Sistemi Distribuiti e Intelligenza Artificiale Distribuita

Cristian Mercadante
UNIMORE | ULTIMO AGGIORNAMENTO: 18/12/2019

Lezione 1-2

Mercoledì 18 settembre 2019

Introduzione

Abbiamo un processo che elabora degli input e deve produrre un insieme di output. Cosa succede quando abbiamo un insieme di processi che elaborano streaming di input diversi, ma devono produrre un output coerente? Il risultato deve tenere contro dei dati di tutti i vari processi. Per fare questo necessariamente i vari processi dovranno coordinarsi. Cercheremo di capire cosa succede quando i processi non eseguono solo algoritmi, ma sono entità finalizzate a un obiettivo, nell'ambito dell'ambiente in cui sono eseguiti. Quindi entità che elaborano input, per elaborare lo stato delle cose nell'ambiente in cui si trovano (fase di percezione) e sulla base di un obiettivo, cerco di ragionare (fase di ragionamento): dato l'obiettivo e ciò che percepisco, devo decidere cosa fare per cambiare lo stato delle cose nell'ambiente, per ottenere l'obiettivo prefissato. Segue quindi un'azione (fase di azione). Questa è la definizione di intelligenza: un'entità computazionale che percepisce lo stato delle cose nell'ambiente, ed è capace di ragionare per capire quali sono le azioni da compiere per raggiungere l'obiettivo prefissato. È la capacità di agire in modo finalizzato nell'ambiente. A u certo punto, ci poniamo lo stesso problema di prima: se abbiamo un insieme di entità di questo tipo, che hanno percezioni diverse dell'ambiente, in che modo i ragionamenti e le azioni conseguenti potranno lavorare assieme per un obiettivo? Può anche darsi che ogni entità abbia obiettivi diversi: quindi mi chiedo come queste entità interagiscano/competano per raggiungere i propri obiettivi. Di nuovo avremo necessità di coordinare le loro attività a tutti i livelli: a livello di percezione, perché la visione del mondo che collettivamente raggiungano sia coerente, e che quindi possano ragionare e compiere azioni.

Una persona che gioca a pingpong ha l'obiettivo di mandare la palla dall'altra parte, mettendo in difficoltà l'avversario. L'input è il campo visivo. La percezione è un'elaborazione del pixel che vede, per capire la velocità e direzione della palla. Deve ragionare come muoversi, se colpire di dritto o di rovescio, ecc. Deve decidere la migliore azione per raggiungere l'obiettivo, per poi eseguirla. Ovviamente ognuna di queste fasi può fallire: percezione sbagliata, azione non eseguibile, ecc.

Introduzione ai Sistemi Distribuiti

Definizioni

Un'**architettura distribuita** è un insieme di processori che non condividono memoria, clock e spazio. Ogni processore ha la sua memoria centrale ad accesso suo esclusivo, eventualmente il processore ha sensori per la sua percezione, e non condivide il clock. Quindi non abbiamo variabili globali, nessun heap o stack su cui condividerne. Dovrà quindi esistere un sistema di comunicazione che permetta di scambiare dati. Il fatto che non ci sia un clock condiviso vuol dire che le macchine hanno il proprio concetto di tempo: non esiste il concetto di tempo condiviso, di simultaneità. Quindi se vogliamo agire orchestrando, non c'è modo di sincronizzarsi, se non comunicando, interagendo. Si può parlare di un'architettura distribuita quando queste entità hanno a disposizione un canale per permettere ai processi di interagire.

Un'architettura diventa un **sistema distribuito** quando tutto il software di gestione del sistema abilita l'interazione fra processi distribuiti in modo tale che la computazione distribuita possa comportarsi in modo coerente come un tutt'uno, come un singolo elemento coerente, secondo quella che è l'ambizione di ottenere trasparenza rispetto alla distribuzione: i processi riescono ad interagire come se non fossero distribuiti, non vedo più la distribuzione. Il sistema distribuito è quindi tutto il sistema operativo di rete che mi abilita l'esecuzione di computazioni distribuite.

Voglio poter risolvere un problema computazionale in termini di entità computazionali (chiamiamoli processi distribuiti) dove ogni processo ha solo una conoscenza parziale di tutti i parametri coinvolti nella **computazione distribuita** (ad es. una porzione di un DB) e/o ha solo una capacità parziale di agire per risolvere il problema (ad es. un robot che deve coordinarsi con altri per poter compiere azioni).

