

Università degli Studi di Perugia

Corso di Laurea Triennale in Informatica



Corso:

SISTEMI DISTRIBUITI E PARALLELI

7 C.F.U.

Anno Accademico:

2022/2023

Docente:

SERGIO TASSO

✉ sergio.tasso@unipg.it

Programma

- ❑ INTRODUZIONE AI SISTEMI DISTRIBUITI
 - Definizione;
 - Vantaggi e svantaggi;
 - Tipologie;
- ❑ COMUNICAZIONI NEI SISTEMI DISTRIBUITI
 - Protocolli a livelli;
 - Modello Client/Server;
 - Chiamata di procedura remota (RPC);
 - Comunicazioni di gruppo;
 - Architettura HLA
- ❑ ELABORAZIONE NEI SISTEMI DISTRIBUITI
 - Processi e thread;
 - Package di thread;
 - Thread e RPC;
 - Java e Thread;
 - Modelli di sistema (workstation e pool di processori);
 - Allocazione dei processori;
 - Schedulazione nei sistemi aperti e distribuiti

Programma

- ❑ Introduzione ai Sistemi Paralleli
 - Parallelismo;
 - Architetture parallele (SISD, SIMD, MISD, MIMD, SPMD);
 - Memorie condivise e distribuite;
 - Memorie e Processori;
- ❑ Paradigmi della Programmazione parallela
 - Message Passing;
 - Data Parallel;
 - Implementazioni (PVM, MPL, MPI, HPF);
- ❑ Analisi Metodologica sulla Programmazione parallela
- ❑ Misura delle prestazioni dei programmi paralleli vs. seriali
 - Definizione di speedup;
 - Legge di Amdhal;
 - Algoritmi per assegnazione statica e dinamica dei task;
 - Altri aspetti della programmazione parallela
 - Esempi ed esercizi

Programma

□ FILE SYSTEM DISTRIBUITI

- Progettazione;
- Implementazione;
- Struttura;
- Caching;
- Replicazione;
- NFS;
- AFS;
- CODA;
- Nuovi FSD per cluster
 - HDFS
 - Ceph
 - GlusterFS
- Il cluster Docker

□ STANDARD CORBA

- Definizioni;
- componenti (ORB, BOA, POA, IDL, SII, DII, DSI) e applicazioni;

Programma

- ❑ COM e DCOM
 - Definizioni e applicazioni;
- ❑ JAVA RMI
 - Definizioni e applicazioni;
- ❑ WIRELESS COMPUTING
 - Tecnologie e configurazioni;
 - Client/Server in ambienti mobili;
- ❑ SERVIZI WEB
 - Definizioni
 - SOAP
 - WSDL
 - UDDI
- ❑ J2EE E I SERVIZI WEB
- ❑ Architetture SOA
- ❑ FORMATO JSON
- ❑ RESTful Web API
- ❑ FRAMEWORK LARAVEL PER COSTRUIRE APP in PHP
- ❑ REPOSITORY e CMS

Testi consigliati



- ❑ David A. Chappell, Tyler Jewell, *Java Web Services*, O'Reilly - HOPS
- ❑ Leonard Richardson, Mike Amundsen, *RESTful Web APIs*, O'Reilly
- ❑ Mike Amundsen, *RESTful Web Clients*, O'Reilly
- ❑ Matt Stauffer, *Laravel Up & Running*, O'Reilly
- ❑ *Materiale multimediale fornito dal Docente*

Modalità d'esame



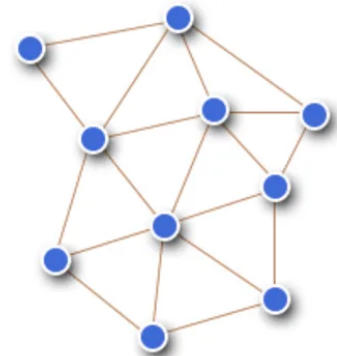
- ❑ Test individuale on-line
- ❑ Progetto di un'applicazione in ambiente distribuito
 - Max 2 studenti

Definizione

- Un **sistema distribuito** consiste in una collezione di calcolatori autonomi connessi via rete e dotati di particolari funzioni che permettono di coordinare in maniera controllata le loro attività e condividere le risorse del sistema.
- Il concetto di sistema distribuito è contrapposto al concetto di **sistema centralizzato** (formato da un unico calcolatore e dalle sue p



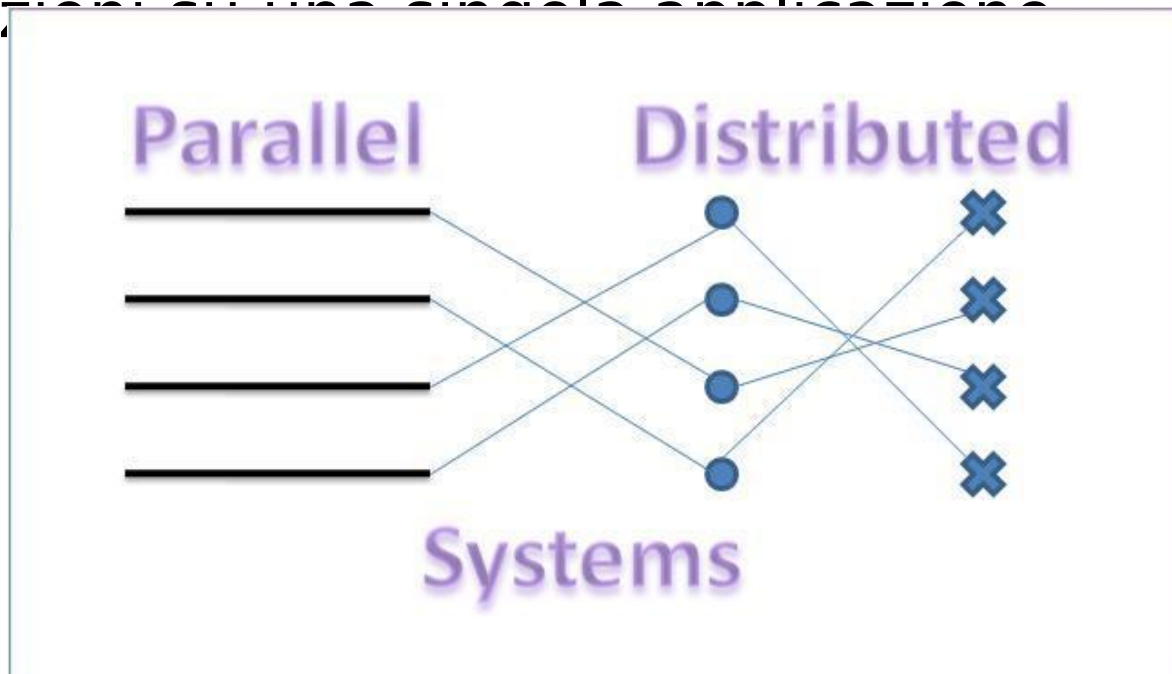
Centralized



Distributed

Definizione

- Una ulteriore distinzione che alcuni fanno è quella fra sistema distribuito che permette a molti utenti di lavorare insieme e **sistema parallelo** il cui obiettivo è quello di raggiungere il massimo incremento di prestazioni su una singola applicazione.



Vantaggi dei sistemi distribuiti rispetto a quelli centralizzati

❑ Economia

- I microprocessori offrono spesso un miglior rapporto prezzo/prestazioni rispetto ai mainframe

❑ Velocità

- Un sistema distribuito può avere una potenza di calcolo complessiva superiore a quella di un mainframe

❑ Distribuzione intrinseca

- Alcune applicazioni richiedono macchine separate e distanti

❑ Affidabilità

- Se una macchina cade, il sistema può nel complesso sopravvivere

❑ Crescita incrementale

- Si può aggiungere potenza di calcolo in passi incrementali

Vantaggi dei sistemi distribuiti rispetto a pc indipendenti

- ❑ Condivisione dei dati
 - Permette a molti utenti di accedere alla stessa base di dati
- ❑ Condivisione dei dispositivi
 - Permette a molti utenti di condividere periferiche dispendiose
- ❑ Comunicazioni
 - Rende la comunicazione fra gli utenti più semplice (email)
- ❑ Flessibilità
 - Distribuisce il carico sulle macchine disponibili in modo più efficiente
- ❑ Crescita incrementale
 - Si può aggiungere potenza di calcolo in passi incrementali

Svantaggi dei sistemi distribuiti



- ❑ Software
 - Software più complesso per i sistemi distribuiti
- ❑ Connessione di rete
 - La rete può saturarsi o causare altri problemi
- ❑ Sicurezza
 - La facilità di accesso vale anche per i dati riservati

Concetto di risorsa



- ❑ La **risorsa** è un qualsiasi elemento che può essere acceduto per un qualsivoglia utilizzo.
- ❑ Si possono distinguere risorse di tipo:
 - **dati** (informazioni)
 - **software** (programmi)
 - **hardware** (dischi, stampanti)
- ❑ Una risorsa si dice **condivisa** quando può essere acceduta da più utilizzatori (**processi**).
- ❑ In base al tipo di risorse esistono diversi metodi per la gestione della loro condivisione.

Risorse condivise: dati

- ❑ Per risorse di tipo dati, tipicamente organizzate in file, l'utilizzo consiste in operazioni di lettura/scrittura.
- ❑ Eventuali concorrenze vengono gestite tramite **indicatori** (semafori o lock o flag o modalità).
- ❑ Alla risorsa(file) viene associato un indicatore che informa eventuali processi concorrenti sullo stato della risorsa, in particolare la risorsa dati può in stato di:
 - **divieto di lettura e scrittura**
 - **divieto di sola scrittura**
 - **divieto di sola lettura**
 - **nessun divieto**
- ❑ Tramite questi indicatori è possibile condividere tra più processi lo stesso dato

Risorse condivise: dati



- ❑ Il divieto di lettura/scrittura sul dato nega l'accesso simultaneo al dato da parte di più processo. In questo modo, il primo processo che apre il file ne resta l'utente esclusivo finché non cessa l'elaborazione.
- ❑ Il divieto di scrittura permette ad altri programmi che aprono lo stesso file di accedervi in sola lettura, mentre il primo che vi accede può anche modificarne il contenuto.
- ❑ Il divieto in sola lettura ha senso solo nel caso in cui processi concorrenti devono essere fatti in modo tale che il risultato di uno (scrittura) sia input per l'altro (lettura) e viceversa (attenzione però a situazioni di stallo!).
- ❑ Esistono poi casi in cui occorre concedere a diversi processi di condividere lo stesso file sia in scrittura che in lettura.

Risorse condivise: software



- ❑ Risorse di tipo software sono i programmi che possono essere condivisi in fase di esecuzione.
- ❑ Un unico programma (utente o di sistema) può essere eseguito da più utenze.
- ❑ In genere l'esecuzione multipla non crea problemi, poiché nuove istanze dello stesso programma lavorano su variabili locali nuove.
- ❑ Se il programma fa uso di dati non locali (globali o assoluti) occorre far uso della condivisione dei dati.

Risorse condivise: hardware



- ❑ Le tecniche di gestione di risorse hardware condivise dipendono strettamente dalla risorsa da gestire.
- ❑ Per stampanti, plotter e periferiche di solo output la tecnica di gestione più diffusa è lo spooling: si creano code di output locali, gestite tramite politiche di tipo FIFO (First In First Out, cioè il primo che si alloca la risorsa è il primo ad usarla) o tramite priorità (ad ogni processo viene associata una priorità, la risorsa viene data subito al processo con priorità più alta).
- ❑ Per dischi ed altre unità di I/O oltre ad un meccanismo di spooling occorre prevedere una protezione sui file con tecniche analoghe a quelle per la condivisione dei dati.

Risorse condivise: hardware



- ❑ La risorsa hardware più importante è sicuramente il processore (o CPU o unità centrale di elaborazione).
- ❑ La tecnica di gestione della CPU discrimina anche il tipo di sistema operativo della macchina, infatti possiamo avere:
 - **s.o. monotask** (l'unico utente può lanciare un solo programma alla volta)
 - **s.o. multitask** (dove un unico utente può lavorare con più programmi contemporaneamente)
 - **s.o. multiutente** (dove più utenti possono lavorare in contemporanea ognuno con diversi programmi)

Risorse condivise: hardware



- in s.o. monotask non esiste politica di gestione essendo la risorsa CPU non condivisa
- in s.o. multitask e in s.o. multiutente in cui la risorsa CPU è comunque condivisa tra più processi (di uno o più utenti), la politica di gestione più applicata è quella del **time-sharing** (tempo condiviso).
- Con il time-sharing i processi vengono accodati in una coda ciclica e a turno o tramite priorità viene assegnato un quanto di tempo ad ognuno di essi.

Hardware e Sistemi operativi



- Mainframe o Super Computer:
 - VM, VMS, MVS, OS/390

- Mini:
 - AS/400, UNIX

- Workstation (server):
 - UNIX (Linux server), Windows Server

- PC:
 - Linux, Windows, MacOS

Condivisione di risorse

□ In Unix: Network File System (NFS)

- accesso in base ai permessi su base

 - ✧ utente

 - ▢ gruppo

 - ▢ altri

□ In Windows: via Netbios

- accesso in base ai permessi su base

 - ▢ utente

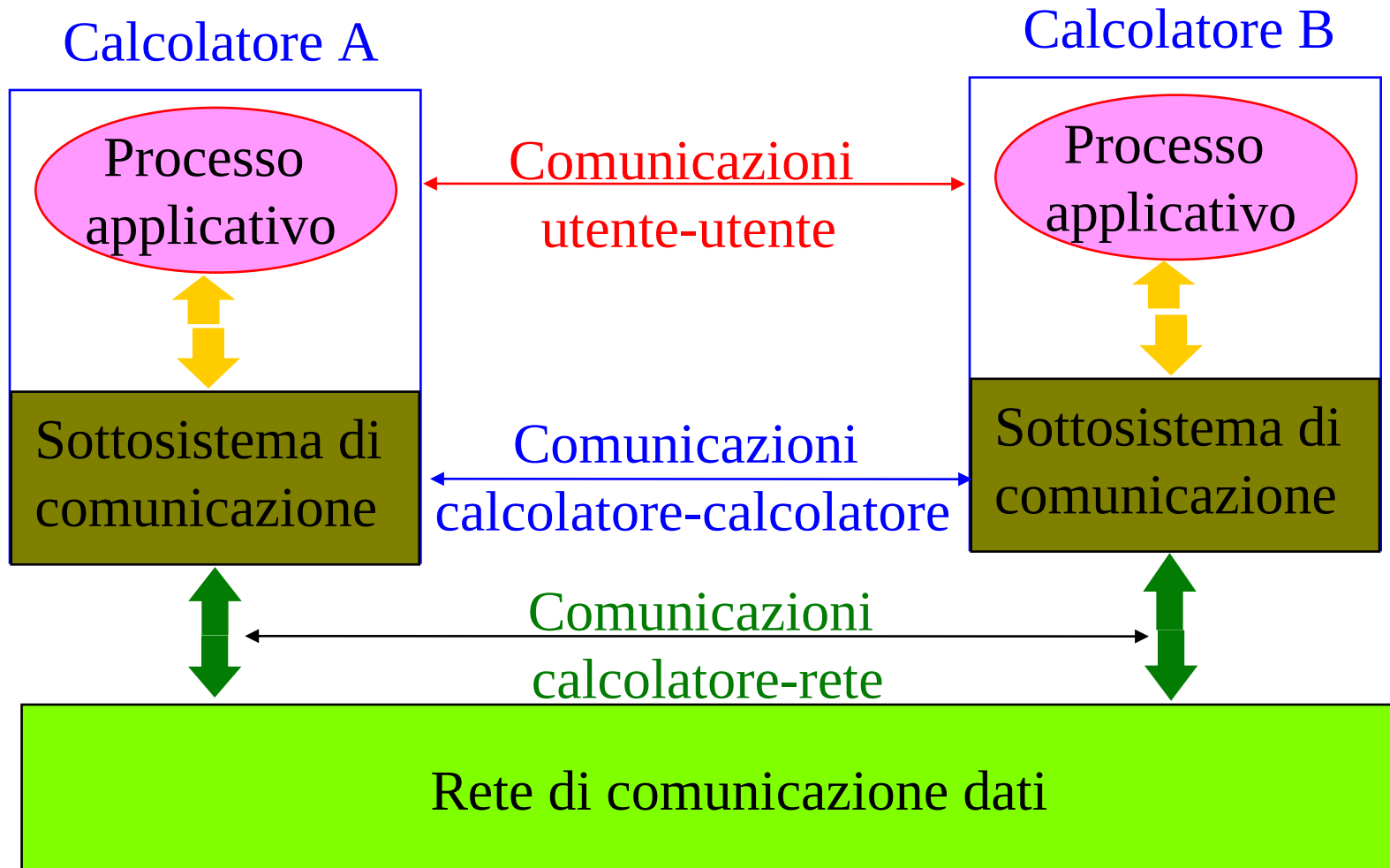
 - ▢ gruppo

 - ▢ altri

Collegamenti e comunicazioni

- ❑ Negli ultimi venti anni si è passati da una nozione di centro di calcolo come stanza con un grande calcolatore (**mainframe**) al concetto di rete di calcolatori indipendenti ma interconnessi (**sistemi distribuiti**).
- ❑ Due calcolatori si dicono *collegati* se sono in grado di scambiare informazioni.
- ❑ Esistono diversi mezzi per collegare in rete più elaboratori: dal filo di rame alla fibra ottica, dalle microonde alle comunicazioni via satellite.
- ❑ Esistono anche diversi tipi di comunicazione tra due o più elaboratori.

Comunicazioni tra calcolatori



Il paradigma Client-Server



- ❑ Il paradigma **client-server** è un modello di interazione tra processi software (allocati anche su computer distinti), tali processi possono essere suddivisi in **client** (che richiedono servizi o *risorse*) e **server** (che offrono servizi o risorse).
- ❑ Il processo **client** è tipicamente dedicato ad interagire con l'utente finale e svolge un ruolo attivo nel generare autonomamente richieste di servizi.
- ❑ Il processo **server** è invece reattivo: svolge una computazione solo a seguito di una richiesta di un client.
- ❑ Il paradigma client-server è ottimo anche per la costruzione di software, indipendentemente dall'allocazione dei processi.

Il paradigma Client-Server

- Interazione asimmetrica tra due processi
 - Il processo client invia una richiesta di servizio al server che esegue il servizio e restituisce un *risultato*.
 - Il flusso di controllo è usualmente *sincrono*
 - La logica di comunicazione è half-duplex: dal client al server e viceversa
 - Non è escluso che client e server possano scambiarsi i ruoli
 - Variante: *function shipping* i processi si scambiano anche codice

Il client



- ❑ L'elaboratore dedicato al **client** deve essere adatto all'interazione con l'utente, cioè corredato di un'interfaccia utente capace di accettare richieste d'accesso alle risorse remote oltre a quelle locali.
- ❑ La macchina deve essere corredata di funzioni comunicative idonee all'invio delle richieste e alla ricezione delle risposte.
- ❑ Spesso si usa un PC dotato di strumenti di produttività come posta elettronica, accesso ad internet, work processing, ecc.

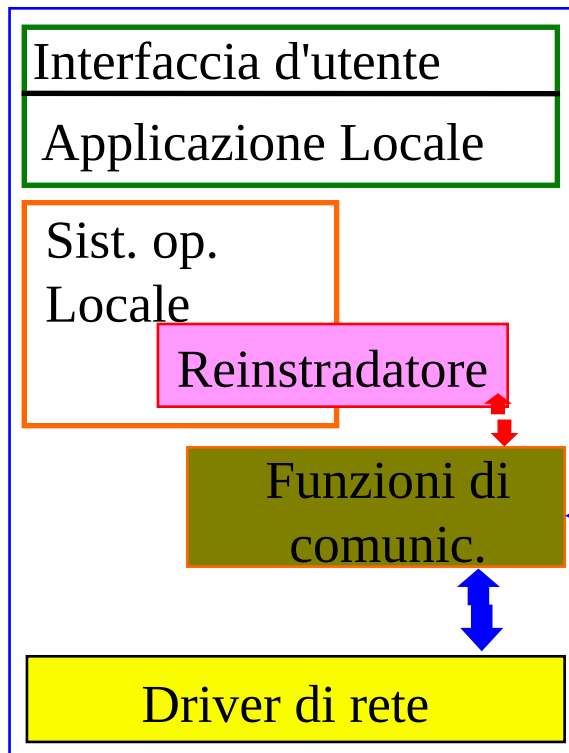
Il server



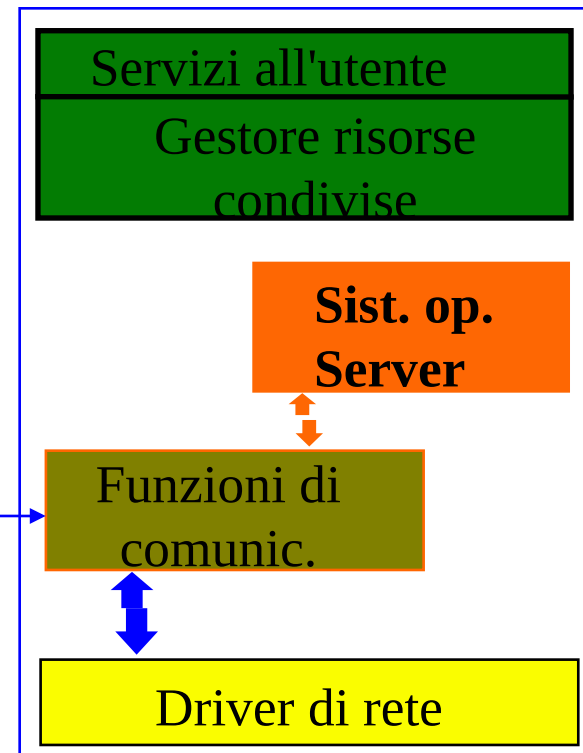
- ❑ L'elaboratore **server** è dimensionato in funzione dei servizi che deve offrire e del carico transazionale.
- ❑ Gestisce ampi buffer di memoria centrale e dispositivi di memoria di massa efficienti e robusti, con elevata capacità di svolgere operazioni di I/O.
- ❑ E' fornito di un sw gestionale capace di interpretare le richieste ed erogare i servizi invocati.
- ❑ E' dotato di un sistema operativo multitasking (capace di gestire richieste multiple simultanee)

Architettura client-server

Client



Server



Protocollo per
richieste di servizi



Le reti:

Client-Server o peer to peer



- ❑ Il concetto di client-server ha quindi insito al suo interno il concetto di rete e di risorsa, poiché i server tramite una rete mettono a disposizione risorse ai client.
- ❑ In alternativa all'organizzazione client server c'è l'architettura **peer to peer** (paritetica) in cui i singoli **nodi** (elementi) della rete sono uguali, cioè sono gestiti nello stesso modo. I singolo nodo può far condividere aree proprie ad altri nodi (trusted = fidati).

Categorie di Sistemi Distribuiti

- ❑ Client-server
- ❑ Object-based
- ❑ Cooperative processing
- ❑ Object-oriented
- ❑ Peer-to-peer
- ❑ Group communication
- ❑ Messaging model
- ❑ Service Oriented Architecture (SOA)
- ❑ Distributed database
- ❑ REST

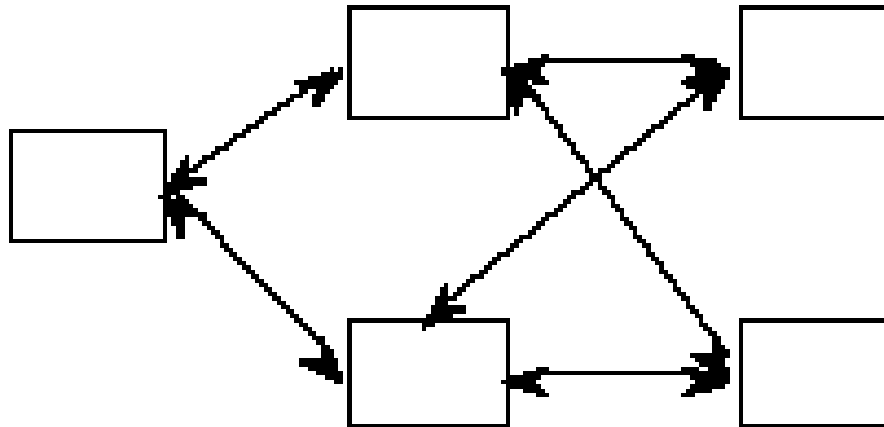
Modelli generali



- **architettura software**
- **architetture distribuite**
 - ✧ client-server
 - ▮ peer-to-peer
 - ▮ 2-tier
 - ▮ 3-tier
 - ▮ 4-tier
- **architetture orientate agli oggetti (OO)**
 - ▮ middleware OO

Architetture software

- **collezione di moduli software (o componenti) interagenti tramite un ben definito paradigma di comunicazione**



Architetture distribuite



❑ Client-server

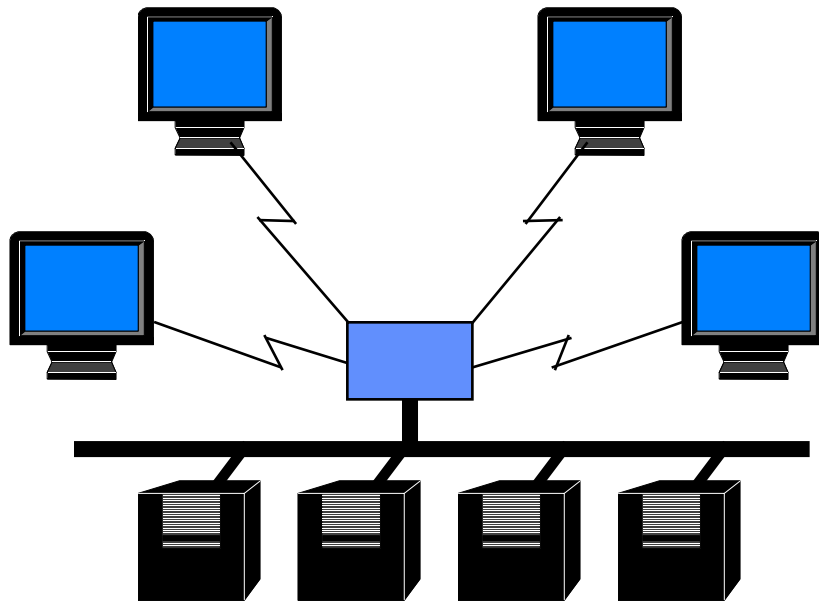
- architettura computazionale in cui i processi client richiedono i servizi offerti da processi server

❑ Peer-to-peer

- architettura in cui i processi possono fungere sia da client che da server

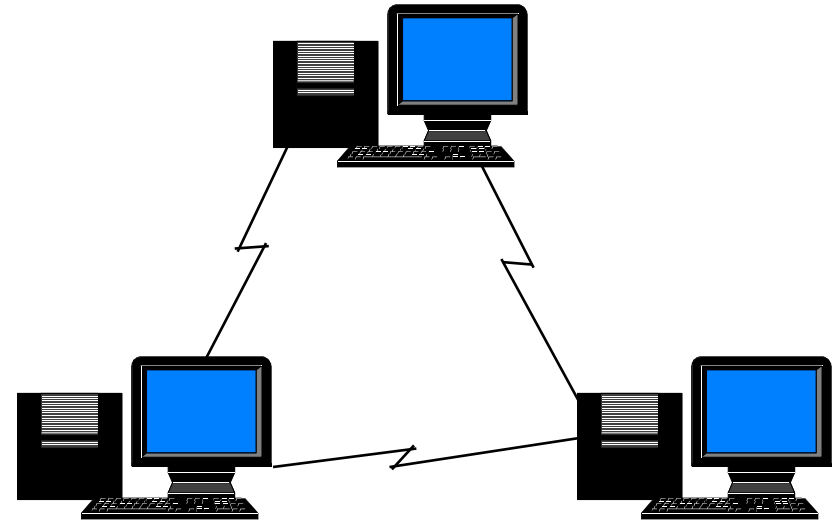
Architetture Client-Server o peer to peer

