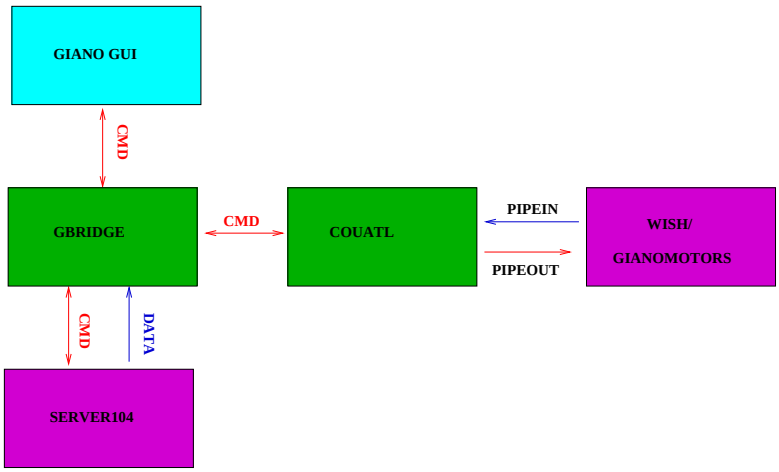

Figura 2.1: *Struttura dell'applicazione di controllo di Giano.*

Invece di avere un canale diretto con l'applicazione GUI, couatl riceve i comandi attraverso il canale verso gbridge-balor. La struttura complessiva del sistema di controllo di Giano, esclusa la sezione relativa alla sensoristica, puó essere schematizzata come in figura 2.1.

Capitolo 3

L'applicazione gbridge-balor

3.1 Il nuovo codename gbridge-balor

Le release più recenti delle nostre applicazioni sono indicate con una coppia di nomi: il primo rappresenta il *function name*, il secondo il *codename*.
Il *function name*, se presente, è il nome originale con cui è nata e conosciuta l'applicazione. Il *codename*, invece, è stato mutuato dal nome di personaggi del gioco "Dungeons & Dragons"[1], scelti in modo che le sue caratteristiche richiamino in qualche modo alla mente le proprietà salienti dell'applicazione.
gbridge è stato ad esempio ribattezzato con il nuovo nome gbridge-balor. Il primo è appunto il nome con cui questa applicazione è stata individuata fin dall'inizio; balor rappresenta invece una classe di demoni tra i piú potenti di "Dungeons & Dragons".
Per questo motivo, tenuto conto del ruolo svolto da questa applicazione all'interno del sistema di controllo di Giano, ci è sembrato un nome adatto da affiancare alla vecchia nomenclatura.
balor è anche il nome con cui gbridge si identifica nei confronti dell'applicazione couatl.
Quest'ultima invece, essendo di recente sviluppo, presenta il solo *codname*.

3.2 Le nuove funzionalità

Tutte le richieste da e verso l'interfaccia GUI da cui opera l'osservatore, passano in modo del tutto trasparente da gbridge-balor: la parte di "intelligenza" risiede nella applicazione couatl per la decodifica e in gianoMotors per l'esecuzione effettiva.
Il compito specifico di gbridge-balor consiste nell'avviare, su richiesta del programma GUI, il *server* couatl e di mantenere attivo e controllare l'attività del canale di comunicazione aperto verso di esso.

[1] Dungeons & Dragons, D & D and Wizards of the Coast and related logos are trademarks or registered trademarks of Wizards of the Coast

Le informazioni che viaggiano attraverso questo canale di comunicazione sono pacchetti formattati secondo il protocollo usato da gbridge e descritto nel documento [3].
Il *socket* aperto tra gbridge-balor e couatl è classificabile come un canale di `tipo comando`. Attraverso questo canale passano le richieste di movimento, i messaggi e gli errori destinati all'operatore, le informazioni con l'aggiornamento dello stato dei motori.
La comunicazione tra le due applicazione avviene tramite *socket locali* del dominio Unix, una forma di comunicazione tra processi in esecuzione sulla stessa macchina.