Una computazione distribuita si basa su un **algoritmo distribuito**. Un algoritmo distribuito implica definire l'algoritmo di computazione di ogni processo, ovvero bisogna decidere chi fa cosa. A questo si aggiunge il coordinamento, cioè con quali messaggi, con quali atti di sincronizzazione questi algoritmi dovranno eseguire.

Un'**IA distribuita** è una computazione che ha luogo fra entità autonome, volta a raggiungere obiettivi comuni o individuali, attraverso modelli di azione coordinata, meccanismi di decisione distribuita e percezione distribuita.

Motivazioni e proprietà dei Sistemi Distribuiti

Un sistema centralizzato è un collo di bottiglia: quando la computazione diventa pesante, un sistema centralizzato non sta al passo e quindi bisogna distribuire le attività computazionali e di processamento dell'input. Un sistema centralizzato è un single-point-

of-failure: se distribuiamo, se anche un nodo viene attaccato o fallisce, il resto della computazione può andare a buon fine. Ci sono alcuni problemi che sono intrinsecamente distribuiti: devo distribuire processi per forza, ad esempio nell'ambito IoT, dove devo distribuire sensori per raccogliere i vari input. Lo svantaggio è che è più complicato gestire, programmare e debuggare un sistema distribuito. Ma i pro riguardano la scalabilità, le prestazioni, l'affidabilità e la capacità di rispondere a problemi che altrimenti non potrebbero essere risolti. I sistemi distribuiti vogliono portare **accesso a risorse** (ad es. nella rete), **trasparenza** (il più possibile totale) nell'accesso, nella locazione, nella migrazione, nella rappresentazione dei dati, ecc. In altri casi però potremmo non voler avere trasparenza, quindi a livello di programmazione algoritmica vogliamo avere trasparenza. Ad esempio, se ho dei sensori nelle varie stanze, voglio poter osservare la distribuzione delle temperature registrata da un determinato sensore. Ovviamente ci sono spesso casi ibridi, in cui voglio avere trasparenza, ma non ovunque. Una cosa molto ricercata in un sistema è la **capacità di apertura**, ovvero la capacità di includere componenti eterogenei, nei vari livelli. Infine, uno dei requisiti più importanti è la **scalabilità**, sulla base delle dimensioni della rete, della complessità della computazione, rispetto ai tempi della comunicazione (voglio evitare che la computazione si blocchi perché la latenza è troppo alta). Si ottiene a livello di sistema operativo di reti, attraverso l'uso di meccanismi di replicazione dei dati, caching. A livello di computazione distribuita la scalabilità si ottiene attraverso lo sviluppo di algoritmi distribuiti scalabili.

Tipologie di Sistemi Distribuiti

Le architetture distribuite sono sistemi basati su nodi computazionali connessi da una rete di telecomunicazioni. Queste possono essere basate su soluzioni tecnologiche di vario tipo. Reti locali connettono totalmente i vari nodi: i costi di comunicazione sono molto bassi, perché avvengono in un unico passo, senza routing; è una rete tollerante ai guasti, perché esistono sempre percorsi alternativi. Una rete di questo tipo però non scalabile, perché ogni nodo ha un numero limitato di schede di rete. Si va allora verso reti mesh che hanno connessione parziale: sono previsti percorsi alternativi solo a volte; è scalabile; ha costi di comunicazione bassi; a seconda del livello di magliatura, l'affidabilità è più o meno alta. Ci sono reti gerarchiche, reti basate su modello a stella, reti connesse a ring.

Se noi vogliamo fregarcene dell'hardware, a livello software è importante capire che nel momento in cui la computazione distribuita è basata sull'insieme dei processi, allora le proprietà della topologia possono influenzare le prestazioni della computazione distribuita. C'è una rete fisica, ma anche una rete di processi, e la forma di quella rete determina come viene eseguita la computazione (efficienza, potenzialità, ecc.). Per questo motivo parleremo di reti sociali, perché vogliamo capire in che modo la forma di una rete ha certe proprietà che hanno effetto sulla computazione e sul *decision making* distribuito.