Architettura client-server



Cluster di workstations
su LAN

Architettura peer to peer



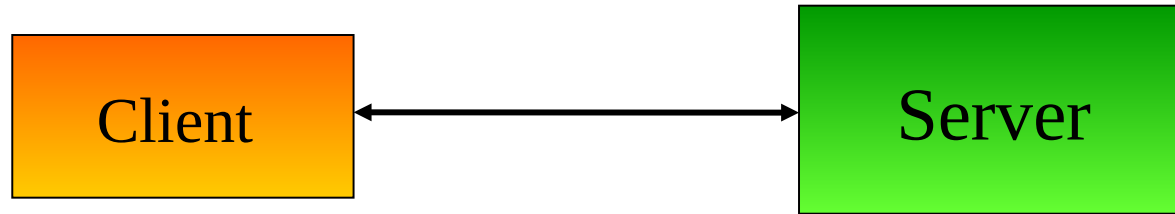
Computer distribuiti

WLAN

LAN

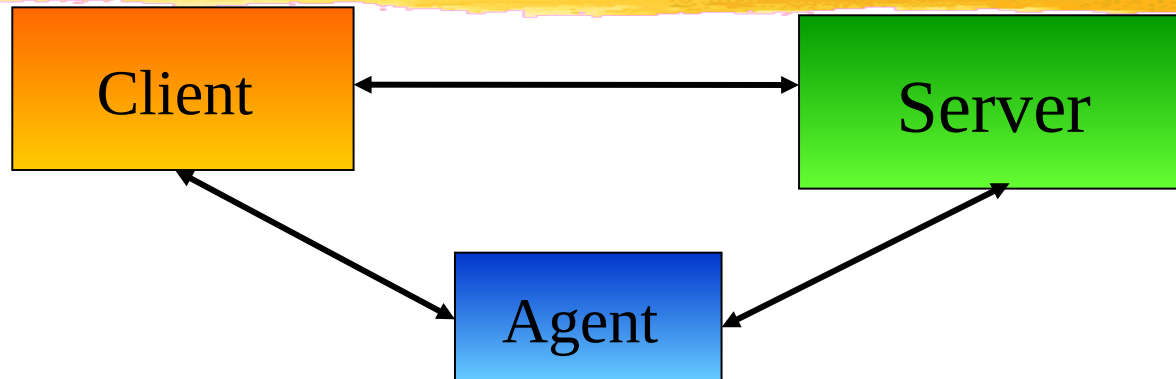
Sistemi aperti e distribuiti

Architettura 2-tier



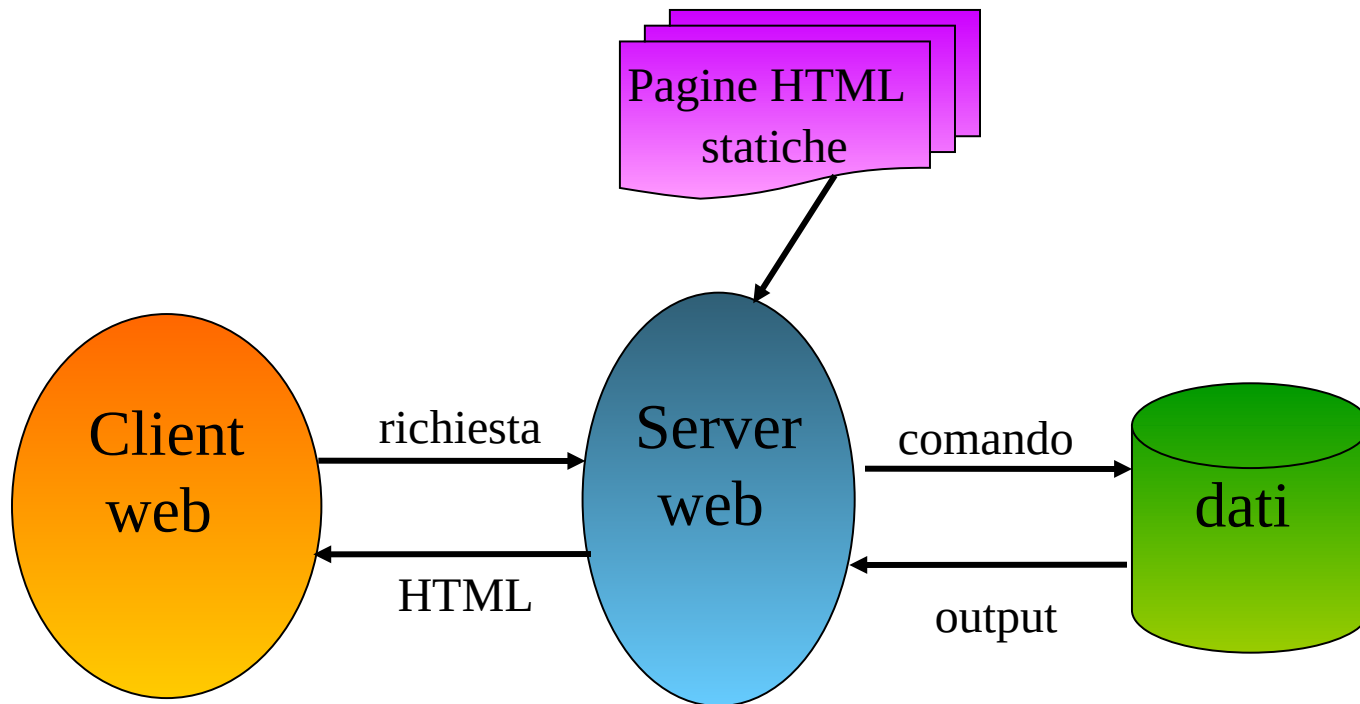
- ❑ Il client interagisce direttamente con il server senza passaggi intermedi
- ❑ architettura tipicamente geograficamente distribuita
- ❑ usata in ambienti di piccole dimensioni (< 50 utenti)
- ❑ svantaggi
 - bassa scalabilità

Architettura 3-tier

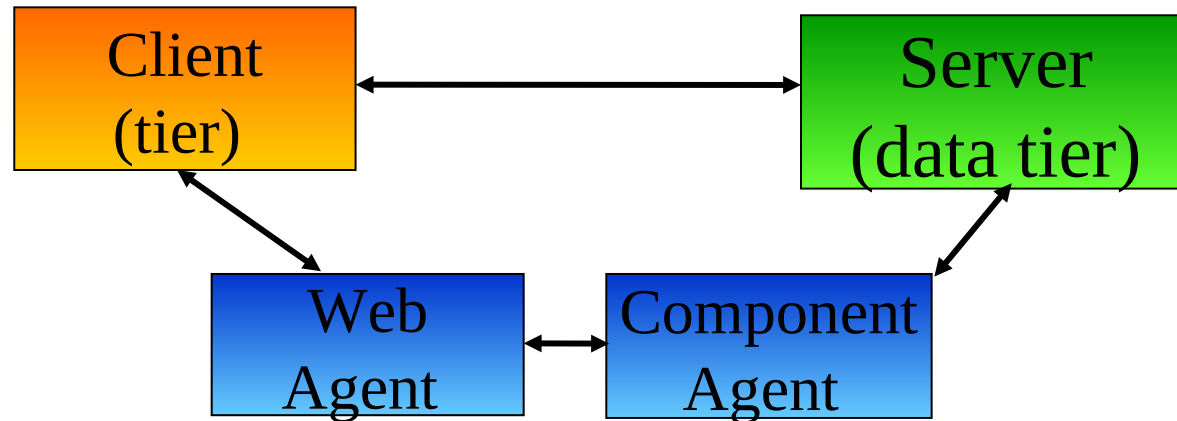


- ❑ Un agente è inserito tra il client e il server.
- ❑ Un agente espleta le attività seguenti
 - filtro (per es. adattare un sistema su mainframe all'ambiente client/server)
 - bilanciamento del carico di lavoro del/i server (per es. monitoraggio delle transazioni per limitare il numero di richieste simultanee)
 - servizi intelligenti (per es. distribuire una richiesta su più server, collezionare i risultati e restituirli al client come risposta singola)

3-tier: modello API

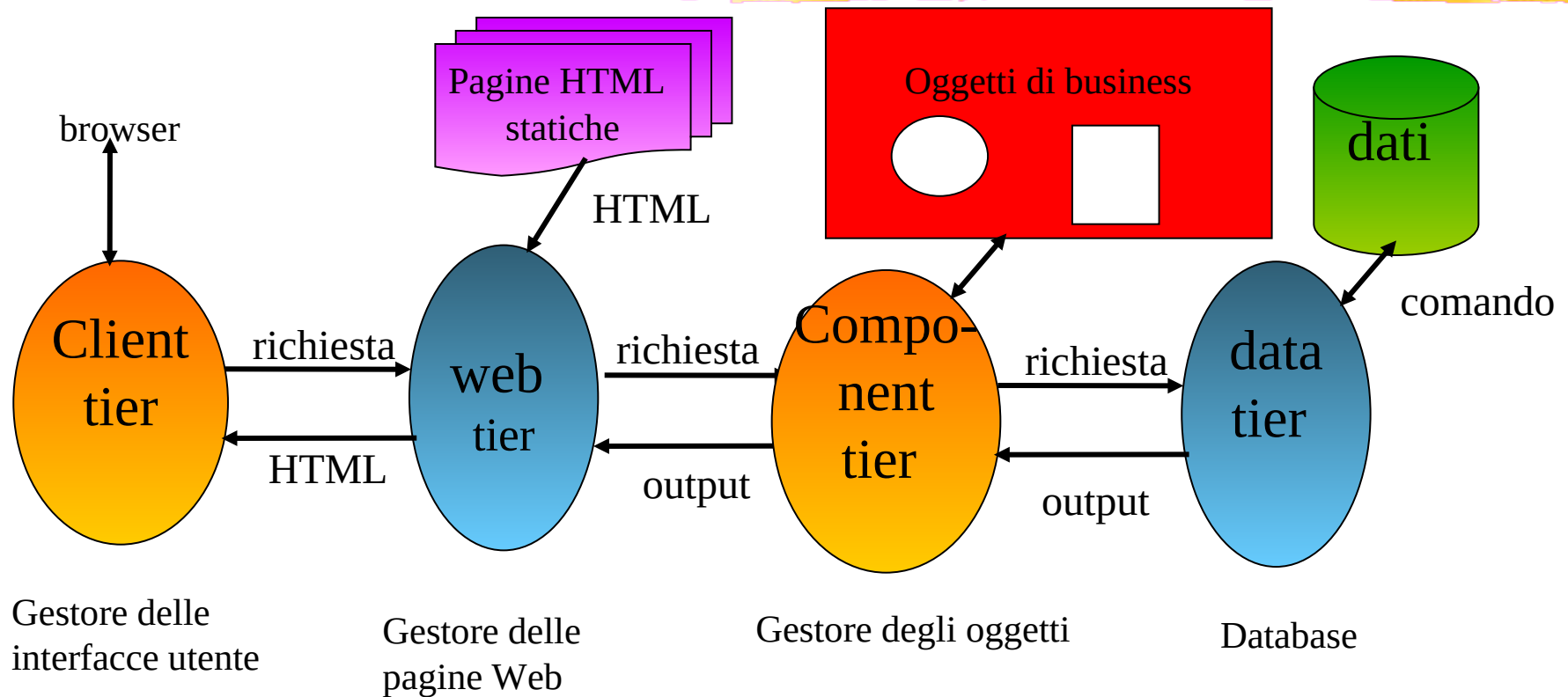


Architettura 4-tier



- Due agenti sono inseriti tra client (client tier) e server (data tier)
 - web tier
 - component tier

4-tier: sistema di commercio in internet

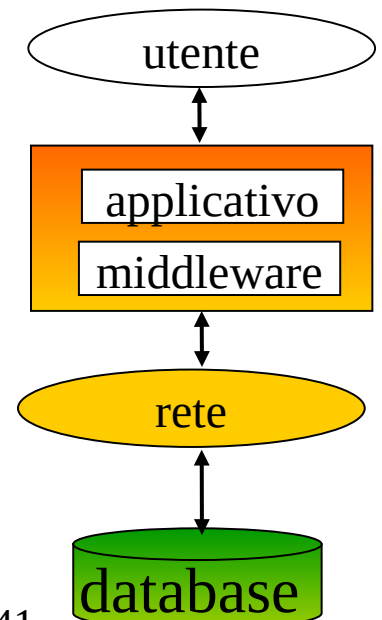


Architetture ad oggetti

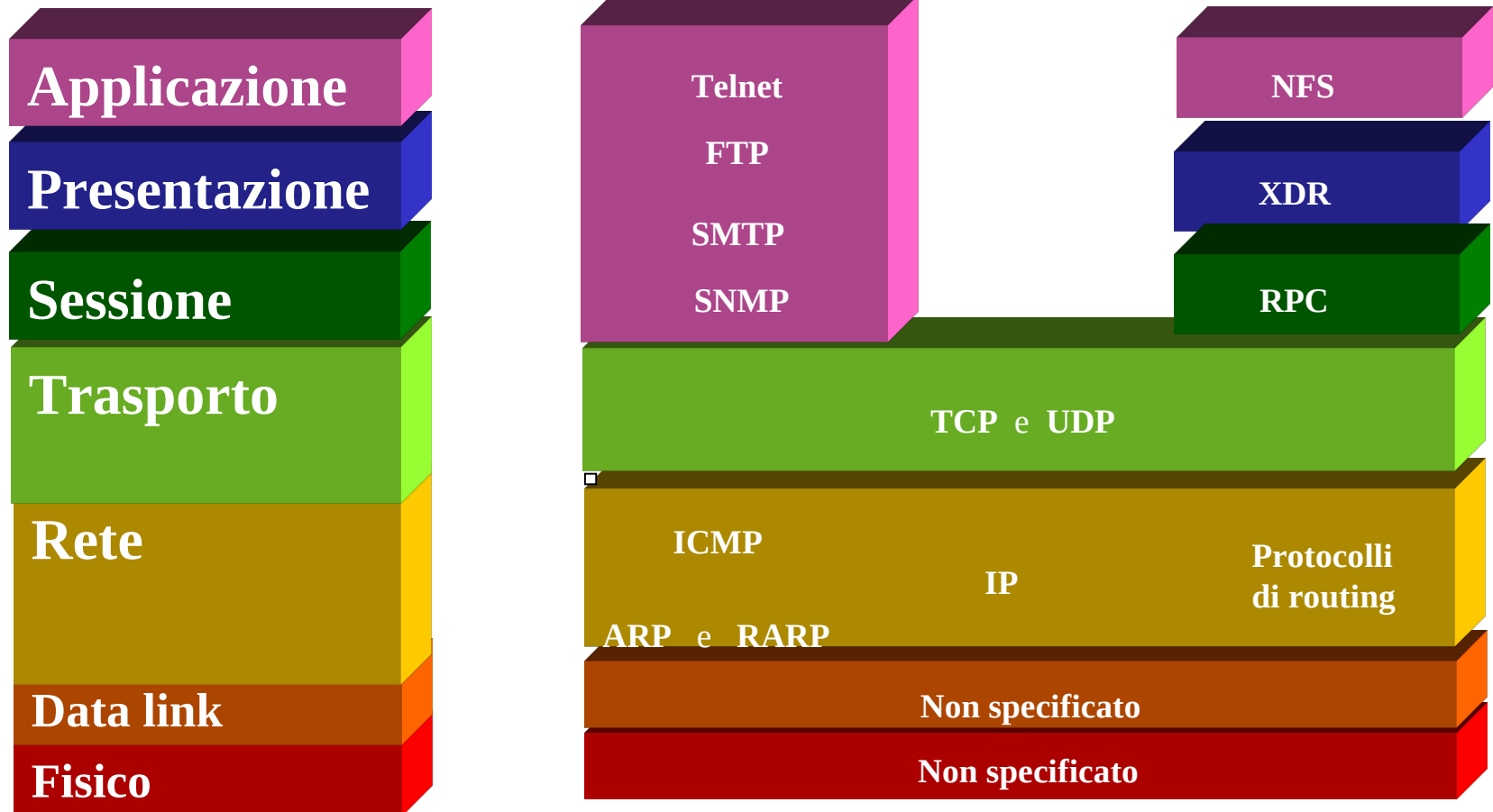
- ❑ Architettura peer-to-peer
- ❑ Ogni agente è un oggetto che eredita la struttura della classe
 - attributi, operazioni e associazioni d'uso
- ❑ Ogni agente rispetta i principi dell'object oriented:
 - identità
 - incapsulamento
 - ereditarietà
 - aggregazione
- ❑ gli agenti comunicano tramite scambio di messaggi
 - locale e distribuito.

Architetture middleware

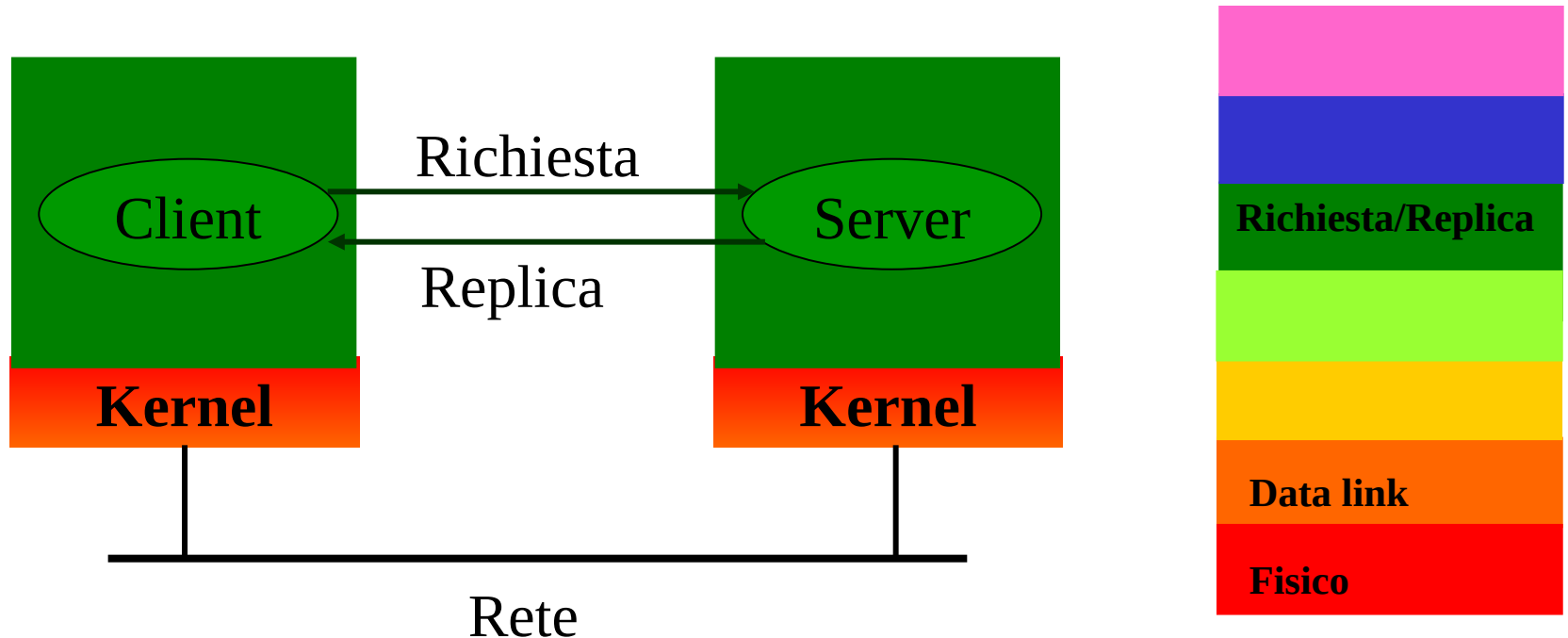
- Il middleware in un ambiente distribuito deve creare l'illusione di un sistema globale in cui tutti i server sembrano comportarsi come un unico calcolatore.
- Obiettivo del middleware è offrire supporto allo sviluppo e all'integrazione di applicazioni Client/Server in ambienti distribuiti eterogenei.



Il modello OSI e il modello TCP/IP



Client server e modello OSI



- Il modello client/server si basa su un semplice protocollo request/reply privo di connessioni.

Client server: vantaggi

- ❑ Il vantaggio principale è la **semplicità**: il client spedisce una richiesta ed ottiene una risposta; non si devono stabilire connessioni prima dell'uso né se ne devono chiudere dopo l'uso e il messaggio di replica serve anche come acknowledgment della richiesta.
- ❑ Altro vantaggio è l'**efficienza**: i livelli OSI interessati sono solo 3 (fisico e data link per trasporto dati e il livello 5 è quello del protocollo request/reply che definisce l'insieme delle richieste e repliche legali, non c'è gestione di sessione perché non ci sono sessioni).

Client/server: chiamate di sistema

- ❑ Per la semplice struttura, i servizi messi a disposizione dal nucleo minimo possono essere ridotti a due chiamate di sistema, una per spedire messaggi ed una per riceverne.
- ❑ Queste chiamate possono essere invocate tramite procedure di libreria per esempio con:
 - `send(dest, &mptr)`
 - ✧ spedisce un messaggio puntato da mptr al processo identificato da dest e fa sì che il mittente si blocchi fino a che non è stato spedito il messaggio.
 - `receive(addr, &mptr)`
 - ▮ fa sì che il chiamante venga bloccato fino a che non arriva un messaggio, quando il messaggio arriva viene copiato in un buffer puntato da mptr e si sblocca il chiamante. Il parametro addr specifica l'indirizzo che il ricevente sta "ascoltando".

Esempio di client/server in C

- L'esempio mostra il tipo di interazioni presenti tra client e server. I dettagli sulle funzioni sono stati omessi.
- Sia il client che il server devono condividere definizioni che vengono raccolte in un file chiamato `header.h` che viene incluso con il comando:
 - `#include "header.h"`

/* Definizioni necessarie ai client e ai server */

#define MAX_PATH	255	/*lungh. massima del nome di un file */
#define BUF_SIZE	1024	/*quanti dati si trasmettono alla volta */
#define FILE_SERVER	243	/*indirizzo di rete del file server */

/* Definizioni delle operazioni ammesse */

#define CREATE	1	/* crea un nuovo file */
#define READ	2	/* legge un pezzo del file e lo restituisce */
#define WRITE	3	/* scrivi un pezzo di un file */
#define DELETE	4	/* cancella un file esistente */

/* Codici di errore */

#define OK	0	/* operazione eseguita correttamente */
#define E_BAD_OPCODE	-1	/* richiesta di operazione sconosciuta */
#define E_BAD_PARAM	-2	/* errore in un parametro */
#define E_IO	-3	/* errore del disco o di I/O */

/* Definizione del formato del messaggio */

```
struct message {  
    long source;           /* identità del mittente */  
    long dest;             /* identità del destinatario */  
    long opcode;           /* quale operazione: READ, CREATE, ecc */  
    long count;            /* quanti byte trasferire*/  
    long offset;           /* dove iniziare a leggere o a scrivere nel file */  
    long extra1;           /* campo extra */  
    long extra2;           /* campo extra */  
    long result;           /* il risultato dell'operazione è restituito qui */  
    char name[MAX_PATH];   /* nome del file su cui si opera */  
    char data[BUF_SIZE];   /* dati da leggere o scrivere */  
};
```

Codice server



```
#include header.h
void main (void)
{
    struct message m1, m2;
    int r;
    while (1) {
        receive (FILE_SERVER, &m1);
        switch (m1.opcode) {
            case CREATE:
                r = do_create(&m1, &m2);
                break;
            case READ:
                r = do_read(&m1, &m2);
                break;
            case WRITE:
                r = do_write(&m1, &m2);
                break;
            case DELETE:
                r = do_delete(&m1, &m2);
                break;
            default:
                r = E_BAD_OPCODE;
        }
        m2.result = r;
        send(m1.source, &m2);
    }
}
```

/ messaggio in arrivo e in partenza */*
/ codice del risultato */*
/ il server gira per sempre */*
/ bloccati in attesa di un messaggio */*
/ verifica il tipo di richiesta e lo esegue */*

Codice client

```
#include header.h
int copy (char *src, char *dst)
{
    struct message m1;
    long position
    long client = 110;
    initialize();
    position = 0;
    do {
        /* prendi un blocco di dati dal file sorgente */
        m1.opcode = READ
        m1.offset = position;
        m1.count = BUF_SIZE;
        strcpy(&m1.name, src);
        send(FILE_SERVER, &m1);
        receive (client, &m1);

        /* scrivi i dati appena ricevuti sul file di destinazione */
        m1.opcode = WRITE
        m1.offset = position;
        m1.count = m1.result;
        strcpy(&m1.name, dst);
        send(FILE_SERVER, &m1);
        receive (client, &m1);
        position += m1.result;
    } while (m1.result > 0);
    return (m1.result >= 0 ? OK : m1.result);
}
```

*/ procedura per copiare un file tramite il server */*

/ buffer del messaggio */*

/ posizione corrente nel file */*

/ indirizzo del client */*

/ si prepara per l'esecuzione */*

/ l'operazione è una lettura */*

/ posizione corrente nel file */*

/ quanti byte leggere */*

/ copia il nome del file da leggere nel messaggio*/*

/ spedisce il messaggio al file server */*

/ bloccati in attesa della risposta*/*

/ l'operazione è una scrittura */*

/ posizione corrente nel file */*

/ quanti byte scrivere */*

/ copia il nome del file da scrivere in buf*/*

/ spedisce il messaggio al file server */*

/ bloccati in attesa della risposta*/*

/ m1.result è il numero dei byte scritti */*

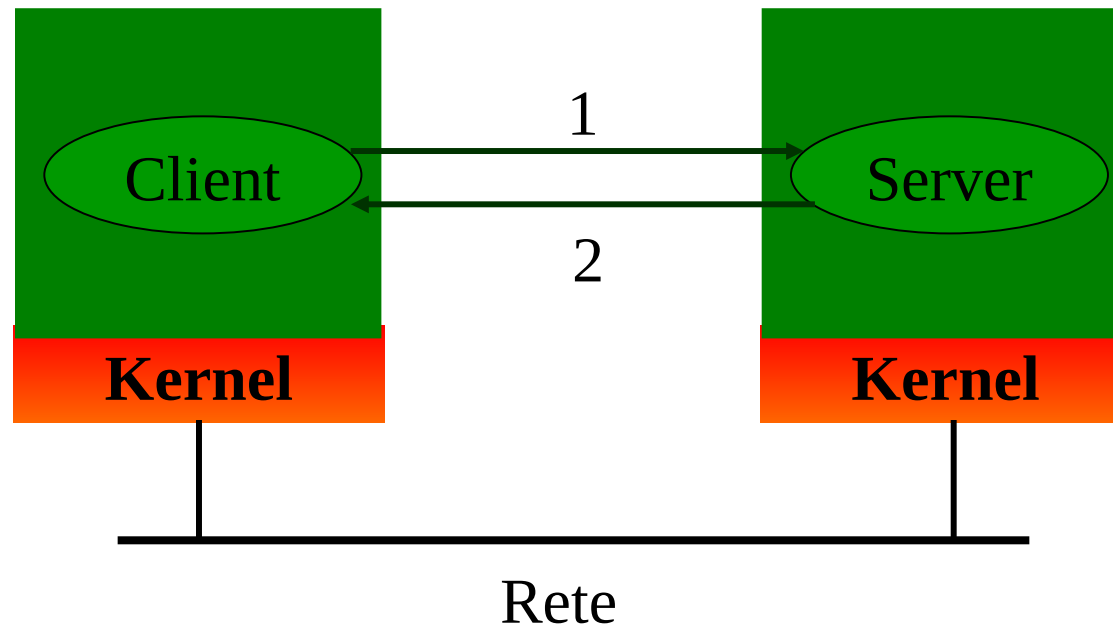
/ itera finché non ha finito */*

/ restituisci OK e il codice di errore */*

Client/server : indirizzamento

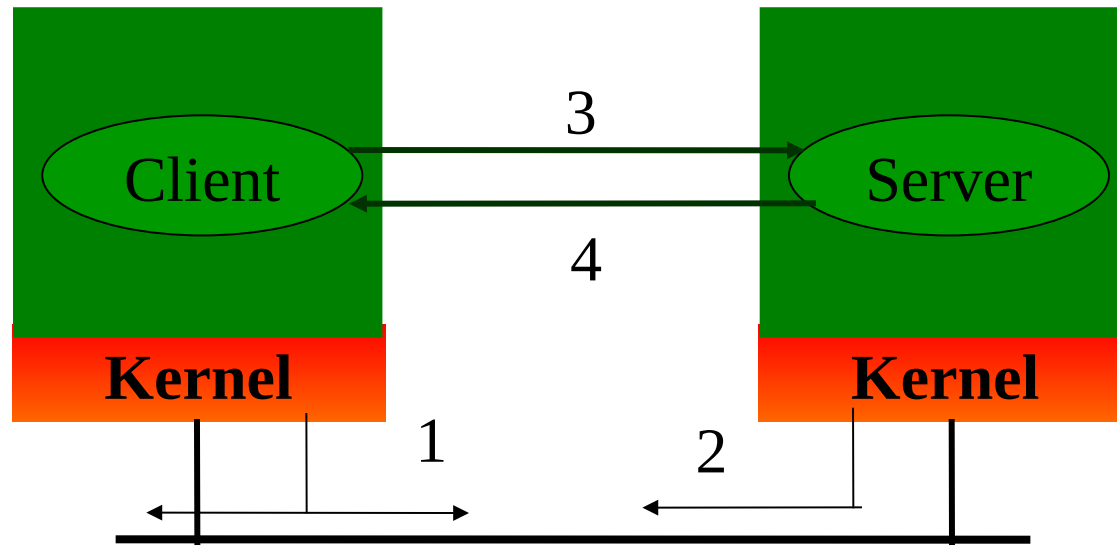
- Esistono vari metodi di indirizzamento tra client e server:
 - indirizzamento **macchina.processo** o **macchina.id-locale**
 - ▮ si usa una combinazione tra il numero della macchina e l'identificativo del processo all'interno del sistema operativo, oppure tra il numero della macchina e un identificativo locale espresso da un intero da 16 o 32 bit scelto in maniera casuale.
 - indirizzamento **con broadcast**
 - ▮ i processi scelgono indirizzi a caso e vengono localizzati all'interno di LAN attraverso il broadcast.
 - ricerca di un indirizzo **mediante un name server**
 - ▮ si inseriscono i nomi ASCII dei server nei client e se ne cerca l'indirizzo a tempo di esecuzione contattando il server dei nomi.

