

# Sistemi Distribuiti – Schema Riassuntivo (BETA)

## Indice generale

Sistemi Distribuiti – Schema Riassuntivo.....	1
<b>Classi di sistemi distribuiti.....</b>	<b>1</b>
RPC (Remote Procedure Call).....	2
<b>Distributed Object Middleware.....</b>	<b>3</b>
RMI (Java Remote Method Invocation).....	3
CORBA : Common Object Request Broker Architecture.....	4
DCOM (Distributed Component Object Model).....	6
.NET.....	6
<b>Paradigmi alternativi di Middleware.....</b>	<b>7</b>
Message Oriented Middleware (message queuing).....	7
Publish-Subscribe.....	8
SUN JMS.....	9
Linda & Tuple Spaces.....	9
Lime.....	10
Jini.....	10
<b>Naming.....</b>	<b>11</b>
DNS.....	11
GSM.....	12
<b>Sincronizzazione.....</b>	<b>13</b>
Temporizzazione.....	13
Orologi Scalari.....	14
Orologi Vettoriali.....	14
Mutua Esclusione.....	14
Leader Election.....	15
Global Snapshot.....	15
Termination Detection.....	17
Transazioni Distribuite.....	17
<b>Consistenza e Replicazione.....</b>	<b>19</b>
Consistency Models.....	19
<b>Fault Tolerance.....</b>	<b>23</b>
Modello sincrono.....	24
Comunicazione di gruppo affidabile.....	25
Tecniche di Recovery.....	26
<b>Sicurezza.....</b>	<b>27</b>
Crittografia.....	28
Kerberos.....	30
Secure Group Communication.....	30
Access Control.....	32
Pagamenti Elettronici.....	33
<b>Mobile Code.....</b>	<b>33</b>
Case Studies.....	35
<b>Peer-to-Peer.....</b>	<b>35</b>
<b>Wireless Sensor Networks.....</b>	<b>35</b>
MICA2 Motes.....	36
Features.....	36

## Classi di sistemi distribuiti

- Distributed OS : forte accoppiamento per multiprocessori e multicomputer omogenei. Atto al management black-box delle risorse hardware. Message passing oppure memoria distribuita condivisa (problemi di consistenza)
- Network OS: accoppiamento lasco per sistemi multicomputer eterogenei. Si offrono servizi a client remoti. Alla base del paradigma client-server (con accesso esplicito o implicito). Le astrazioni di programmazione sono ancora di basso livello.
- Middleware: Layer superiore al NetOS che implementa servizi generici. Offre trasparenza di distribuzione. Utilizza servizi locali e di rete, per offrire alle applicazioni servizi di alto livello. Si lavora indipendentemente dalle disomogeneità presenti in rete (interoperabilità)

### RPC (Remote Procedure Call)

Abilita accesso remoto tramite chiamate di procedure di più alto livello rispetto alle primitive di rete. Problematiche principali : **Marshalling** (conversioni tra rappresentazioni dati) e **Serialization** (flattening dei dati). Si sfrutta un **IDL**, ovvero un linguaggio di supporto che definisce in maniera interoperabile i dati passati da un linguaggio ad un altro (separo interfaccia ed implementazione ed effettuo il mapping dei dati). IDL permette anche la generazione automatica di codice (stubs, ad esempio)

Problema: passaggio parametri. Soluzioni:

1. **Pass by reference**: bisogna fare attenzione alla validità generalmente *locale* del riferimento. (Spesso si vieta il passaggio by reference)
2. **Call by Value/Result**: copio parametro nel valore locale della procedura, alla fine sincronizzo la variabile originale con il valore locale (semantica non trasparente, difficile da usare in dati fortemente strutturati, ottimizzazioni per dati read/write only)

Implementazioni notevoli:

1. **SUN RPC** (Standard de facto): utilizza XDR, un IDL estesa per specificare dati ed interfacce. Trasporto over TCP/UDP, dati passati per copia, solo un parametro di input ed output. Sicurezza tramite DES.
2. **DCE RPC** : implementa servizi di alto livello (Directory, Sincronizzazione, Sicurezza), molte semantiche di invocazione per le procedure remote.

**Dynamic Binding**: trovare chi offre un dato servizio, dov'è il server che lo offre e come stabilire una comunicazione con esso

1. SUN RPC: processo demone (**portmap**) che correla le chiamate alle porte. I servizi comunicano a portmap che cosa forniscono e su che porta. Clients possono fare richieste verso portmap. Non risolvo il primo problema, posso però appoggiarmi a directory servers oppure effettuare chiamate multicast
2. DCE RPC: Simile a SUN, ma implementa anche il Directory server
  - Servizio si registra presso **DCE Daemon locale** e anche sul **Directory Server**
  - Client effettua lookup su Directory Server, che gli dice dove trova il server con quel servizio
  - Da qui in poi è come SUN

**Dynamic Activation**: i processi possono rimanere attivi anche in assenza di richieste, si sprecano

risorse

- SUN RPC: demone inetd locale, attiva i servizi quando vengono richiesti. Mapping tra servizi richiesti e processi relativi è in un file di configurazione. Le richieste vengono inoltrate a inetd che redireziona sul processo richiesto oppure lo attiva se non è presente. La disattivazione può seguire differenti politiche.

**Lightweight RPC:** utilizzare in modo trasparente metodi di IPC per processi locali, in modo da non creare socket inutili quando posso usare cose più efficienti (pipes, shm, etc...). Questo permette risparmio di risorse (meno thread/processi) e diminuisce le copie di parametri (una invece di quattro).

Nell'invocazione lightweight, stub copia parametri su uno shared stack privato, poi effettua una chiamata di sistema. Il kernel si occupa del context switch, esegue il servizio; i risultati vengono copiati sullo stack e tramite un'altra syscall si effettua il context switch a favore del processo client.

**Asynchronous RPC:** si può evitare la semantica bloccante sulla richiesta (client si sospende in attesa della reply) passando ad una semantica asincrona (di cui esistono diverse varianti, ad esempio ACK sulla richiesta oppure PROMISE)

**Batched RPC :** implementato da SUN. Chiamate che non richiedono risultati sono bufferizzate sul client. Esse vengono mandate tutte assieme quando arriva una chiamata RPC non-batched (o dopo un forcing). Si economizza sulla banda.

**Queued RPC:** implementato in ROVET Toolkit (MIT). E'utile in contesti a forte mobilità (caduta temporanea di un nodo), server accoda le risposte per un nodo caduto e le rimando tutte assieme quando torna online (else errore dopo un timeout)

## Distributed Object Middleware

Stessa idea di RPC, costrutti differenti (che mirano a sfruttare i vantaggi della programmazione a oggetti anche in contesto distribuito).

- **Si possono passare i riferimenti agli oggetti remoti:** bisogna mantenere le relazioni di aliasing
- **IDL** sono molto più ricche (riflettono la varietà dei meccanismi OO)

### RMI (Java Remote Method Invocation)

Esprime un'architettura minimale sullo stile di SUN RPC. I servizi di alto livello sono implementati in altri moduli basati su RMI (JINI, JNDI ...). Aspetti innovativi: semantiche di passaggio parametri, class downloading, proxies dinamici.

- Gli oggetti remoti devono estendere un'interfaccia che estende *java.rmi.Remote* (come IDL)
- Gli oggetti possono essere copiati e distribuiti se implementano *java.io.Serializable*
- Gli oggetti remoti devono essere esportati (listen sul socket per chiamate). E'fatto automaticamente se l'oggetto estende *UnicastRemoteObject*, oppure staticamente tramite *UnicastRemoteObject.exportObject*

**Come ottenere un riferimento remoto:**

1. Attraverso passaggio parametri
2. Querando un directory service (**rmiregistry**)

Il riferimento remoto è un oggetto contenente il client stub (proxy). Una volta acquisito lo stub, l'invocazione è del tutto trasparente rispetto a un'invocazione locale.

**Passaggio Parametri:** semantiche diverse per il caso locale e per quello remoto. **method(obj)**

1. se **obj** è remoto, si usano semantiche by reference. Modifiche all'oggetto implicano accesso diretto tramite la rete. Se **obj** è serializzabile, passo la copia.
2. se **obj** è locale, si passa by copy. Le modifiche sono locali e non scatenano comunicazioni sulla rete

Per oggetti logicamente condivisi conviene usare remoto. Se l'accesso è read-only, conviene serializzare (se le dimensioni lo consentono). Differenza esplicita, permette al programmatore di scegliere, ma è una differenza statica. Non si può fare nulla a runtime a seconda delle esigenze.

**Rmiregistry:** lookup service, provvede per il binding dinamico dei servizi. I client possono ottenere un proxy dato il nome simbolico, oppure una lista dei nomi simbolici registrati. Primitive fornite: **lookup, list, bind, rebind, unbind**

**RMI e Concorrenza:** essa non è contemplata nel modello. La sincronizzazione deve essere forzata dal programmatore se è necessaria, sono gli oggetti implementati che devono garantire la proprietà di *thread-safety*

**RMI** si appoggia a skeletons, proxies e dispatchers

- **Proxy:** rende possibili invocazioni trasparenti. Si occupa della connessione e del marshalling, preserva la compatibilità di tipo (implementa la stessa interfaccia). Un proxy per ogni processo
- **Dispatcher:** manager delle connessioni lato server, e invia le invocazioni verso gli skeletons (possibilità di socket reusing e thread pooling)
- **Skeleton:** marshalling lato server e instradamento della chiamata verso l'oggetto remoto.

Nelle ultime versioni di Java gli Stubs non sono più necessari (ci si basa su proxy dinamici, classi generate automaticamente su interfacce specificate a runtime)

**Attivazione Dinamica degli Oggetti:** Gli oggetti remoti estendono *java.rmi.activation.ActivableObject*. Durante l'attivazione, lo stub genera una *FaultingReference* ed un *Activation ID*. Se l'oggetto non è già presente, esso viene attivato e gli viene passato l'*Activation ID*.

**RMID:** è il demone che si occupa del listening delle chiamate, ha una tabella di descrittori di Attivazione e può gestire più JVM.

**Code Downloading:** invocazione **method (Type A)**. Grazie al polimorfismo, a runtime il client può fare invocazione attuale **method (SubType B)**. Non è detto che la classe di SubType si trovi anche sul server!! Soluzione: Il server si accorge che non ha il bytecode relativo alla sottoclasse, e quindi lo scarica dal client o da un punto indicato dal client (**mobile code**)

Questo viene realizzato ridefinendo il comportamento del classloader e dei meccanismi di serializzazione.

**RMI** non maschera totalmente la distribuzione (semantica passaggio parametri, interfacce ed eccezioni remote), inoltre è visto come un mattone base su cui costruire servizi di più alto livello.

### **CORBA : Common Object Request Broker Architecture**

È un insieme di funzionalità molto più complesso di RMI. Obiettivo: interoperabilità, standardizzando la comunicazione tra applicazioni eterogenee. Ci sono più implementazioni ognuna

con le proprie estensioni.

### Architettura

- **ORB** (Object Request Broker): il core che gestisce le funzionalità del middleware
- **Object Services**: servizi base necessari per lo standardizzazione
- **Common facilities**: funzionalità utili non obbligatorie (printing, help, internationalization)
- **Domain interfaces**: servizi non obbligatori dipendenti da un certo dominio applicativo (finanza, gestione produzione, telecomunicazioni ...)
- **Application interfaces**: interfacce implementate dal programmatore

**CORBA IDL**: è uno standard, si usa anche al di fuori di questo sistema specifico. Utilizzata per specificare gli oggetti applicativi. Un oggetto così specificato può essere implementato in una varietà di linguaggi (anche non OO). Il modello a oggetti di CORBA è fortemente tipizzato. Supporto per ereditarietà multipla (sempre pubblica) e binding dinamico. Il dispatching è sempre dinamico. Non si hanno costruttori e distruttori, queste operazioni infatti non sono controllate dal client. Non si può effettuare overloading dei metodi.