Quando parliamo di sistemi operativi di rete, tipicamente distinguiamo fra due classi distinte: i **network operating systems** e i **distributed operating systems**. Tutti i sistemi operativi sono NOS e sono OS che mettono a disposizione meccanismi per abilitare la comunicazione distribuita fra processi. Oltre ad avere le funzionalità degli OS, hanno anche la parte di gestione delle interfacce di rete, permettendo di aprire connessioni TCP e UDP. Un NOS diventa un DOS quando non solo abilita la comunicazione, ma permette anche di organizzare la gestione delle risorse a livello di architettura distribuita. In realtà oggi i sistemi operativi non sono DOS, ma NOS sui quali è possibile aggiungere uno strato per realizzare sistemi distribuiti. Si parla quindi di **middleware**. Abbiamo hardware, su cui abbiamo osware (che abilita la connessione, ma non fa una gestione distribuita). Il middleware è un qualcosa che sfrutta le capacità di rete degli OS al fine di realizzare sistemi distribuiti. Noi quando programmiamo usiamo le astrazioni di file system, stream di input, ecc. Il middleware sfrutta queste astrazioni per realizzarne altre di più alto livello. Nel panorama enterprise, il middleware facilita la comunicazione e monitora le richieste degli utenti, controlla la consistenza dei dati, redistribuisce le richieste, evita i deadlock, ecc.

Paradigmi architetturali

Quando si adotta un sistema di MW che supporta applicazioni distribuite, a seconda del MW ci sarà un supporto a diversi paradigmi di computazione distribuita. Ci sono MW che supportano modelli di computazione distribuita basato su interazioni client-server (come il MW di Java che utilizza il paradigma ad oggetti), e (più generalmente) paradigmi di computazione distribuita basata su *message passing*. Il primo modello prevede che ci siano una richiesta e una risposta fra due nodi; il secondo modello prevede un messaggio sul quale costruire protocolli di comunicazione fra processi. I modelli basati sul paradigma client-server supportano architetture orientate agli oggetti e invitano allo sviluppo di modelli a strati. Al contrario, modelli basati su conversazioni fra processi, in cui non c'è più una chiara distinzione fra client e server, ma abbiamo peer, supportano diverse architetture: ad eventi, centralizzate sui dati, P2P e multi-agente.

I modelli client-server sono dominanti oggi. Si definisce chiaramente una serie di processi che devono agire come server, che devono stare passivi, sempre pronti ad ascoltare le richieste dei clienti, che invece proattivamente contattano il server, gli fanno la richiesta ed aspettano la risposta. Questo modello fa in modo che il server possa incapsulare i suoi servizi, in modo trasparente (senza svelare come fa il servizio). Il cliente deve solo sapere quale servizio invocare. A livello di MW deve essere messo a disposizione un servizio col nome di "invocazione di procedura remota", che oggi viene chiamata "invocazione remota di metodo" (per

la diffusione di linguaggi ad oggetti). Nella comunicazione fra client e server c'è anche la sincronizzazione: tipicamente il client blocca la sua attività nell'attesa di una risposta dal server.

Nelle applicazioni object-based, l'architettura che ne risulta è una rete di oggetti, dove ogni oggetto può agire sia da client che da server, nel momento in cui qualcuno chiede a lui l'invocazione dei servizi. Ogni oggetto può invocare metodi di qualsiasi altro oggetto, basta che ne abbia il riferimento. I riferimenti possono essere passati come parametri, quindi si può arrivare a far conoscere tutti i metodi di tutti gli oggetti. Ogni oggetto può agire quindi sia da client che da server. Il risultato di questo è un'applicazione in cui la topologia delle interazioni fra oggetti può diventare caotica: non c'è una disciplina dell'ingegnerizzazione delle interazioni e degli schemi di interazione. Perciò nelle applicazioni client-server object-based fatte bene si tende a dare agli oggetti una disciplina relativamente a chi può invocare chi, quale client può invocare quale server, secondo un'impostazione layered. Il layer 1 è solo server, ovvero risponde a richieste, senza necessità di invocare altre richieste. Il layer 2 può invocare solo servizi del layer 1 e agisce da server solo per oggetti del layer 3. E così via. In questo modo lo stack di attivazioni è al massimo lungo N (come il numero di layer). Il modello base di oggi è il modello **3-Tier**. Il livello dei dati è costituito da servizi per accedere ai dati. Il livello dei servizi (business layer) è quello dove c'è la computazione vera e propria: per realizzare gli algoritmi, i nodi fungono da client nei confronti del livello dati. Infine, il livello di presentazione stabilisce le interfacce utente, che scatenano l'invocazione di servizi del livello logico. Con questi livelli, il flusso è molto disciplinato.