Indirizzamento macchina.processo



- 1: Richiesta per 243.0
- 2: Replica per 199.0

Indirizzamento con broadcast



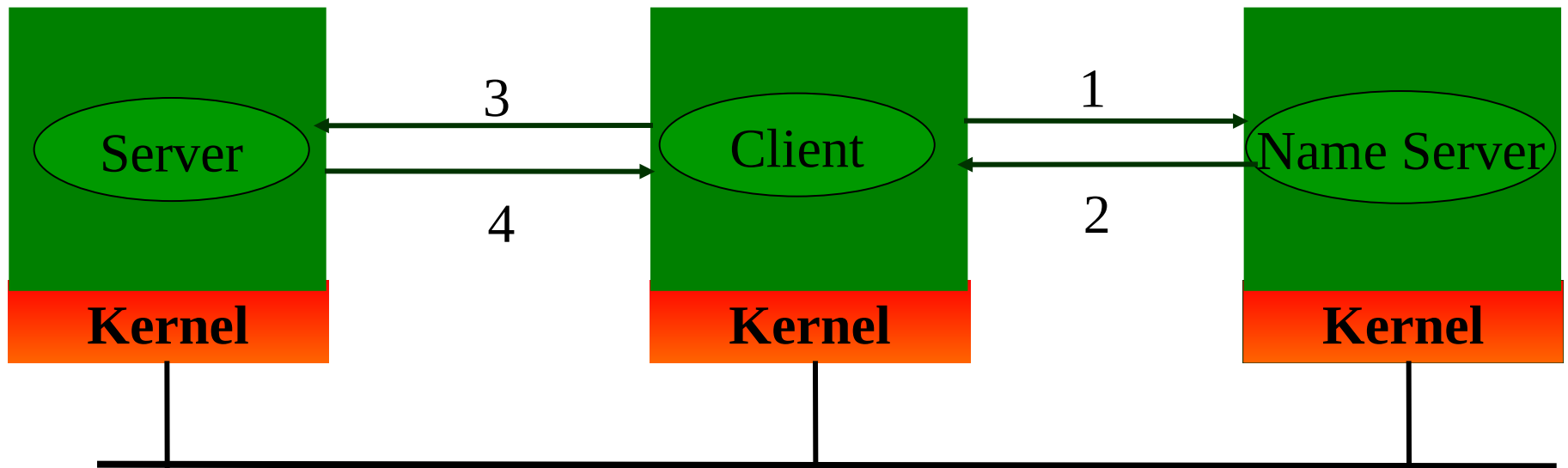
1: Broadcast

2: Ci sono

3: Richiesta

4: Replica

Ricerca tramite Name Server



Rete

- 1: Ricerca
- 2: Replica del Name Server
- 3: Richiesta
- 4: Replica

Primitive bloccanti o sincrone

- Le primitive viste fino ad ora vengono dette **bloccanti** o **sincrone**:

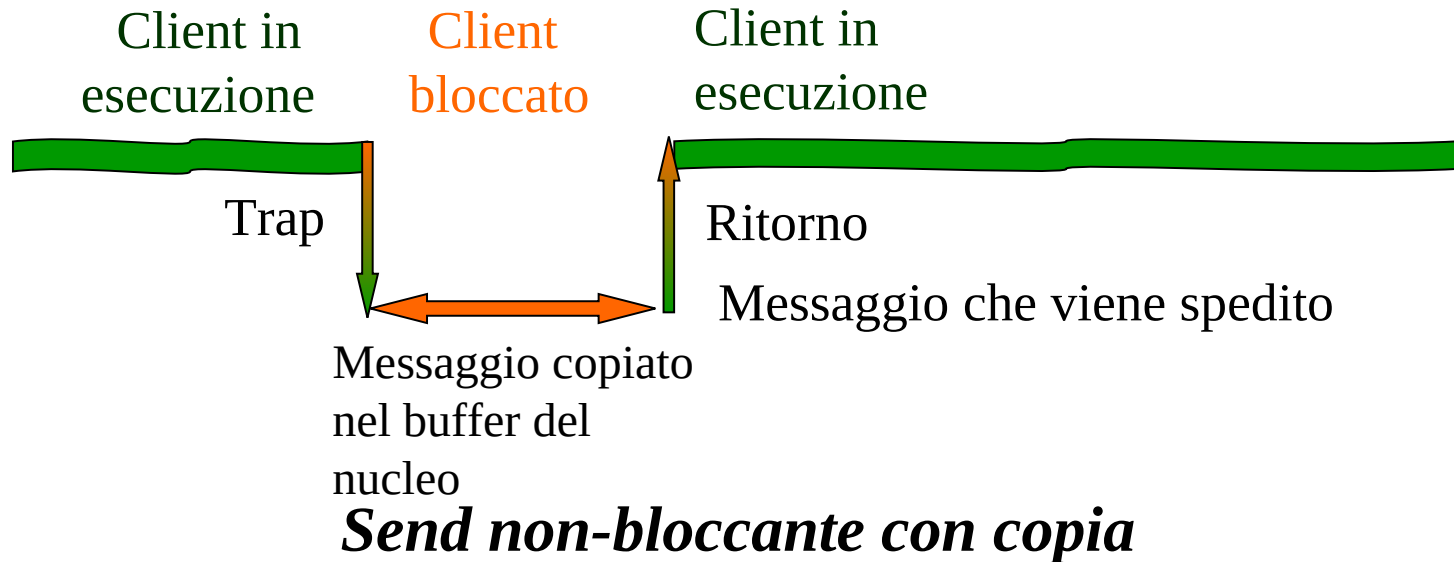
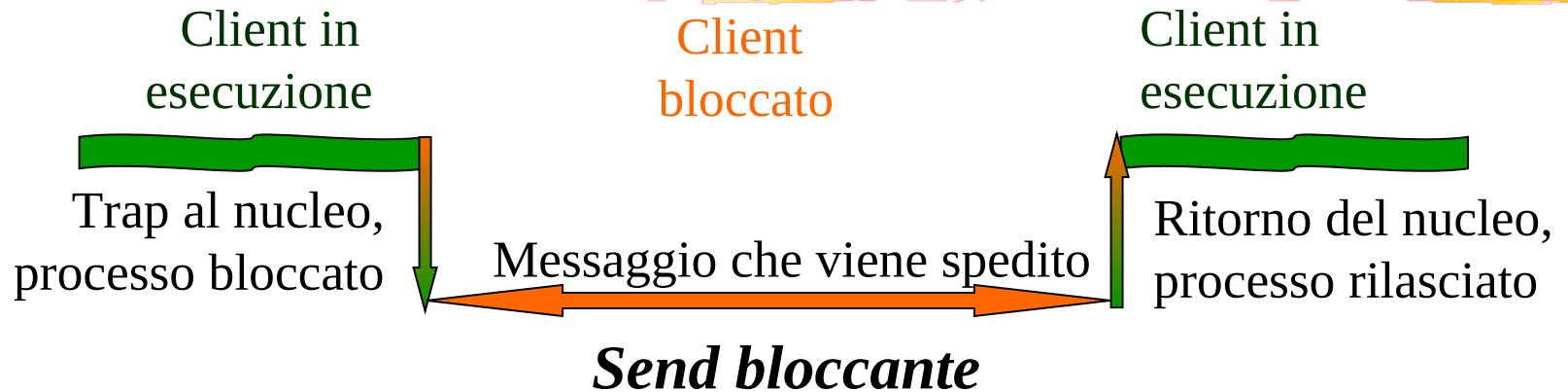
Quando un processo chiama la **send** specifica la destinazione e un buffer da spedire. Mentre il messaggio viene spedito il processo mittente si blocca (viene sospeso): l'istruzione che segue la send non viene eseguita finché il messaggio non è stato completamente spedito.

Analogamente una chiamata ad una **receive** non termina fino a quando il messaggio non è stato ricevuto e posto nel buffer puntato dal parametro. In questo caso il processo può rimanere sospeso anche molto tempo in attesa di ricevere un messaggio.

Primitive non bloccanti o asincrone

- Un'alternativa è rappresentata dalle primitive **non-bloccanti** o **asincrone**:
 - La **send non bloccante** restituisce immediatamente il controllo al chiamante, prima che il messaggio viene spedito. Problemi sorgono però sul riutilizzo del buffer che può essere sovrascritto prima che sia stato spedito il messaggio precedente. Le due possibili soluzioni sono:
 - ✧ send non bloccante **con copia** del messaggio da spedire
 - ▮ send non bloccante **con interruzione** al processo che l'ha generata per avvertire l'avvenuta partenza
 - La **receive non bloccante** comunica semplicemente al kernel dove si trova il buffer e restituisce immediatamente il controllo al chiamante. In questo modo sorge il problema di testare (**test**) o aspettare (**wait**) l'avvenuta ricezione. Una variante è la **conditional_receive** che o prende un messaggio o segnala un fallimento, ma comunque restituisce il controllo o subito o dopo un certo timeout.

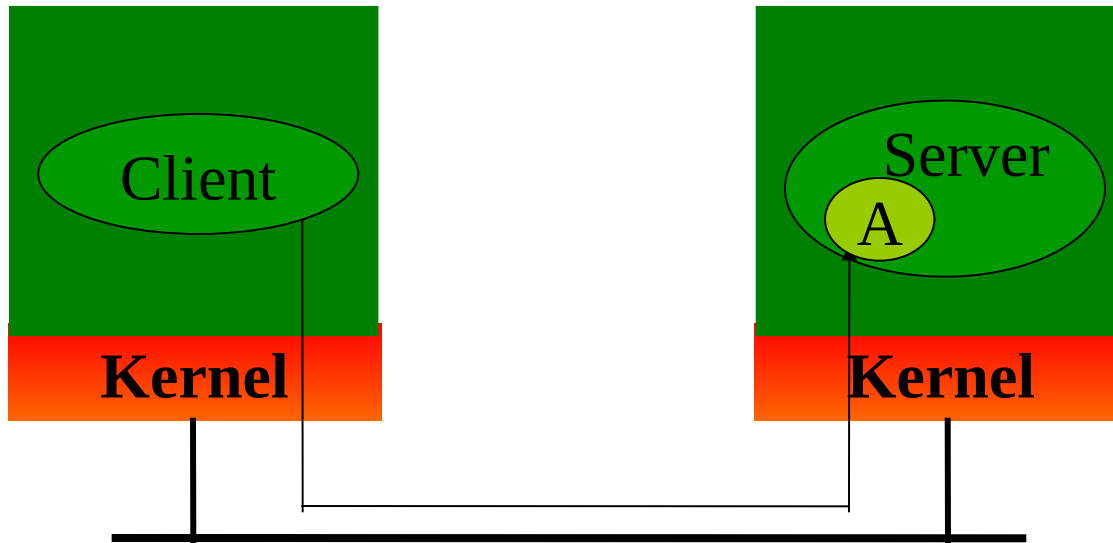
Send bloccante e non bloccante



Primitive bufferizzate e non

- ❑ Se per send e receive non bloccanti la send viene svolta e ultimata prima di una receive, cioè il buffer predisposto dalla receive arriva troppo tardi (probabilmente il messaggio è andato su altri buffer predisposti), occorre sperare che il client dopo un timeout rispedisca un'altra volta il messaggio.
- ❑ Un'alternativa a questa gestione del buffer è quella di prevedere una struttura dati chiamata mailbox (casella postale): un processo interessato a ricevere messaggi chiede al kernel di creargli una mailbox e specifica l'indirizzo associato da ricercarsi nei pacchetti che arrivano dalla rete. Tutti i pacchetti con tale indirizzo finiranno nella mailbox. In tal caso la receive preleva semplicemente un messaggio dalla mailbox. Questa tecnica è chiamata a **primitive bufferizzate**.

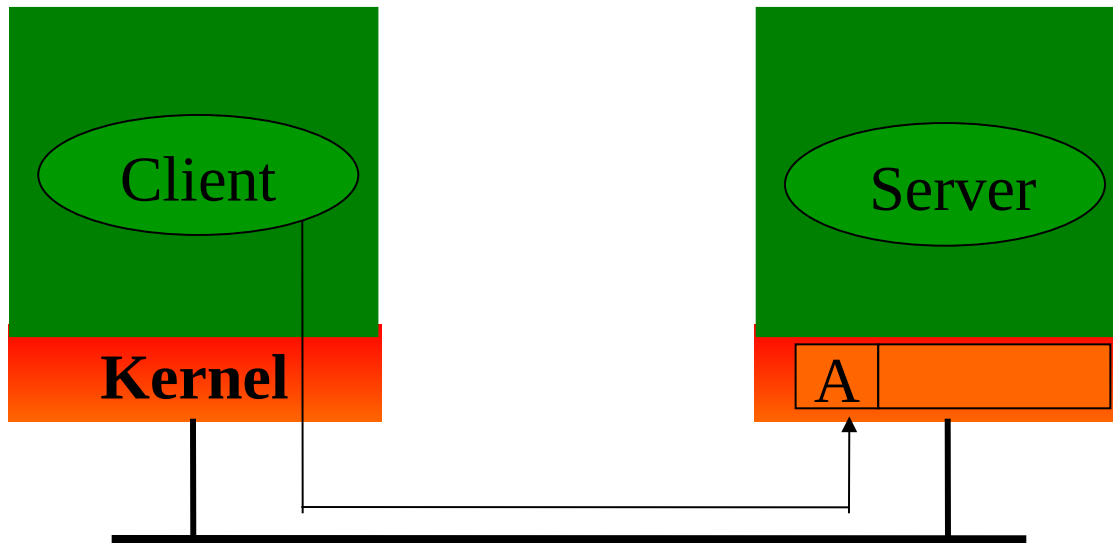
Primitive non bufferizzate



dove A è l'indirizzo
che si riferisce ad un
processo

Scambio messaggio senza buffer

Primitive bufferizzate



dove A è l'indirizzo
che si riferisce ad una
mailbox

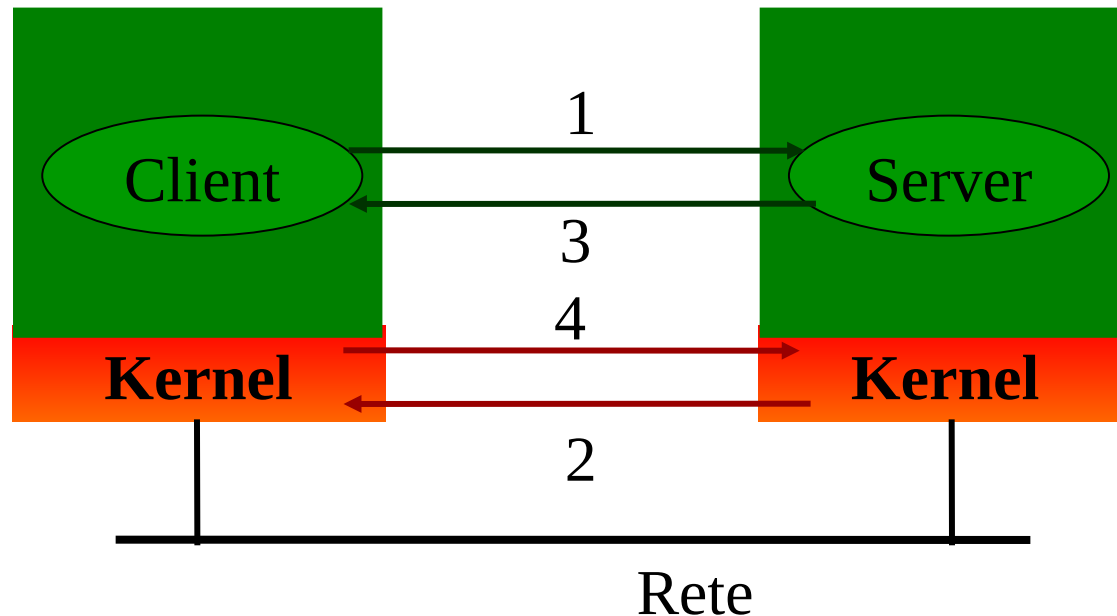
Scambio messaggio con buffer

Primitive affidabili e non

- ❑ Per rendere affidabile la comunicazione tra client e server occorre avere garanzie sul buon arrivo di un messaggio. Si hanno tre diversi approcci:
 - Il primo consiste nel ridefinire la semantica delle primitive evidenziandone l'**inaffidabilità**.
 - Il secondo consiste nel richiedere al kernel della macchina che riceve il messaggio di spedire un **messaggio di avvenuta ricezione** (**ack** per acknowledgement) alla macchina che ha spedito il messaggio.
 - Il terzo consiste nel cercare di sfruttare gli scambi di messaggi già esistenti tra client e server, in tal modo il **messaggio di replica** dal server al client server anche di conferma di avvenuta ricezione del messaggio di richiesta tra client e server. Alcuni sistemi inviano un **ack** finale dal client al server di avvenuta ricezione della replica per liberare il server.

Primitive affidabili

ACK individuale per ognuno dei messaggi



1: Richiesta (dal client al server)

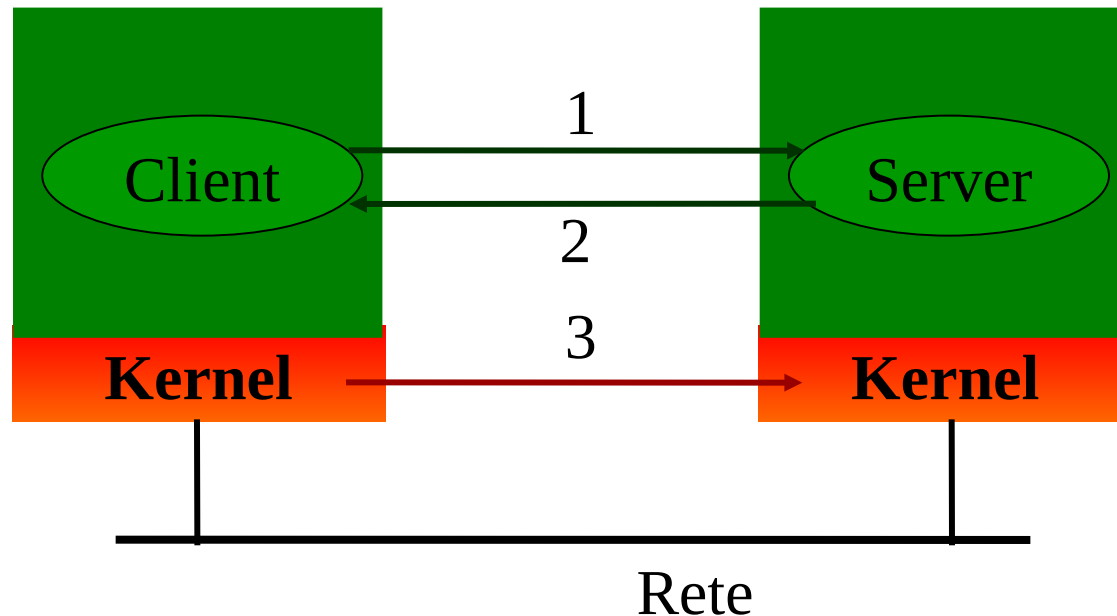
2: ACK (da nucleo a nucleo)

3: Replica (dal server al client)

4: ACK (da nucleo a nucleo)

Primitive affidabili

Messaggi di replica usati come ack delle richieste



1: Richiesta (dal client al server)

2: Replica (dal server al client)

3: ACK (da nucleo a nucleo)

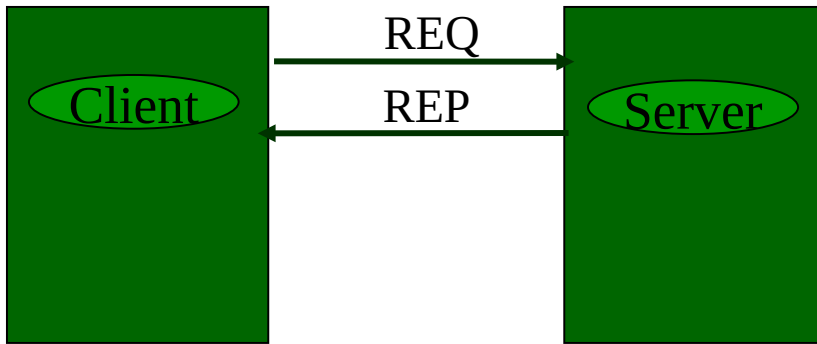
Riassunto problematiche del modello Client/Server

Elemento	Opzione 1	Opzione 2	Opzione 3
Indirizzamento	Numero macchina	Indirizzo di processo sparsi in rete LAN	Ricerca i nomi ASCII tramite server
Bloccante o non	Primitive bloccanti	Non bloccanti con copie nel nucleo	Non bloccanti con interruzione
Bufferizzazione	Non bufferizzate: si scartano i messaggi inattesi	Non bufferizzate con messaggi inattesi conservati	Mailbox
Affidabilità	Inaffidabile	Richiesta-ACK Replica_ACK	Richiesta-Replica ACK

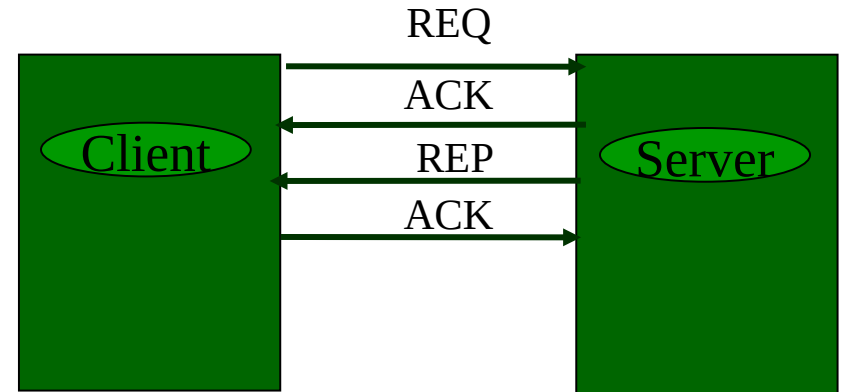
Tipi di pacchetti usati nei protocolli Client/Server

Codice	Tipo di pacchetto	Da	Per	Descrizione
REQ	Richiesta	Client	Server	Il client vuole un servizio
REP	Replica	Server	Client	Replica del server al client
ACK	Ack	Entrambi	Server	Il pacchetto precedente è arrivato
AYA	Sei vivo?	Client	Server	Controlla se il server è caduto
IAA	Sono vivo	Server	Client	Il server non è caduto
TA	Prova di nuovo	Server	Client	Il server non ha spazio
AU	Indirizzo sconosciuto	Server	Client	Nessun processo sta usando questo servizio

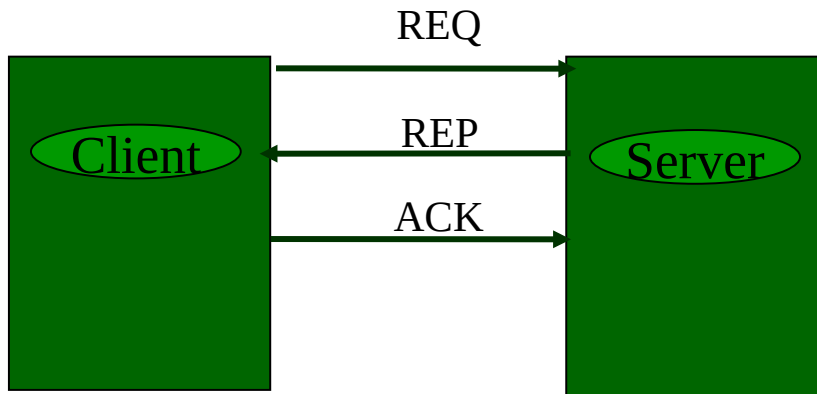
Esempi di scambi pacchetti in comunicazioni client server



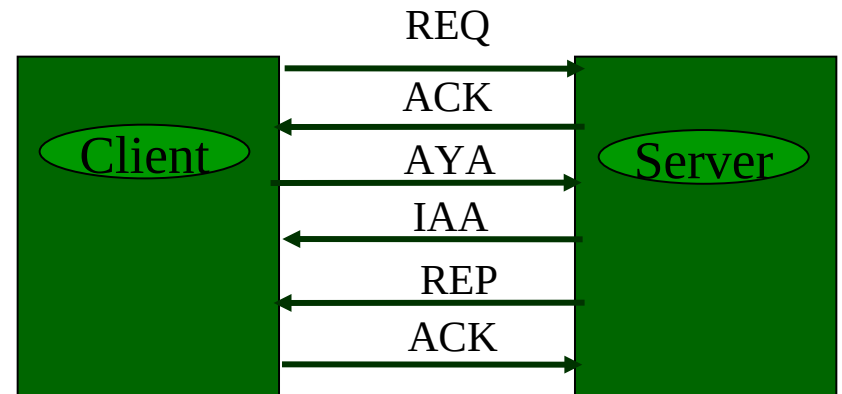
Senza controllo



Controllo ad ogni messaggio



Replica come controllo



*Client controlla sempre
se il server è presente*

Chiamata di procedura remota

- ❑ Il metodo della **chiamata di procedura remota** o **RPC** (Remote Procedure Call) si basa sul concetto di permettere ad un programma di chiamare procedure che sono allocate su altre macchine.
- ❑ Quando il processo sulla macchina A chiama una procedura sulla macchina B, il processo in A viene sospeso e avviene l'esecuzione della procedura sulla macchina B.
- ❑ Si possono passare informazioni tra il chiamante e la procedura chiamata attraverso i parametri e, al contrario, tra la procedura e il chiamante tramite il risultato. Al programmatore è completamente trasparente questo scambio di messaggi e di I/O.
- ❑ La RPC è una tecnica ampiamente usata che sta alla base di molti s.o. distribuiti.

RPC

- Per capire come avviene la chiamata di procedura remota occorre ricordare come funziona una normale chiamata di procedura tipo:

count = read (fd, buff, nbytes)

dove *fd* è un intero, *buff* è un vettore di caratteri e *nbytes* è un intero.

Lo stack viene così modificato:



RPC lato client

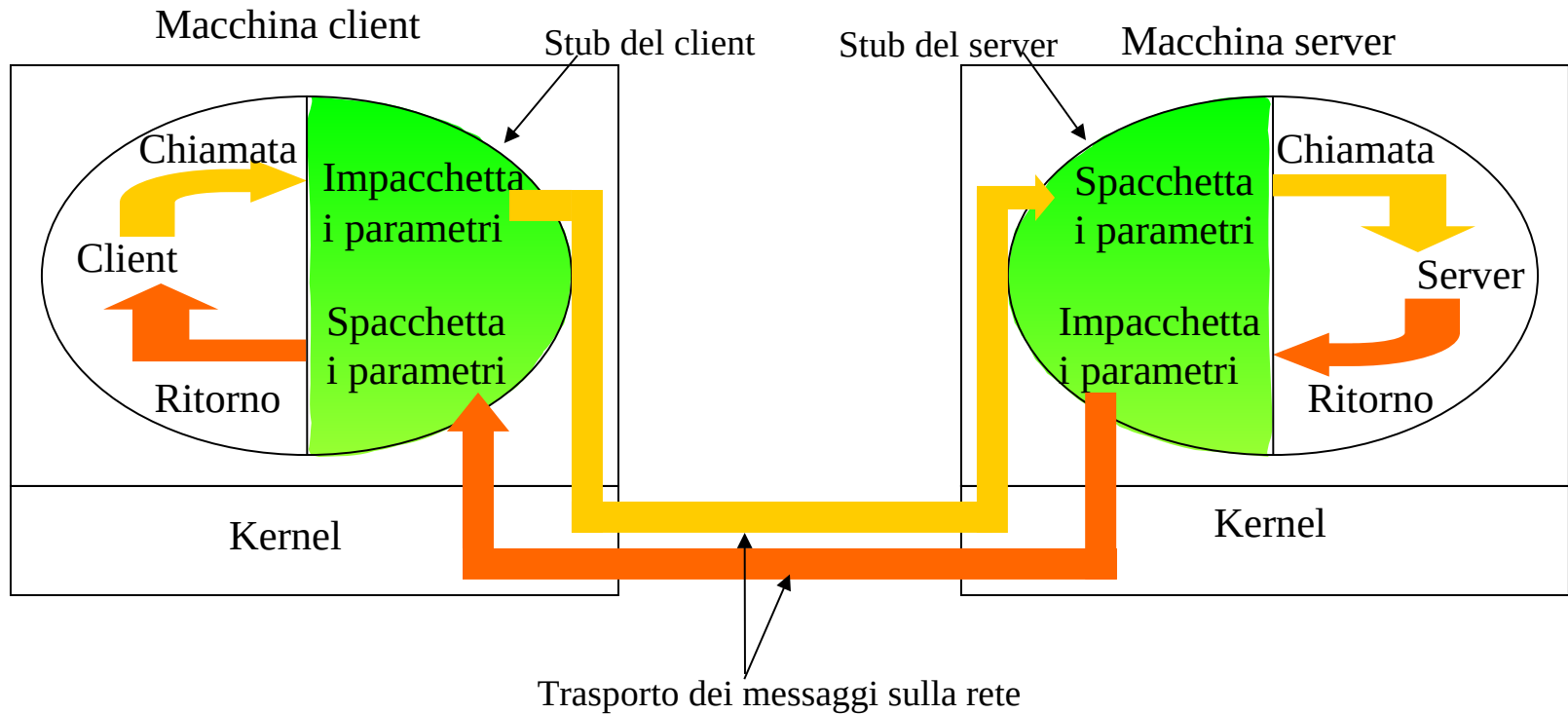
- ❑ Se la *read* è una RPC (cioè una chiamata a “un qualcosa” che girerà sulla macchina del file server) si inserisce nella libreria una diversa versione della *read*, detta **client stub** (“mozzicone” di procedura del client) che viene chiamata come l'originale (*read*).
- ❑ Tale versione stub provoca una trap del kernel, ma non mette i parametri nello stack, né chiede al kernel di restituirle i dati. Impacchetta invece i parametri in un messaggio e chiede al kernel di inviarlo al server. Subito dopo la chiamata alla *send*, il client stub chiama una *receive*, in modo da bloccarsi in attesa di risposta.
- ❑ Quando arriva la risposta il kernel si accorge che è per il client, copia il messaggio nel buffer e il processo viene sbloccato, il client stub spacchetta i parametri e tutto riprende in locale.

RPC lato server



- Quando nel server arriva un messaggio, il kernel lo passa al server stub che avrà chiamato una receive per restare in attesa di messaggi. Il server stub spacchetta i parametri contenuti nel messaggio e poi chiama il server che soddisferà la richiesta riempiendo di dati il buffer del server stub. Il server stub riottenuto il controllo impacchetta il buffer in un messaggio e lo invia tramite una send al client, rimettendosi poi con una receive in attesa di un prossimo messaggio.

RPC



Chiamate e messaggi in una RPC

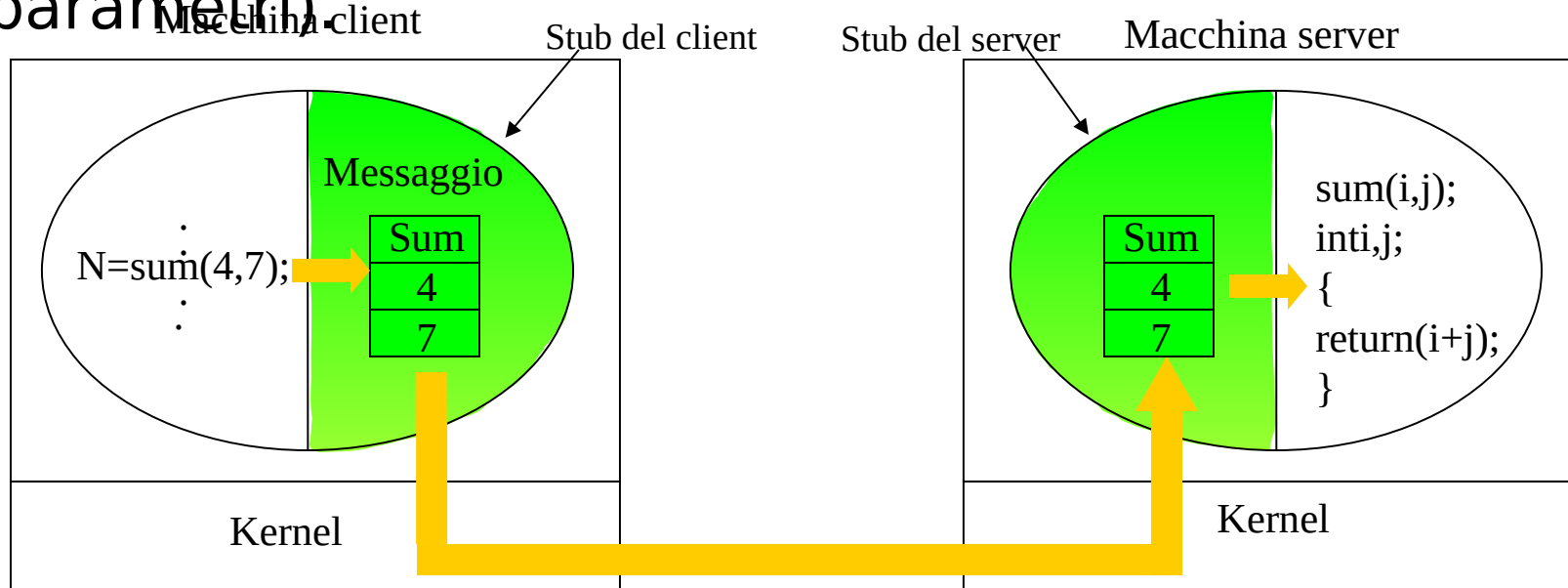
Passi di una RPC



- La procedura client chiama il client stub,
- il client stub costruisce un messaggio ed effettua una trap al kernel,
- il kernel spedisce un messaggio al kernel remoto,
- il kernel remoto trasmette il messaggio al server stub,
- il server stub spacchetta i parametri e chiama il server,
- il server svolge il lavoro e restituisce i risultati al server stub,
- il server stub impacchetta i risultati in un messaggio ed esegue una trap al kernel,
- il kernel remoto spedisce il messaggio al kernel del client,
- il kernel del client restituisce il messaggio al client stub,
- il client stub spacchetta il messaggio e restituisce il controllo al client.