Gli attributi sono pubblici (definiti nell'interfaccia) oppure sono privati. Si può specificare se sono *readonly* (non modificabili dal client) oppure *const* (non modificabili). Le operazioni accettano parametri *in,out,inout* (permette di ottimizzare allocazione e comunicazione).

Gli **oggetti** vengono passati **per riferimento**, i **non-oggetti** sono passati **per copia**. Recentemente, è stata introdotta la possibilità di passare **object-by-value** (si può comunque copiare lo stato dell'oggetto dentro una *struct*).

### Semantiche di invocazione

Richiesta	Semantica di fallimento	Comportamento client
Sincrona	At-most-once	Blocking
One-way	Best effort delivery	Non blocking
Deferred synchronous	At-most-once	May block on reply delivery

(chiamate asincrone) **Future**: blocco sul reply , **Promise**: controllo a polling non bloccante

**ORB**: rende possibile la comunicazione. Si occupa di manipolare i riferimenti agli oggetti, del marshalling, naming. E' disaccoppiato dalle applicazione tramite una interfaccia. Sul server è presente un **Object Adapter** che si occupa dell'attivazione e della concorrenza. Sul client sono presenti **DII** (Dynamic Invocation Interface) e **DSI** (Dynamic Skeleton Interface) che supportano l'invocazione dinamica (di risorse non note staticamente).

**Interface Repository** può essere contattato per ottenere informazioni su interfacce metodi e parametri

**Implementation Repository** contiene informazioni sull'implementazione degli oggetti nel sistema e sul modo di organizzarli (simile a inetd)

**ORB Interoperabilità**: risolto introducendo due protocolli nell'architettura

1. **General Inter-ORB Protocol (GIOP)**: framework di comunicazione indipendente dal protocollo di trasporto di rete attuale. Si ha un *tagged profile* per ogni protocollo supportato;
2. **Internet Inter-ORB Protocol (IIOP)**: specializzazione di GIOP su TCP
3. **Interoperable Object Reference (IOR)**: astrazione standardizzata dei riferimenti ad oggetti. Si utilizzano dei *tagged profiles* che contengono il supporto per differenti protocolli

4. **Object Adapter / Portable Object Adapter (POA)**: l'object adapter genera ed interpreta i riferimenti remoti, realizzando effettivamente l'invocazione dei metodi. POA supporta anche la persistenza.

Ciascun POA può incapsulare differenti policy riguardo all'attivazione, alla persistenza ed alla concorrenza. Sono collegati a dei *Servants*, ovvero gli oggetti nativi che implementano la logica applicativa. Tuttavia la logica applicativa viene gestita tramite le politiche specificate dai POA.

**CORBA Naming**: più sofisticato del naming in RMI. Gestisce *nomi strutturati* e *namespaces* (i quali, assieme al processo di risoluzione, sono distribuiti).

I nomi sono composti da sequenze di oggetti contenenti coppie di stringhe (ID,tipo) e sono organizzati in un grafo con radice, sul quale è possibile fare ricerche con un apposito componente.

### **DCOM (Distributed Component Object Model)**

Ottenuto facendo comunicare remotamente componenti COM (Component Object Model), OLE (Object Linking and Embedding), ActiveX. COM+ è un insieme di tutti i precedenti.

**Interfacce COM**: Specificate con **MIDL** (Microsoft IDL) e sono binarie. Sono tabelle contenenti puntatori al codice che implementa i metodi (non ho problemi di interoperabilità su macchine con SO omogeneo - Winzozz)

Gli **Oggetti COM** implementano una o più interfacce. Un'**implementazione COM** è una classe qualsiasi che implementa un'interfaccia COM. Un **oggetto COM** è un'istanza di un'implementazione COM.

Ogni interfaccia ha un UUID (Universally Unique ID), non ho concetto di riferimento ad oggetto (si usano puntatori), l'ereditarietà non può essere multipla.

**Invocazione COM**: sono possibili diverse semantiche

- **Invocazione sincrona**
- **Callbacks**
- **Cancel Objects** (permette di cancellare un'invocazione sincrona, simile ad Deferred di CORBA)

Esse vengono implementate in diversi modi a seconda della locazione

- Stesso processo: chiamata locale via DLL
- Stesso host: lightweight RPC
- Host differenti: RPC

Semantiche at-most-once, non ci sono eccezioni (solo un codice d'errore HRESULT)

**Architettura DCOM**:

**Windows Registry**: contiene un mapping tra CLSID e un file locale contenente l'implementazione della classe

**SCM (Service Control Manager)**: gestisce l'accesso al registro, può iniziare un nuovo processo server. Tiene traccia dei meccanismi di invocazione da utilizzare

I puntatori di interfaccia sono gestiti passando UUID e le informazioni di binding. Di default gli oggetti DCOM sono transitori (oggetti persistenti vengono implementati tramite **monikers**)

## **.NET**

E' il sostituto di DCOM. Microsoft specifica **Common Language Infrastructure (CLI)**, .NET ne è l'implementazione commerciale di Microsoft. L'obiettivo è assicurare interoperabilità tra più linguaggi usando una singola virtual machine **CLR (Common Language Runtime)**

Questo viene realizzato attraverso:

- **CTS (Common Type System)**: definisce un modello comune per i dati e un ricco set di tipi.
- **CLS (Common Language Specification)**: definisce un subset di CTS e regole per compilatore e sviluppatori, per garantire l'interoperabilità
- **MSIL (Microsoft Intermediate Language)** è il bytecode della CLR, contenente anche metadati.

L'interoperabilità viene garantita sul sottoinsieme CLS di operazioni per ogni linguaggio. Viene garantita anche safety. Per il resto è operabile in modalità "unmanaged" (ovvero senza garanzie, il programmatore si prende carico di sicurezza e safety delle proprie implementazioni)

Viene introdotto un nuovo linguaggio (C#) che ha un mapping 1:1 sul CLS

**Application Domains**: sono "processi logici". All'interno dello stesso processo fisico posso identificare più sottoinsiemi di thread che rappresentano processi logici. Si usano metodi IPC per comunicare tra questi processi logici. Utili per generare diversi processi applicativi senza sprecare risorse.

**.NET Remoting**: è il framework di invocazione metodi (simile a Java RMI). Si generano i proxies in modo dinamico. Interfacce ed implementazione possono essere in linguaggi diversi. Le interfacce non sono obbligatorie. Gli oggetti vengono passati by value o by ref a seconda del tipo. Non c'è registro, invocazione è specificata tramite indirizzo esplicito. Attivazione avviene con molte opzioni (**SAO** Server Scivated Object: single call oppure singleton; **CAO**: Client Activated Object: ciclo di vita è affidato al client).

## **Paradigmi alternativi di Middleware**

Il paradigma classico (imperativo) ha un supporto esclusivamente P2P, usa comunicazione sincrona ed è "session oriented". Tutto questo lo rende molto rigido.

I paradigmi alternativi si basano principalmente sui **messaggi** (evento/messaggio). Si ha quindi il supporto multicast e i componenti sono meno rigidamente correlati.

**Modello di riferimento**: rappresentato dal **message passing**, replicando quindi una funzionalità tipicamente di rete a livello applicativo (**overlay network**)

La comunicazione può essere

- *Asincrona*: il sender continua immediatamente dopo l'invio del messaggio
- *Sincrona*: sender bloccato finchè il ricevente ha ricevuto/salvato/processato il risultato
- *Transiente*: il sender e il ricevente devono essere entrambi attivi perchè il messaggio venga consegnato
- *Persistente*: il messaggio è salvato nel sistema di comunicazione finchè non può essere consegnato.

**Tipi di comunicazione transiente**: Asincrona, Sincrona sull'ACK di ricezione, Sincrona sull'ACK di presa in carico, Sincrona sulla risposta.

Si ha via via un'aumento di garanzie ma una diminuzione sulla possibilità di concorrenza.

**Tipi di comunicazione persistente:** Asincrona (“send & forget”), Sincrona (sull'accettazione del messaggio da parte del sistema di comunicazione)

### **Message Oriented Middleware (message queuing)**

Si fornisce una comunicazione persistente asincrona punto-punto. Spesso vengono comunque implementate anche altre semantiche. Comunicazione fortemente disaccoppiata nello spazio-tempo, tipicamente si hanno garanzie solo sulla presa in consegna del messaggio da parte del sistema non sulla sua lettura.

Ogni componente ha una coda di ingresso ed una coda di uscita.

#### **Primitive di comunicazione:**

- **Put:** appendo il messaggio ad una coda specificata
- **Get:** blocco finchè non c'è un elemento sulla coda specificata, e poi lo prendo
- **Poll:** come sopra, non bloccante
- **Notify:** installo un handler, vengo chiamato quando un messaggio viene messo nella coda specificata

Se necessario è possibile implementare un'architettura client-server: ad esempio per implementare sistemi di gestione del carico (cosa difficoltosa con RPC), oppure per ambiti mobili dove il client o il server possono perdere il collegamento tra di loro molto facilmente.

#### **Servizi MOM:**

- *Gestione delle Code:* persistenti, transienti, gruppi di code, code remote
- *Trasporto messaggi:* punto-punto, punto-multipunto, priorità, garanzie di consegna variabili
- *Ricezione messaggi:* code condivise, politiche eterogenee (FIFO, priorità, filtri)
- *Directory Services, Security, Transazioni*

#### **Problematiche architetturali:**

- Code identificate da nomi simbolici: necessità di servizio lookup, oppure pre-deployed static topology
- Code manipolate da gestori appositi (*relay*)
- Relay organizzati in overlay network: possibile miglioramento della fault tolerance, implementa multicast senza usare IP-multicast
- **Message Brokers:** gateway di livello applicativo per la conversione dei messaggi tra vari formati eterogenei di altri sottosistemi

### **Publish-Subscribe**

I componenti di un'applicazione possono pubblicare in modo asincrono **notifiche di eventi** oppure possono dichiarare il loro interesse in talune classi di eventi, mandando una **sottoscrizione**. L'API è composta da due sole primitive (*publish*, *subscribe*) e le notifiche sono semplici messaggi.

Le sottoscrizioni sono registrate da un **Event Dispatcher** che è responsabile dell'instradamento degli eventi verso chi li ha sottoscritti (centralizzato oppure distribuito).

La comunicazione è transiente asincrona, implicita e multipunto.

Il grado di disaccoppiamento è molto elevato: facile modificare la topologia della rete, ed è appropriato per ambienti molto dinamici.

Si hanno dei linguaggi per descrivere le sottoscrizioni:

1. **Subject (topic) based:** classe determinata a priori, analogo al multicast
2. **Content based:** espressioni (*event patterns*) che permettono ai client di filtrare gli eventi a seconda del loro contenuto. Determinati dai client. Un evento può matchare più patterns.

I tradeoff delle due strategie implicano complessità di implementazione vs espressività

#### **Architettura del Dispatcher:**

1. **Centralizzato:** componente singolo si prende in carico tutte le sottoscrizioni e tutti i forwarding
2. **Distribuito:** un set di **message brokers** è organizzato in un **overlay network**. Essi cooperano per recepire le sottoscrizioni e per effettuare il routing dei messaggi. Le strategie per effettuare queste cose possono variare.

Ognun dispatcher inoltra le sottoscrizioni ai vicini. Gli eventi seguono le route disegnate dalle sottoscrizioni. Possono esserci ottimizzazioni dovute a relazioni di copertura.

Per migliorare l'efficienza e la scalabilità, generalmente le overlay network sono disposte ad albero. In questo caso sia i messaggi che le sottoscrizioni sono inoltrate dai broker verso la root. I messaggi scendono verso altri broker solo se una "matching subscription" è stata ricevuta su quella route.