Spesso ci sono servizi che per essere realizzati chiamano altri servizi, quindi interazioni fra oggetti del business layer. Pertanto, si può inserire un quarto layer. Sopra i dati c'è il livello dei microservizi, che fanno solo una cosa, sono servizi di base. Al di sopra c'è il livello dell'orchestrazione fra servizi, dove ci sono servizi che sono creati sulla base della composizione di microservizi. Il flusso delle comunicazioni è così mantenuto verticale, mai orizzontale.

La distribuzione può essere sia orizzontale, dove metto i vari microservizi in diverse macchine geograficamente. Può essere anche verticale, perché potrei decidere di mettere i microservizi vicino sulla stessa macchina dove risiedono i dati; oppure potrei avvicinare l'orchestrazione nel browser, quindi nella presentazione (ad es. AJAX).

Sempre di più si va verso una direzione in cui non ci sono componenti che agiscono solo da client o solo da server, ma che possono trovarsi nell'esigenza di servizi o in situazioni in cui erogano servizi. Si parla di modelli **peer-to-peer**, in cui non c'è una divisione dei ruoli. Bisogna focalizzarsi sull'ingegnerizzazione dei protocolli fra i componenti: quali messaggi si possono mandare, quali ricevere e qual è la sequenza di messaggi che i componenti si devono scambiare per realizzare le funzionalità del sistema. Questi messaggi possono avere varie forme: nel client-server, l'unica forma era richiesta-risposta, mentre nel P2P possono scegliere il meccanismo di interazione. Un modello di questo tipo è il modello ad eventi, dove i componenti annunciano al mondo quello che sta succedendo e decidono di ascoltare certi tipi di eventi piuttosto che altri. Un altro modo è quello di lasciare dei segni nello spazio in cui gli oggetti agiscono: c'è una sorta di spazio dati virtuale che i processi possono modificare e vedere per capire come agire di conseguenza. Poi c'è il modello multi-agente, dove il problema è quello di coordinarsi per far sì che gli oggetti possano percepire cosa succede nell'ambiente per raggiungere un determinato obiettivo (come da introduzione).

Network programming in Java (1)

Tutti i sistemi operativi permettono di connettersi ad internet tramite primitive per avviare connessioni UDP e TCP. In Java c'è la libreria *java.net* che mette a disposizione classi da manipolare per far interagire processi. La **socket** è l'astrazione di base che permette di accedere ai servizi di base per accedere a Internet. La socket è la porta, il gate, verso la rete, che viene messo a disposizione di un processo. Diventa un canale su cui possiamo inviare dati verso altri nodi della rete, e su cui poterne ricevere. Ogni OS mette a disposizione un certo numero di porte, tipicamente 65536. La socket ha un certo numero di porta. Questo permette a un processo di essere univocamente identificabile in internet: indirizzo IP e numero porta. La socket non è altro che uno stream (come *istream* del C++).

La comunicazione fra due entità è dal punto di vista logico sempre client-server: quando parliamo con qualcuno e questo ci ascolta, lui ha un orecchio che agisce da server che ascolta, mentre io parlo e fungo da client. Se un processo vuole ascoltare dei messaggi, deve avere un thread fermo ad aspettare. Viceversa, se vuole parlare deve sapere chi ascolta. Tipicamente questo avviene con un'opportuna gestione dei thread da parte del server, che scatena un thread per rispondere, mentre continua ad ascoltare. Quindi ogni nodo deve avere un thread pronto a ricevere messaggi e un processo principale che parla.

Ci sono due tipi di socket: **Socket Stream** (basate su TCP) e **Socket Datagram** (basate su UDP). Usiamo SS quando ci interessa la proprietà FIFO e la sicurezza nella consegna. Assumiamo che ci sia un processo (server) che decide di mettersi in ascolto. Significa creare un oggetto della classe SS. Questo significa connettersi a una porta e mettersi ad ascoltare chi vuol parlar con me. Da un'altra parte, un processo (client) che vuole comunicare, crea un oggetto della classe **Socket**, richiedendo di connettersi al processo che ha aperto la SS. Questo atto scatena il *three-way-handshake*, che nel TCP stabilisce il canale di comunicazione e determina i parametri di questo canale. Una volta effettuata la connessione, i processi hanno a disposizione di un input stream e un output stream ciascuno. Attenzione: se tento di leggere qualcosa che non c'è, oppure se entrambi i processi scrivono in output ed

attendono in input, abbiamo un deadlock. Di fatto, quando un SS riceve una richiesta, istanzia una Socket (come quella del client). La SD invece permette di inviare datagrammi non collegati fra di loro.