Passaggio dei parametri

- L'impacchettamento dei parametri nei messaggi, effettuato dal client stub e dal server stub, viene detto **parameter marshalling** (inquadramento dei parametri)



Calcolo remoto della funzione sum

Passaggio parametri

- ❑ Alcuni problemi di interpretazione possono nascere nel caso che il client e il server abbiano rappresentazione interne dell'informazione diverse, ad esempio:
 - ASCII e EBCDIC,
 - interi in complemento a 1 o a 2,
 - little endian della Intel (numera i byte da destra a sinistra) e big endian della SPARC (da sinistra a destra)
- ❑ La soluzione a tali problemi può essere data dall'introduzione:
 - del **tipo** di informazione trasmessa e
 - di standard di rete che stabiliscono una **forma canonica** per la rappresentazione dell'informazione in base al tipo.

Codifica dei dati per la trasmissione



- ❑ I due codici più usati sono:
 - ASCII (American Standard Committee for Information Interchange)
 - ✧ codice a 7 bit
 - EBCDIC (Extended Binary Coded Decimal Interchange Code)
 - ▮ codice a 8 bit

Codice ASCII

Bit	7	0	0	0	0	1	1	1	1
	6	0	0	1	1	0	0	1	1
	5	0	1	0	1	0	1	0	1
4 3 2 1	1								
0 0 0 0	NUL	DLE	SP	0	@	P	\	p	
0 0 0 1	SOH	DC1	!	1	A	Q	a	q	
0 0 1 0	STX	DC2	"	2	B	R	b	r	
0 0 1 1	ETX	DC3	#	3	C	S	c	s	
0 1 0 0	EOT	DC4	\$	4	D	T	d	t	
0 1 0 1	ENQ	NAK	%	5	E	U	e	u	
0 1 1 0	ACK	SYN	&	6	F	V	f	v	
0 1 1 1	BEL	ETB	'	7	G	W	g	w	
1 0 0 0	BS	CAN	(8	H	X	h	x	
1 0 0 1	HT	EM)	9	I	Y	i	y	
1 0 1 0	LF	SUB	*	:	J	Z	j	z	
1 0 1 1	VT	ESC	+	;	K	[k	{	
1 1 0 0	FF	FS	,	<	L	\	l		
1 1 0 1	CR	GS	-	=	M]	m	}	
1 1 1 0	SO	RS	.	>	N	^	n	~	
1 1 1 1	SI	US	/	?	O	_	o	DEL	

Codice EBCDIC

Bit				4	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
				3	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1	1
				2	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	1
				1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	1
8	7	6	5																		
0	0	0	0	NUL	SOH	STX	ETX	PF	HT	LC	DEL			SMM	VT	FF	CR	SO	SI		
0	0	0	1	DLE	DC1	DC2	DC3	RES	NL	BS	IL	CAN	EM	CC		IFS	IGS	IRS	IUS		
0	0	1	0	DS	SOS	FS		BYP	LF	EOB	PRE			SM			ENQ	ACK	BEL		
0	0	1	1			SYN		PN	RS	UC	EOT					DC ₄	NAK		SUB		
0	1	0	0	SP										ø	.	<	(+			
0	1	0	1	&										!	\$	*)	;	¬		
0	1	1	0	-	/										'	%	-	>	?		
0	1	1	1											:	#	@	,	=	"		
1	0	0	0		a	b	c	d	e	f	g	h	i								
1	0	0	1		j	k	l	m	n	o	p	q	r								
1	0	1	0			s	t	u	v	w	x	y	z								
1	0	1	1																		
1	1	0	0		A	B	C	D	E	F	G	H	I								
1	1	0	1		J	K	L	M	N	O	P	Q	R								
1	1	1	0			S	T	U	V	W	X	Y	Z								
1	1	1	1	0	1	2	3	4	5	6	7	8	9								□

Binding dinamico

❑ Come fa il client a localizzare il server?

- Un metodo è quello di cablare all'interno del programma del client l'indirizzo di rete del server.
 - ✧ Questo metodo risulta poco flessibile, se il server si sposta o viene replicato è necessario ricompilare molti programmi.
- Un metodo che molti sistemi distribuiti usano è quello che viene chiamato **binding dinamico** (associazione dinamica)
 - ▮ Questo metodo si basa sul riconoscimento, all'atto delle chiamate di procedure remote, del server a disposizione. Questo aggancio avviene in maniera dinamica in fase di esecuzione.

Binding dinamico

- Riprendendo l'es di pag. 45,46,47 vengono dati oltre al nome del server (file_server), la versione (3.1) e la lista delle procedure messe a disposizione dal server (read, write, create, delete) dove per ogni procedura si danno anche i tipi di parametri: di in o out rispetto al server. Un parametro di in viene spedito dal client al server, al contrario uno di out dal server al client. Un parametro di in out (non presente nell'esempio) dovrebbe essere spedito dal client al server, modificato dal server e rispedito indietro al client).

```
#include <header.h>
```

```
specification of file_server, version 3.1:
```

```
long read (in char name[MAX_PATH], out char buf[BUF_SIZE],  
           in long byte, in long position);
```

```
long write (in char name[MAX_PATH], out char buf[BUF_SIZE],  
            in long byte, in long position);
```

```
long create (in char name[MAX_PATH], in int mode);
```

```
long delete (in char name[MAX_PATH]);
```

Binding dinamico

- Quando un server va in esecuzione l'interfaccia del server viene esportata mediante la chiamata alla `initialize()` eseguite fuori dal ciclo principale. Questo significa che il server manda un messaggio ad un programma detto **binder** per rendergli nota la propria presenza. Questa procedura è detta registrazione del server. Per registrarsi, il server passa al binder il proprio nome, il proprio numero di versione, un identificatore unico (un intero da 32 bit) e una **handle** (aggancio) che permette di localizzarlo. La handle dipende dal sistema e può essere: un indirizzo internet (nome o IP), un indirizzo X.500, un identificatore di processo. In aggiunta il server può anche passare al binder informazioni sull'autenticazione.
- Lato client invece, quando un client chiama per la prima volta una procedura remota lo stub del client vede che ancora non è stata associata a nessun server e quindi manda un messaggio al binder chiedendo di importare nell'es. la versione 3.1 dell'interfaccia. Il binder se esiste un server adatto restituisce la sua handle e il suo identificatore al client stub.

Binding dinamico

L'interfaccia del binder

Chiamata	Dati in ingresso	Dati in uscita
Registrazione	Nome, versione, identificatore unico, handle	
Cancellazione di una registrazione	Nome, versione, identificatore unico	
Ricerca	Nome, versione	Handle, identificatore unico

Semantica della RPC nei fallimenti

- Si distinguono cinque diverse classi di fallimento che si possono verificare in sistemi con RPC:
 - il client non riesce a individuare il server,
 - il messaggio di richiesta dal client al server va perso,
 - il messaggio di replica dal server al client va perso,
 - il server cade dopo aver ricevuto una richiesta,
 - il client cade dopo aver spedito una richiesta.

Ciascuna di queste classi pone problemi diversi e richiede soluzioni diverse.

Il client non riesce a individuare il server

- ❑ Il client può non riuscire ad individuare un server adeguato se:
 - il server è fuori servizio
 - il client è stato compilato usando una versione di stub diversa da quella usata per compilare il corrispettivo stub del server
- ❑ Per segnalare queste ed altre anomalie vengono usati o codici appositi di fallimento (errno) oppure un altro metodo è quello che prevede che ogni errore dia origine ad una eccezione (exception) o ad un segnale (signal).

Messaggi di richiesta persi

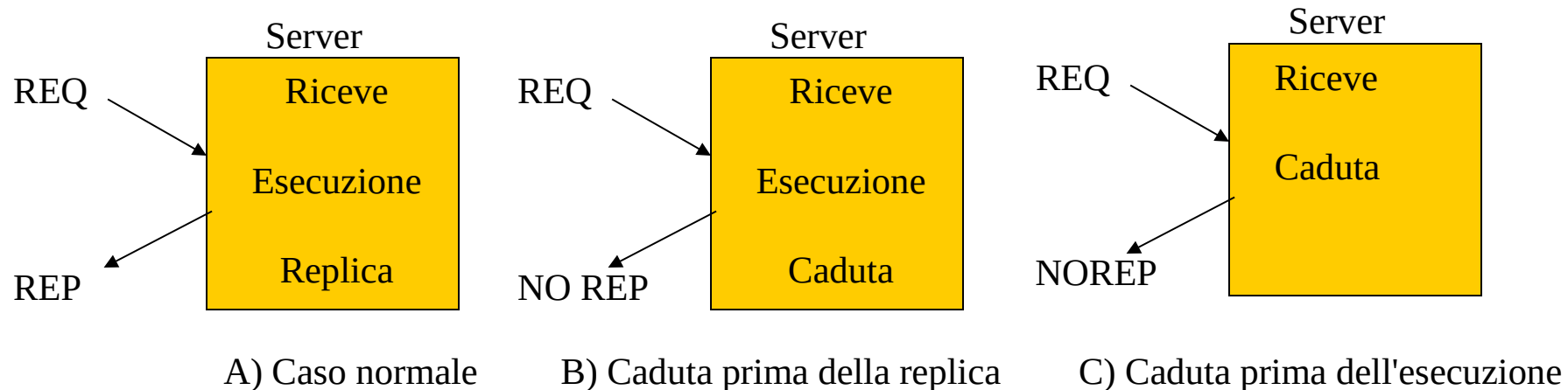
- ❑ Questo tipo di inconveniente viene risolto facendo in modo che il kernel del client faccia partire un timer quando si invia un messaggio e, se il timer scade prima che sia arrivato il messaggio di risposta, il kernel provvede a rispedire il messaggio.
- ❑ Se dopo ripetuti tentativi (numero massimo) il client non ha avuto risposta si rientra nel caso precedente (il client non è in grado di individuare il server).

Messaggi di risposta persi

- ❑ Questo tipo di inconveniente viene inizialmente risolto come il precedente facendo in modo che il kernel del client rispedisca il messaggio di richiesta. Ma queste iterazioni di richieste possono essere soddisfatte tutte e ciò può provocare anomalie. Richieste che non provocano anomalie nella loro ripetizione vengono dette **idempotenti**.
- ❑ Per richieste non idempotenti occorre che il server si premunisca dal ripetere la loro espletazione. Questo può essere fatto in due modi: o facendo introdurre dal kernel del client un numero di sequenza a tutti i messaggi spediti oppure inserendo un bit aggiuntivo al messaggio che dica se è un messaggio originale o una ripetizione.

Cadute del server

- A fronte di una richiesta dal client si possono aver le tre situazioni seguenti:



Nel caso B) il sistema deve riportare un fallimento al client, mentre nel caso C) il client deve semplicemente ritrasmettere la richiesta.

Cadute del server

- Esistono tre tecniche distinte nell'affrontare il problema:
 - Si ritenta la trasmissione finché non arriva una replica: chiamata **semantica at least once** (almeno una volta),
 - Si smette subito e si riporta un fallimento: chiamata **semantica at most once** (al più una volta),
 - Non si garantisce nulla.
- Ciò che si vorrebbe è ottenere una **semantica exactly once** (esattamente una volta), ma non c'è modo di ottenerla nel caso generale.

Cadute del client

- ❑ Cosa succede se un client cade prima che il server spedisca la replica? Rimane attiva nel server una computazione con nessun padre che la aspetta. Tale computazione non richiesta viene detta **orfano**. Gli orfani possono provocare molti problemi: sprecano cicli di CPU, possono bloccare l'accesso a file o impegnare altre risorse preziose.
- ❑ Alla ripartenza del client cosa succede? Cosa occorre prevedere? Esaminiamo quattro possibili soluzioni:
 - Prima che lo stub del client spedisca la richiesta questa va registrata in un apposito log salvato su disco. Al reboot del client si controlla il log e si uccide l'orfano. Questa soluzione è chiamata **sterminio**. Molte RPC orfane potrebbero però al loro interno aver prodotto anche nipoti orfani complicando ulteriormente il problema.

Cadute del client

- o Una soluzione detta **reincarnazione** consiste nel dividere il tempo in epoche numerate. Alla ripartenza del client si passa ad una nuova epoca e ciò viene comunicato via broadcast a tutti i server.
- o Una soluzione che è una variante della precedente è chiamata **reincarnazione gentile**: quando arriva il broadcast di un'epoca, ciascuna macchina controlla se ha in corso una computazione remota, nel qual caso cerca di individuarne il possessore. La computazione viene uccisa solo se non viene individuato il possessore.
- o L'**esaurimento** è una soluzione che associa a ciascuna RPC un quanto di tempo standard T per compiere il suo lavoro. Se il server non è in grado di finire, deve comunque richiedere esplicitamente un ulteriore quanto, altrimenti la RPC viene uccisa.

Problemi implementativi: protocolli

- ❑ Il successo o il fallimento di un sistema distribuito dipende spesso dalle sue prestazioni che a loro volta dipendono dalla velocità della trasmissione e quest'ultima dipende dalla sua implementazione reale.
- ❑ Il primo aspetto riguarda i **protocolli** di comunicazione da scegliere per una RPC: spesso si adottano i protocolli già esistenti per la comunicazione (IP e UDP), più oneroso, anche se sicuramente vantaggioso sarebbe implementare protocolli ad hoc per le RPC.
- ❑ Ecco alcuni cenni sul protocollo IP alla base anche dell'UDP.

RIPASSO:

Internet Protocol, IP

- L'utente considera un'internet come una singola rete virtuale che interconnette tutti gli host e attraverso cui e' possibile comunicare
- Il software dell'internet e' progettato intorno a tre servizi di rete disposti su scala gerarchica:
 - ***Servizio di consegna del pacchetto senza connessione***
 - ***Servizio di trasporto inaffidabile***
 - ***Servizi di applicazione***

RIPASSO:

Internet Protocol (IP)

- ❑ *Consegna senza connessione:* ogni pacchetto e' consegnato indipendentemente dagli altri
- ❑ *Servizio inaffidabile:* i pacchetti possono andar persi o fuori sequenza
- ❑ *Consegna best-effort:* si fa di tutto per consegnare i pacchetti: l'inaffidabilità si verifica solo per malfunzionamenti hw

RIPASSO:

Internet Protocol, IP (II)

- ❑ L'IP definisce l'esatto formato dei dati, mentre attraversano l'internet TCP/IP
- ❑ L'IP svolge la funzione di **routing** scegliendo il percorso che dovranno seguire i dati
- ❑ Definisce un insieme di regole che inglobano l'idea della consegna non affidabile dei pacchetti e come host e gateway debbono elaborare i pacchetti, come e quando generare messaggi di errore, le condizioni in cui scartare i pacchetti, etc
- ❑ L'unita' fondamentale di trasferimento e' detta *datagramma IP*, il quale e' diviso in area di intestazione e area dati

RIPASSO: IP address



- ❑ Un indirizzo IP su 32 bit permette di identificare univocamente una rete ed uno specifico host appartenente alla rete
- ❑ L'indirizzo si divide in due parti:
 - rete
 - host
- ❑ Esistono 4 tipi di classi di IP address.

RIPASSO:

Classi di indirizzi IP (x.y.z.q)

Classe A: con $x < 128$



Classe B: con $128 \leq x \leq 191$



Classe C: con $192 \leq x \leq 223$



Classe D: con $x > 223$



multicast

RIPASSO:

Indirizzi riservati

❑ Livello rete

○ Classe A:


- ✧ 0.y.z.q (default route)
- ▮ 127.y.z.q (loopback address)

❑ Livello host

- ▮ x.0.0.0 e x.255.255.255 per la classe A
 - ▮ x.y.0.0 e x.y.255.255 per la classe B
 - ▮ x.y.z.0 e x.y.z.255 per la classe C
- che identificano rispettivamente la rete stessa e il “broadcast address”

RIPASSO:

Nota



- ❑ Gli indirizzi IP vengono assegnati alle interfacce di rete non agli host!

E' quindi errato, anche se comune, chiamare gli indirizzi IP "indirizzi degli host".

Infatti un computer che funziona da gateway ha diversi indirizzi per ogni rete che collega, e quindi viene indirizzato in maniera diversa da singoli dispositivi in base alla rete su cui questi risiedono.

RIPASSO:

Sottoreti



- ❑ un IP address può essere localmente modificato usando parzialmente i bit relativi agli host come bit di indirizzo per sottoreti.
- ❑ Una sottorete è definita applicando all'indirizzo IP una maschera di bit chiamata **subnet mask**.
 - Se un bit su una subnet mask è a **1**:
 - ▮ l'equivalente indirizzo IP è interpretato come un network bit (cioè appartiene ad un indirizzo di rete)
 - Se un bit nella maschera è a **0**:
 - ▮ il relativo bit nell'indirizzo IP fa parte di un indirizzo di host (host bit)

RIPASSO:

Sottoreti (effetti della subnet mask)



Classe	IP address	Subnet mask	Interpretazione
B	128.66.12.1	255.255.255.0	host 1 della sottorete 128.66.12.0
B	130.97.16.132	255.255.255.192	host 4 della sottorete 130.97.16.128
C	192.178.16.66	255.255.255.192	host 2 della sottorete 192.178.16.64
B	132.90.132.5	255.255.240.0	host 4.5 della sottorete 132.90.128.0
A	18.20.16.91	255.255.0.0	host 16.91 della sottorete 18.20.0.0

RIPASSO:

Tabella di routing (1)



- ❑ I gateway instradano i dati tra diverse reti
- ❑ Gli host prendono decisioni di instradamento nel modo seguente:
 - Se l'host di destinazione è sulla rete locale, i dati vengono spediti all'host di destinazione;
 - Se l'host di destinazione è su una rete remota, i dati vengono mandati al gateway locale.
- ❑ Il protocollo IP basa le sue decisioni di instradamento su parte **rete** dell'indirizzo IP

RIPASSO:

Tabella di routing (3)

- In ambiente Unix il comando per visualizzare la tabella di routing è:

`netstat -nr`

dove `-r` è l'opzione per la tabella

`n` è per vederla in modo numerico

RIPASSO:

Tabella di routing (4)

Routing tables

Destination	Gateway	Flags	Refcnt	Use	Interface
127.0.0.1	127.0.0.1	UH 1	298	lo0	
default	128.66.12.1	UG 2	50360	en0	
128.66.12.0	128.66.12.2	U	40 98379	en0	
128.66.2.0	128.66.12.3	UG 4	1179	en0	
128.66.1.0	128.66.12.3	UG 10	1113	en0	
128.66.3.0	128.66.12.3	UG 2	1379	en0	
128.66.4.0	128.66.12.3	UG 4	1119	en0	

dove: Destination = rete od host di destinazione

Gateway = gateway da usare per la specifica destinaz. Flags

= U : route in linea e attiva

H : route per host spec. (non per una rete)

G : route che usa un gateway

D : route aggiunta da una ICMP redirect

Refcnt = n.ro di volte che la route è stata usata

Use = n.ro pacchetti trasmessi su quella route

Interface = nome dell'interf. di rete usata per la route

RIPASSO:

Tabella di routing (5)

Source Host

Gateway

Destination Host

Application	
Transport	
Destination	Gateway
128.66.1.0	128.66.12.3
128.66.12.0	128.66.12.2
default	128.66.12.1
Network access	
128.66.12.2	

Destination	Gateway
128.66.1.0	128.66.1.5
128.66.12.0	128.66.12.3
default	128.66.12.1

Network access	
128.66.12.3	128.66.1.5

Application	
Transport	
Destination	Gateway
128.66.1.0	128.66.1.2
default	128.66.1.5

Network access	
128.66.1.2	

128.66.12.0

128.66.1.0

RIPASSO:

Numeri di protocollo (1)

- ❑ Il numero di protocollo è un byte presente nella 3.a parola del datagram header che specifica a quale tipo di protocollo sono passati i dati dal livello IP
- ❑ In Unix i tipi di protocollo sono definiti nel file `/etc/protocols`, visualizzabile con il comando:
 - `cat /etc/protocols`

RIPASSO:

Numeri di protocollo (2)

Internet (IP) Protocols

ip	0	IP	# internet protocol, pseudo protocol number
icmp	1	ICMP	# internet control message protocol
igmp	2	IGMP	# internet group multicast protocol
ggp	3	GGP	# gateway-gateway protocol
tcp	6	TCP	# transmission control protocol
pup	12	PUP	# PARC universal packet protocol
udp	17	UDP	# user datagram protocol

dove: la prima colonna riporta il nome ufficiale del protocollo,
 la seconda colonna il numero di protocollo,
 la terza eventuali aliases (nome maiuscolo)
 dopo il carattere # commenti o nomi estesi

RIPASSO:

Numeri di porta (2)

- Per vedere i numeri di porta definiti in un ambiente Unix dare:

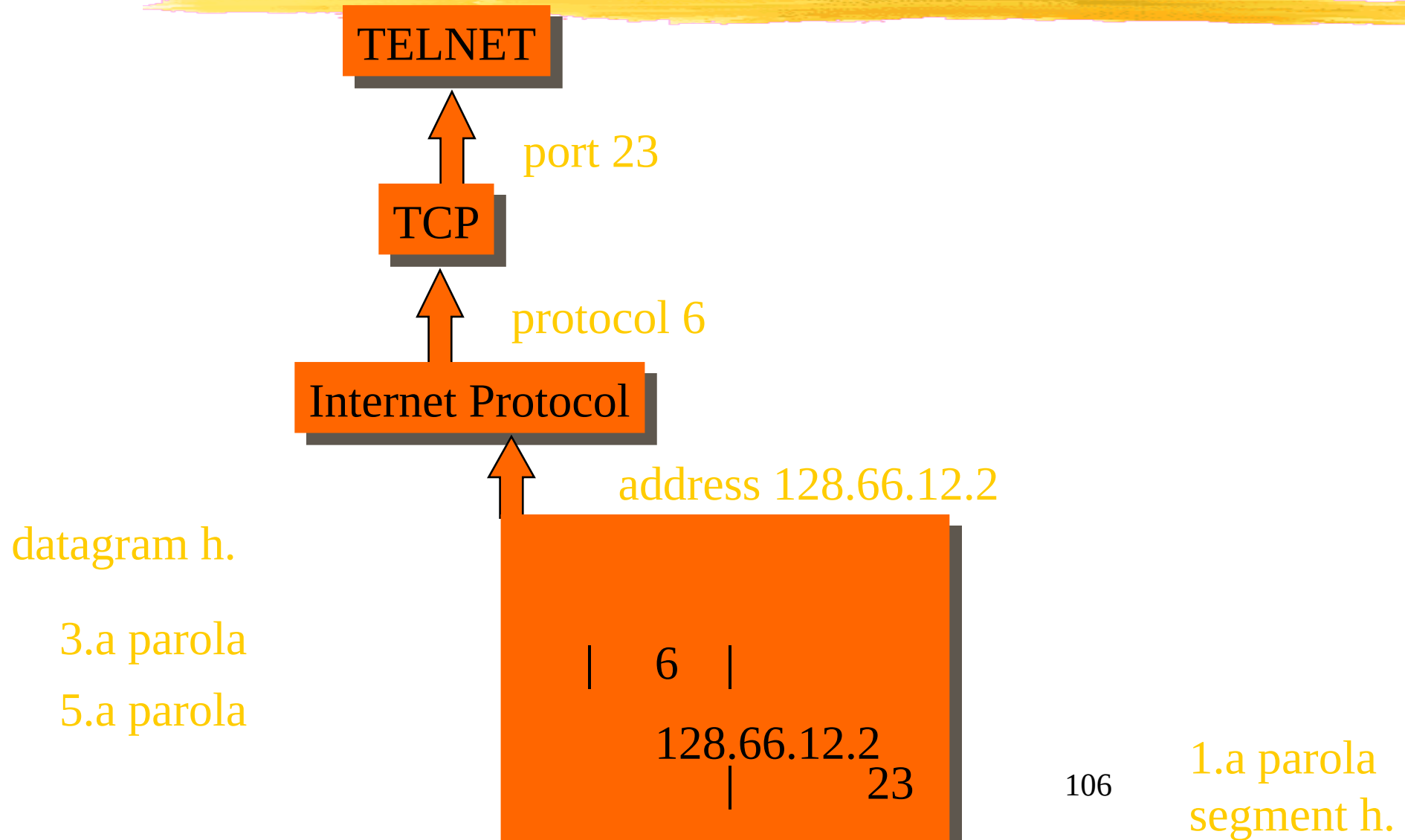
- `cat /etc/services`

```
# Network services
#
echo          7/udp          ping
echo          7/tcp
systat        11/tcp
netstat       15/tcp
ftp-data      20/tcp
ftp           21/tcp
telnet        23/tcp
smtp          25/tcp          mail
...
```

dove: la prima colonna riporta il nome del processo
 la seconda colonna indica in n.ro di porta / tipo di protocollo
 la terza colonna l'applicazione sotto cui gira il processo

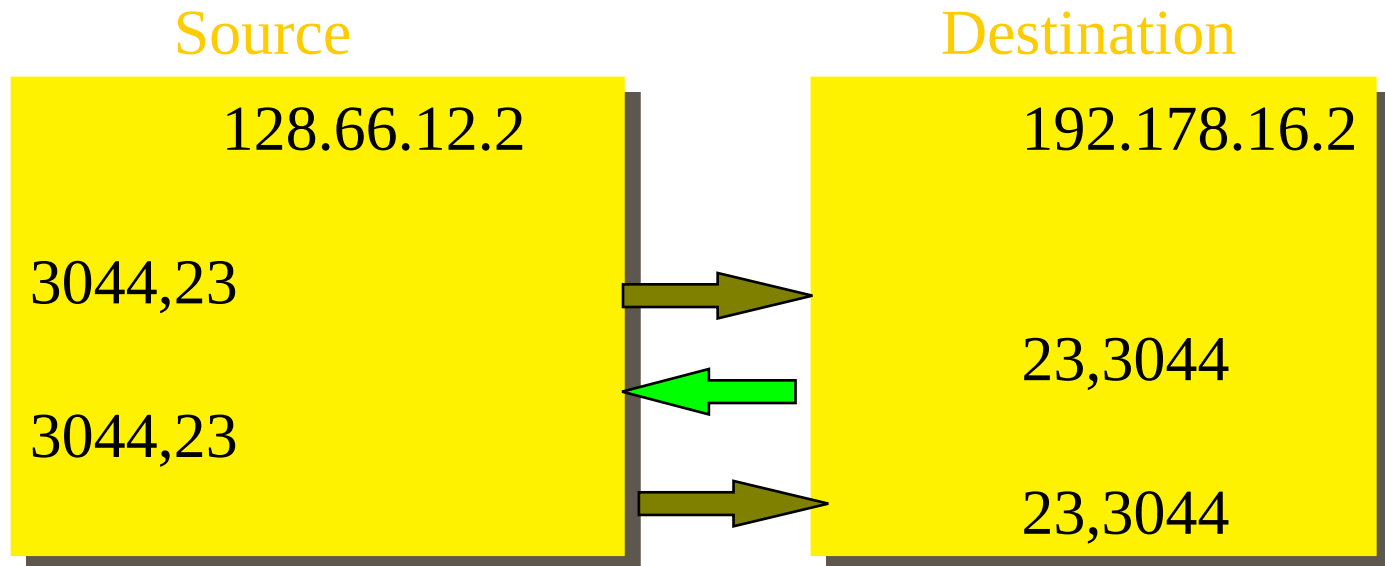
RIPASSO:

Numeri di porta (3)



RIPASSO: Sockets

- ❑ Un Socket è una combinazione di un indirizzo IP e un numero di porta.
- ❑ Un socket identifica univocamente un unico processo di rete sull'intera rete Internet.