3.3 Avvio di couatl

gbridge-balor manda in esecuzione il *server* couatl prima di entrare nel ciclo degli eventi, invocando la chiamata di sistema `system`.
Solo in seguito ad una esplicita richiesta di connessione al sistema dei motori da parte dell'operatore, gbridge-balor prova a connettersi al *socket* locale di ascolto aperto dall'applicazione *server* couatl.
La connessione attiva viene aggiunta alla lista delle connessioni sulle quali gbridge-balor esegue il monitoraggio dell'attività, come descritto nel documento [4].
Durante l'esecuzione del *ciclo degli eventi* gbridge-balor intercetta le richieste di movimento generate dall'operatore e le instrada verso couatl. In modo analogo recapita al programma GUI le risposte provenienti dal programma gianoMotors attraverso la "mediazione" di couatl.

Capitolo 4

L'applicazione couatl

L'applicazione couatl presenta una struttura molto simile a quella dell'applicazione *middle-ware* gbridge-balor. Come quest'ultima, anch'essa è un' applicazione guidata da eventi, monitorati durante l'esecuzione di un *loop* principale. Prima di descrivere le funzionalità base del programma couatl, introduciamo alcuni concetti sulle operazioni e gli strumenti che andremo ad usare: il *fork* e le *pipe*.

4.1 Fork

Il processo che esegue la *fork* è definito *processo genitore* mentre il nuovo processo è noto come *processo figlio*.
Una caratteristica importante dell'operazione `fork` è che i file aperti nel processo genitore prima del `fork` sono condivisi dal processo figlio. Viene quindi fornito un semplice mezzo per passare specifici file aperti dal processo genitore al figlio.
L'impiego piú comune del `fork` si ha nello sviluppo di applicazioni *client-server*, oppure quando si vuole eseguire un nuovo programma.
Prendiamo in considerazione solo il secondo caso, che è quello a cui facciamo ricorso nella nostra applicazione. In questo caso il processo originale (genitore) effettua una copia di sé stesso con la chiamata *fork*. Poi una delle due copie, generalmente il processo figlio, esegue un *exec* per eseguire il nuovo programma. Questa è la tipica situazione usata per i programmi come le shell.

4.2 Concetti base sulle pipe

Una *pipe* è uno strumento che fornisce un flusso unidirezionale di dati ed è creata dalla chiamata di sistema *pipe()*. Questa chiamata restituisce due descrittori di file: uno aperto in lettura, l'altro in scrittura.
Le *pipe* sono generalmente utilizzate per comunicare tra due diversi processi in esecuzione sulla **stessa macchina**. Il modo con cui questo avviene è in

generale il seguente: un processo crea una *pipe* e successivamente esegue un *fork* con cui crea una copia di sé stesso. Subito dopo il processo genitore chiude l'estremità di lettura della pipe mentre il processo figlio ne chiude l'estremità in scrittura. Il risultato è un flusso di dati unidirezionale fra i due processi.
Se si desidera un flusso di dati bidirezionale, è necessario creare due *pipe* distinte ed utilizzarne una per ciascuna direzione.

4.3 Descrizione di couatl

L'applicazione couatl funziona utilizzando i concetti esposti nei paragrafi precedenti.
Come prima operazione apre un *socket* locale di ascolto. Quando ha luogo la connessione con l'applicazione balor, couatl procede a creare due *pipe* distinte e successivamente avvia un processo separato attraverso l'uso di `vfork`: il processo figlio così creato eredita le due *pipe*.