Lezione 3

Giovedì 19 settembre 2019

Network programming in Java (2)

In Java, quando definisco una socket devo dare due parametri: l'indirizzo del nodo o il suo nome simbolico, e il numero di porta. Quando riusciamo a connetterci, l'oggetto così creato automaticamente determina la creazione automatica di un input stream e un output stream. Possiamo quindi recuperarne i riferimenti con i rispettivi getter. Quando non serve più la connessione, la chiudiamo con il metodo *close*. Per gestire gli indirizzi IP in Java esiste la classe *InetAddress*, che ha una serie di metodi di classe per manipolare gli indirizzi IP, anche a livello simbolico.

Vediamo adesso alcuni esempi semplici per connettersi a servizi standard di internet. Tipicamente, un computer in rete, al boot del sistema operativo vengono messi a disposizione una serie di server che ascoltano. Tipicamente abbiamo il servizio/server di *ping* (per dire che sono vivo), *echo* (che riceve una linea di testo da parte di chi si è connesso e la rimanda indietro), *daytime* (mi connetto a un server remoto e mi aspetto che lui mi invii la data e l'ora corrente), *telnet*, *smtp*, *http*. Vogliamo connetterci a un server *daytime*: noi faremo un client per ricevere la data, sulla porta 13.

Quando creo una socket, potrei avere un'eccezione, ad esempio quando la connessione non c'è. Quindi devo gestire le opportune eccezioni. Anche quando leggo e scrivo sugli stream. Dall'altra parte c'è un processo che è stato creato e per prima cosa ha aperto una ServerSocket sulla porta 13. Quindi il SS fa un ciclo infinito per ascoltare. L'ascolto non significa accettare automaticamente una connessione. Il metodo *accept* è un metodo bloccante, finché da qualche parte qualcuno non si connetterà con noi. In quel momento, la comunicazione non avviene attraverso la SS, ma viene creato un altro oggetto Socket.

La conversazione presuppone che sia client che server conoscano il protocollo di comunicazione, ovvero sapere quali frasi bisogna dire e in che ordine. Se i processi non rispettano il protocollo, ad esempio, se il client tenta di inviare una stringa, il server non la leggerà mai, perché questo vuole scrivere, ma il client continua a scrivere. Il risultato: deadlock.

La classe ServerSocket riceve un numero di porta, ma posso anche specificare il numero massimo di richieste che posso mettere in coda (lunghezza della coda): infatti, se il server è impegnato a rispondere, una richiesta in ingresso viene messa in coda. Nel momento in cui una SS accetta una richiesta di connessione, è possibile vedere con chi ci si è connessi, per poi decidere se rifiutare o accettare la connessione.

Questi meccanismi sono abbastanza di basso livello, perché non coinvolgono middleware. Pensando ai tre livelli logici, il livello DB avrà delle SS per l'accesso a DB. Il business layer agirà da client e quindi avrà una Socket aperta, ma anche una SS per le richieste di connessione dal livello presentazione.

Nel caso più generale di una rete in cui tutti possono interagire con altri, ogni processo avrà la sua SS aperta e in attesa di connessioni per ricevere messaggi da chi li vuole inviare, e aprirà una Socket tutte le volte che vorrà inviare un messaggio a un processo.

Gli oggetti Java sono serializzabili e inviabili su uno stream, tramite le classi *ObjectInputStream* e *ObjectOutputStream*.

Il problema nella programmazione distribuita è che non posso sapere quando le cose avverranno. Il server si mette in attesa, ma non sa quando risponderanno: non c'è un clock condiviso e non so quando il client vorrà parlarmi, quindi non ha il controllo sul quando si sbloccherà. La richiesta delle connessioni deve essere messa in un thread separato, così non si blocca il processo principale. Nella programmazione di rete quindi le operazioni che rischiano di essere bloccanti devono essere messe su thread separati.