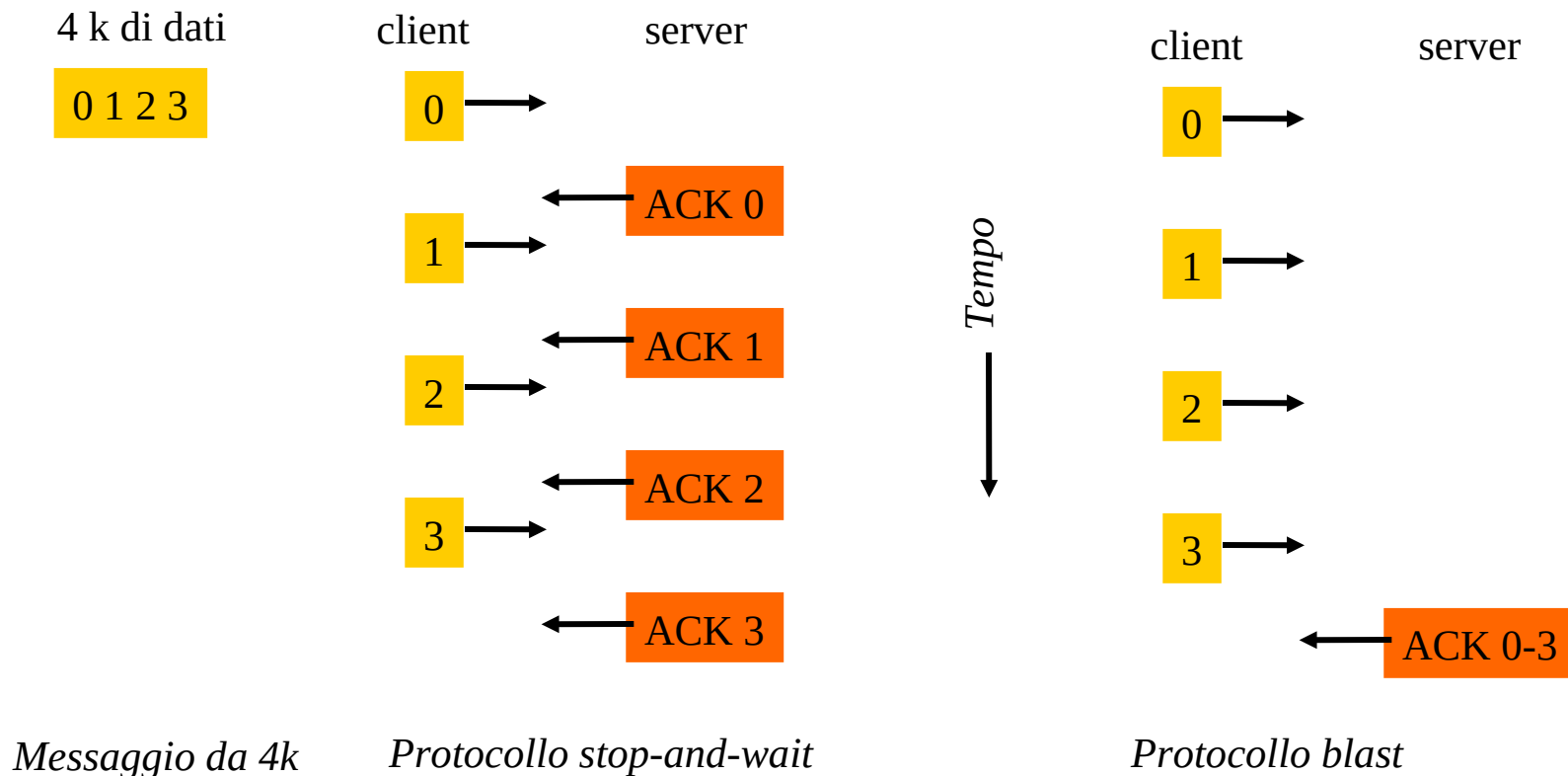
Problemi implementativi: protocolli e ack

- Poiché i protocolli esistenti spesso frammentano molto grosse RPC in numerosi pacchetti (spesso di poco più di 1k), protocolli di tipo **stop-and-wait**, cioè che per ogni pacchetto aspettano l'ack di risposta risultano lenti e carichi di overhead.
- Un'alternativa, chiamata protocollo **blast**, stabilisce che il client spedisca i pacchetti più in fretta possibile, il server spedirà poi un unico ack per tutti i pacchetti ricevuti.
- Questo ultimo tipo di protocollo introduce però qualche problema nel caso che uno o più pacchetti della sequenza vadano persi: nel caso stop-and-wait si ritrasmettevano subito i pacchetti persi, mentre nel blast occorre o ritrasmettere l'intera sequenza o bufferizzare i pacchetti e chiedere al client di ritrasmettergli il pacchetto perso (tecnica detta ripetizione selettiva).

Problemi implementativi: protocolli e ack

□ Esempio:

- supponiamo di voler spedire un messaggio composto da 4k di dati e che il sistema sia in grado di trattare al più pacchetti da 1k

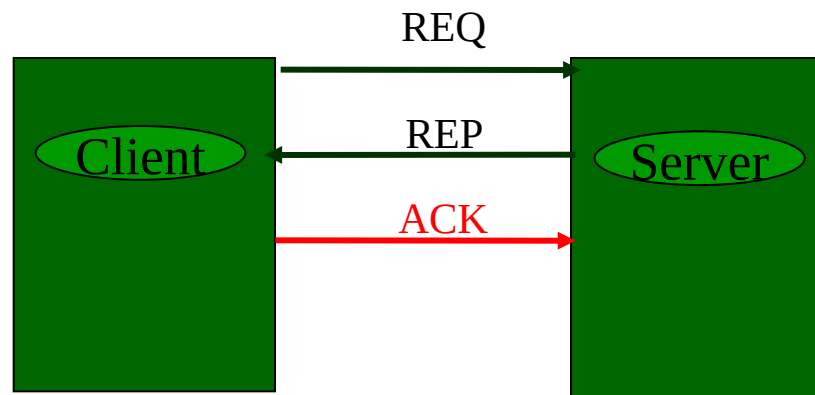


Problemi implementativi: protocolli e ack

- A parte il controllo sull'errore esiste un'altra considerazione che riguarda il **controllo del flusso** (flow control).
- Molti chip di interfaccia sono in grado di spedire pacchetti consecutivi, ma non di ricevere un numero non precisato di pacchetti, per via della limitata capacità del buffer.
- Quando arriva un pacchetto e il ricevente non è in grado di accettarlo si ha un **errore di overrun** (sorpasso) e il pacchetto in arrivo viene perso.
- Errori di overrun sono impossibili in protocolli di tipo stop-and-wait, ma possibilissimi in protocolli blast. In tal caso si può pensare che il mittente imponga un ritardo minimo tra due successivi pacchetti (attesa attiva di qualche microsecondo).

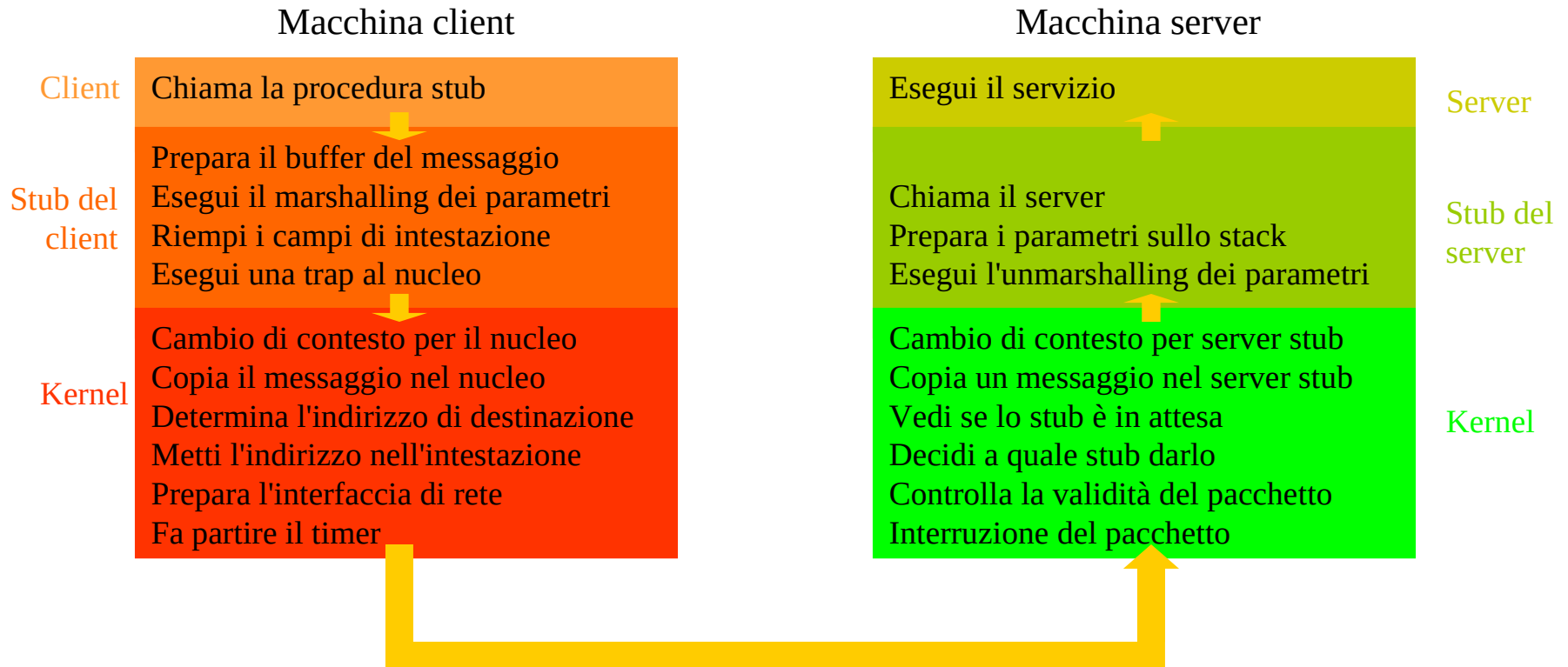
Problemi implementativi: protocolli e ack

- Un ulteriore inconveniente, illustrato sotto, è quello che in caso di protocollo REQ-REP-ACK si perda l'ultimo ACK e il server non sa quando "cestinare" la REP già inviata, poiché non avendo avuto l'ACK non esclude che il client gli possa richiedere una ritrasmissione della REP.
- Una soluzione prevede l'inserimento di un timeout oltre il quale si cestina la REP, un'altra soluzione prevede che una successiva richiesta del client funga da ACK per la richiesta precedente.



Problemi implementativi: cammino critico

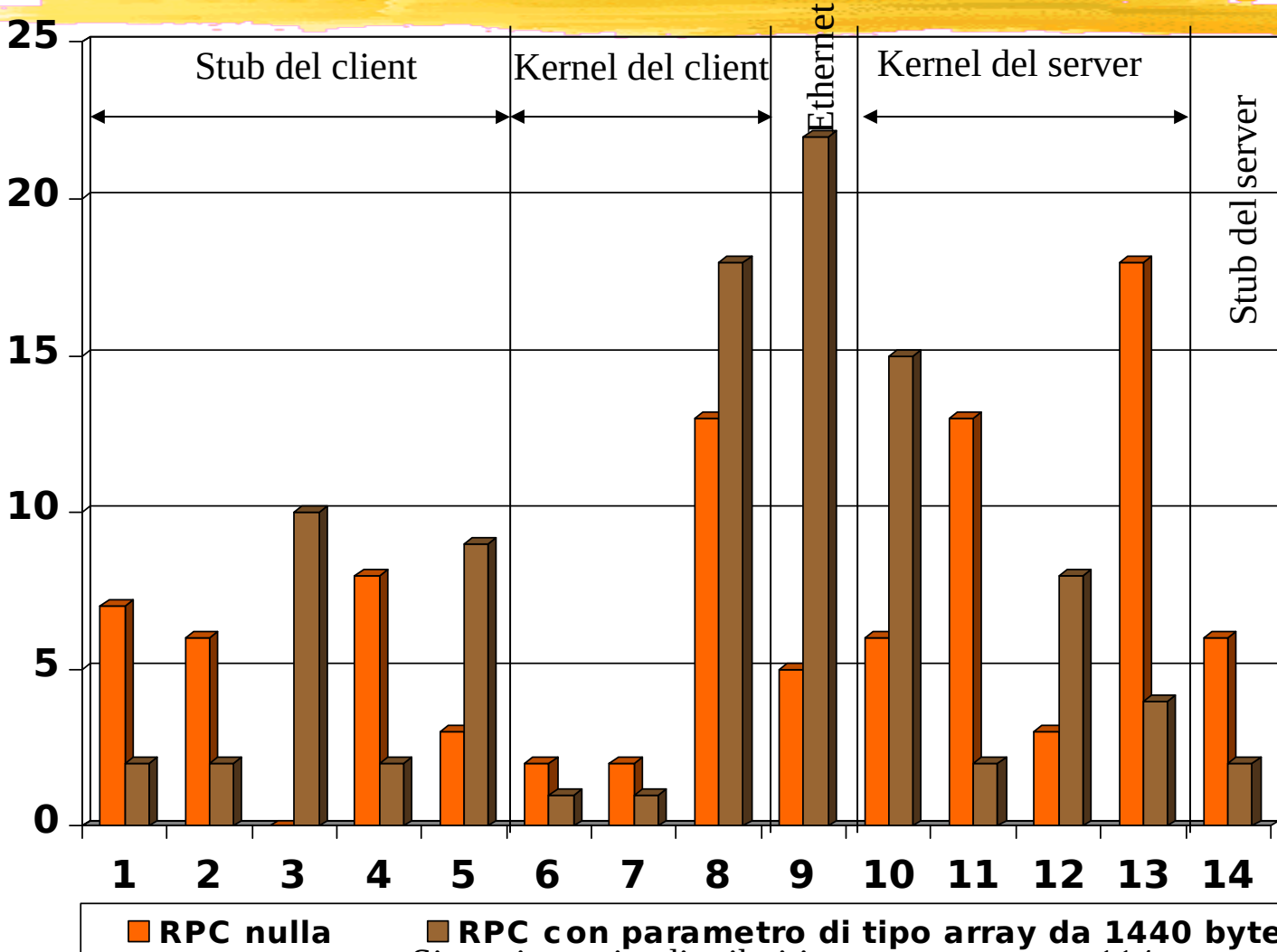
- La sequenza di istruzioni eseguita da ogni RPC di un client verso un server viene detta **cammino critico** (**critical path**).



Problemi implementativi: cammino critico

- In un cammino critico dove si spende più tempo?
 - I risultati, espressi nel seguente istogramma a 14 passi (ciascuno corrispondente ad uno dei passi che vanno dal client al server, il ritorno è simile), sono relativi ad una RPC nulla (cioè senza trasmissione dati) confrontati con quelli di una RPC con un parametro di tipo vettore di 1440 elementi. Sebbene l'overhead fisso sia lo stesso in tutti e due i casi, occorre spendere più tempo, nel secondo caso, per il marshalling del messaggio e per i vari spostamenti dello stesso.

- | | | |
|-----------------------------------|---|---------------------------------|
| 1: Chiama la stub | 6: Trap al nucleo | 11: Routine di servizio dell'in |
| 2: Prendi un buffer per messaggio | 7: Accoda il pacchetto per trasmissione | 12: Calcola il chec |
| 3: Marshalling dei parametri | 8: Sposta il pacchetto al controller con Qbus | 13: Cambio contes |
| spazio ut. | | |
| 4: Riempi le intestazioni | 9: Tempo di trasmissione su Ethernet | 14: Codice del server stub |
| 5: Calcola il checksum UDP | 10: Prendi il pacchetto dal controller | |



Problemi implementativi: copia

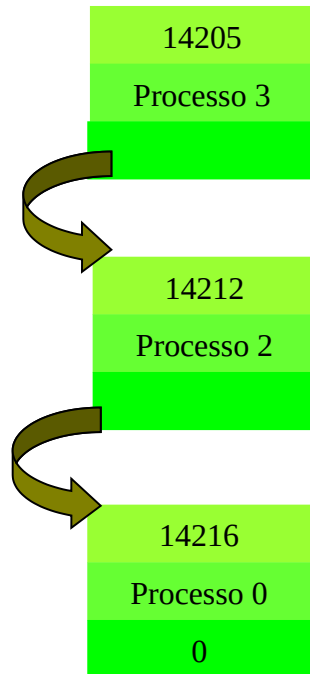
- Un problema che incide fortemente nei tempi di esecuzione di una RPC è la copia della stessa. Sull'ultimo esempio non si vede poiché i buffer fanno parte sia dello spazio utente che dello spazio del kernel, ma in molti sistemi questi due spazi sono disgiunti. Il numero di volte che si deve copiare un messaggio varia da una a otto a seconda dell'hw, del sw e del tipo di RPC.
- Una caratteristica hardware chiamata **scatter-gather** aiuta ad eliminare le operazioni di copia non necessarie. Un chip di rete con tale caratteristiche riesce ad assemblare pacchetti con informazione tutte sparse in vari buffer. In tale modo posso aver l'intestazione separata dai dati della sub del client, poiché è l'hw che raccoglie l'intero pacchetto senza necessità di copie, anche se per il destinatario non sempre è possibile.

Problemi implementativi: gestione dei timer

- ❑ I timer di attesa per risposte e ACK hanno gestioni molto spesso onerose che vanno a influire sui tempi totali. Esistono due metodi per gestire tali timer.
- ❑ Il primo è una **lista ordinata** contenente il numero di processo e il tempo in cui scade uno specifico timer. Lo svantaggio di tale tecnica è la gestione della lista (inserimento e cancellazione).
- ❑ Il secondo è una tabella dei processi che prevede l'inserimento del tempo in cui scade lo specifico timer. Qui la gestione è meno onerosa, ma si prevede una scansione ad intervalli di tempo di tutta la lista. Algoritmi di questo tipo si dicono di tipo **sweep** (a spazzola).

Problemi implementativi: gestione dei timer

Tempo attuale 14200



Timeout in una lista ordinata

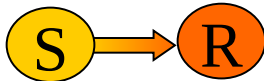
Tempo attuale 14200

0	14216
1	0
2	14212
3	14205

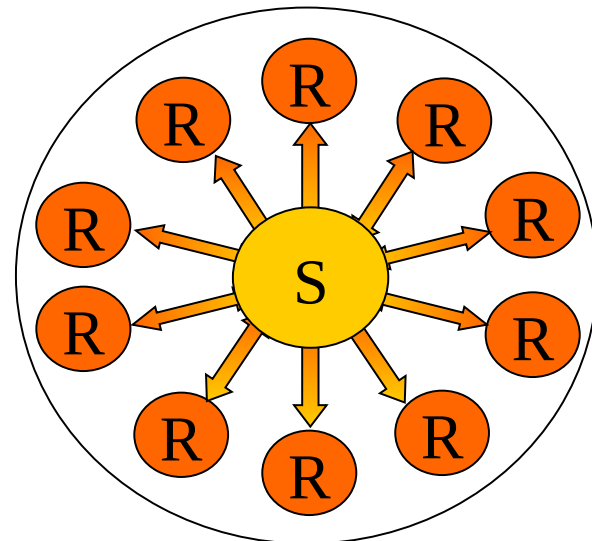
Timeout nella tabella dei processi

Comunicazioni di gruppo

- Un **gruppo** è una collezione di processi che agiscono insieme secondo modalità determinate dal sistema e specificate dall'utente.
- La proprietà principale che hanno i gruppi è che quando un messaggio viene spedito al gruppo, viene ricevuto da tutti i componenti del gruppo. Questa è una forma di comunicazione **uno-a-molti**, e si pone in contrasto con la comunicazione **punto-a-punto**.



Comunicazione punto-a-punto



Comunicazione uno-a-molti

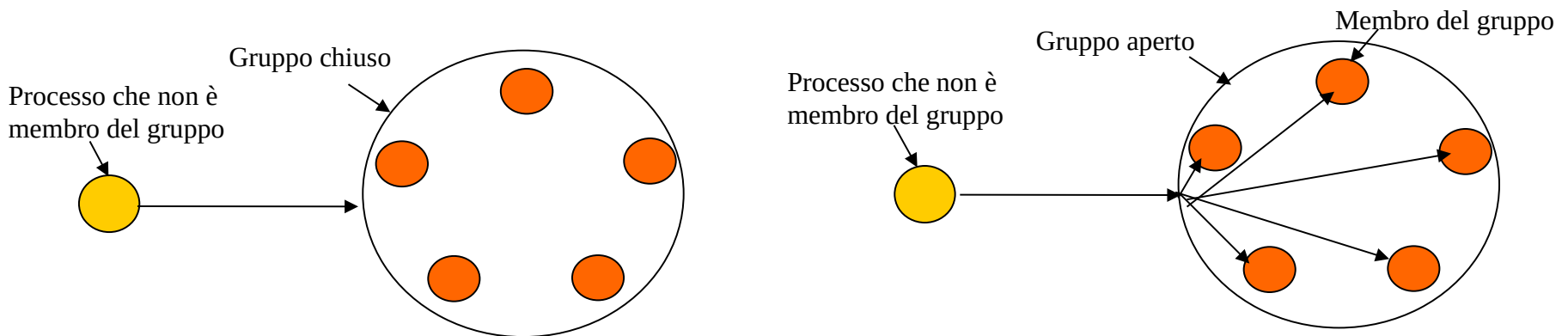
Comunicazioni di gruppo



- ❑ L'implementazione della comunicazione di gruppo dipende molto dall'hardware. Su certe reti è possibile creare un indirizzo di rete speciale che può essere riconosciuto da più di una macchina. Questa tecnica viene detta **multicasting**.
- ❑ Reti che non hanno il multicast, hanno a volte almeno il **broadcast**, cioè permettono che pacchetti che contengono un particolare indirizzo vengano spediti a tutte le macchine.
- ❑ In reti che non hanno né multicast, né broadcast, si può attuare la comunicazione di gruppo facendo sì che il mittente trasmetta un pacchetto separato per ogni membro del gruppo (**unicasting**).

Gruppi chiusi ed aperti

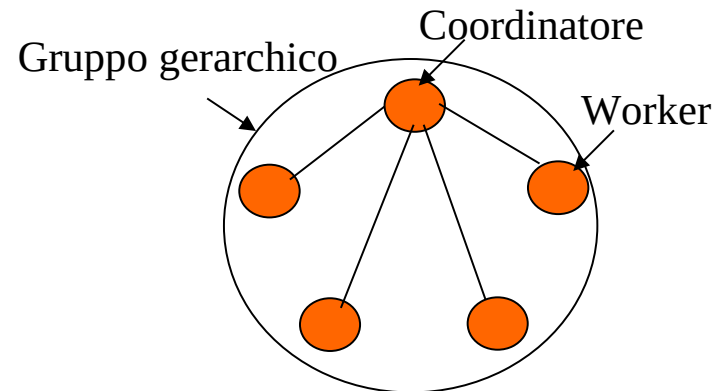
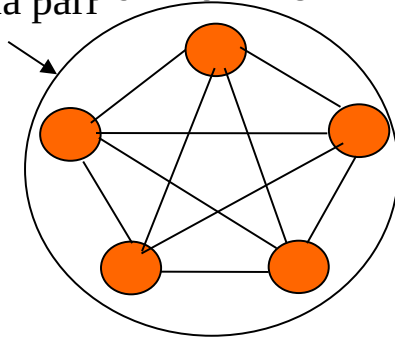
- I sistemi che mettono a disposizione la comunicazione di gruppo possono essere classificati a seconda di quali possono essere i mittenti e quali i destinatari delle comunicazioni.
- Alcuni sistemi mettono a disposizione i **gruppi chiusi**: in cui solo i membri del gruppo possono spedire messaggi al gruppo stesso. Chi non è membro non può spedire messaggi al gruppo anche se può spedire messaggi ai singoli membri del gruppo.
- Altri sistemi mettono a disposizione i gruppi aperti: ciascun processo del sistema può spedire un messaggio ad un qualunque gruppo.



Gruppi alla pari e gruppi gerarchici

- In alcuni gruppi tutti i processi sono uguali (**gruppi alla pari**): le decisioni sono prese in maniera collettiva. Il gruppo alla pari è simmetrico e non ha un singolo punto di caduta. La gestione delle decisioni è però complessa.
- In altri gruppi esiste invece una gerarchia (**gruppi gerarchici**): un processo fa da coordinatore e tutti gli altri si comportano da worker. La perdita del coordinatore porta ad un blocco immediato, ma fino a che esso continua a girare può prendere decisioni senza infastidire nessun altro.

Gruppo alla pari

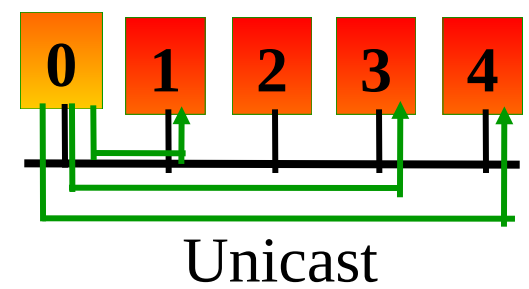
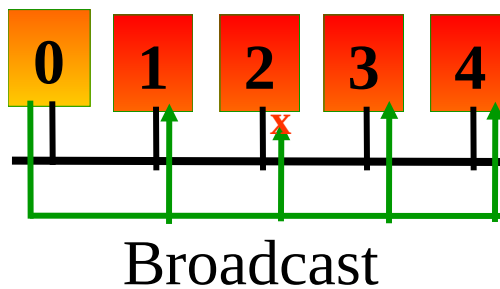
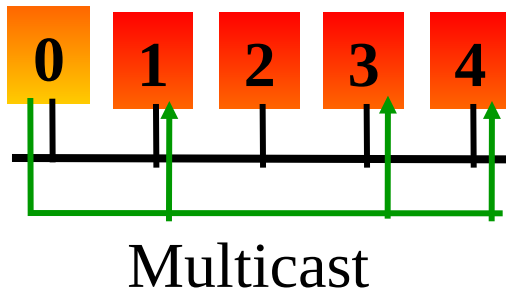


Appartenenza al gruppo

- Quando è presente la comunicazione di gruppo è necessario avere un metodo per creare e cancellare dei gruppi, così come per associarsi a o abbandonare un gruppo. Una possibilità è quella di avere un **server dei gruppi** (**group server**) al quale possono essere indirizzate tutte le richieste.
- Questo ha dei vantaggi gestionali nel mantenimento di una base di dati completa relativa a tutti i gruppi, ma l'accentramento di gestione comporta lo svantaggio del singolo punto di caduta (se il group server cade, la gestione dei gruppi cessa).
- L'alternativa al group server consiste nel gestire l'appartenenza ai gruppi in maniera distribuita. Tale approccio deve prevedere che anche i gruppi chiusi devono aprirsi per la fase di associazione, inoltre potrebbero causarsi disallineamenti all'atto della caduta o dissociazione di un membro appartenente a uno o più gruppi.

Indirizzamento dei gruppi

- Un primo metodo prevede un indirizzo unico per il gruppo. Se l'hardware della rete permette il multicast i messaggi indirizzati al gruppo verranno spediti alle macchine del gruppo. Se non è possibile attuare il multicast si può usare il broadcast ed inviare i messaggi a tutte le macchine della rete, sarà poi il kernel delle macchine non del gruppo a scartarli. Infine se non si ha a disposizione né multicast, né broadcast il kernel della macchina mittente avrà una lista delle macchine appartenenti al gruppo e spedisce ad ognuna di queste messaggi punto-a-punto.



Indirizzamento dei gruppi

- Un secondo metodo di indirizzamento dei gruppi è quello di richiedere che il mittente metta a disposizione una lista esplicita di destinazioni (es. indirizzi IP). Quando si usa questo metodo il parametro della chiamata alla *send* è un puntatore alla lista di indirizzi. Lo svantaggio è che i processi membri del gruppo devono essere a conoscenza di chi appartiene al gruppo.
- Un terzo e nuovo metodo di indirizzamento, chiamato **indirizzamento predicativo**, prevede che ciascun messaggio spedito con uno dei metodi precedenti contenga al suo interno un predicato (un'espressione booleana) da valutare. Se il predicato è VERO il messaggio viene accettato, se vale FALSO viene scartato. (Per esempio si può spedire un messaggio alle sole macchine che hanno 64 MB di memoria e vogliono accettarlo).

Primitive di send e receive

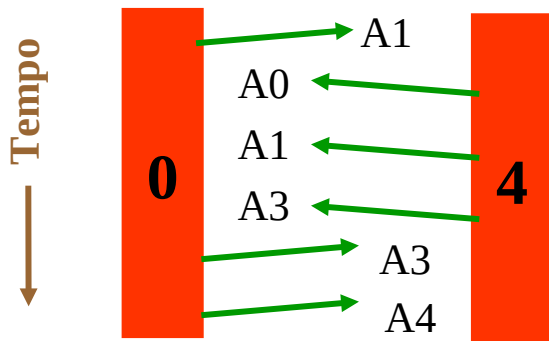
- ❑ Le primitive send e receive nascono per la comunicazione punto_a_punto.
- ❑ Si può usare le primitive send con speciali destinazioni, se l'indirizzo è quello di gruppo il messaggio viene spedito a tutti i componenti del gruppo.
- ❑ Le primitive send e receive possono anche essere adattate alla comunicazione di gruppo creando nuove procedure di libreria quali una *group_send* e una *group_receive*.
- ❑ Per dare alla comunicazione un sapore di RPC, un processo dopo aver spedito un messaggio, deve ripetutamente chiamare una *getreply* per poter raccogliere tutte le risposte una alla volta.

Atomicità

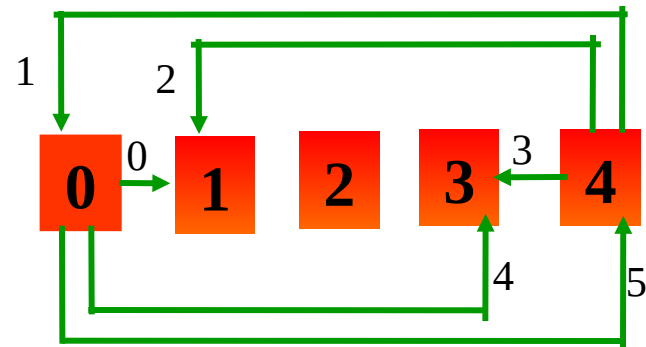
- ❑ La caratteristica delle comunicazioni di gruppo è che tutti i componenti del gruppo devono ricevere il messaggio seguendo la proprietà del "tutto-o-niente" che viene detta **atomicità** o **broadcast atomico**.
- ❑ Nella realtà le comunicazioni viste fino ad ora non assicurano l'atomicità in modo trasparente.
- ❑ Esiste però un algoritmo che dimostra che il broadcast atomico è almeno possibile: il mittente inizia spedendo un messaggio a tutti i membri del gruppo; vengono caricati i timer e il messaggio viene ritrasmesso se necessario. Quando un processo riceve un messaggio, se non lo ha già visto un'altra volta, spedisce anch'esso il messaggio a tutti i membri del gruppo. E così via. Sicuramente alla fine tutti i membri avranno il messaggio.