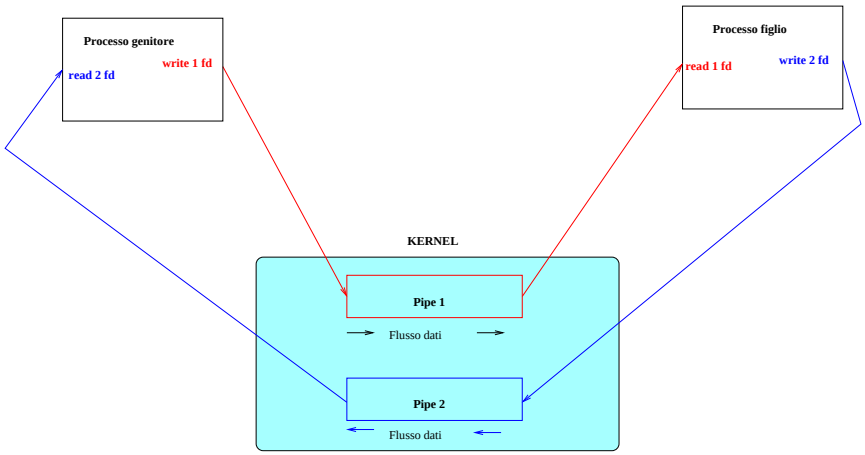

Figura 4.1: *Impiego di due pipe per ottenere un flusso bidirezionale.*

Per ottenere due canali di comunicazione che garantiscano un flusso bidirezionale (vedi fig.4.1), il processo padre couatl chiude l'estremità di lettura della prima pipe e quella in scrittura della seconda, mentre il processo figlio esegue l'operazione opposta: chiude l'estremità in scrittura della prima e quella in lettura della seconda.

4.4 Il processo figlio: la shell `wish`

Una volta eseguita la chiusura delle estremità delle due pipe, il processo generato usa la funzione *dup2* per connettere il proprio *standard input* e *standard output* rispettivamente alle *pipe* di scrittura e di lettura.

In questo modo tutto ciò che il processo genitore scrive sulla *pipe* di scrittura verrà mostrato sullo *standard input* del processo figlio e viceversa, tutto ciò che il processo figlio scriverà sullo *standard output* sarà accessibile al processo padre attraverso la *pipe* di lettura.
Il processo figlio esegue una chiamata `execlp` specificando come argomento il comando `wish` che vogliamo mandare in esecuzione. Se la chiamata ha successo il processo figlio, detto in questo caso *processo chiamante*, viene sostituito col **nuovo programma** `wish`.
L'invio da parte di un processo esistente di una chiamata di sistema `exec` o di una sua versione distinta (*execlp*, *execl*,... etc) costituisce l'unico modo in cui un programma puó essere eseguito da un sistema `Unix`. Spesso in casi come questo si parla impropriamente di *nuovo processo*, ma il programma che viene eseguito da `exec`, cioè la *shell* `wish`, è *un nuovo programma* eseguito nel contesto del processo chiamante.

4.5 Il processo padre: il *server* couatl

Il funzionamento del processo padre couatl, è molto semplice: dopo aver chiesto alla shell `wish` di caricare il programma gianoMotors, entra in un loop di lettura sotto il controllo della chiamata di sistema *poll*.
I due descrittori associati alle *pipe* di comunicazione, come pure il *socket* di connessione con l'applicazione balor, sono registrati e esaminati continuamente in polling.
Quando viene rilevata una qualche attività di lettura sulla *pipe* corrispondente, il programma legge quanto inviato sullo *standard output* dal programma gianoMotors.
Quando invece couatl riceve sul *socket* di comando una richiesta di movimento, questa viene tradotta nel corrispondente comando `Tcl/Tk` come stringa ASCII e inviata alla *shell* `wish` sulla *pipe* di uscita se risulta pronta per portare a termine l'operazione.

4.6 Operazioni di lettura da *pipe*

L'operazione di lettura da una *pipe* risulta essere piú "delicata" rispetto a quella portata a termine su un *socket*: non si puó infatti specificare la lettura di un numero di bytes superiore a quelli effettivamente presenti nella *pipe*, pena il fallimento dell'operazione in corso[1].
Tra i programmi couatl e gianoMotors non sussite un protocollo di comunicazione simile a quello usato con i *socket* di `tipo comando` che ci consenta di

[1]La chiamata di sistema *read* o *recv* fallisce restituendo `-1` con un *errno* uguale a ENOTSOCK: questo codice di errore implica un tentativo illegale di eseguire un'operazione specifica di un descrittore di *socket*, effettuata su un descrittore non corrispondente a un *socket*.

sapere, *prima* dell'operazione di lettura, quanti bytes dobbiamo effettivamente leggere.
Per ovviare al problema a cui abbiamo accennato sopra, ogni operazione di lettura da *pipe* è preceduta da una chiamata *ioctl* sulla *pipe* in questione. Questa chiamata di sistema restituisce il numero di bytes disponibili ad essere letti.