**Riconfigurazione topologica:** può essere dovuta a modifiche fisiche oppure logiche della rete. Bisogna risolvere tre problemi

1. Ricostruire la Overlay Network: può essere supervisionata da un admin e dipende dallo scenario di utilizzo del middleware
2. Riarrangiare le tabelle di sottoscrizione:
3. Minimizzare la perdita di eventi: ad esempio utilizzando algoritmi epidemici

#### **Protocollo di Strawman:**

- Gli endpoint che vedono sparire/apparire un link si comportano come se avessero ricevuto disiscrizioni/sottoscrizioni da parte di quell'area della rete.
- **Bisogna evitare che un numero elevato di sottoscrizioni vengano rimosse per essere poi subito reinserite** (risolvibile usando concetti di disiscrizione ritardata con timeout)

#### **SUN JMS**

E' un set di specifiche con un'implementazione di riferimento. E' un building block, non si interessa di risolvere problematiche non strettamente funzionali quali sicurezza, fault tolerance. Accorpa MOM e Publish-Subscribe (comunicazione persistente, topic-based ma è permesso anche content-based filtering sui client)

#### **Linda & Tuple Spaces**

Modello di condivisione dati utilizzato tradizionalmente in ambiti di computazione parallela. La comunicazione è *implicita, persistente, generativa* e *content-based*. Si realizza un elevatissimo grado di disaccoppiamento tra gli endpoint.

**Architettura:** i dati sono organizzati in sequenze ordinate di campi tipizzati (*tuple*), raccolte in uno spazio persistente, globale e condiviso (*tuple space*). Le primitive standard sono:

- **out(t)**: scrittura della tupla t nel tuple space
- **rd(p)**: ritorna una copia di una tupla (scelta non deterministicamente se sono più di una) che corrisponde al pattern (o *template*) p. Chiamata blocking finchè la condizione non è verificata
- **in(p)**: simile a rd(p), ma elimina la tupla trovata dal tuple space

Ci sono molte varianti, anche asincrone, delle primitive di recupero tuple, e anche varianti di ritiro di gruppo etc... queste estensioni non sono di implementazione triviale se è necessario garantire certe proprietà quali l'atomicità.

**Problematiche Architettureali**: il modello non scala facilmente su una WAN, esso è solamente proattivo(i processi devono prendere l'iniziativa esplicitamente). Molte implementazioni commerciali introducono anche primitive di tipo *reattivo* (ad esempio *notify*)

## **Lime**

Cerca di realizzare il supporto per la mobilità (fisica e logica). Si basa sul concetto di **tuple space transitori**, che vengono connessi/disconnessi tramite operazioni di **engagement/disengagement**. E' possibile ottenere features addizionali rispetto ai sistemi tradizionali, come reazioni, scoping e indirizzamento.

## **Jini**

Piattaforma distribuita della SUN. Il modello di riferimento è session oriented. Serve per scoprire chi sta fornendo un certo servizio: una volta che questo è stato scoperto, si usa RMI per fare la richiesta come già visto in precedenza.

Sfruttamento del **codice mobile**: i proxies dei servizi vengono scaricati dinamicamente, e abilitano l'uso di un servizio da parte di un client non in possesso a priori del bytecode necessario.

**Lookup**: ogni servizio di lookup mantiene le seguenti triplette di informazioni:

- ServiceID: identificatore del servizio associato a questa tupla
- Service: un riferimento (anche remoto) all'oggetto che implementa il servizio
- AttributeSets: un set di tuple che descrivono il servizio

**ServiceRegistrar** definisce l'interfaccia per accedere al servizio di lookup. Ricerca basata sul tipo di servizio, vengono forniti metodi per effettuare lookups template-based oppure per abilitare notifiche di nuovi servizi presso chi è interessato (Viene utilizzato un JavaSpaces tuple space, che effettua solo exact-matching)

**Entry**: una tupla (da utilizzare con o senza il tuple space). Il matching viene effettuato su comparazione del form serializzato. I templates vengono confrontati con le entry presenti nelle query.

## **Protocolli di Discovery:**

- Multicast request protocol: ricerca dei servizi di lookup
- Multicast announcement protocol: utilizzati dai servizi di lookup per annunciare la loro presenza
- Unicast discovery protocol: permettono l'interazione con un certo servizio di lookup
- Esiste anche un protocollo di join molto semplificato

**Modello JINI:** viene utilizzato un modello molto semplice basato su **eventi** e **notifiche**: i client possono registrare dei listener su oggetti remoti. Però non ho un supporto per una distribuzione scalabile degli eventi. E' possibile interporre più oggetti sfruttando le interfacce. E' possibile gestire una *prenotazione delle risorse* tramite il meccanismo dei **lease**: possono essere esclusivi (controllo di concorrenza) oppure condivisi. La durata dei lease è negoziabile (a differenza di RMI), può essere rinnovata da chi detiene il lease.

## Naming

I meccanismi di naming sono utilizzati per fornire un nome logico ad una risorsa: la struttura di questo nome influenza il modo con cui esso può essere utilizzato (e risolto) in un sistema distribuito. Da un lato si può utilizzare un **nome puro** (totalmente opaco) oppure un **indirizzo** (che identifica esattamente il punto d'accesso). Lo scope di un nome può essere globale oppure locale (dipendente dal contesto in cui viene utilizzato). I nomi sono generalmente organizzati in **namespaces** (meccanismi di aliasing, mounting, merging).

**Risoluzione:** è il processo tramite il quale si ottiene un riferimento all'entità, dato il suo nome. Un meccanismo di **chiusura** fornisce il punto di inizio per la risoluzione dei nomi. Questo viene svolto da un **name service**.

## DNS

Il **Domain Name Service** è utilizzato per risolvere i nomi associati ai domini internet. E' un name service organizzato gerarchicamente su più layer (globale, amministrativo, manageriale): ogni layer è caratterizzato da differenti condizioni di lavoro (# di nodi, tempi di risposta, repliche, uso di caching etc).

La risoluzione può avvenire **iterativamente** (la gerarchia viene navigata dal client con richieste multiple) oppure **ricorsivamente** (la richiesta arriva al server di gerarchia più alta, che provvede a fare chiamate ricorsive verso altri server inferiori finché non ottiene la risposta). Nel secondo caso si ha più carico sui name server, ma si risparmia sulla comunicazione e si può sfruttare più efficacemente i meccanismi di caching

DNS è organizzato come un albero. Ogni sottoalbero è un dominio, e appartiene ad un'autorità separata. Ogni server è responsabile per una zona. I DNS Record sono di vari tipi e contengono informazioni diverse. E' possibile gestire la ridondanza dei nodi con priorità.

I server globali (al vertice della gerarchia) sono mirrorati e usano indirizzi IP Anycast per bilanciare il carico; vengono inoltre usati server secondari refreshati periodicamente con informazioni aggiornate e servizi di caching. Sono possibili temporanee inconsistenze (la cui durata temporale aumenta all'aumentare della gerarchia del server)

**DNS ed entità mobili:** se l'host mobile resta nel dominio originale, basta solo aggiornare il database relativo a quel dominio. Se invece il cambiamento interessa anche il dominio ci sono due modi:

- Vecchio server DNS fornisce anche l'indirizzo della nuova locazione (overhead amministrativi)
- Vecchio server DNS fornisce nome simbolico della nuova locazione (lookup inefficiente)

In ambiti mobili queste soluzioni non sono accettabili: si passa quindi allo **sdoppiamento** in **naming service** e **location service** (*nome -> access point -> indirizzo fisico*)

Soluzioni semplici: in ambito LAN si possono utilizzare meccanismi di tipo *broadcast* (**find msg**, simile ad ARP, molto overhead di traffico e di processing), di tipo *multicast* (si cerca di ridurre gli overheads) oppure utilizzando *forwarding pointers* (logiche next-op a catena, poco robuste)

**Forwarding Pointers e oggetti distribuiti:** i proxies inoltrano le richieste alle istanze attuali; si possono mantenere le catene di risoluzione corte con aggiustamenti a runtime e cancellazione degli skeleton non referenziati da nessuno. Le rotture della catena vengono gestiti con un punto d'accesso noto.

**Home Nodes:** si utilizza un home node (assunto come stabile, eventualmente replicato) che conosce sempre la locazione dell'unità mobile. Questo fa incrementare la latenza sul lookup (si aggiunge sempre l'hop verso l'home node). Si possono adottare anche soluzioni two-tier (registro locale or home node) oppure gerarchiche per incrementare l'efficienza.

**Come distribuire le informazioni di locazione:**

- Il nodo root ha entry per ciascuna entità
- Le entries puntano al prossimo subdomain
- Le foglie contengono gli indirizzi di un'entità
- Entità possono avere più indirizzi in differenti domini (foglie) per gestire replicazione

Il **lookup** inizia nel dominio dove risiede il client, e si propaga verso la radice finchè non si trova una entry di livello superiore che matcha. A quel punto si ridiscende l'albero verso la foglia che contiene l'indirizzo del dominio.

**Ottimizzazioni:**

- **Caching:** se effettuato direttamente sugli indirizzi è inefficiente. E'possibile creare delle shortcuts se si hanno a disposizione informazioni sulla mobilità
- **Scalabilità:** il nodo root deve avere entries per tutte le entità. I record sono piccoli ma c'è un collo di bottiglia sul nodo root per le lookup. Si può distribuire la root ma a quel punto l'allocazione delle entità diventa non triviale.

**GSM**

- **SIM** (Subscribe Identity Module): inserita nel terminale mobile, disaccoppia il subscriber dal terminale specifico
- **BTS** (Base Transceiver Station) collega il terminale all'infrastruttura
- **BSC** (Base Station Controller) controlla un gruppo di BTS e si interfaccia sul Network
- **MSC** (Mobile Switching Centre) switch che collega il network radio con la rete telefonica pubblica
- **HLR** (Home Location Register) è un database per i dati sui subscriber (statici e dinamici, come la last location known e lo stato del terminale)
- **VLR** (Visitor Location Register) database temporaneo per i subscriber che entrano nella zona di gestione di un MSC
- **AuC** (Authentication Centre) database che detiene informazioni sensibili di autenticazione per i subscriber

**Rimozione delle entità non referenziate:** come gestire la rimozione di entità non raggiungibili dal rootset? Nel caso non distribuito esistono meccanismi di **garbage collection**, la cui logica non è direttamente trasferibile nel caso distribuito (non si ha conoscenza globale dello stato d'uso delle risorse, e la rete non è infallibile)

- **Reference counting:** l'oggetto (nel proprio skeleton) mantiene traccia di quanti altri oggetti hanno ricevuto il suo riferimento. Bisogna garantire semantica **exactly-once** (ACK & DUP

deletion). Si hanno race conditions sul passaggio riferimenti tra processi distribuiti

- **Weighted Reference Counting:** si comunicano solamente i decrementi del contatore. Si richiede un contatore addizionale. Quando si rimuove un reference, si sottrae il contatore parziale a quello globale: quando i due sono uguali, allora l'oggetto può essere rimosso. In questo caso però si possono generare solamente un numero fisso di references
- **WRC indiretto:** si interpone un processo con un proprio skeleton prima dell'oggetto. Questo rimuove la limitazione del numero di references, ma introduce un hop addizionale sull'accesso
- **Reference Listing:** si tiene traccia dell'identità dei proxies che accedono all'oggetto. Si ottiene *idempotenza* sull'inserimento/rimozione (posso usare comunicazione non reliable). E'più facile mantenere la lista consistente rispetto alle network failures. Si hanno race conditions quando si copiano references. Utilizzato in JAVA RMI

**Identificare entità irraggiungibili:** si usano garbage collections tracing-based: richiede conoscenza di tutte le entità (poco scalabile)

Su sistemi uniprocessore si usa la tecnica di **mark & sweep**: nella prima fase si marcano le entità accessibili seguendo le referenze. Nella seconda fase c'è la ricerca esaustiva nella memoria per trovare entità non marcate da rimuovere. Esistono sistemi di **mark & sweep distribuito**: richiede però che il grafo di raggiungibilità rimanga stabile durante la garbage collection: transazione distribuita che blocca l'intero sistema, problemi di scalabilità.

## Sincronizzazione

La problematica di sincronizzazione di attività concorrenti in contesto distribuito non è triviale:

- Mancanza di clock fisico globale
- Mancanza di global shared memory
- Failures parziali

## Temporizzazione

Come fare in modo che il global clock sia lo stesso per ogni macchina in distribuito?