Ordinamento dei messaggi

- Per rendere le comunicazioni di gruppo facili da capire ed usare occorre che siano verificate due proprietà: atomicità e ordinamento dei messaggi.
- Il problema dell'ordinamento dei messaggi può essere rappresentato dall'esempio:



Messaggi spediti dal processo 0 e 4
intercalati nel tempo

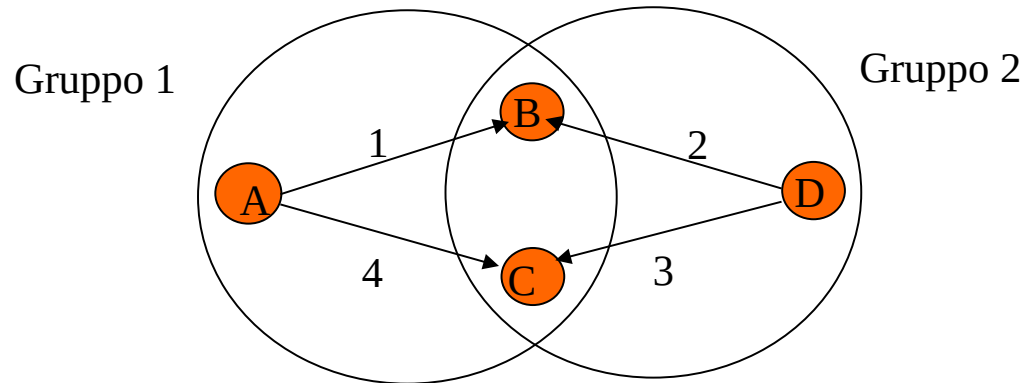


Rappresentazione grafica dei 6 messaggi
con il tempo di arrivo

- Per avere un **ordinamento globale del tempo** occorre spedire i messaggi con ritardi anche a livello di programmazione.
- Un **ordinamento consistente del tempo** non assicura il corretto ordinamento ma tutti i componenti del gruppo ricevono nello stesso ordine.

Gruppi che si sovrappongono

- Un processo può essere membro di diversi gruppi nello stesso istante: ciò può provocare problemi di inconsistenza.
- Esempio:

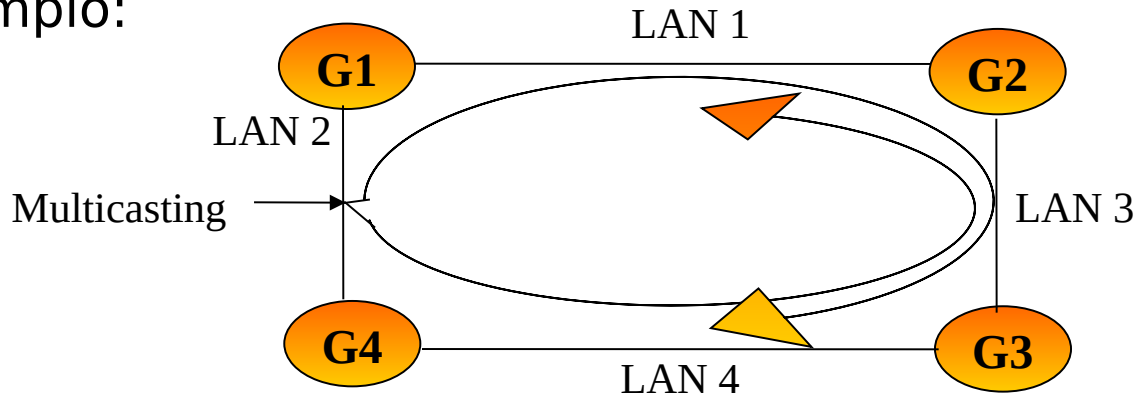


I processi B e C prendono messaggi da A e D in ordine inverso

- In questo caso pur essendoci un ordinamento globale del tempo all'interno di ciascun gruppo, non c'è coordinamento tra gruppi diversi.

Scalabilità

- ❑ Cosa succede se i membri di un gruppo sono moltissimi? O quando si hanno migliaia di gruppi? O quando non è più sufficiente una LAN, ma occorrono diverse LAN interconnesse tramite gateway?
- ❑ La presenza di gateway complica molto il multicasting.
- ❑ Esempio:



- ❑ La propagazione di pacchetti multicast da parte dei gateway porta ad una crescita esponenziale dei pacchetti.
- ❑ Un ulteriore problema delle reti di reti è che è possibile avere più pacchetti sulla rete contemporaneamente, eliminando così l'utile ordinamento globale assoluto.

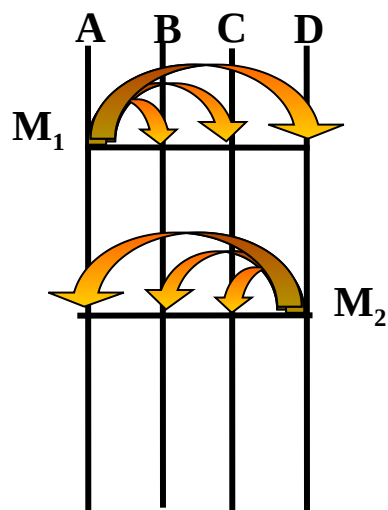
Comunicazioni di gruppo in ISIS

- ❑ **ISIS** è un sistema sviluppato a Cornell da Birman e Joseph che si presenta come un insieme di strumenti che servono a costruire applicazioni distribuite.
- ❑ ISIS non è un sistema operativo completo, ma è un insieme di programmi che girano su UNIX o altri s.o. esistenti.
- ❑ Pur essendo "datato" è comunque interessante da studiare perché è stato ampiamente descritto in letteratura e perché è stato usato per numerose applicazioni reali.
- ❑ L'idea di ISIS è la **sincronia** e le principali primitive di comunicazione sono forme diverse di **broadcast atomico**.

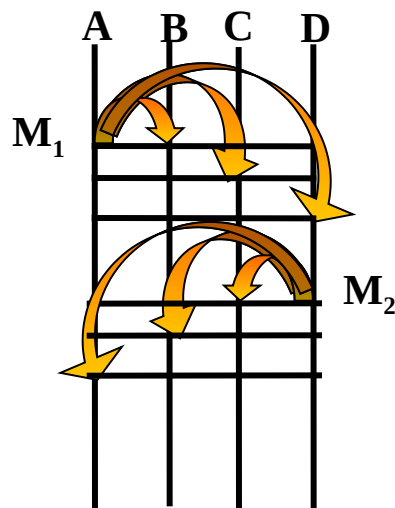
Comunicazioni di gruppo in ISIS

- ❑ Un **sistema sincrono** è un sistema nel quale gli eventi accadono in maniera strettamente sequenziale e dove ciascun evento (per esempio un broadcast) si completa in un tempo praticamente nullo.
- ❑ Un **sistema lascamente sincrono** è un sistema nel quale gli eventi impiegano una quantità finita di tempo, ma tutti gli eventi appaiono nello stesso ordine a tutte le parti del sistema. In particolare i processi ricevono i messaggi nello stesso ordine.
- ❑ Un **sistema virtualmente sincrono** è un sistema nel quale il vincolo dell'ordinamento è stato rilasciato, ma in modo tale per cui, in determinate circostanze, questo rilascio non ha importanza.

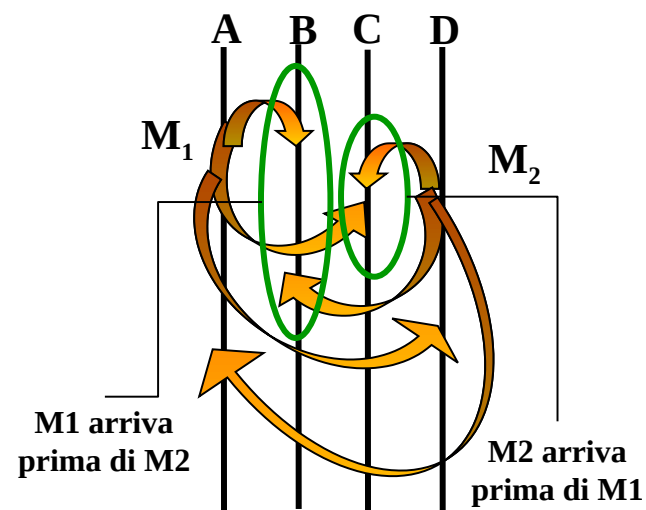
Comunicazioni di gruppo in ISIS



Sistema sincrono



Sistema lascamente sincrono



Sistema virtualmente sincrono

Comunicazioni di gruppo in ISIS

- ❑ In un sistema distribuito due eventi si dicono **in relazione causale** se la natura del comportamento del secondo potrebbe essere in relazione con il primo.
- ❑ In un sistema distribuito due eventi che non sono in relazione si dicono **concorrenti**.
- ❑ Ciò che significa realmente la sincronia virtuale è che se due messaggi sono in relazione causale, tutti i processi devono riceverli nello stesso ordine. Se essi sono concorrenti, non ci sono garanzie ed il sistema è libero di trasmetterli in un diverso ordine ai vari processi.
- ❑ Nei casi in cui ha importanza i messaggi vengono trasmessi con lo stesso ordine, altrimenti se non ne ha possono essere spediti con un ordine diverso.

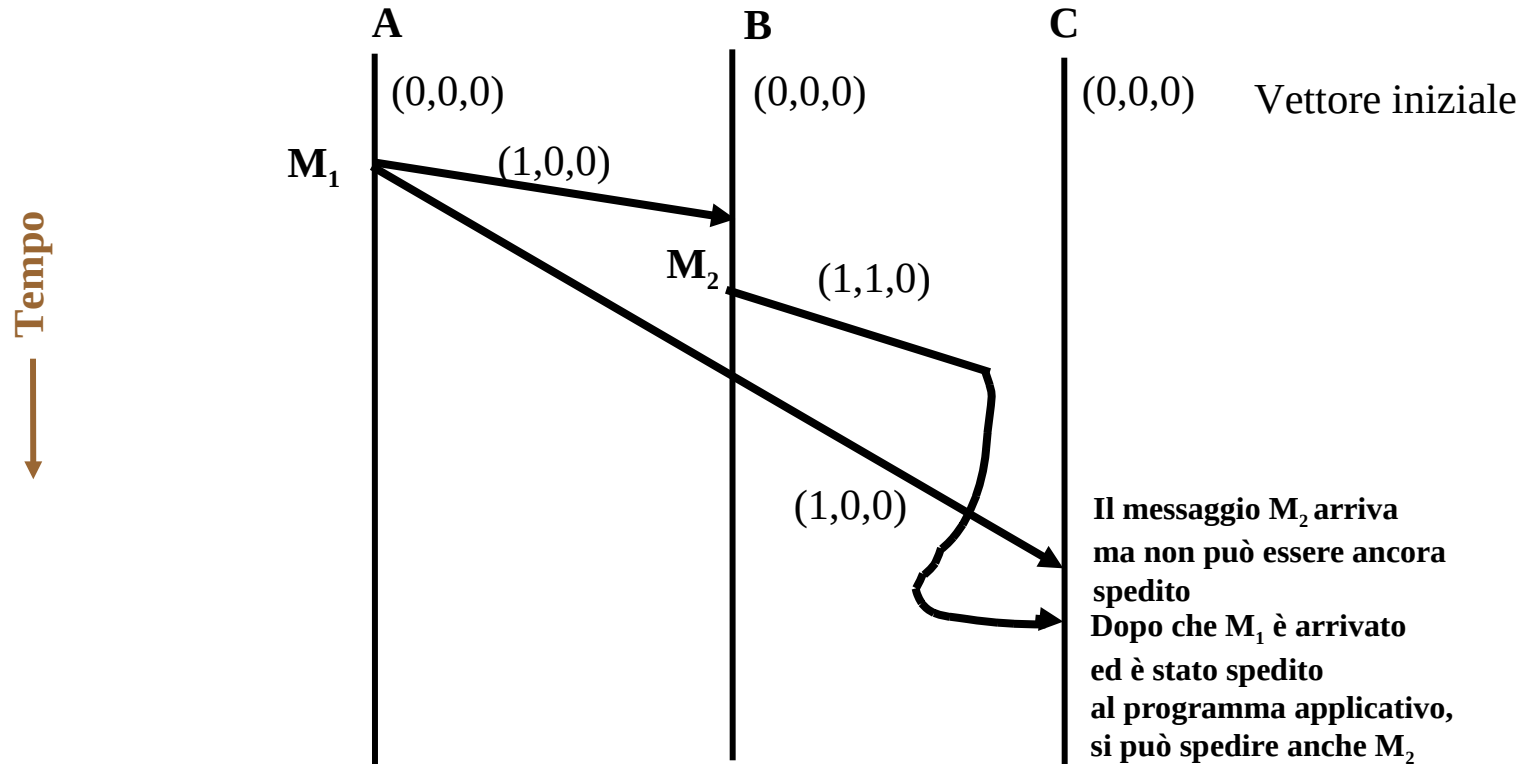
Le primitive di comunicazione in ISIS

- ❑ Le primitive di broadcast usate in ISIS sono:
 - ABCAST
 - CBCAST
 - GBCAST
- ❑ **ABCAST** attua una comunicazione lascamente sincrona e viene usata per trasmettere dati ai membri di un gruppo.
- ❑ **CBCAST** attua una comunicazione virtualmente sincrona e anch'essa viene usata per trasmettere dati in un gruppo.
- ❑ **GBCAST** assomiglia a ABCAST eccetto che viene utilizzata per gestire l'appartenenza ai gruppi e non per trasmettere dati.

Le primitive di comunicazione in ISIS

- Analizziamo in dettaglio la CBCAST:
 - Se un gruppo ha n membri, ciascun processo mantiene un vettore a n componenti, una per ogni membro del gruppo. L' i -esimo elemento del vettore è il numero dell'ultimo messaggio ricevuto in sequenza dal processo i . I vettori vengono gestiti dal supporto a tempo di esecuzione e non dagli stessi processi utente, e vengono inizializzati a zero. Quando un processo ha un messaggio da spedire esso incrementa il suo elemento nel proprio vettore e spedisce il vettore come parte del messaggio.

Le primitive di comunicazione in ISIS

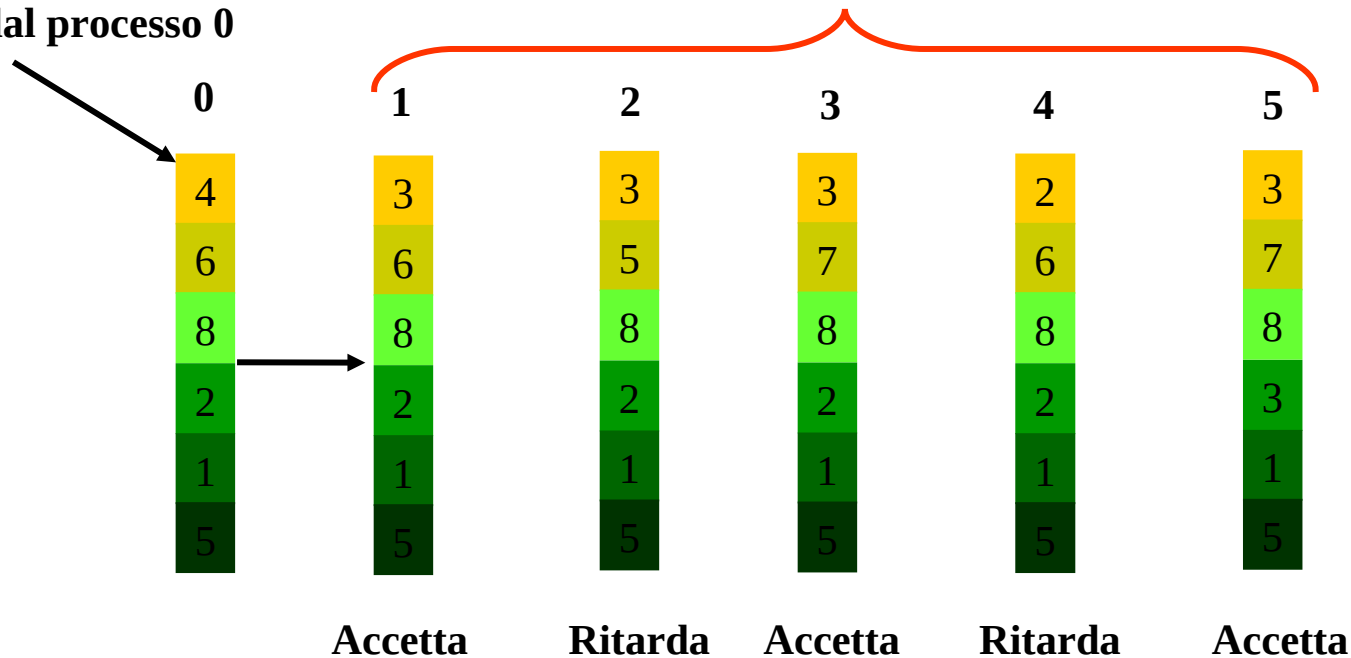


I messaggi possono essere spediti solo quando tutti i messaggi precedenti in relazione casuale sono stati spediti

Le primitive di comunicazione in ISIS

Vettore in un messaggio
spedito dal processo 0

Stato dei vettori delle altre macchine



Esempi di vettori usati dalla CBCAST

High Level Architecture

A thick, horizontal, yellow brushstroke with a textured, painterly appearance, spanning most of the width of the slide.

Introduzione

Indice



- ❑ High Level Architecture (HLA): Background
- ❑ Regole
- ❑ Specifica dell'Interfaccia
 - Panoramica
 - Sottoscrizione basata sulla Classe
 - Aggiornamenti degli Attributi

HLA: Motivi

Il Dipartimento della Difesa degli U.S.A. (Department of Defense - DoD) è stato tormentato da “simulazioni a tubo di stufa”: simulazioni individuali disegnate e tagliate per una specifica applicazione

- ❑ Non facilmente adattabili ad altri usi, risultandone un riuso limitato del software, molte duplicazioni di sforzi
- ❑ Non sfruttano facilmente le proprietà sviluppate in altri modelli e programmi di simulazione del DoD

Obiettivo della High Level Architecture: definire una infrastruttura di simulazione comune che supporti l'interoperabilità e il riuso delle simulazioni della difesa

- Simulazioni Analitiche (per es. wargames)
- Training (platform-level, command-level)
- Test e Stime

Simulazione Distribuita nel DoD



- ❑ SIMNET (SIMulator NETworking) (1983-89)
 - Progetto DARPA e U.S. Army
 - Simulatori di battaglie interattive su rete
 - Da alcune decine a centinaia di simulatori
- ❑ DIS (Distributed Interactive Simulation) (1990-96)
 - Rapida espansione basata sul successo di SIMNET
 - Decine di migliaia di entità simulate
 - Standard IEEE
- ❑ Aggregate Level Simulation Protocol (ALSP) (ultimi anni '80 e '90)
 - Applicazione dei concetti delle simulazioni in rete ai modelli di wargaming

HLA Development Process

- ❑ 10/93-1/95: tre proposte di architetture sviluppate nell'industria
- ❑ 3/95: DMSO (Defense Modeling and Simulation Office) costituisce l'Architecture Management Group (AMG)
- ❑ 3/95-8/96: sviluppo dell'architettura di fondo
 - AMG costituisce i gruppi tecnici di lavoro (IFSpec, time management, data distribution management)
 - Prototipi della Run-Time Infrastructure (RTI)
 - Federazioni dei prototipi: preparazione del livello platform, preparazione del livello command, test e stima, analisi analitica
- ❑ 8/96-9/96: adozione dell'architettura di fondo
 - Approvazione da parte dell'AMG, Executive Council for Modeling and Simulation (EXCIMS), U.S. Under Secretary of Defense (Acquisition and Technology)
 - 10 September, 1996: Baseline HLA viene approvato come architettura tecnica standard per tutte le simulazioni del DoD degli U.S.A.
- ❑ 9/96-oggi: continua lo sviluppo e la standardizzazione
 - Diversi livelli di adozione Varying levels of adoption
 - Commercializzazione del software RTI
 - Standardizzazione (IEEE 1516)

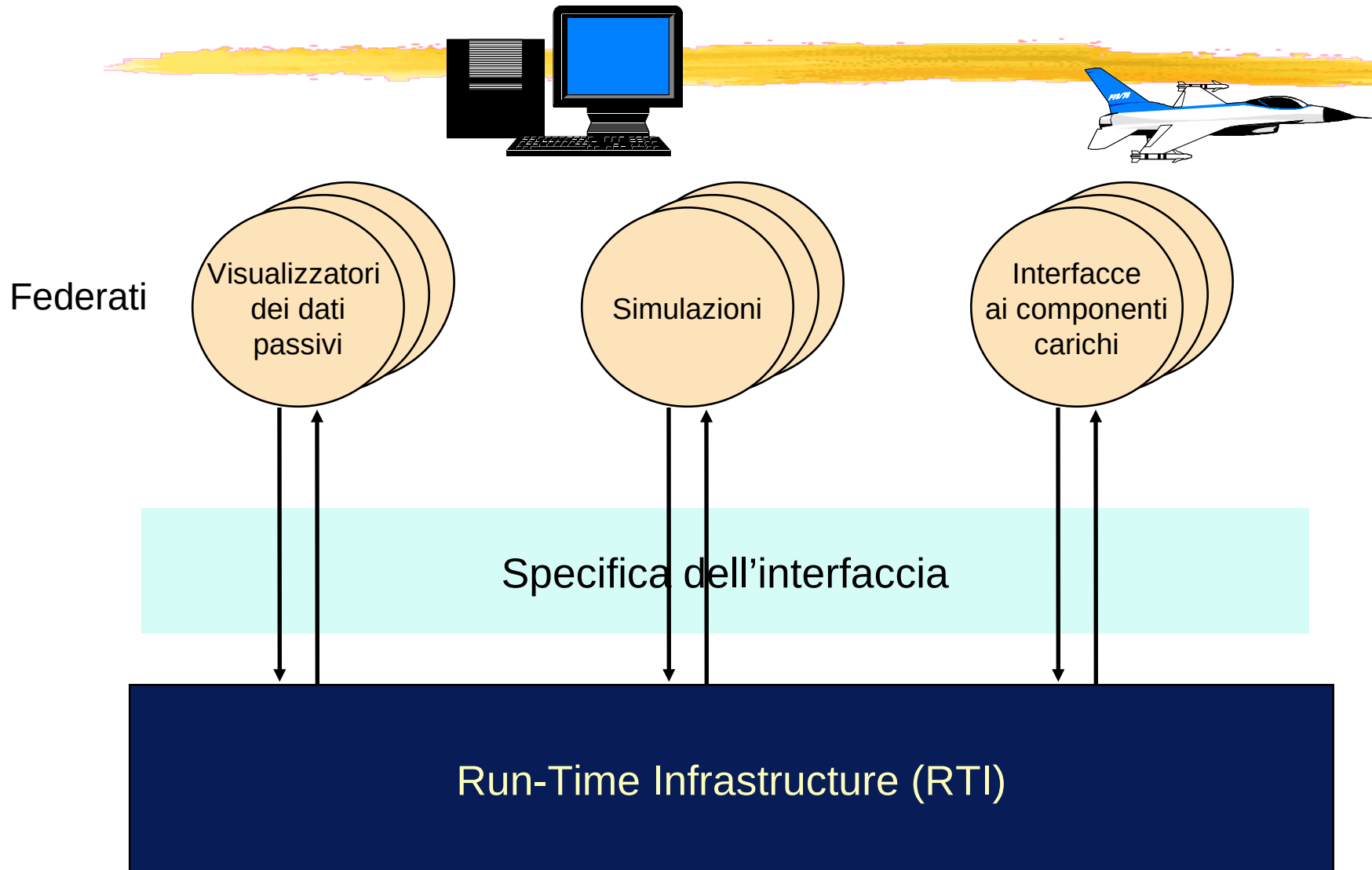
High Level Architecture (HLA)

- Basata su un approccio composto di “sistema di sistemi”
 - Nessuna singola simulazione può soddisfare a tutte richieste degli utenti
 - Supporta l'interoperabilità e il riuso tra le simulazioni del DoD
- *federazioni* di simulazioni (*federates*)
 - Simulazioni software pure e semplici
 - Simulazioni del tipo human-in-the-loop (simulatori virtuali)
 - Componenti carichi (es. Sistemi di armi strumentali)

HLA consiste di

- **Regole** che le simulazioni (*federates*) devono seguire per ottenere una corretta interazione durante una esecuzione della federazione
- **Object Model Template (OMT)** definisce il formato per specificare l'insieme degli oggetti comuni usati da una federazione (*federation object model*), i loro attributi, e le relazioni fra essi
- **Interface Specification (IFSpec)** fornisce una interfaccia alla *Run-Time Infrastructure (RTI)*, che tiene insieme i federati durante l'esecuzione del modello

Una Federazione HLA



Regole della Federazione



- 1 Le Federazioni avranno un HLA Federation Object Model (FOM), documentato in accordo con il HLA Object Model Template (OMT).**
- 2 In una federazione tutta la rappresentazione delle istanze degli oggetti associati alla simulazione sarà nei federati, non nella infrastruttura a runtime (RTI).**
- 3 Durante una esecuzione della federazione tutti gli scambi dei dati di FOM tra i federati associati accadranno via RTI.**
- 4 Durante una esecuzione della federazione i federati associati interagiranno con RTI in conformità con la specifica dell'interfaccia HLA.**
- 5 Durante una esecuzione della federazione un attributo dell'istanza ad un dato tempo apparterrà al più ad un federato.**

Regole dei Federati

- 6 I Federati avranno un HLA Simulation Object Model (SOM) documentato in accordo con il HLA Object Model Template (OMT).**
- 7 I Federati potranno aggiornare e /o riflettere ogni attributo dell'istanza e/o ricevere interazioni come specificato nel loro SOM.**
- 8 I Federati potranno trasferire e/o accettare possessi di attributi delle istanze in maniera dinamica durante un'esecuzione della federazione come specificato nei loro SOM.**
- 9 I Federati potranno variare le condizioni (es. le soglie) sotto cui essi forniranno aggiornamenti degli attributi delle istanze come specificato nel loro SOM.**
- 10 I Federati potranno gestire il tempo locale in modo che permetta loro di coordinare gli scambi di dati con altri membri di una federazione.**

Specifica dell'interfaccia

Categoria	Funzionalità
Federation Management	Crea e cancella le esecuzioni della federazione. Associa e rilascia le esecuzioni della federazione Controlla: checkpoint, pause, resume, restart
Declaration Management	Stabilisce l'intento a pubblicare e sottoscrivere gli attributi degli oggetti e le interazioni
Object Management	Crea e cancella istanze degli oggetti Controlla la pubblicazione degli attributi e delle interazioni Crea e cancella le riflessioni degli oggetti
Ownership Management	Trasferisce il possesso degli attributi degli oggetti
Time Management	Coordina l'avanzamento del tempo logico e le sue relazioni con il tempo reale
Data Distribution Management	Supporta un routing efficiente dei dati

Alternative del Message Passing

- ❑ I meccanismi del message passing tradizionale: Il sender identifica esplicitamente i receiver attraverso:
 - Il processo di destinazione, la port, ecc.
 - Scarsamente adottati per le simulazioni federate
- ❑ Broadcast
 - Receiver scarta i messaggi che non lo interessano
 - Usata in SIMNET, DIS (inizialmente)
 - Non scala bene su grandi federazioni
- ❑ Meccanismo di Publication / Subscription
 - Analogo ai newsgroup
 - Il Produttore delle informazioni ha modo di descrivere il dato che sta producendo
 - Il Receiver ha modo di descrivere il dato che è interessato a ricevere
 - Usato in High Level Architecture (HLA)

Quindi



- ❑ La High Level Architecture è un esempio di un approccio per realizzare simulazioni distribuite
- ❑ Le HLA Rules definiscono i principi generali che pervadono l'intera architettura
- ❑ HLA Interface Specification definisce un insieme di servizi a run-time per supportare simulazioni distribuite
- ❑ Data distribution è basata su un meccanismo di publication / subscription

Data Distribution

A horizontal yellow brushstroke with a textured, painterly appearance, spanning most of the width of the slide.