Le Socket utilizzate finora sono TCP, quindi affidabili, su conversazione. Ogni messaggio inviato da un processo all'altro prevedere un ACK e quindi se dopo un certo time-out il processo che ha inviato il messaggio non riceve l'ACK, allora rinvia il messaggio. Inoltre, c'è la garanzia di delivery FIFO, perché i messaggi hanno numero di sequenza. Se arrivano fuori sequenza, allora li bufferizza. Per quanto riguarda il protocollo di comunicazione, possiamo stabilire che ogni connessione consista in un unico messaggio da processo P1 a P2 (modello per processi peer). Se c'è bisogno di molteplici messaggi, ogni messaggio ha comunque una sua connessione. Se dopo P2 vuole rispondere a P1, allora la sua Socket contatterà la SS di P1, invierà il messaggio e chiuderà la connessione. L'altro modo consiste nel basarsi nell'interazione richiesta-risposta: per ogni richiesta è prevista una risposta e poi la connessione si chiude. Poi è possibile inventarsi un protocollo a piacere: ad esempio, parlo fino a che non arriva la parola "stop". L'importante è che entrambe le parti siano consce del protocollo, altrimenti si rischia il deadlock.

L'altra possibilità è usare il protocollo UDP, dove non esiste la connessione. I processi P1 e P2 dovranno creare una propria DatagramSocket e impacchettare il messaggio che vogliono, specificando il destinatario e affidando, con un'operazione di *send*, il messaggio alla rete. P2, quando riceverà il messaggio nel suo buffer, può fare quello che vuole: può fregarsene oppure fare un'operazione di *receive*: prendo i datagrammi che mi sono arrivati. Quando creiamo una DS, se vogliamo spedire un messaggio, non è necessario specificare il numero di porta. Ma se vogliamo ricevere un messaggio, allora bisogna specificarlo. Il metodo *send*, non è bloccante: imbuco una lettera e faccio altro. La *receive* è invece bloccante: se non mi sono arrivati dei datagrammi, aspetto che arrivino.

La creazione del datagramma è invece un'operazione complicata. Bisogna comporre il messaggio in un *ByteArrayOutputStream*, che incapsuleremo in una classe di filtraggio, ad esempio in un *DataOutputStream*. Bisogna poi creare un *DatagramPacket* e specificare lunghezza, IP e numero di porta. Viceversa, quando riceviamo un DP, dobbiamo estrarre il payload: si fanno le operazioni inverse.

Lezione 4

Giovedì 26 settembre 2019

Stati consistenti globali di Sistemi Distribuiti (1)

Il problema che ci poniamo è quello di riuscire ad osservare e a capire cosa sta succedendo in una computazione distribuita, in cui una serie di processi, in posti diversi, ognuno con clock diviso, eseguono questo algoritmo interagendo gli uni con gli altri. Ci poniamo il problema di capire se una certa affermazione sia vera o no, se una certa situazione si sia verificata (sicuramente, plausibilmente, ecc.). Questo problema si chiama **Global Predicate Evaluation (GPE)**: capire se una certa affermazione booleana (predicato) che fa riferimento agli stati della computazione distribuita, ovvero al valore delle variabili che processi stanno elaborando, sia vera o falsa. Parliamo per esempio di deadlock: vogliamo capire se il sistema è andato in deadlock. Sia data una serie di processi, ognuno che esegue le sue computazioni: a un certo punto P1 e P2 si mandano messaggi a vicenda e aspettano una risposta l'uno dall'altro: si verifica un deadlock.

Molti sistemi distribuiti si basano sul passaggio di token, ovvero di una variabile che viene fatta circolare fra i processi. La computazione prevede che solo un processo abbia il token. Però in molti casi il token potrebbe essere duplicato o eliminato per sbaglio. Quindi vorremmo poter controllare il token. Invece, un algoritmo distribuito che gestisce transazioni bancarie deve garantire che la quantità di denaro globale sia sempre la stessa, e che il denaro non venga duplicato.

Assumiamo un sistema distribuito asincrono, dove esiste sicuramente una scala di tempo assoluta, ma i vari processi eseguono sulla base di un loro clock locale (che può avere degli slittamenti): non c'è modo di affermare con precisione e con certezza se due eventi stanno avvenendo nello stesso momento o di sapere la loro posizione temporale reciproca. L'unico modo per sincronizzare le attività dei processi è quello di inviare messaggi da un processo all'altro. Nel momento in cui P1 invia un messaggio a P2 e P2 lo riceve, solo in questo momento possiamo avere la certezza che questa operazione del processo P2 è avvenuta dopo l'operazione di P1 (la *receive* può avvenire solo dopo che P1 ha fatto la *send* e le operazioni precedenti). Quindi la comunicazione fra processi è l'unico modo per avere una relazione di prima-dopo fra i processi.