**Drift Rate:** quanti secondi l'orologio perde ogni tot secondi

**Skew Rate:** ritardo massimo tollerabile prima di dover risincronizzare

- Sincronizzare i clock rispetto a un clock di riferimento (*accuracy*)
- Sincronizzare tutti i clock tra di loro (*agreement*)

**Algoritmo di Cristian:** ogni server fa richiesta periodica a un time server. Si assume che i messaggi siano "rapidi" ( $T_{tx}$  nullo) rispetto all'accuratezza richiesta. Se  $T_{tx}$  non è nullo, si assume  $T_{tx} = RTT$  da cui  $T_{client} = T_{utc} + 0.5 RTT$

**Algoritmo Berkeley UNIX:** il time server colleziona il tempo di tutti i client, fa la media, e ritrasmette il delta di aggiustamento.

**Network Time Protocol (NTP):** UTC synch su reti di larga scala. Protocollo implementato sopra UDP in internet. Si effettua *sincronizzazione gerarchica* delle subnet organizzate in *strati*. Nello strato 1 stanno i server collegati a sorgenti UTC (informazioni più accurate). Più si scende nell'albero, più si perde accuratezza.

**Funzionamento:** due macchine si scambiano un certo numero di messaggi. Si stima il ritardo tra i

due orologi e dell'aggiustamento necessario per ottenere la precisione voluta. Si creano due stime (**ritardo** =  $t+t'$  e **offset** =  $q + (\text{ritardo}/2)$ )

La coppia  $\langle \text{ritardo}, \text{offset} \rangle$  viene fatta passare in filtri statistici che decidono come impostare la sincronizzazione e verificano l'affidabilità della stima.

Nei sistemi distribuiti il problema principale tuttavia non è rappresentato dalla precisione del timestamp temporale, ma dal mantenimento dell'ordinamento sugli eventi e delle relazioni causa-effetto.

## Orologi Scalari

Concetto introdotto da Lamport. Si basano sulla relazione **happens-before** ( $A \rightarrow B$ )

- Se A e B avvengono nello stesso processo, e A avviene prima di B, allora  $A \rightarrow B$
- Se  $A = \text{send}(\text{msg})$  e  $B = \text{recv}(\text{msg})$ , allora  $A \rightarrow B$
- **happens-before** gode della proprietà transitiva
- IF (NOT( $A \rightarrow B$ ) AND NOT( $B \rightarrow A$ )) THEN A e B sono **concorrenti**

I clock sono rappresentati da numeri interi, e non hanno relazione con il tempo fisico. Ciascun processo ha un orologio logico: parte da zero e si incrementa ad ogni occorrenza di un evento. Ogni messaggio inviato in rete ha il timestamp del server da cui è stato inviato. Alla ricezione, il ricevente imposta il proprio clock come **MAX (msg TS, myClock) + 1**

Si ottiene solo ordinamento parziale: ordinamento totale si può imporre propagando anche il PID assieme al valore del clock sui messaggi (convenzionalmente)

**Totally Ordered Multicast**: l'obiettivo è preservare un ordinamento coerente su tutti i nodi nella ricezione dei messaggi (senza sapere qual'è l'ordinamento). Voglio che tutti i nodi abbiano la stessa vista, indipendentemente da quale sia la vista. *Assunzione*: i link sono stabili e FIFO, i nodi inoltre sanno a priori da chi si aspettano gli ACK.

Per risolvere il problema, i riceventi accumulano i messaggi ricevuti in un buffer, ordinandoli per timestamp. L'ordinamento è quindi realizzato se il messaggio è consegnato all'applicazione nell'ordine imposto dal buffer ed al ricevimento di tutti gli ACK.

I clock scalari assicurano che  $A \rightarrow B \Rightarrow \text{CLK}(A) < \text{CLK}(B)$ , non necessariamente il contrario!

## Orologi Vettoriali

Gli orologi scalari non catturano tutte le possibili relazioni di causalità. **Orologi Vettoriali**: per ogni processo si ha un vettore V di N valori (N processi). Il primo valore è l'orologio scalare del processo, gli altri rappresentano la conoscenza su "dove sono arrivati" gli altri processi secondo le informazioni fin qui disponibili.

Inizialmente  $V_i[j] = 0$  per ogni  $i, j$  (i e j sono gli indici dei processi; i fissato)

- **Per ogni evento locale su  $P_i$** :  $V_i[0] = V_i[0]++$
- **Su ogni messaggio inviato da  $P_i$** , si attacca  $T = V_i$
- **Per ogni messaggio ricevuto su  $P_i$  da  $P_j$  contenente T**:  $V_i[j] = \text{MAX}(V_i[j], T[j]) + 1$

Grazie a questo meccanismo si può stabilire che (**CAUSALITA'**):

$A \rightarrow B \Leftrightarrow V(A) < V(B)$

$A \parallel B \Leftrightarrow V(A) \parallel V(B)$

## **Mutua Esclusione**

E' necessaria per garantire consistenza sui dati condivisi

- **Safety**: un solo processo può eseguire nella sezione critica
- **Liveness**: prima o poi chiunque richiede una risorsa deve ottenerla

**Soluzione Semplice**: server centralizzato che gestisce i lock con tokens. Facile da implementare ma si ha un bottleneck sulle performances e un singolo punto di fallimento.

**Utilizzo degli Scalar Clock**:

- P<sub>j</sub> manda messaggio **m** di richiesta risorsa con il timestamp T<sub>m</sub> a tutti i processi
- Richiesta è messa in una coda locale ordinata su T<sub>m</sub> (ed eventualmente su PID)
- Chi riceve **m** manda un ACK a P<sub>j</sub> con TS > T<sub>m</sub>
- Per rilasciare la risorsa, viene mandato un messaggio a tutti
- Quando il messaggio di rilascio è ottenuto, si rimuove la richiesta dalla coda

La risorsa è garantita a P<sub>j</sub> quando

1. Il suo messaggio di richiesta è prima di tutti gli altri nella coda
2. La sua richiesta è stata ACK da tutti gli altri processi con un messaggio > T<sub>m</sub>

**Token Ring**: i processi sono logicamente organizzati in un anello. L'accesso alla risorsa è garantito da un token che è inoltrato in una data direzione dell'anello. Chi ha il token può accedere. Esso inoltre deve circolare indipendentemente dalle richieste d'accesso.

## **Leader Election**

In molti algoritmi distribuiti è richiesto un processo che agisca da coordinatore. Come si fa a far accordare tutti su chi è il leader? Assunzione minima: è possibile distinguere un nodo dall'altro. Inoltre ogni processo sa che esiste ogni altro processo (non necessariamente sa se è up o se è crashato). Staticamente si decide che il leader è quello con il PID più elevato.

**Bully Election**: se avviene un crash del coordinatore, il primo processo che se ne accorge invia un messaggio di **election** a tutti i nodi con il PID maggiore del proprio. Chi lo riceve, risponde con un ACK. Chi riceve un messaggio di election, ne manda uno nuovo ai processi con ID superiore. Ovviamente ad un certo punto un processo invierà un messaggio di election e non otterrà risposta (Perchè l'unico che potrebbe mandargliela, il vecchio leader, è crashato. Oppure è il leader stesso che è tornato online e non esiste nessun processo con PID maggiore). A quel punto, il processo diventa leader e manda un messaggio broadcast **Coordinator** a tutti gli altri per informarli.

**Ring Based**: si assume topologia ad anello (fisica o logica). Quando un processo si accorge della caduta del coordinatore, manda un messaggio election al più vicino "in avanti" contenente il proprio PID. Un processo P riceve un msg di election: se non è tra i PID nel messaggio, aggiunge il proprio e propaga. Se trova il proprio PID nel messaggio, cambia il messaggio in Coordinator e lo fa ricircolare.

Un processo che riceve un messaggio Coordinator, legge la lista dei PID e considera leader quello più alto (e nel frattempo sa anche chi è ancora online).

## **Global Snapshot**

Lo stato globale di un sistema distribuito consiste nello stato locale di ciascun processo assieme

all'immagine dei messaggi in transito sui link. Serve per effettuare debugging, rilevamento terminazioni e deadlock e così via.

Uno **snapshot distribuito** riflette uno stato globalmente consistente nel quale il sistema distribuito avrebbe potuto trovarsi (non ci devono essere messaggi in arrivo "dal futuro", se è stato ricevuto un ACK allora deve essere stato mandato anche il messaggio relativo)

**Soluzione semplice:** fermo tutti i processi, lascio svuotare i canali, registro lo stato, riavvio il sistema. Non va bene!

**Snapshot Distribuito (Chandy – Lamport):** si assumono canali FIFO, link e nodi affidabili.

- Uno dei processi inizia lo snapshot: salva il proprio stato interno e invia un token su tutti i canali in uscita (segnalando agli altri che sta per iniziare uno snapshot). Da questo momento, registra tutti i messaggi in ingresso.
- Un processo riceve un token:
  - Se non sta già registrando, fa come sopra
  - Inoltre ferma la registrazione relativa al canale dove ha ricevuto il token
- Registrazione dei messaggi:
  - Se un messaggio arriva su un canale che sta registrando, si salva l'arrivo e si continua a processare normalmente
  - Altrimenti effettuo il processing normale del messaggio senza loggarlo
- Lo snapshot è completo quando i token sono arrivati su tutti i canali entranti

Questo meccanismo implementa la scelta di un **taglio consistente** nella storia d'esecuzione, non richiede il blocco della computazione (performance), ed è possibile svolgere più snapshots concorrenti associando ID ai makers. Possibili variazioni (snapshot incrementale)

**Caso Agenti mobili:** le applicazioni devono comunicare comandi e dati ad agenti mobili. Serve un meccanismo per garantire la consegna (uni o multicast) di questi messaggi.

**Utilizzo dei Proxy:** durante il movimento, l'informazione al proxy sulla posizione dell'agente può non essere aggiornata, i messaggi potrebbero "inseguire" l'agente per un tempo molto lungo.

**Spanning Tree Broadcast:** gli agenti possono perdere il messaggio spostandosi da una regione dove il messaggio non è ancora giunto ad una dove è già giunto

**Snapshot Delivery:** lo snapshot ha associata una vista consistente dello stato del sistema. I nodi in processing "intrappolano" l'agente in una regione dalla quale non può uscire senza aver ricevuto una copia del messaggio. **Ciascun messaggio nel sistema appare esattamente una volta in ciascuno snapshot locale.**

Nodo <-> Mobile Agent Server

Message <-> Mobile Agent

Marker <-> Messaggio Applicativo

Record del messaggio <-> Consegna del messaggio

Termine dello snapshot <-> Cancellazione del messaggio

Il messaggio viene immagazzinato quando arriva la prima copia. Il messaggio viene consegnato quando messaggio e agente sono nello stesso punto. Il messaggio viene cancellato quando una copia è arrivata su tutte le direzioni entranti.

Questo metodo impone una semantica *exact-once* indipendente dal movimento dell'agente. Funziona se si conoscono in anticipo i vicini sulla rete. Questo non è sempre vero per reti particolarmente dinamiche.

???? AGENTI MOBILI ???

### **Termination Detection**

Riconoscere quando una computazione è completata oppure è in deadlock. Tutti i processi dovrebbero essere idle e non dovrebbero esserci messaggi in rete.

Si può usare lo snapshot distribuito, ma i canali devono essere vuoti quando termina.

- Un processo termina la sua computazione dello snapshot, invia un messaggio al processo dal quale ha ricevuto il marker
- DONE message (solo se non si è osservato alcun messaggio tra recording e marker) altrimenti CONTINUE message
- Se l'iniziatore arriva tutti i messaggi DONE, la computazione è terminata, se no è necessario un altro snapshot

**Diffusing Computation:** tutti i processi sono inizialmente fermi tranne il processo di init. I processi vengono attivati quando ricevono un messaggio. La condizione di terminazione è la stessa.

**Algoritmo di Dijkstra-Scholten (Termination Detection):**

- Si crea un albero dei processi attivi
- Quando un nodo termina di processare ed è una foglia, lo elimino dall'albero
- Quando rimane soltanto la radice ed ha completato il processing, allora il sistema ha terminato di eseguire

Bisogna creare un albero e mantenerlo aciclico. Bisogna distinguere i nodi foglia dagli altri.