Contenuti



- ❑ Concetti fondamentali
 - Name space
 - Description expressions
 - Interest expressions
- ❑ Static Data Distribution: HLA Declaration Management
 - Class-based filtering

Background

- Domanda : Quando un processo genera informazioni (es. aggiornamenti di stato) che possono essere di interesse per altri processi, chi dovrebbe ricevere il messaggio?
- Esempio: veicoli mobili in un ambiente virtuale
 - Veicoli mobili inviano messaggi di “aggiornamento” che indicano la nuova posizione
 - Ogni veicolo che può “vedere” il veicolo in movimento dovrebbe ricevere il messaggio
 - Come fa il sender/RTI a conoscere quali altri federati dovrebbero ricevere il messaggio?
 - ✧ Data distribution è essenzialmente un problema di instradamento dei messaggi

Pimitive di Comunicazione

□ Unicast

- Un sender, il messaggio è ricevuto da una destinatione

□ Broadcast

- Un sender, il messaggio è ricevuto da tutte le destinationi

□ Multicast

- Un sender, il messaggio è ricevuto da molte destinazioni (ma non necessariamente tutte)
- Operazioni (analoghe ai newsgroup)
 - ✧ Join group
 - Leave group
 - Send message to group
- Possono essere implementate con unicast, o multicast di rete
- Best effort vs. multicast affidabile

Data Distribution



- ❑ Domanda: Chi riceve ogni messaggio che viene inviato?
- ❑ Approccio: Inviare in broadcast ogni messaggio, il receiver è responsabile di filtrare (eliminare) i messaggi non desiderati
- ❑ $O(N^2)$ messaggi con N federati; può usare una grande quantità di ampiezza di banda di comunicazione
- ❑ Il tempo speso per ricevere e filtrare messaggi non ricercati diventa un collo di bottiglia

Data Distribution

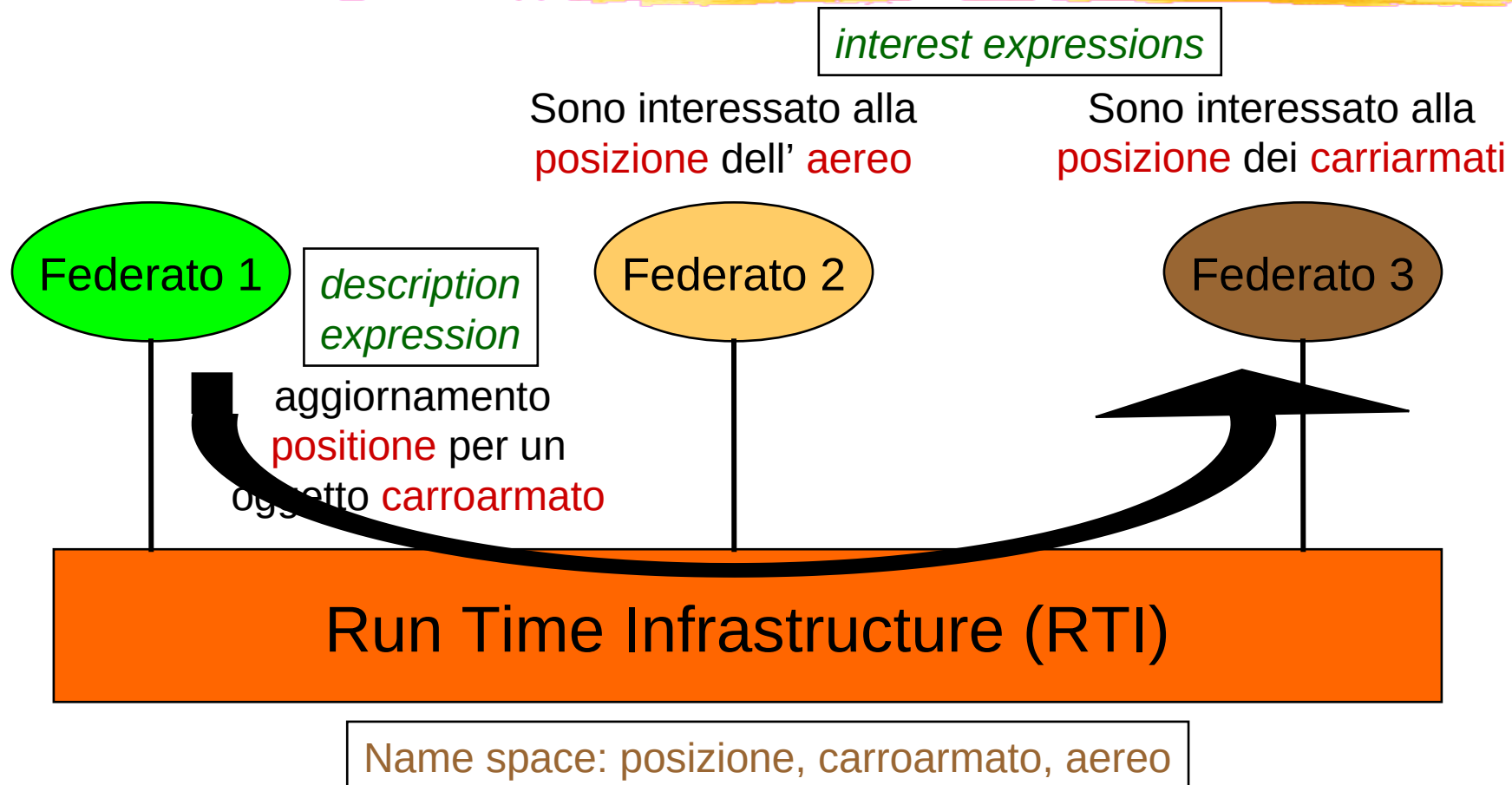


Scopo: instradare i dati prodotti da un simulatore solo ai simulatori che sono interessati a riceverli (e idealmente non ai simulatori che non sono interessati a riceverli)

Questo implica che ci sia

- ❑ Un qualche modo per un simulatore di specificare quali dati è interessato a ricevere (**interest expressions**)
- ❑ Un qualche modo di descrivere i dati che vengono prodotti (**description expressions**)
- ❑ Un linguaggio comune (vocabolario) per specificare espressioni di descrizione e di interesse (**name space**)

Esempio



Analogia: Internet Newsgroups



❑ Description expression

- Nome del newsgroup in cui il messaggio è pubblicato
- Annunci della classe pubblicata in “git.cc.class.cs4230”

❑ Interest expression

- Insieme dei newsgroups a cui un utente è iscritto
- Iscritto a “git.cc.class.cs4230” e “git.cc.class.cs2200”

❑ Name space

- Insieme di tutti i nomi dei newsgroup

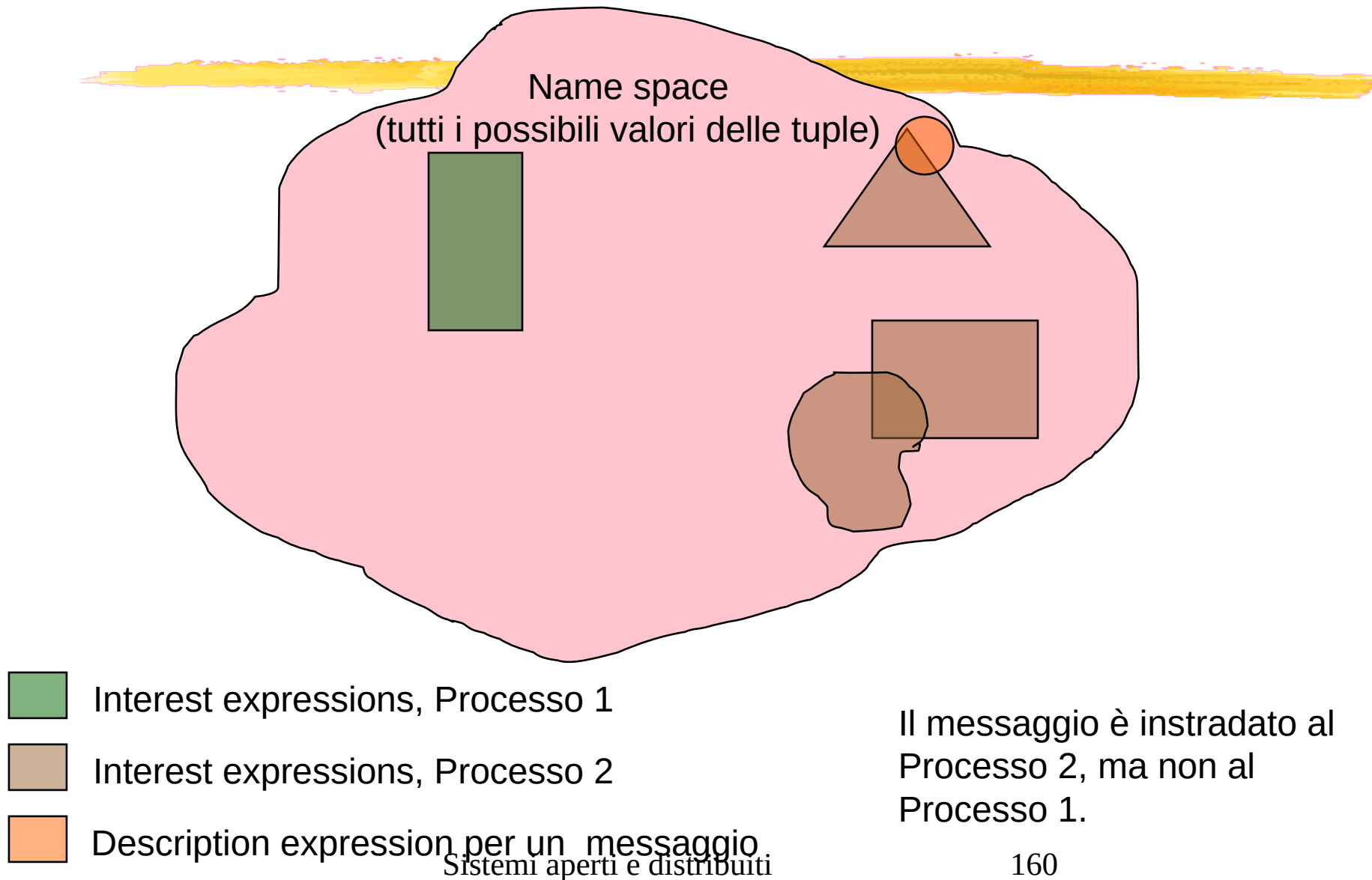
Name Space

- ❑ “Vocabolario” usato per creare
 - Data description expressions
 - Interest expressions
- ❑ Un name space è un insieme di tuple $(V_1, V_2, \dots V_N)$ dove V_i è un tipo base o un'altra tupla
 - Esempio: (class, location)
 - ✧ Class: enumerated type <carroarmato, aereo, nave>
 - ▮ Location: tupla (int: X-coordinata, int: Y-coordinata), dove $0 < X < 1000$ e $0 < Y < 1000$
 - Valori nel name space: (carroarmato,(30,200)); (aereo,(10,20))
 - Valori dello space della i-esima tupla può corrispondere a
 - ▮ Variabili di stato della simulazione (es., valori degli attributi dell'oggetto), o
 - ▮ Nuovo costrutto definito proprio allo scopo di instradare i dati (HLA: routing space regions; newsgroup name)
- ❑ Può includere proprietà **statiche** degli oggetti (es., nomi delle classi, nomi degli attributi) o proprietà **dinamiche** (valori degli attributi)

Interest & Description Expressions

- **Interest expression:** sottoinsieme del name space
 - Interessato a tutti gli aerei
 - ✧ (aereo, (X, Y)) per ogni X e ogni Y
 - Interessato ai carriarmati che sono “vicino a”
 - ▮ (carroarmato, (X, Y)) dove $10 < X < 20$ e $130 < Y < 150$
- **Description expression:** sottoinsieme del name space
 - (carroarmato, (15, 135))
 - (aereo, (X, Y)) dove $35 < X < 38$ e $98 < Y < 100$
- **Instradamento dei dati**
 - Un simulatore riceve un messaggio se la description expression del messaggio si sovrappone con una interest expression del simulatore

Concetti di Data Distribution



Data Distribution Statica vs. Dinamica

□ Data Distribution Statica

- Il name space include solo proprietà statiche che non cambiano durante l'esecuzione
- Esempio: Gestione delle dichiarazioni nel HLA
 - ✧ **Class-based** data distribution
 - ▮ Filtro basato sui tipi di dati
 - ▮ "Dammi gli aggiornamenti dell'attributo posizione per tutti gli oggetti della classe carroarmato"
 - Non si può filtrare basandosi su quantità calcolate dinamicamente
 - ▮ Non si può dire: "Dammi gli aggiornamenti degli oggetti carriarmati che sono vicini a me"

□ Data Distribution Dinamica

- Il name space include quantità dinamiche che possono cambiare durante l'esecuzione
- **Value-based** data distribution
 - ▮ "Dammi gli aggiornamenti degli oggetti carriarmati che sono vicini a me"
- Esempio: Gestione della Data Distribution nel HLA
 - ▮ Routing spaces
 - ▮ Iscrivere all'attributo posizione degli oggetti carriarmati che sono nel mio settore del campo di battaglia e distribuirli

Software RTI

□ Interfaccia di RTI all'applicazione (federato)

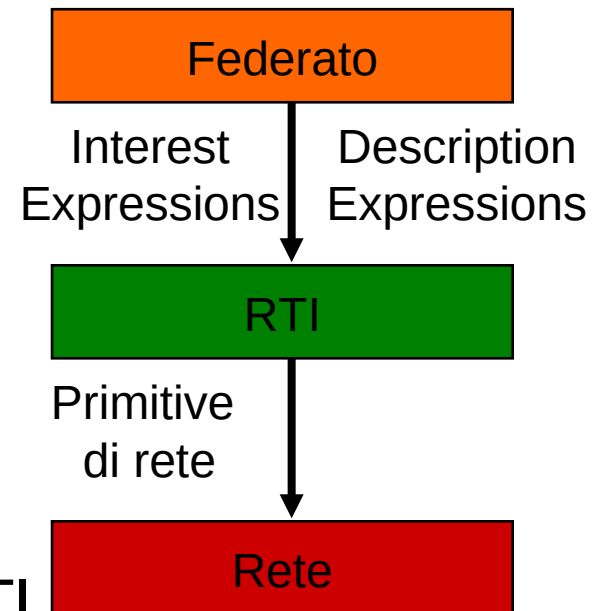
- Name space
- Interest expressions
- Description expressions

□ Primitive di rete

- Unicast
- Multicast
- Broadcast

□ Problemi dell'architettura di RTI

- A cosa dovrebbe assomigliare l'interfaccia del federato?
- Come è mappata l'interfaccia nel meccanismo di comunicazione sottostante?

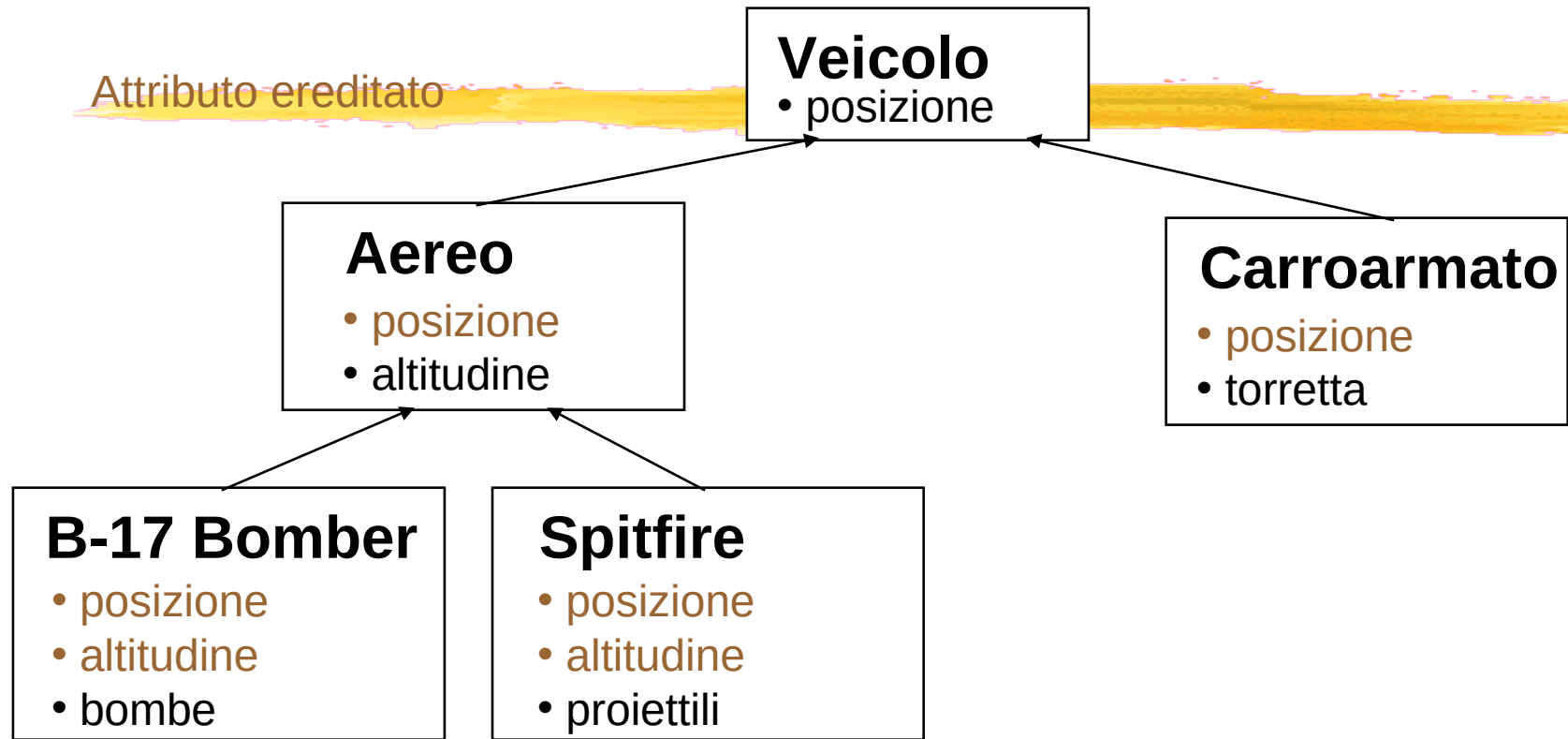


Class-Based Data Distribution



- ❑ Servizi di Gestione delle Dichiarazioni nel HLA
- ❑ Federation Object Model (FOM) definisce una gerarchia di classi di oggetti che descrive tutti i dati scambiati fra i federati
 - Classi degli oggetti
 - Attributi
- ❑ Description expressions e interest expressions specificano punti nella gerarchia della classe dell'oggetto

Esempio Gerarchia della Classe

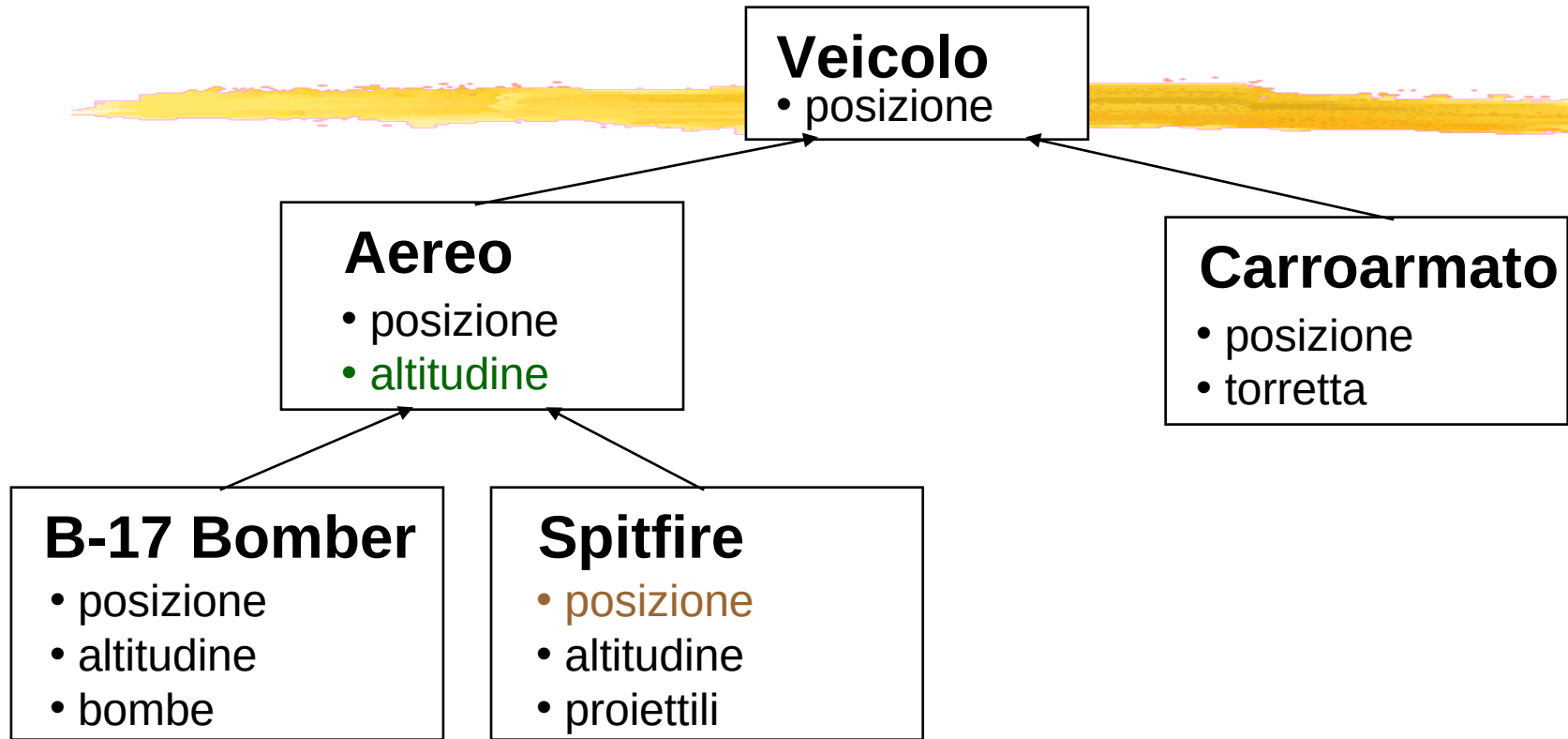


❑ Ogni classe eredita attributi dalla classe padre

❑ Name space: <classe, attributo>

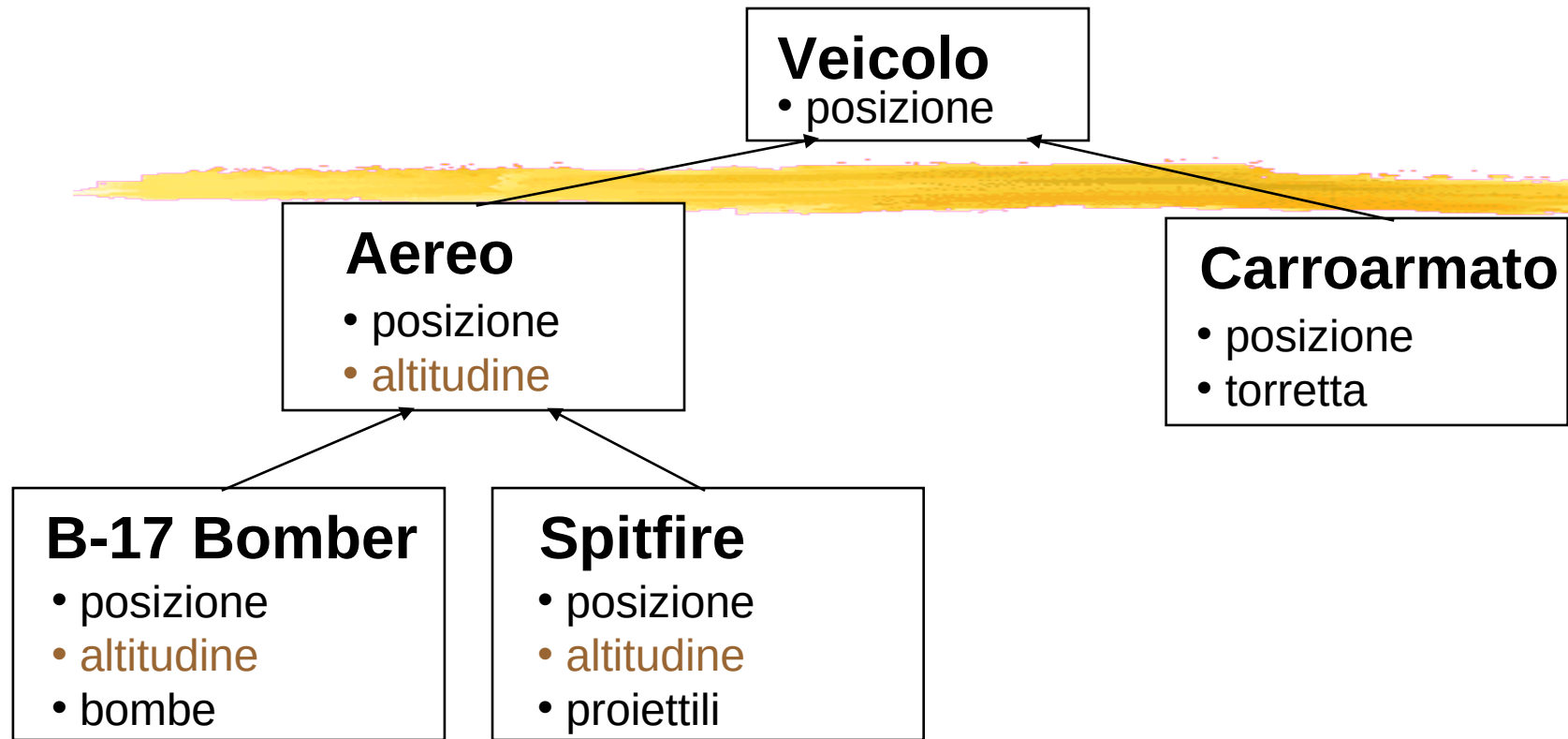
<Veicolo,posizione>, <Aereo,posizione>, <Aereo,altitudine>,
<Carroarmato,posizione>, <Carroarmato,torretta>, <B-
17,posizione>, <B-17,altitudine>, <B17,bombe>,
<Spitfire,posizione>, <Spitfire,altitudine>, <Spitfire,proiettili>

Description Expressions



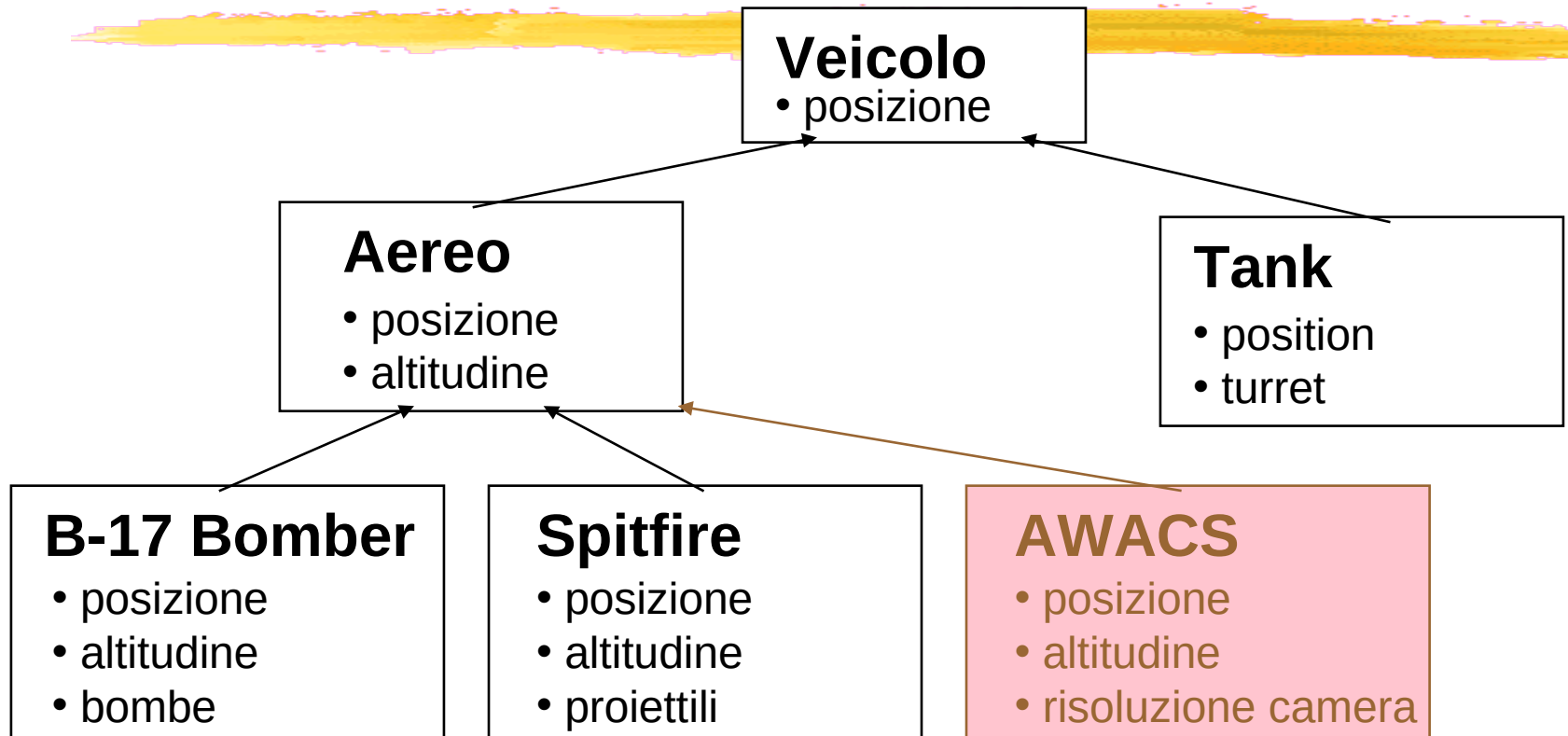
- ❑ Il servizio di Aggiornamento dei Valori degli Attributi invia un messaggio
- ❑ Description expression: un attributo di una istanza dell'oggetto
 - Singolo punto dell' <attributo della classe> nel name space
 - Esempi: <Spitfire, posizione> o <Aereo, altitudine>

Interest Expressions



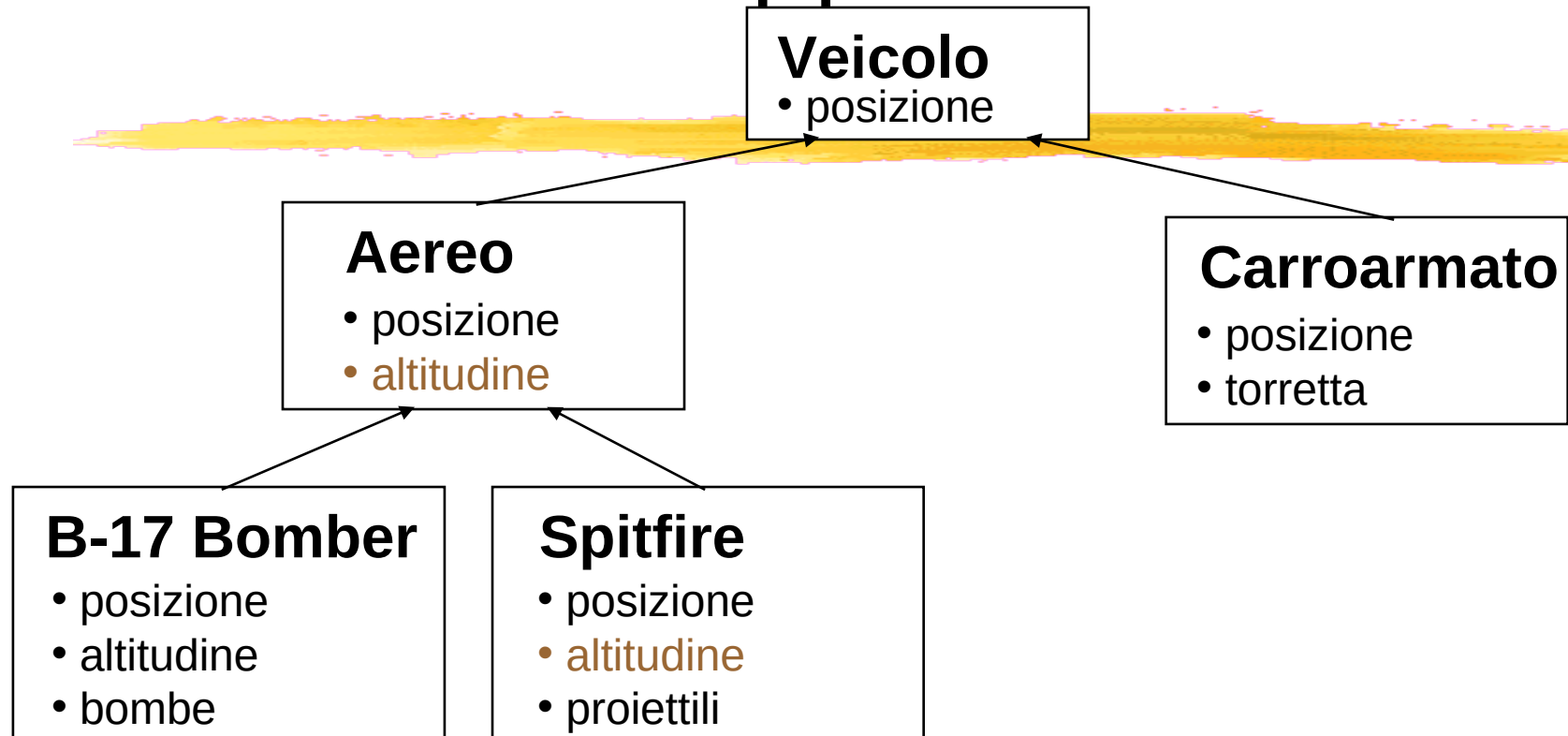
- ❑ Sottoscrivere Attributi della Classe dell'Oggetto [classe, attributo(i)]
- ❑ Interest expression: Sottoalbero con radice al punto di sottoscrizione
 - Subscribe (Aereo, altitudine): riceve aggiornamenti sull'attributo altitudine di oggetti di tipo Aereo, B-17, Spitfire
 - $\langle \text{Aereo}, \text{altitudine} \rangle$, $\langle \text{B-17 Bomber}, \text{altitudine} \rangle$, $\langle \text{Spitfire}, \text{altitudine} \rangle$
 - In tutti i casi il messaggio appare come un aggiornamento dell'oggetto Aereo