Approccio reattivo

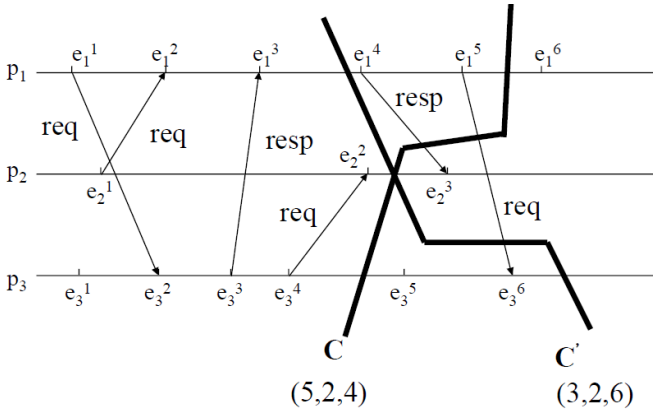
Ma chi osserva una computazione distribuita? Osservare significa essere in grado di capire che tipo di operazione stanno facendo i vari processi. L'**approccio reattivo** è quello in cui i vari processi inviano un messaggio a un processo P0 che agisce da osservatore. Questo si pone quindi a ricevere messaggi dagli altri processi ogni qual volta che questi fanno operazioni di rilievo. Ad esempio, quando un processo è a rischio deadlock, prima di fare una *send* (ad esempio), avvisa P0 dell'operazione che sta per fare. P0 potrebbe essere un processo esterno, ma anche un processo che agisce come tutti gli altri. L'**approccio proattivo** prevede invece che sia P0 a mandare un messaggio agli altri processi, per capire cosa stia succedendo negli altri processi.

Una computazione distribuita è realizzata da un insieme N di processi (P1 – Pn) collegati attraverso canali di comunicazione unidirezionali. Quindi assumiamo che ogni processo abbia un canale di output e un canale in input. In quest'ottica distinguiamo allora gli eventi che contraddistinguono l'attività di un processo in **eventi interni** (che un processo Pi fa per fatti suoi) e_i^j. Ci sono relazioni causali fra eventi: per un processo *i*, l'evento *k* è avvenuto prima dell'evento *l*, se *k<l* (*happened before*). Un processo può inviare e ricevere messaggi: la *receive* è in relazione *happened before* con la *send*. Due eventi non sono in relazione causale fra di loro se possono essere avvenuti concorrentemente. Non possiamo sapere l'ordine perché, se ci mettiamo nel punto di vista dell'osservatore, l'unico modo per osservare è aspettare messaggi dai processi. P0 riceve un messaggio da P1 che gli dice "ho eseguito l'evento 1". Poi riceve da P3 un "ho eseguito l'evento 1". E così via. Il problema è che in un sistema distribuito non c'è la garanzia che

l'esecuzione e la conseguente osservazione di un fenomeno avvenga in modo ordinato: potrebbe esserci un ritardo nella rete che ha fatto sì che l'ordine di osservazione non corrisponda all'ordine di esecuzione. È un fenomeno relativistico.

L'implicazione è che osservare lo stato globale di un sistema distribuito è difficile. Vuol dire fare una fotografia del sistema. Nel caso dei sistemi distribuiti diciamo di **osservare un taglio dell'osservazione**, ovvero osservare lo stato di ogni processo, sulla base dei messaggi che sono arrivati a P0 dai vari processi. Quindi, dati N processi, ognuno che esegue eventi, un taglio C definisce un punto della storia di un processo: se *h* sono le storie di un processo, un taglio include una storia *h* per ogni processo. Il problema è: vanno bene tutti i tagli? No, perché alcuni tagli potrebbero far riferimento ad azioni che nella realtà non hanno coerenza: ad esempio, un evento di *receive* che non ha la corrispondente *send*. Supponiamo che P3 abbia mandato tutti i messaggi di notifica a P0, compresa la *receive* di un messaggio da P1, ma che P1 non abbia ancora notificato a P0 di aver fatto quella *send*: se mi affidassi a questo per valutare cos'è successo nella computazione, potrei osservare dei predicati che non corrispondono alla computazione reale.