**Realizzazione:** ogni nodo tiene traccia dei propri figli (quelli a cui ha inviato un messaggio). Se un nodo era già sveglio, allora era già parte dell'albero e non deve essere aggiunto come figlio del mittente. Quando un nodo non ha più figli ed è idle, allora dice al padre di toglierlo come figlio.

Snapshot Distribuito

- Overhead : 1 messaggio per link
- Costi per collezionare i risultati
- Se il sistema non ha terminato, devo rifare tutto!

Dijkstra – Scholten

- Overhead dipendente dal numero di messaggi nel sistema
- Non coinvolge i processi mai attivati
- Terminazione rilevata quando si riceve l'ultimo ACK

### **Transazioni Distribuite**

Si protegge una risorsa condivisa dall'accesso di processi concorrenti. Le transazioni sono sequenze di operazioni (BEGIN\_T, END\_T, ABORT\_T, READ, WRITE)

Proprietà ACID

- **Atomicità:** la transazione è vista come un'unità indivisibile dall'esterno
- **Consistenza:** la transazione non viola gli invarianti del sistema
- **Isolamento:** transazioni concorrenti non interferiscono tra di loro
- **Durata:** dopo il commit, le modifiche sono permanenti

#### *Tipi di transazioni:*

- **Flat:** transazioni ACID come prima
- **Innestate:** costruiti da sottotransazioni. E' possibile fare l'abort delle sottotransazioni una volta committate. La Durability si applica solo alla transazione top-level. Concettualmente le sottotransazioni operano su una copia privata dei dati. In genere le sottotransazioni girano su host differenti
- **Distribuite:** essenzialmente è una transazione FLAT compiuta su dati distribuiti. E' necessario implementare il locking distribuito.

#### *Come implementare le transazioni*

- **Private Workspace:** copio i dati che vengono modificati in uno spazio separato di memoria (creo **shadow blocks** dei file originali). Se la transazione è abortita, si cancella il private workspace, altrimenti si copiano nel workspace originale. Si ottimizza replicando gli indici e non tutto il file.
- **WriteAhead Log:** I file sono modificati sul posto, ma si mantiene un log che comprende la transazione che ha effettuato il cambiamento, su quale file/blocco e i valori (vecchio e nuovo). Si modifica il file solo dopo che il log è stato scritto. Se la transazione ha successo, si scrive il commit sul log, altrimenti si effettua il *rollback*

**Controllo di concorrenza:** permette a più transazioni di essere eseguite simultaneamente, mantenendo la consistenza. Si implementa con diversi componenti: Transaction Manager, Scheduler e Data Manager

**Serializzabilità:** date le operazioni elementari di scrittura e lettura, è possibile riorganizzarle in modo da eseguire più transazioni in parallelo. Le operazioni devono essere riorganizzate in modo consistente dal sistema, liberando il programmatore da meccanismi espliciti di mutua esclusione.

**2PL Locking:** si effettua il locking delle risorse in modo incrementale, non permettendo ad una transazione di acquisire nuovi lock una volta che ne ha rilasciato qualcuno. Nella versione *strict* i lock vengono rilasciati tutti assieme (evita abort a cascata). Può causare deadlock

- **2PL Centralizzato:** il TM contatta un LM centralizzato, riceve i grant di lock e interagisce direttamente con i dati, poi ritorna i grants al LM
- **2PL Primario:** esistono più LM; ogni dato ha una copia primaria su un certo host. Il LM di quell'host è responsabile della copia primaria e dei suoi lock.
- **2PL Distribuito:** i dati possono essere replicati su più host. Il LM su un host è responsabile della replica locale e per contattare il Data Manager locale.

**Timestamping:** si assegna un timestamp ad ogni transazione (Tread, Twrite). Tra due operazioni in conflitto viene scelta quella con il TS più basso. Le transazioni abortite ripartono con un nuovo timestamp. Non c'è deadlock ma c'è possibilità di starvation.

**Ordinamento Timestamp Ottimistico** (variante): si assume che i conflitti siano rari. Faccio quello che voglio quando mi pare, e metto a posto i conflitti in un secondo tempo. I data items vengono marcati con il tempo iniziale della transazione. Al momento del Commit, se alcuni item sono

cambiati dall'inizio, si fa l'abort. Deadlock-free e permette il massimo parallelismo. Sotto condizioni di carico imponente ci sono un sacco di rollback.

**Deadlock Distribuiti:** nei sistemi distribuiti è più difficile trattare i deadlock. Sono presenti diversi metodi di approccio:

- Ignorare il problema
- Detection & Recovery (killo uno dei processi coinvolti)
- Prevention (impedisco di creare le condizioni potenziali per un deadlock)
- Avoidance: non usato nei sistemi distribuiti. Implica conoscenza a priori del sistema sull'uso delle risorse

Se si usano transazioni distribuite, questo aiuta perchè è meno distruttivo abortire una transazione che killare un processo.

**Centralized Deadlock Detection:** ogni macchina mantiene un grafo locale delle risorse e lo manda ad un coordinatore. Il coordinatore riceve periodicamente le informazioni, oppure quando avviene una modifica, oppure le ottiene on-demand. Non funziona bene a causa dei **falsi deadlock** dovuti ai ritardi di propagazione dei messaggi sulla rete.

**Distributed Deadlock Detection (Chandy-Misra-Haas):** i processi possono richiedere più risorse simultaneamente. Quando un processo si blocca, manda un messaggio di **probe** ai processi che detengono le risorse. Si vede il deadlock se il messaggio di probe ritorna a chi l'ha generato. A quel punto:

- L'iniziatore decide di auto killarsi. Può causare abort non necessari di molti processi se c'è più di un iniziatore nel loop
- L'iniziatore sceglie il processo con il PID più alto e lo killa (bisogna mettere il PID sul probe)

**Distributed Prevention:** si eliminano i deadlock by-design

- **wait-die algorithm:** quando un processo A sta per bloccarsi su una risorsa già in utilizzo, si permette a A di aspettare solamente se A è più vecchio di B. Altrimenti killo A.
- **Wound-wait algorithm:** se è possibile l'interruzione dei processi, si effettua la preemption sul processo più giovane. Avviene l'abort di una transazione, ma non la kill del processo.

## Consistenza e Replicazione

La replicazione dei dati permette di

- Aumentare le performances (accesso più locale, workload sharing)
- Aumentare la disponibilità (dati non disponibili per failure, dati "intermittenti" nei mobile agents)
- Ottenere robustezza (fault tolerance)

Esempi dov'è usata la replicazione: Distributed Shared Memory, Web Caches, Distributed File Systems, DNS ...

Il problema principale è **mantenere la consistenza:** la modifica ad una replica deve riflettersi su tutte le altre. Accedere ad una replica deve essere indistinguibile dall'accesso all'originale. Tutto questo con un limitato overhead sulla comunicazione!

Tradeoff sulla scalabilità e sulla performance.

## Consistency Models

Incentrati su un data store distribuito. *Una read deve mostrare il risultato dell'ultima write.* Un modello di consistenza è il contratto tra i processi e il data store. Indebolisce il modello di programmazione in maniera più o meno marcata, togliendo garanzie per favorire performances e facilità d'uso

- **Consistenza Stretta:** *"ogni lettura su X ritorna il valore della write più recente su X."* Si mantiene il global ordering. Il concetto di "più recente" richiede tempo globale, ma nella pratica questa garanzia è possibile solo su una macchina uniprocessore.
- **Consistenza Sequenziale:** *"Le operazioni di tutti i processi vengono eseguite in un qualche ordine sequenziale, e le operazioni di ogni processo appaiono in ordine di programma all'interno di questa sequenza".* Le operazioni all'interno di un processo non possono essere riordinate. Tutti i processi vedono lo stesso interleaving. Non c'è concetto di tempo.
- **Linearizzabilità:** *"Il sistema ha consistenza sequenziale; inoltre, se  $TSop1(x) < TSop2(Y)$  allora  $OP1(X)$  precede  $OP2(Y)$  nella sequenza di operazioni".* Viene chiamata anche "consistenza atomica". Più forte di quella sequenziale, meno di quella stretta. Utile solo per le verifiche formali
- **Consistenza Causale:** *"Le write che hanno una relazione potenziale di causalità devono essere viste da tutti i processi nello stesso ordine. Inoltre le write concorrenti possono essere viste in qualsiasi ordine su macchine diverse".* Simile alla causalità dei vector clock. Le read sono causalmente correlate alle write prima e dopo di esse.
- **Consistenza FIFO:** *"Le write eseguite da un singolo processo sono viste da tutte le altre nell'ordine in cui sono state eseguite; le write di processi differenti possono essere viste in qualsiasi ordine".* Si scarta la causalità tra processi differenti. Chiamato anche Consistenza Pipelined RAM (PRAM). Facile da implementare ma controintuitivo

In certi casi non è detto che tutte le writes debbano essere visibili in tutti i processi. Si introducono le **synchronization variables**: le writes diventano visibili solamente quando i processi richiedono questo attivamente impostando la variabile. Si forza la consistenza solo quando questa è veramente necessaria, in modo esplicito.

- **Consistenza debole**
  1. L'accesso alle variabili di sincronizzazione è sequenzialmente consistente
  2. Non sono permesse operazioni su variabili di sincronizzazione finchè tutte le write precedenti non sono completate
  3. Non si permettono write/read se prima tutte le operazioni sulle variabili di sincronizzazione non hanno terminato

Si forza la consistenza su un gruppo di operazioni. Si limita solamente lo span temporale di validità della consistenza. Negli altri istanti i dati possono anche essere inconsistenti

- **Rilascio della consistenza:** il data store non può distinguere tra una richiesta di sincronizzazione per distribuire le write o per leggere dati consistenti. Si ha quindi un overhead non necessario. Si risolve introducendo differenti operazioni di sincronizzazione
  1. *Acquire:* si indica l'ingresso in una regione critica
  2. *Release:* si indica l'uscita da una regione critica

Può essere definita sui singoli item, non necessariamente si applica a tutto il data store.

- Prima di effettuare una read o una write, tutte le acquire precedenti dal processo

devono essere state completate con successo

- Prima che sia permessa una release, tutte le read e le write precedenti del processo devono essere state completate
- L'accesso alle variabili di sincronizzazione ha consistenza FIFO

Sulla release, i dati protetti che sono cambiati devono essere propagati alle altre copie: **Eager release** (tutte le update vengono pushate sulle altre repliche al momento della release) **Lazy release** (sulle acquire, i processi devono procurarsi la versione più recente dei dati dagli altri processi)

- **Entry Consistency**: si associa esplicitamente ciascun oggetto condiviso con una variabile di sincronizzazione. L'accesso avviene sempre con metodi *acquire-release*. Si riduce l'overhead sugli update ma si aumenta la complessità dell'accesso. Accesso **non esclusivo** oppure **esclusivo**.
  1. Acquire su una synchvar non è permesso ad un processo finché tutti gli update di quel dato protetto sono stati effettuati rispetto a quel processo
  2. Prima dell'accesso esclusivo ad una variabile di sincronizzazione, nessun processo deve detenere la variabile di sincronizzazione, anche in modo non esclusivo
  3. Dopo che è stato effettuato l'accesso esclusivo ad una synchvar, allora nessun altro processo può accedere a quella synchvar finché non ha eseguito rispetto al detentore di quella variabile (????????)

Sulle acquire, tutti i dati protetti devono essere resi visibili. Per poter rilasciare accesso esclusivo, nessun altro processo deve detenere lock di qualsiasi tipo. Dopo che un dato è stato utilizzato in modalità esclusiva, allora tutti i futuri accessi devono avvenire tramite il processo di acquire

I modelli fin qui considerati sono **data-centric**: fornire una vista di sistema consistente, di fronte a update simultanee e concorrenti. Tuttavia ci sono situazioni in cui non si hanno update simultanee (oppure sono facili da risolvere) e si effettuano molte read. In questi sistemi molto spesso è sufficiente utilizzare la **Consistenza Finale**: si garantisce che alla fine le update saranno propagate a ciascuna replica.