Estensibilità



Nuove sottoclassi possono essere aggiunte senza richiedere modifiche ai sottoscrittori a più alti livelli nella gerarchia delle classi

Altro Approccio



❑ Interest expression: singolo punto nell'albero

○ **<Aereo, altitudine>**

❑ Description expression: Cammino in su nell'albero fino al punto in cui l'attributo viene definito

○ **Update (Spitfire, altitudine): <Spitfire,altitudine>, <Aereo,altitudine>**

Riassunto



- ❑ Meccanismi di data distribution sono necessari per evitare comunicazioni di tipo broadcast
- ❑ Concetti fondamentali
 - Name space
 - Interest expressions specificano quali informazioni il simulatore vuole ricevere
 - Description expression descrive il dato contenuto dentro il messaggio
- ❑ RTI deve “confrontare” interest expressions e data description expressions per instradare i dati

Data Distribution



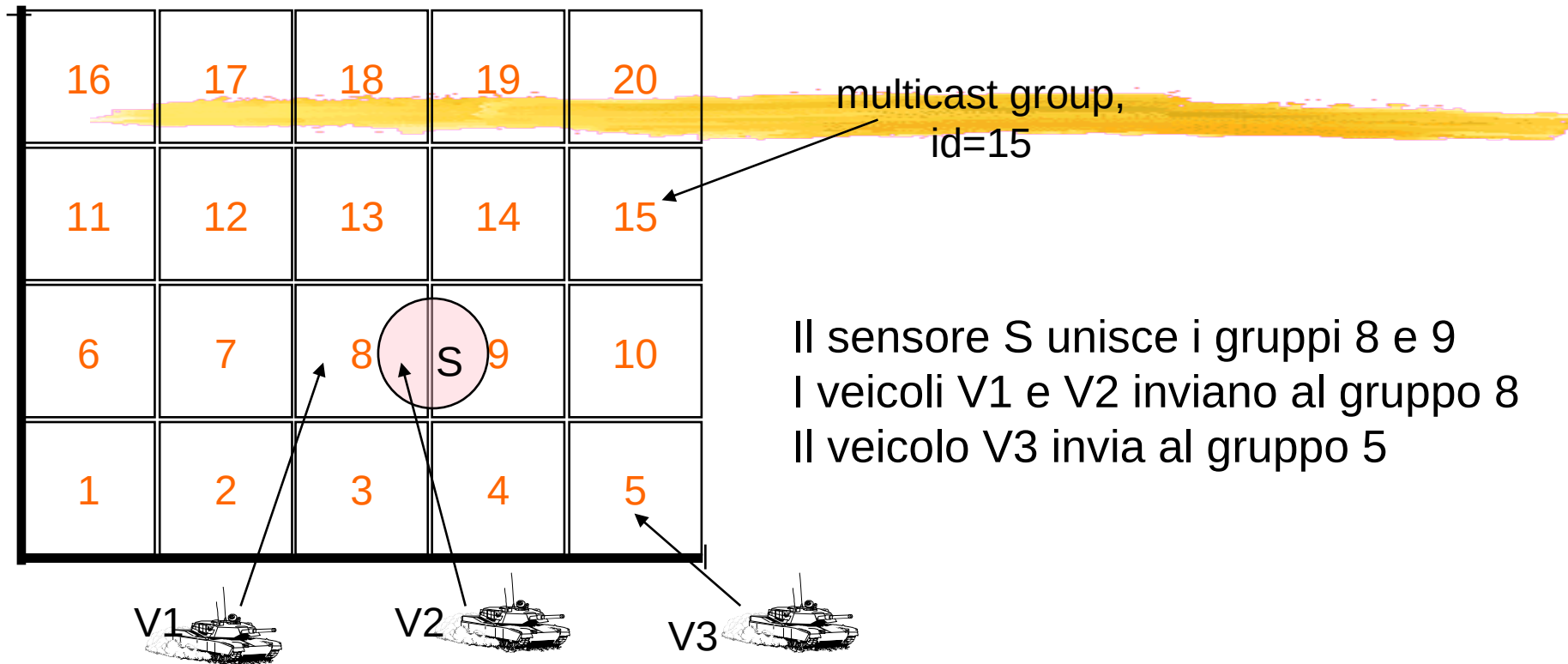
Data Distribution Dinamica

Contenuti



- ❑ Data Distribution Dinamica (Value based) : HLA Data Distribution Management
 - Routing space
 - Publication Region
 - Subscription Region
- ❑ Implementazione del DDM
 - Cell-Based
 - Region-Based
 - Combinando Cell e Region

Usare una Grid per catturare la località

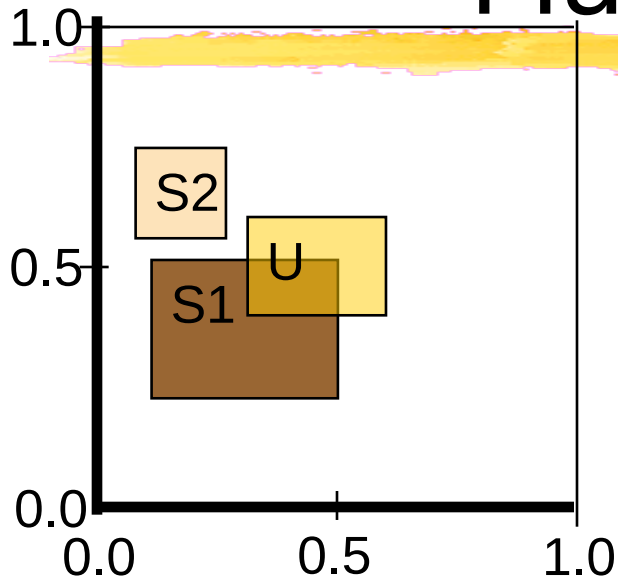


- ❑ Divide il play-box in celle non sovrapposte (rettangolari, esagonali) di griglia
- ❑ Crea un multicast group per ogni cella
- ❑ Sottoscriviti a celle(a) che tu puoi “vedere”
- ❑ Invia un messaggio alla cella in cui risiede il veicolo
- ❑ Richiede un filtraggio aggiuntivo al ricevente

HLA Data Distribution Management (DDM)

- ❑ HLA DDM fornisce un meccanismo più generale
- ❑ Name Space
 - **Routing space**: sistema di coordinate N-dimensionale
 - Separato dallo stato della simulazione, usato solamente per il routing
- ❑ Interest expressions
 - **Subscription region**: rettangolo N-dimensionale nel routing space
 - Associa una regione alle richieste di sottoscrizione
- ❑ Description expressions
 - **Update region**: rettangolo N-dimensionale nel routing space
 - Associata ad ogni **istanza** dell'oggetto
- ❑ Un messaggio che aggiorna un attributo di un'istanza di un oggetto è instradato al federato se:
 - Il federato è sottoscritto alla classe e all'attributo dell'oggetto, **e**
 - La regione di aggiornamento associata con l'attributo aggiornato si sovrappone con la regione di sottoscrizione del federato per quella classe/attributo

HLA Data Distribution Management

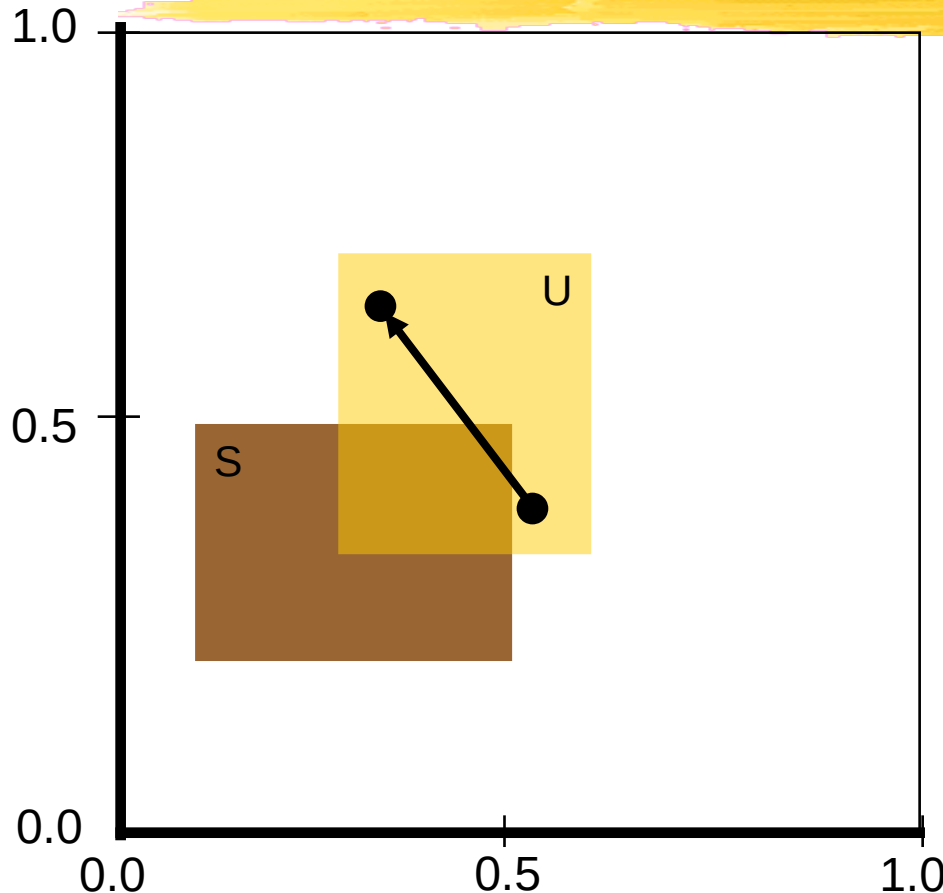


- Federato 1 (sensore): sottoscrive a S1
 - Federato 2 (sensore): sottoscrive a S2
 - Federato 3 (obiettivo): aggiorna la region U
- I messaggi di aggiornamento sono inviati da obiettivo al federato 1 non al 2

Description expressions

- Update region nel routing space (U)
- Una update region è associata ad ogni aggiornamento dell'attributo
- Un federato riceve un messaggio se
 - Se si era sottoscritto all'attributo(i) che viene aggiornato, e
 - La sua subscription region si sovrappone con la update region

Update Regions vs. Points

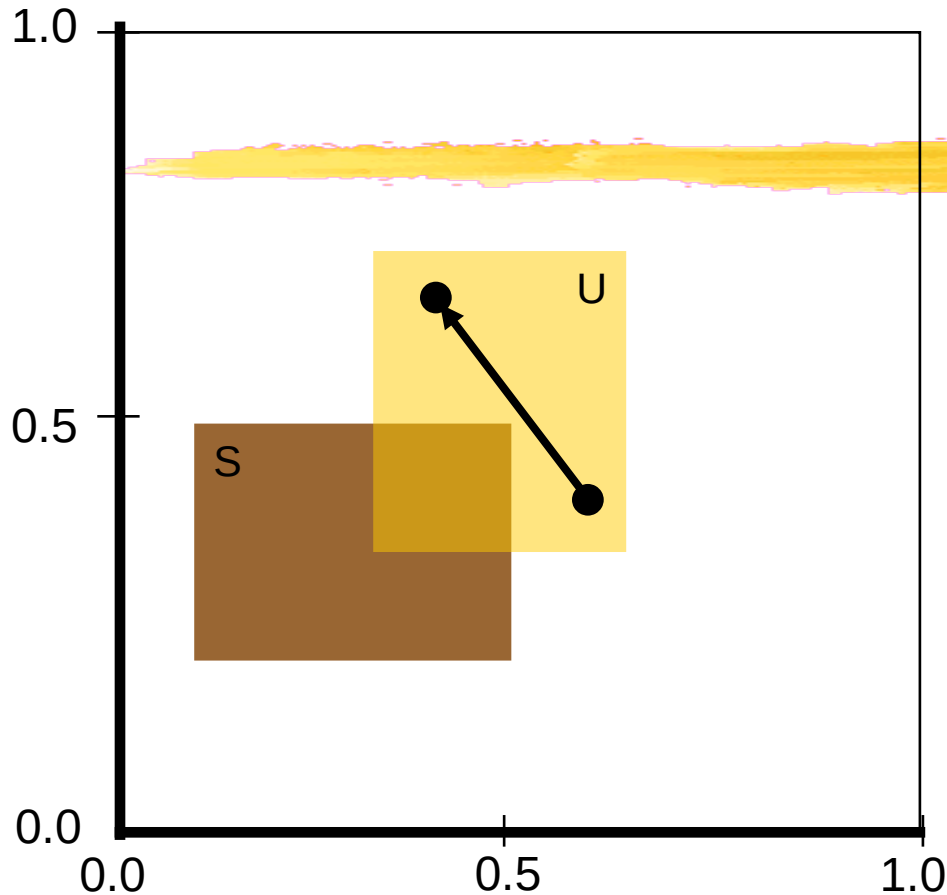


- Il Routing space rappresenta il playbox
- La Subscription region rappresenta il sensore
- Gli aggiornamenti corrispondono a posizioni del veicolo in movimento

□ Update points: Il sensore non viene informato del veicolo

□ Update regions: Il sensore viene informato del veicolo

Precisione di filtering



- Veicolo fuori dal range, ma gli aggiornamenti sono ancora instradati al federato del sensore
- I messaggi devono essere filtrati dal ricevente
- Il range del sensore potrebbe non essere rettangolare

In generale DDM è un compromesso tra

- Accuratezza di filtering
- Considerazioni di implementazione (mapping a multicast groups)
- Facilità di utilizzo

Servizi di DDM di HLA

Routing spaces (name space) e regions

- ❑ Definire routing spaces nel file di inizializzazione della federazione
- ❑ Creare region, Modificare Region, Cancellare Region
 - Usati sia per subscription regions che per update regions

Subscription regions (interest expressions)

- ❑ Sottoscrivere/De-sottoscrivere Attributi di Classi di Oggetti con Region
 - Usati in aggiunta al class-based filtering

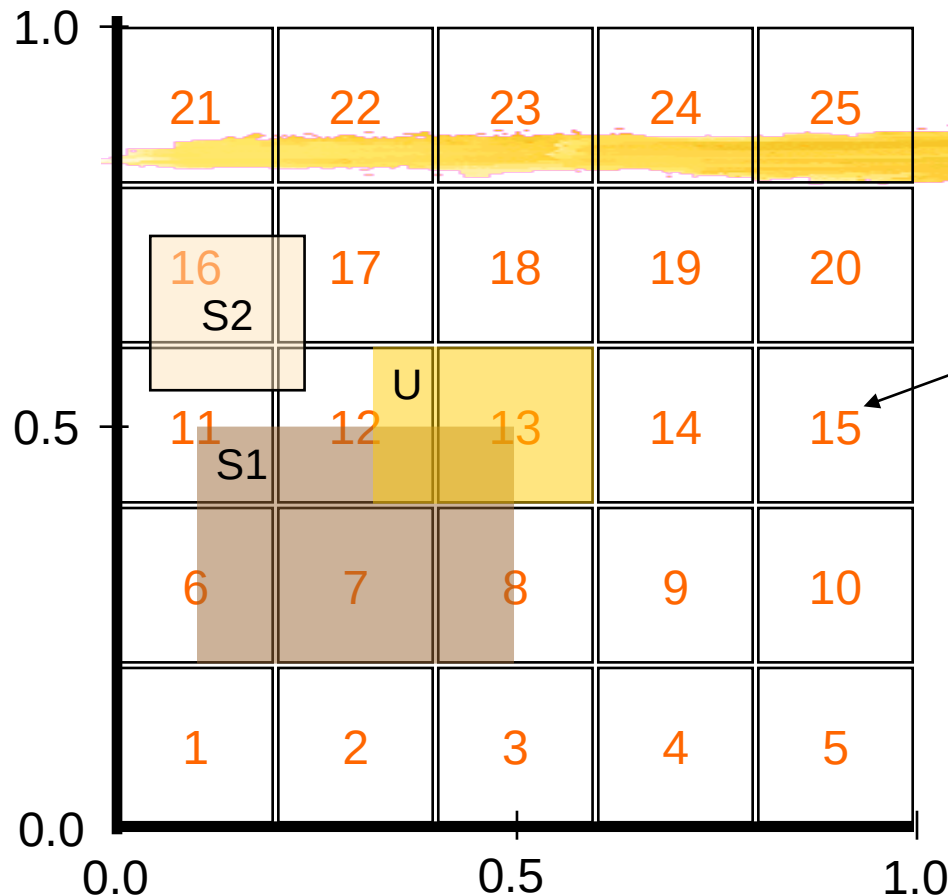
Update regions (description expressions)

- ❑ Registrare Istanze di Oggetti con Region ○ Associare Region con Aggiornamenti
- ❑ De-associare Region per Aggiornamenti
- ❑ Aggiornare Valori degli Attributi

Approccio di Implementazione

- ❑ Mappare il name space ai gruppi multicast
 - Non tutti i punti nel name space necessitano di essere mappati a gruppi
 - Un punto nel name space potrebbe mappare a gruppi multipli
- ❑ Interest expression
 - Interest expression sono definite come punti del name space
 - Unire gruppi che si sovrappongono alla interest expression
- ❑ Description expression
 - Description expression sono definite come punti nel name space
 - Inviare messaggi a gruppi che si sovrappongono alla description expression

Implementazione Grid-Based



S1 sottoscrive a 6,7,8,11,12,13

S2 sottoscrive a 11,12,16,17

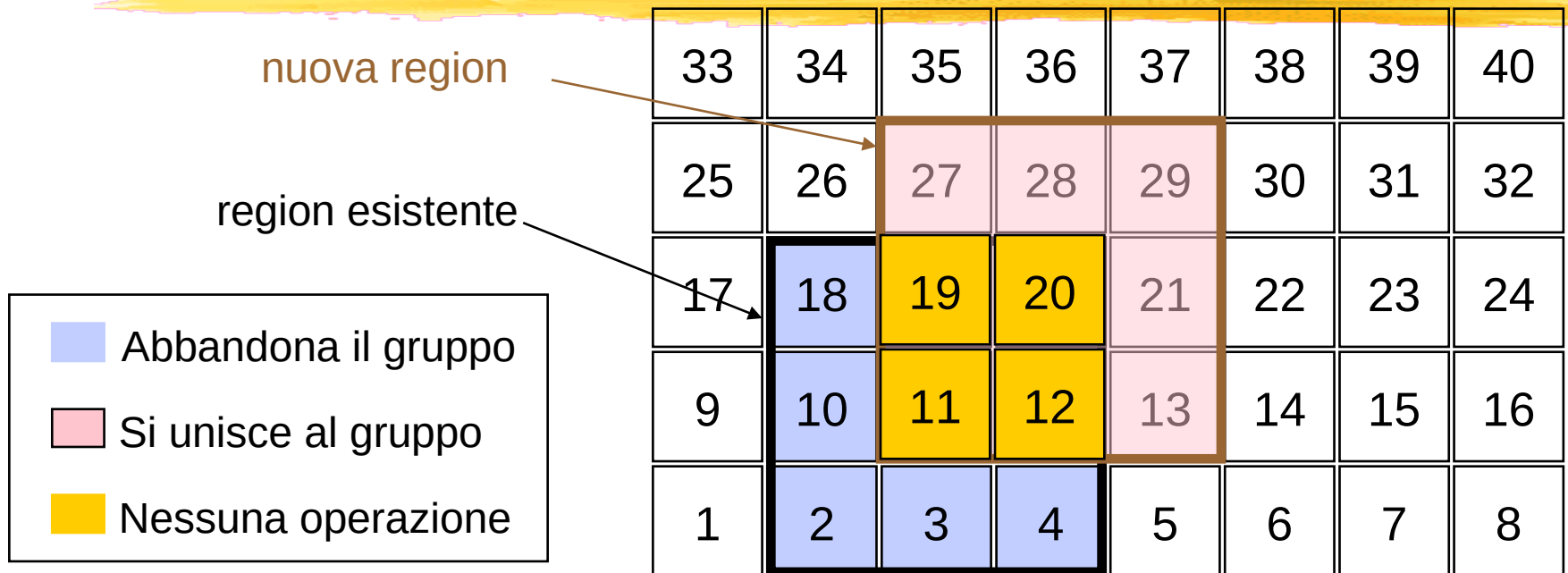
U pubblica a 12, 13

Messaggi non cercati a S2

Messaggi duplicati a S1

- Partizionamento del routing space in celle grid, mappare ogni cella ad un multicast group
- subscription region: Unire ogni group che si sovrappone alla subscription region
- attribute update: inviare *Update* a ogni gruppo che si sovrappone alla update region
- E' necessario un ulteriore filtering per evitare messaggi non cercati e duplicati

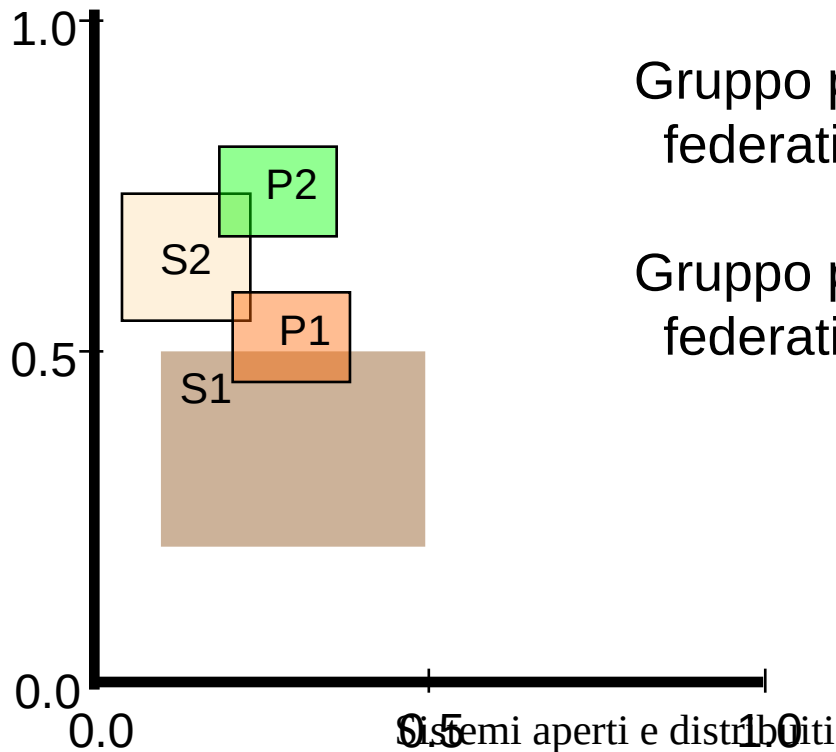
Cambiare una Subscription Region



- Provoca operazioni *Leave* per (celle nella vecchia region – celle nella nuova region)
- Provoca operazioni *Join* per (celle nella nuova region – celle nella vecchia region)

Approccio 2: Gruppi Region-Based

- Definire un multicast group per publication region
- Appartenenza al gruppo: Ogni federato sottoscritto a una region che si sovrappone alla publication region è un membro del gruppo
- Update: Invia il messaggio al gruppo associato con la publication region

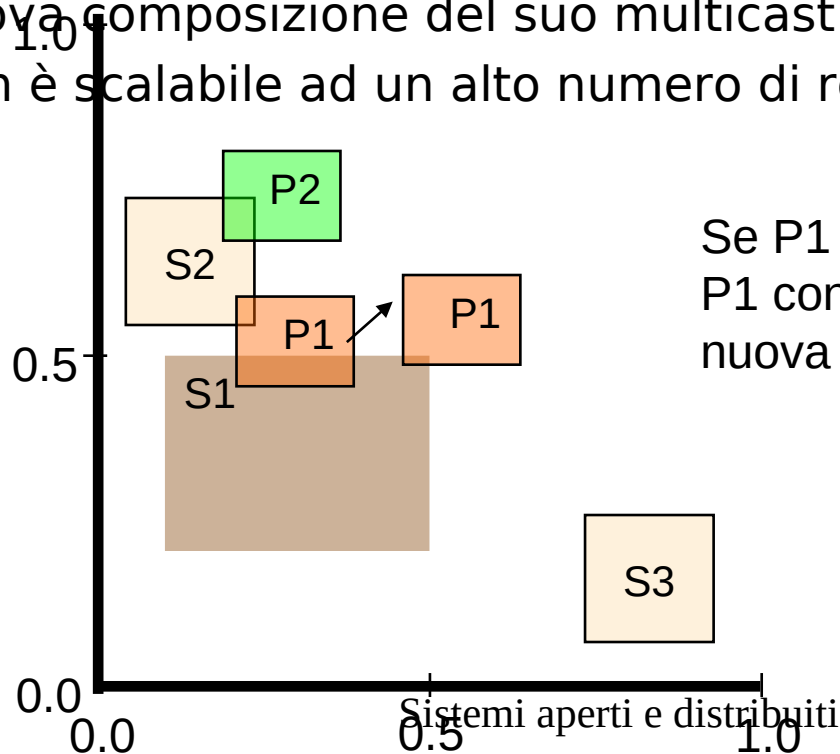


Gruppo per P1:
federati sottoscritti alle region S1, S2

Gruppo per P2:
federati sottoscritti alla region S2

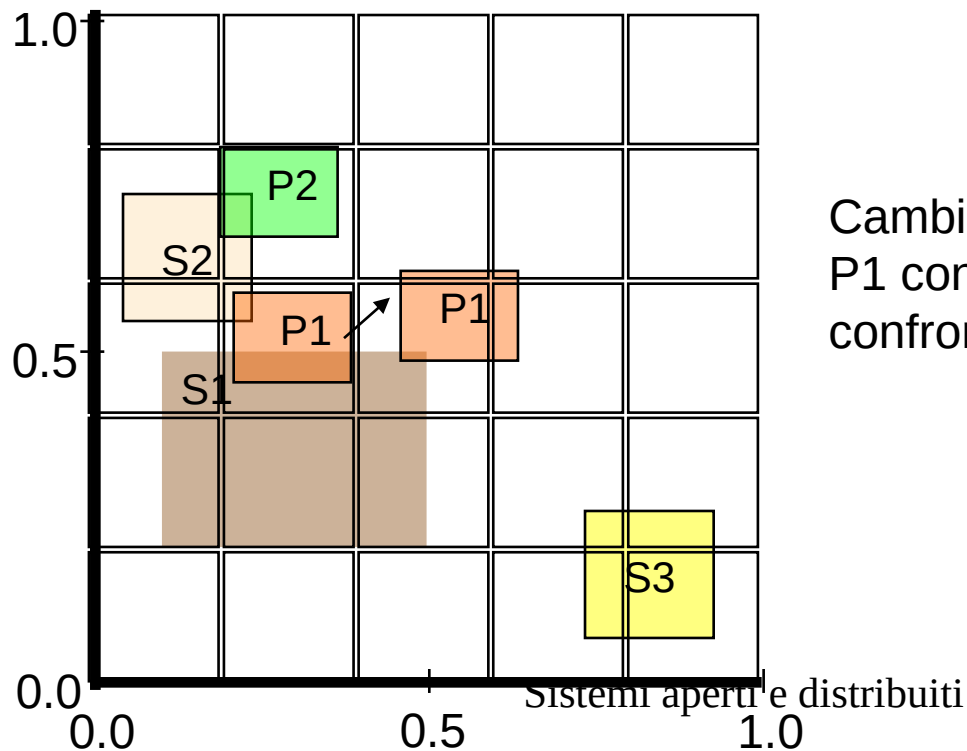
Il Problema del Matching

- ❑ Quando una subscription region cambia, questa deve essere confrontata con tutte le publication region per determinare se il federato dovrebbe join/leave multicast groups
- ❑ Quando una publication region cambia, questa deve essere confrontata con **tutte** le subscription regions per determinare la nuova composizione del suo multicast group
- ❑ Non è scalabile ad un alto numero di region



Approccio 3: Regions con Grids

- Un gruppo è definito per ogni publication region (come nell'approccio region-based)
- Viene sovrapposta una grid sul routing space
- Matching: occorre solo controllare le publication/subscription regions nelle celle della grid che si sovrappongono alla region vecchia e alla region nuova



Cambiando P1: deve confrontare P1 con S1 e S2, ma non è necessario confrontarlo con S3

Problemi pratici



- ❑ Numero limitato di multicast groups
- ❑ Movimenti veloci: rapidi join e leave
 - I tempi dei Join/leave possono essere lunghi
 - Occorre effettuare prima le operazioni di gruppo Predict e Initiate
- ❑ Visualizzatori di ampie aree: troppo traffico!
 - Per ridurre il traffico bisogna togliere le informazioni dettagliate
 - Routing space multipli con diverse dimensioni della griglia e diversi livelli di dettaglio delle informazioni che coprono la playbox

Riassunto

- ❑ Data distribution management fornisce un filtering dei dati value-based
 - Interest e description expression dinamiche
 - Design implica molti compromessi
 - ✧ Efficienza del filtering
 - ▮ Facilità di utilizzo
 - ▮ Complessità di implementazione
- ❑ Implementazioni
 - Mappare il name space a multicast group
 - Mappare le interest expressions a multicast group joins
 - Mappare la declaration expression a multicast group sends
 - Mappare i cambiamenti delle interest expressions a group joins e leaves