Assumiamo che P2 mandi un messaggio a P1 e che aspetti che P1 gli risponda. Se P1 non invia la risposta, allora P2 non procede. Potrei arrivare al deadlock se si crea un ciclo nelle serie di richieste e risposte. Se P1 invia a P2, che invia a P3, che invia a P1, allora ho 3 processi in attesa di risposta: deadlock. Supponiamo che l'osservatore osservi il taglio C'. Osserverà che P2 ha inviato una richiesta a P1, ma non gli ha ancora risposto. P3 ha inviato una richiesta a P2. Vedo poi che P3 ha ricevuto una richiesta da P1: osservo quindi un deadlock. In verità, questo blocco non c'è, perché se fossimo in grado di vedere ciò che è effettivamente avvenuto, vedremmo che l'invio da P1 a P3 è avvenuto dopo che P1 ha risposto a P2, quindi non c'è più il ciclo. Quindi si dice che quel taglio non è consistente, ovvero non corrisponde a ciò che è successo realmente. Il taglio C invece è consistente.



Il modo in cui P0 osserva la computazione corrisponde all'ordine in cui riceve i messaggi dai vari processi. Una singola computazione distribuita potrà avere diversi **run** possibili, a seconda del modo in cui gli arrivano i messaggi. Quindi formalmente definiamo un **taglio consistente** un taglio tale che, dati due eventi *e* ed *e'*, se *e* appartiene al taglio (cioè sta nella storia di un processo a sinistra di un taglio) e c'è un evento *e'* che è avvenuto prima di *e* nello stesso processo, allora anche *e'* appartiene al taglio.

$$\forall\, e \; and \; e', (e \in C) \wedge (e' \rightarrow e) \Longrightarrow e' \in C$$

Nel caso di due eventi appartenenti a processi diversi, non ho modo di stabilire qual è avvenuto prima, ma di sicuro non posso ossevare che *e* appartiene al taglio, ma non *e'*, se $e \rightarrow e'$. Un processo P0 come osservatore potrà, sulla base dei messaggi che gli arrivano dai vari processi, osservare in modo diverso la stessa computazione (diverse run). Osservando, si accerta che il taglio sia consistente prima di capire lo stato del sistema.

Una prima forma di incosistenza è la ricezione di messaggi fuori ordine dallo stesso processo. Questo chiaramente non può corrispondere a un run reale del processo in esame. Ma questo si risolve semplicemente, perché io sono in grado di capire se sono arrivati out-of-order, cioè posso mettere in ordine in messaggi, ovvero stabilire la regola di **FIFO delivery**. Questa mi garantisce che P0 non osservi mai questa situazione. Quindi posso ristabilire un'osservazione che corrisponde a un run reale per ogni singolo processo, ma ancora non necessariamente consistente, perché non posso fare questa cosa quando i messaggi sono fuori ordine fra processi diversi. Non c'è modo di stabilire che l'ordine di ricezione dei messaggi da parte dell'osservatore corrisponda all'ordine di esecuzione.

Com'è possibile osservare dei tagli consistenti? Osservare una computazione consistente sarebbe possibile nell'ipotesi (che non abbiamo) di avere un **real-clock (RC) globale condiviso**: se tutti i processi inviano un messaggio a P0 attaccandoci anche il clock globale, allora posso ricostruire l'ordine di esecuzione degli eventi nei vari processi. Però questo non toglie che un messaggio possa arrivare in ritardo e che P0 possa fare un'osservazione non consistente. Se però ho un limite di tempo superiore al tempo di consegna dei messaggi, allora abbiamo ottenuto che in presenza di un orologio universale, noi possiamo garantire un'osservazione consistente, ovvero $e \rightarrow e' \Rightarrow RC(e) < RC(e')$. Se a livello algoritmico costruisco un **orologio logico (LC)** che mi garantisce questa proprietà, allora riesco a fare un'osservazione consistente. L'orologio logico è un numero intero che ogni processo mantiene e che tiene traccia del suo ordine di esecuzione degli eventi. È un contatore banale. Mi garantisce che all'interno dello stesso processo valga questa proprietà e che P0 possa riordinare gli eventi (ovviamente). Fra processi diversi invece vale questa regola: il processo che manda un messaggio, invia anche il suo LC, mentre chi riceve fa l'update del proprio LC, aggiungendovi 1. P0 poi riceve le informazioni: guardando gli orologi logici dei messaggi ricevuti, può semplicemente mettere in