**Client-centric Consistency**: si provvedono garanzie sugli accessi visti dalla prospettiva di un singolo cliente (Bayou):

- **Monotonic Reads**: "Se un processo legge il valore di  $x$ , allora tutte le letture successive da parte di quel processo forniranno sempre quel valore o uno più recente"
- **Monotonic Writes**: "Un'operazione di scrittura su  $x$  da parte di un processo sarà completata prima di alcuna write successiva su  $x$  da parte di quel processo"
- **Read your Writes**: "L'effetto di una operazione di write su  $x$  sarà sempre visto da operazioni di read  $x$  successive, da parte dello stesso processo"
- **Writes Follow Reads**: "Un'operazione di write da parte di un processo su  $X$ , che segue una lettura precedente dello stesso valore su  $x$ , è garantita eseguirsi su quello stesso valore letto oppure su uno più recente"

Problemi

- Come / dove posizionare le repliche?

- Come propagare gli updates?
- Come mantenere le repliche consistenti?

Posizionamento:

1. **Permanenti**: configurate staticamente
2. **Server-initiated** : create dinamicamente (bilanciamento del carico) per muovere i dati vicino ai client. Basate sul profiling, richiedono conoscenze topologiche
3. **Client-initiated**: client caches. Possono essere anche condivise tra più client. Non c'è coinvolgimento del data-store (in linea di principio)

Propagazione degli Update

1. **Propagazione notifica**: update una copia e propago solo la notifica. Generalmente assieme a protocolli di invalidazione. Funziona bene se `#read << #write`
2. **Propagazione dati**: funziona bene se `#read >> #write`
3. **Active Replication**: propago informazioni per abilitare l'update su tutte le copie. Piccolo overhead comunicazionale, ma può implicare molto processing.

Come propagare

1. **Push (server)-based** : l'update è propagato a tutte le repliche, indipendentemente dai loro bisogni (bene se `#read >> #write`). Mantiene alto grado di consistenza. Tipicamente multicast.
2. **Pull (client)-based**: update on-demand. (bene se `#read << #write`). Tipicamente unicast.

Si possono utilizzare dei meccanismi di **lease** per switchare le politiche da pull a push a seconda dei bisogni.

3. **Algoritmi Epidemici**: il server con l'update è detto **infettivo**. Quelli che non sono ancora stati raggiunti dall'update è detto **suscettibile**. Un server updatato che non vuole propagare è **rimosso**. Simile alla propagazione delle malattie. Proprietà intrinsecamente distribuite e ridondanti (scalabile, fault-tolerance, resistente a cambi topologici). Di natura probabilistica (non offre garanzie deterministiche ma permette forti risparmi)

Strategie di propagazione negli algoritmi epidemici:

- **Anti-entropy**: scelta casuale di un server e scambio degli update (push, pull, push-pull). Notifiche sugli update visti o mancati. Alla fine tutti i server ricevono l'update
- **Gossiping** (rumor spreading): un update innesca un altro update verso un differente server. Se quello conosceva già l'update effettuato, si riduce la probabilità di effettuare un nuovo gossip. Nessuna garanzia di propagazione, ma alta velocità di diffusione.

Si possono effettuare mix dei precedenti, con differenti strategie e variazioni. Le cancellazioni possono essere gestite come updates ("death certificates")

**Primary-based consistency protocols**: tutte le scritture passano attraverso un singolo server primario, che deve garantire la consistenza degli accessi. Può essere complementato da server backup:

- **Remote Write**: server remoto prende in carico la gestione dell'accesso ai dati. Si implementa facilmente la consistenza sequenziale. Tuttavia ho degrado di performances con clients bloccati mentre avviene la propagazione dell'update.
- **Local Write**: il dato migra al server che richiede l'update, il quale diventa il nuovo primario

(come trovarlo?). Si possono usare local-write con backup per supportare operazioni disconnesse in ambiti mobili.

- **Replicated-Write protocols:** le operazioni di scrittura vengono eseguite su repliche multiple. **Active replication:** update inviati a ciascuna replica, va preservato l'ordine delle operazioni. **Quorum based:** le update avvengono solo se un quorum di server concorda sul version number da assegnare. La lettura richiede un quorum per assicurare che sia letta la versione più recente.  $N_r$  (read quorum)  $N_w$  (write quorum).  $N_r + N_w > N$  evita conflitti RAW.  $N_w > N/2$  evita conflitti WAW

Esempio: Bayou (vedi ultime due slides)

## Fault Tolerance

Come definire la fault tolerance in un sistema distribuito? Availability, Reliability, Safety, Maintainability. Un sistema fallisce quando non può provvedere servizi. Il fallimento è il risultato di un errore nello stato del sistema. Un errore è causato da un fault.

I fault possono essere

- **Transitori:** accadono una volta e spariscono
- **Intermittenti:** compaiono e scompaiono senza ragione apparente
- **Permanenti:** persistono finchè non si ripara il guasto

Tipi di server failures in un sistem distribuito:

- **Crash:** halt del server, risultati corretti fino all'halt
- **Omission:** server non risponde alle richieste (non riceve messaggi o non riesce a inviarli)
- **Timing:** il server risponde in un intervallo temporale al di fuori di quello previsto
- **Response:** il server risponde in modo incorretto (valore errato oppure stato errato)
- **Byzantine:** il server produce errori arbitrari in tempi arbitrari

In molti casi è difficile stabilire l'origine del guasto, o persino capire se si sta verificando un guasto oppure tutto funziona correttamente. E'difficile anche distinguere un server halted da un server lento.

La tecnica principale per mascherare le failures è la **ridondanza**

- Informazione (codici di correzione)
- Tempo (retry)
- Componenti (ridondanza per i guasti fisici)

Un gruppo di processi può lavorare in ridondanza per garantire il funzionamento di un sottoinsieme di essi in qualsiasi istante. Essi possono essere organizzati in gruppi **flat** oppure in un gruppo gerarchico (**coordinatore -lavoratori**).

1. Tenere traccia di quali sono i costituenti di ciascun gruppo
2. In distribuito bisogna contare su messaggi multicast affidabili di join/leave
3. Tenere conto dei crash e della consistenza del gruppo

Quanti processi devono essere in gruppo?

- Se i processi falliscono silenziosamente, bisogna avere  $k+1$  processi per avere un sistema  $k$ -fault tolerant
- Se i processi falliscono in modalità bizantina, servono  $2k+1$  processi per avere  $k$ -fault tolerance

Nella pratica non si è mai sicuri che non più di  $k$  processi falliranno simultaneamente.

**Macchina a stati distribuita:** un set di client manda comandi ad un server. Il server è distribuito: ogni componente riceve comandi, li esegue sui propri dati locali e deve essere prodotto un singolo risultato condiviso da tutti. Il problema del **consenso distribuito** non è semplice in caso di fallimenti.

**Consenso distribuito per errori crash:** un set di processi deve accordarsi su un dato valore. Il valore è soggetto ad una condizione di validità

1. *Agreement* : non ci sono due processi che decidono su valori differenti
2. *Validity*: se tutti i processi iniziano con il valore  $v$ , allora  $v$  è l'unica decisione possibile
3. *Termination*: tutti i processi non-faulty alla fine prendono una decisione.

### **Modello sincrono**

I processi evolvono in round sincroni. Ad ogni round un processo può mandare/ricevere messaggi e cambiare stato. I messaggi sono ricevuti nello stesso round (oppure con un offset limitato). I processi possono crashare: come risolvono?

Strutture dati:

- $v$  set di possibili valori
- $v_0$  valore di default
- $v_p$  valore iniziale del processo
- $W$  subset di  $v$  mantenuto da ciascun processo

#### **Algoritmo Floodset:**

$W \leftarrow [v_p]$

Ripetere per  $f+1$  round: broadcast di  $W$ , e aggiungere i valori ricevuti a  $W$

Decisione:

- se  $|W| = 1$ , si decide sul valore di  $W$
- se  $|W| > 1$ , si decide su  $v_0$

#### **Algoritmo Improved Floodset:**

Strutture dati uguali al caso precedente, decisione uguale alla precedente. Cambia la ripetizione

Ripetere per  $f+1$  round. Se  $W$  cambia, si manda in broadcast il nuovo valore, poi si aggiungono i valori ricevuti a  $W$

Questo funziona finchè non introduciamo gli errori bizantini (invio di messaggi arbitrari, cambiamenti arbitrari di stato).

**Consenso con fallimenti bizantini:** cambia leggermente

- *Agreement*: non avviene che due processi non-faulty decidano su valori differenti
- *Validity*: se tutti i processi non-faulty iniziano con lo stesso valore  $v$ , allora  $v$  è l'unico valore

di decisione possibile per tutti i processi non-faulty

- *Termination*: tutti i processi non-faulty prima o poi decidono

Nel caso di crash failures si prova che non c'è soluzione al problema di "Consenso Distribuito con un processo faulty in distribuito". Questo estende le considerazioni ai casi Bizantini.

### **Comunicazione di gruppo affidabile**

È un problema di importanza critica se si vuole supportare la resistenza ai guasti dei processi usando le repliche. Ottenere multicast affidabile con più collegamenti affidabili punto punto non è sufficiente.

- Che succede in caso di crash durante l'invio del messaggio?
- Che succede in caso di modifica del gruppo durante l'invio di un messaggio?

**Modello iniziale**: I gruppi sono fissati, i processi non sono faulty. Tutti i membri del gruppo ricevono le comunicazioni multicast: facile da implementare con multicast inaffidabile tramite i meccanismi di **acking** (*positivo*: gran numero di ack in circolazione – *negativo*: meno messaggi ma chi invia deve effettuare caching dei messaggi)

**In caso di processi faulty**: tutti i non-faulty devono ricevere il messaggio. Deve esserci accordo su chi è membro del gruppo

**Scalable Reliable Multicast**: soluzione iniziale, implementa un controllo feedback non gerarchico. I messaggi sono di NACK, ed è presente un coordinatore per ogni gruppo di riceventi. Flessibilità sulle politiche del coordinatore, si può richiedere ritrasmissioni al coordinatore superiore (si forza il coordinatore a rimuovere un messaggio dal suo buffer se ha ricevuto un ACK da tutti i riceventi nel suo gruppo e da tutti i coordinatori figli). **Problema**: implementare e mantenere la gerarchia. Gestire l'ingresso/uscita di processi a runtime (i messaggi "in flight" o vengono dati a tutti o non devono essere ricevuti da nessuno; inoltre bisogna preservare un ordinamento consistente).

**Modello di riferimento**: Network – OS Locale – Layer di comunicazione – Applicazione. Si identificano gli istanti in cui i messaggi arrivano dalla rete, sono ricevuti dal Layer di comunicazione e sono inoltrati all'applicazione.

1. I processi crashati sono eliminati dal gruppo e devono riconciliare il proprio stato al rientro nel gruppo
2. Messaggi provenienti da processi corretti, devono essere inoltrati a tutti i processi corretti
3. Messaggi provenienti da processi faulty sono consegnati a tutti i membri corretti oppure a nessuno
4. L'ordinamento sugli eventi è preservato solamente quando è significativo

**Group View**: è il set di processi a cui deve arrivare il messaggio (in ordine consistente rispetto agli altri multicast e rispetto a ciascun altro processo). La Group View si modifica quando un processo entra o esce (volontariamente/crash). Tutte le trasmissioni multicast devono avvenire nel periodo che intercorre tra le view changes: si garantisce che le multicast "in transito" durante una View Change saranno completate prima che la modifica diventi effettiva.

Ordinamenti per i messaggi multicast:

- multicast non ordinato
- multicast FIFO
- multicast ordinato casualmente

Si può combinare ciascuno dei precedenti introducendo una proprietà di ordinamento totale (in quel caso si dice che il multicast è anche atomico)

L'implementazione di queste specifiche porta ad ottenere **Sincronia Virtuale** (una forma di multicast affidabile). Un'implementazione di questi metodi si trova nel sistema ISIS (vedi slide 69).