Capitolo 5

Il programma gianoMotors

Abbiamo eseguito alcune modifiche minori al codice del programma gianoMotors per cercare di raggiungere una buona interazione tra questo programma e l'applicazione couatl.
Abbiamo infatti che i messaggi inviati da gianoMotors sul proprio *standard output*, costituiscono di fatto il protocollo di comunicazione tra le due applicazioni.
Quando gianoMotors è usato come *demone*, le risposte inviate da questa applicazione al proprio *standard output* sono ricevute sulla *pipe* di lettura da couatl e forniscono a quest'ultimo l'*unico* modo per ricavare informazioni sul sistema e inviarle, per mezzo di balor, al programma di controllo dello strumento.
In alcuni casi abbiamo aggiunto delle stampe oltre a quelle prodotte di default da gianoMotors in modo da ottenere maggiori dettagli sullo stato del sistema oppure sulle posizioni dei motori.
In altri, invece, abbiamo scritto piccole procedure `Tcl/Tk` che stampano sullo *standard output* la posizione delle ottiche o che consentono la la selezione delle divers e configurazioni. Per mandare in esecuzione questo codice è sufficiente inviare alla *shell* `wish` una stringa ASCII con il nome della procedura.
Le funzioni che abbiamo aggiunto vengono chiamate solo quando gianoMotors è usato come *demone* di controllo dei motori e mai quando eseguito in modalità laboratorio.
Come accennato sopra, l'interazione con la GUI di gianoMotors quando eseguito in modalità demone, è possibile solo attraverso couatl.
La pressione di un tasto oppure la selezione di un *item* da una lista sono eventi naturalmente generati dall'intervento dell'operatore sulla GUI del programma per mezzo della tastiera o il mouse quando gianoMotors lavora in modalità normale, ma possono essere "simulati", scrivendo sulla *pipe* di uscita, o i nomi di procedure scritte *ad hoc* oppure specifici comandi `Tcl/Tk` quando gianoMotors agisce da *demone*.
Ad esempio, se vogliamo mandare in esecuzione la procedura attivata dalla pressione del tasto *b1*, couatl deve scrivere sulla *pipe* di uscita la stringa

ASCII "b1 invoke" che chiama la *callback* registrata in corrispondenza di quello specifico evento.
Questo è quanto fa couatl per mandare in esecuzione la procedura iniziale di *startup*: invoca con il metodo precedente la *callback* associata al tasto STARTUP della GUI di gianoMotors.
Per finire, ci preme comunque sottolineare che gli interventi che abbiamo effettuato sul codice `Tcl/Tk`, non hanno modificato in alcun modo il comportamento del programma quando eseguito in modalità *stand-alone* come programma di laboratorio: riteniamo infatti fondamentale che l'applicazione gianoMotors continui a mantenere lo stesso comportamento di prima.

Indice

Bibliografia

[1] "**Progetto Giano**. Inventario e memorandum del software di controllo", E.Rossetti, 2010, Memo.

[2] "Using Tcl and Tk from Your C Programs", M. Welsh, 1995, Linux Journal.

[3] "**Progetto Giano**. Protocollo di comunicazione tra GBridge e Giano GUI", C.Baffa, E.Giani, 2006, Memo.

[4] "**Progetto Giano**. Gbridge: il software middle-ware di Giano", E.Giani, C.Baffa, 2010, Arcetri Technical Report 7/2010

[5] "Dungeons & Dragons - Monster Manual 3.5", 2003, J. Tweet, M. Cook e S. Williams, TSR.