### **Tecniche di Recovery**

Necessarie quando un processo riprende il lavoro dopo una failure, per riconciliare lo stato:

- **Backward recovery**: si riporta il sistema ad un'immagine salvata precedente (e corretta).
- **Forward recovery**: si conduce il sistema in un nuovo stato corretto dal quale poter resumare la computazione.

Backward recovery: si utilizzano meccanismi di **checkpointing** e **logging**. La soluzione più semplice prevede il *checkpointing indipendente* eseguito da ogni processo. Questo causa molto spesso un effetto domino con i rollback, e inoltre non è facilmente implementabile (problema di dipendenze di messaggi, temporizzazioni, latenze, relazioni di causalità).

Alternativamente, si può gestire il processo di recovery con un coordinatore: in caso di failure & recovery, il processo faulty manda una richiesta di recovery in broadcast. Tutti i processi si fermano e inviano le informazioni. Il processo faulty calcola le azioni di recovery e le ritrasmette a ciascun partecipante. Le azioni possono essere calcolate tramite due approcci: **grafo delle dipendenze di rollback**, **grafo di checkpointing**.

**Rollback-dependency graph**: ho una relazione tra il checkpoint  $C_{i,x}$  e  $C_{j,y}$  se:

- $i = j$  e  $y = x+1$   
*oppure*
- $i \neq j$  e si ha un messaggio inviato da  $I_{i,x}$  verso  $I_{j,y}$

Si marcano i nodi corrispondenti agli stati di failure, e poi si marcano tutti quelli da essi raggiungibili. Strategia simile per il **checkpoint graph**.

**Checkpointing coordinato**: la soluzione precedente è parecchio laboriosa. Introducendo un coordinatore si eliminano checkpoint "inutili".

- Coordinatore invia CHCKP-REQ
- Processi riceventi fanno il CHKPT e mettono in coda i messaggi uscenti che le applicazioni inoltrano verso la rete.
- Quando hanno finito, mandano ACK
- Quando tutti hanno finito, il coordinatore invia CHKPT-DONE

Alternativamente si può usare lo snapshot incrementale: si richiedono i checkpoint solo ai processi che dipendono dal coordinatore per la recovery, ovvero se ha ricevuto un messaggio con dipendenza causale da uno inviato dal coordinatore.

Inoltre è possibile implementare ad un **algoritmo di snapshot globale distribuito** che non blocchi i processi durante l'esecuzione (al prezzo di maggior complessità). Oppure si possono sfruttare i vantaggi di entrambi gli approcci (indipendente e coordinato) introducendo il **checkpointing indotto dalla comunicazione**: i processi prendono indipendentemente dei checkpoints ma mandano alcune informazioni tramite messaggio per permettere gli altri di capire quando devono fare pure loro il checkpoint.

A partire dal checkpoint si usano i messaggi di logging per ricostruire le azioni svolte. Bisogna fare

attenzione alle problematiche introdotte dalla distribuzione per effettuare correttamente il REDO delle azioni rispetto al caso locale!

Distinzione in

- **Messaggi stabili:** sono stati scritti in storage persistente (non possono più essere persi)
- **Messaggi instabili:** per ognuno definisco DEP(m) [insieme dei processi cui m oppure m' è stato inviato - dove m' è un messaggio dipendente da m] e COPY(m) [processi che hanno ricevuto una copia di m non ancora trasferita in storage stabile]

Detto Q un processo "sopravvissuto" a uno o più crash, si dice che Q è **orfano** se esiste un m tale per cui Q è in DEP(m) e tutti i processi in COPY(m) sono crashati. Si può evitare questa condizione mettendo tutti i processi in DEP(m) anche in COPY(m)

1. **Pessimistic Logging:** si assicura che ogni messaggio instabile m è inviato al più ad un processo. Il ricevente sarà quindi sempre in COPY(m) [a meno che rifiuti il messaggio] e non invierà altri messaggi finché non scrive m nello storage. Si assicura che non venga mai creato un orfano
2. **Optimistic Logging:** messaggi sono loggati in modalità asincrona, assumendo che saranno loggati prima di qualsiasi fault. Se tutti i processi in COPY(m) crashano, allora tutti quelli in DEP(m) saranno rollati finché non sono più DEP(m). Si creano orfani che vengono rollbackati fino a che non lo sono più

## Sicurezza

Si introducono due ulteriori proprietà del sistema

- **Confidenzialità:** informazioni sono leggibili solo da chi è autorizzato
- **Integrità:** modifica alle informazioni (dati, programmi, hardware) può essere effettuata solo nelle modalità consentite dal modello.

La necessità di regolamentare le autorizzazioni induce la creazione di una **Security Policy** (definizione di cosa è permesso e di cosa non lo è). Essa è implementata da meccanismi di sicurezza:

1. **Crittografia:** confidenzialità e integrità
2. **Autenticazione:** identificazione delle parti
3. **Controllo accessi:** access lists, role-based system
4. **Auditing:** analisi del traffico, IDS

### *Rischi di sicurezza:*

I messaggi possono essere soggetti a

1. **Intercettazione:** sniffing, dumping
2. **Interruzione:** disruption, dos
3. **Modifica:** tampering, defacing
4. **Forging:** injection, replay attack

Le problematiche di implementazione e di design si articolano su svariati aspetti:

- **Focus** del controllo. Proteggere i dati da invocazioni errate o invalide, proteggere i dati da invocazioni non autorizzate, proteggere i dati eliminando gli utenti non autorizzati ...

- **Layering**: a che livello dei protocolli inserire i meccanismi di sicurezza?
- **TCB** (Trusted Computing Base): rappresenta il set di meccanismi base dei quali "mi fido" per poter forzare una certa politica (ad esempio, suppongo che il mio kernel sia ragionevolmente "sicuro"). Un modo per ridurre TCB è separare i servizi fidati da quelli malfidati usando una RISSC (Reduced Interfaces for Secure System Components)
- **Semplicità**: in genere meccanismi semplici e chiari portano a introdurre meno errori. Spesso però meccanismi semplici non sono sufficienti.

## Crittografia

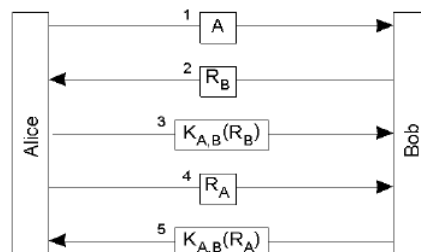
Un plaintext **P** è processato da una funzione di crittazione **F** e genera un cyphertext **C**. La funzione deve essere invertibile! Un intrusore può captare passivamente **C**, alterare messaggi in transito oppure introdurne di nuovi nel sistema.

**Crittografia Simmetrica**:  $K_{a,b}$  è la chiave condivisa da A e B. Avendo N peer, servono  $O(N^2)$  chiavi differenti.

**Crittografia asimmetrica**: si ha un paio di chiavi accoppiate  $K_{a+}$  e  $K_{a-}$ .  $K_{a-}$  può decrittare solo da  $K_{a+}$ , e  $K_{a+}$  può crittare solo da  $K_{a-}$ . Deve essere computazionalmente impossibile calcolare  $K_{a+}$  dato  $K_{a-}$  o viceversa: si può distribuire quindi una delle due chiavi senza inficiare la sicurezza dell'altra.  $K_{a-}$  rimane privata,  $K_{a+}$  diventa pubblica.

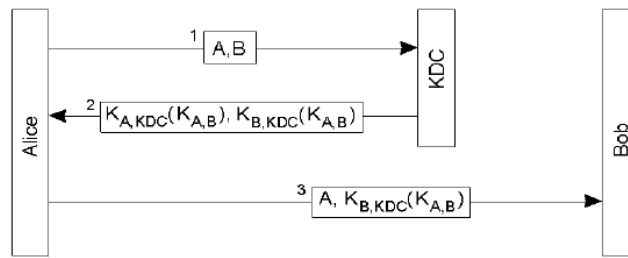
**Hashing**: dato un messaggio **m**, lo passo in una **funzione di hashing H** e ottengo un **hash h**. Possono avvenire collisioni (dati due messaggi diversi, posso ottenere lo stesso hash). La funzione di hashing è **one way** (è praticamente impossibile trovare **m** dato **h**). La funzione di hash può avere diverse proprietà a seconda dell'uso che ne faccio. **Weak collision resistance**: dato **h** e **m** tali che  $h = H(m)$ , deve essere difficile trovare due messaggi diversi che abbiano lo stesso hash. **Strong collision resistance**: data **H**, deve essere difficile trovare due messaggi diversi che generano lo stesso hash.

**Protocolli Challenge-Response**: autenticazione con chiave segreta condivisa

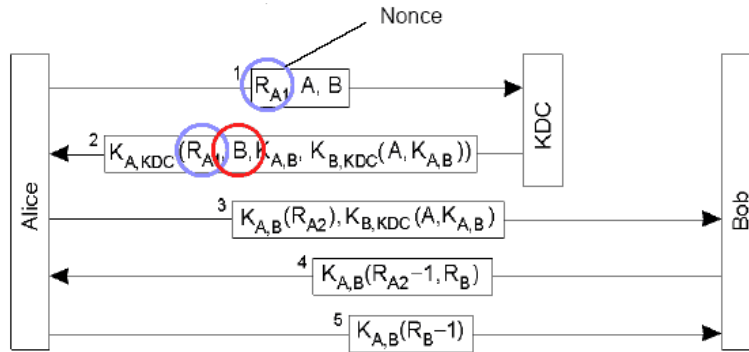


Se si tenta di accorciare lo scambio dei messaggi accorpono dati tra le varie richieste, si rende l'algoritmo non sicuro! Suscettibile a **Reflection Attack**: un attaccante può richiedere la risposta ad un challenge. **Key Wearing**: un attaccante può sollecitare l'uso della chiave (per indebolirne la segretezza). Dopo l'autenticazione, su un canale sicuro si stabilisce una chiave simmetrica di sessione per fornire integrità e confidenza dei messaggi. La session key permette di limitare il key wearing della chiave di autenticazione. Essa viene distrutta alla fine della sessione.

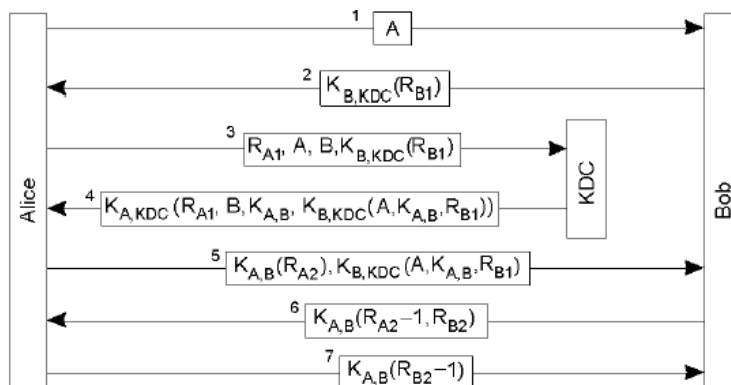
**Key Distribution Server**:



Questo protocollo non è sicuro per molti tipi di attacco. E' suscettibile su attacchi di tipo replay  
 Nella pratica si usa una versione migliorata, detta **Protocollo di Needham-Schroeder**

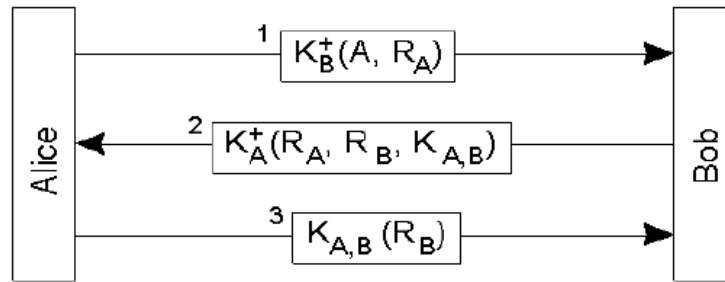


E' possibile un ultimo tipo di attacco con il replay del terzo messaggio quando la chiave  $K_{a,b}$  è compromessa



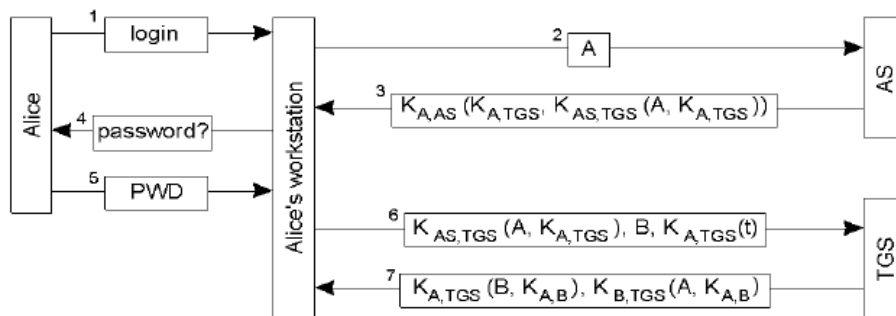
Questa versione protegge contro il riuso di una chiave di sessione generata precedentemente.

**Autenticazione con crittografia a chiave pubblica:**



## Kerberos

In Kerberos l'autenticazione utilizza il protocollo Needham-Schroeder. Il KDC è diviso in due sezioni. Authentication Server (AS): comunicazione semplificata dalla destinazione, che è sempre il TGS, e il Ticket Granting Service (TGS)



Le problematiche di Kerberos includono: password guessing, software malizioso e i timestamps (nella versione 4)

## Secure Group Communication

Nel caso interessi rendere sicura una comunicazione tra più partecipanti, sono adottabili differenti strategie:

- Crittografia simmetrica (una chiave per ogni paio di partecipanti)
- Crittografia asimmetrica: è richiesto parecchio overhead computazionale
- Crittografia simmetrica con chiave singola S

Il problema principale da gestire è rappresentato dalle operazioni di **join/leave** da un gruppo. Si richiedono **Backward & Forward Secrecy**: chi entra nel gruppo non può leggere i messaggi emessi prima della sua ammissione, e chi esce dal gruppo non deve più poter leggere i messaggi pubblicati dal gruppo.

Soluzione: si cambia la chiave del gruppo. Come crittografare la nuova chiave?

- **Join**: si critta con la vecchia chiave di gruppo e con quella di chi si connette  $O(1)$
- **Leave**: si critta con la chiave di tutti i rimanenti membri  $O(n)$

In questo caso è evidente che l'operazione di Leave è parecchio costosa.

Come scegliere la nuova chiave? Se la sceglie il server, ho il problema della **distribuzione della nuova chiave**. Se la scelgono i partecipanti, il problema diventa l'**agreement sulla chiave** tra i membri del gruppo.

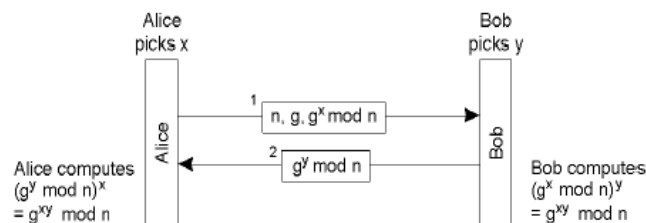
Come implementare una distribuzione efficiente della chiave?

- Utilizzando una topologia **gerarchica** di chiavi logiche: nelle foglie ho i membri con le loro chiavi. Nel nodo root ho la Data Encryption Key (DEK). Ciascun membro conosce le Key Encryption Keys fino alla root. Dopo una leave, si cambiano tutte le chiavi conosciute dal membro che esce dal gruppo: si cambia DEK e tutte le chiavi presenti sul percorso dal membro uscente al nodo. Le nuove chiavi si possono distribuire in modo efficiente sfruttando i sottoalberi stabili (che non vengono mutati).
- **Centralized Flat Table**: ho 2 KEK per ciascun bit di member ID + 1 DEK. Ciascun nodo ha una chiave per ogni bit del proprio ID. Se esce un membro, si guarda il suo ID e si cambiano tutte le chiavi associate ai bit della codifica dell'ID. Per quanto riguarda DEK, si encripta con tutte le altre chiavi. In questo modo tutti tranne chi è uscito possono decrittare la nuova chiave. Infine si crittano le nuove KEKs con le vecchie e con il nuovo DEK. E' possibile effettuare attacchi di collusione (se più nodi si mettono d'accordo).

**Secure replicated Servers**: abbiamo dei client che si connettono in modo trasparente ad un server replicato. Dobbiamo essere in grado di filtrare e risposte corrotte da un intruso. In una soluzione semplice, si usano  $2c+1$  server replicati, e ciascuno firma la propria risposta. Il client verifica la firma e decide a maggioranza. Così però il client deve conoscere l'identità e la chiave pubblica di tutti i server.

Soluzione più usata: **Schemi a soglia (n,m)**: si divide un segreto in m pezzi. n pezzi sono sufficienti a ricostruire il segreto.

Altro problema: Come distribuire le chiavi iniziali in modo sicuro? Algoritmo **Diffie-Hellman** (metodo per scambiare chiavi simmetriche su un canale insicuro)



Diffie-Hellman funziona solo contro attacchi passivi: se l'attaccante è attivo, c'è possibilità di attacco **Man in the middle**. Serve quindi autenticazione ed integrità su entrambi i messaggi Diffie-Hellman: in questo modo si vedono i due messaggi di DH come se fosse uno scambio di chiavi pubbliche. Quindi Diffie Hellman:

1. È un modo di trasformare lo scambio di chiavi segrete in uno scambio di chiavi pubbliche
2. Lo scambio di chiavi pubbliche richiede solamente integrità (autenticazione).

In ogni caso serve autenticazione per scambiare le chiavi pubbliche. L'autenticazione delle chiavi pubbliche si può effettuare tramite l'utilizzo dei **certificati**: essi sono collezioni di informazioni sull'identità, dotati di una chiave pubblica ben conosciuta e firmati da una **Certification Authority**.

Sono possibili più modelli di sicurezza:

- **Gerarchico**: root CA appartenenti ad autorità centrali (governi o enti internazionali). Si ha una gerarchia ad albero, gli utenti con CA sono le foglie dell'albero
- **PGP**: gli utenti possono autenticare altri utenti firmando la loro chiave pubblica con la propria. Gli utenti possono definire di chi si fidano per autenticare altri.

I certificati sono validi in un determinato lasso temporale, e se viene compromessa la chiave segreta si può revocare. Le CA pubblicano regolarmente le CRL (Certificate Revocation List), che vanno consultate dai client per rimanere aggiornati.

## **Access Control**

Esso è realizzato tramite un *reference monitor*, che prende le richieste da parte di un soggetto verso un oggetto e le trasforma in richieste autorizzate. Sono possibili varie implementazioni, che influenzano il modo in cui si accede al monitor:

**AC Matrix:** una matrice di utenti e oggetti con relative capability (matrice sparsa). Si può implementare un ACL per ogni oggetto (**Access Control List – ACL**), oppure ogni capability ha un'entry nella ACM (**Capabilities List – CL**).

Nel primo caso si crea una richiesta di accesso R al subject S: sul server se S appare nell'ACL e R appare in ACL(S) allora fornisco l'accesso.

Nel secondo caso si crea una richiesta di accesso R per l'oggetto O. Si passa anche la Capability. A quel punto sul server se R appare in C allora si fornisce l'accesso

Per risparmiare memoria è possibile introdurre il concetto di **gruppo** per introdurre una gerarchia nell'ACL. Il lookup rimane comunque costoso, ma si può semplificare associando ad ogni soggetto una lista dei gruppi cui appartiene.

**Role-Based AC:** diversamente da prima, ogni utente è associato ad uno o più ruoli. Quando l'utente si logga, ottiene il proprio (o uno dei propri) ruoli. Diversamente dal meccanismo dei gruppi visto in precedenza, gli ruoli possono essere cambiati dinamicamente a runtime in modo semplice.

Esempio di AC : Amoeba (vedi slides)

**Codice Mobile:** bisogna fare in modo che l'agente (data e codice) non venga alterato, che i suoi dati sensibili non vengano rubati, oppure che venga distrutto. Inoltre si vuole proteggere l'host: il codice scaricato deve avere diritti molto limitati per non fare danni. In ogni caso non è possibile fornire completa protezione per l'agente (a meno di non implementare tecniche fortemente restrittive)

**Append Log:** E'possibile capire tuttavia se l'agente è stato alterato (codice o dati): si implementa in modo che i dati possano essere solamente aggiunti, non cancellati oppure modificati. Nel log si inserisce inizialmente una checksum  $C = K+(Nonce)$ . Dopo un server S aggiunge X : allora nel log metto X e firmo (X,S). Il nuovo checksum diventa  $C\_New = K+(C\_Old,sign(X,S),S)$

**Sandbox:** a protezione dell'host si controlla l'esecuzione del codice mobile. Per il codice compilato si fanno check a runtime sulle operazioni potenzialmente problematiche. Per il codice interpretato è possibile controllare la sicurezza di ogni istruzione. *Approccio Java:* TCL (Trusted ClassLoader), Byte Code Verifier, Security Manager.

**Playground:** si usa una macchina dedicata per eseguire il codice mobile. Possibili diversi livelli di protezione: JVM Sandbox, Playground machine OS, connettività limitata da Playground al resto della rete interna.

**Signed Code:** il Security Manager può utilizzare policy differenti se il codice è autenticato

Con Java è possibile usare tecniche avanzate:

- Utilizzo delle **object references** come capabilities. In questo caso non è possibile forgiare delle references (per via della forte type safety del linguaggio). Richiede un redesign delle API (Java-E)
- Management dei **name spaces**: si possono risolvere gli imports in modo dinamico a seconda dei differenti settings di sicurezza, per ogni package. (es. Java.io.File --> security.File)

**Stack Introspection:** ciascun codebase può avere la propria politica di sicurezza. Che succede quando si hanno tante procedure sullo stack con differenti politiche? Di default, si controlla che la chiamata sia permessa in ciascun metodo sullo stack.

## Pagamenti Elettronici

E-Cash e SET (vedere le slides di sicurezza)

## Mobile Code

- Si cerca di rilocalizzare il codice vicino alle risorse (minor overhead comunicazionale)
- Customizzazione dell'accesso alle risorse remote con **Client Agents** (flessibilità)

**Code Mobility:** abilità di rilocalizzare a runtime i componenti di un'applicazione distribuita. Può comportare mobilità del codice, dello stato o di entrambi

Un servizio può essere fornito quando si ha la co-localizzazione dei seguenti oggetti:

1. **Know How**
2. **Risorse**
3. **Esecutore**

Il modello *codice mobile* prevede la presenza di **componenti** (risorse e computazionali), **interazioni** (eventi) e **siti** (supporto locale all'esecuzione e all'interazione). Alcuni paradigmi sono

1. **Client-Server** (non ho mobilità)
2. **Remote Evaluation** (Invio il Know-How sul sito remoto)
3. **Code on demand** (richiedo il Know-How al sito remoto)
4. **Mobile Agent** (in remoto ho un componente di risorse, io invio il resto)

Paradigm	Before		After	
	Site A	Site B	Site A	Site B
Client Server	A	Know-how Resources B	A	Know-how Resources B
Remote Evaluation	A Know-how	B Resources	A	Know-how Resources B
Code On Demand	A Resources	B Know-how	Know-how Resources A	B
Mobile Agent	A Know-how	Resources	-	Know-how Resources A

### Issues:

- Come inserire la mobilità in un linguaggio?
- In che modo il linguaggio è influenzato dalla mobilità?
- Come gestire lo spostamento e l'esecuzione?
- Come provvedere alla comunicazione?
- Come gestire gli aspetti di sicurezza?

Il modello di riferimento comprende un **Computational Environment**, nel quale si hanno più **execution units** (composte di **stato**, **code segments** e **data space**) e **risorse**. Bisogna chiedersi che cosa migrare e come gestire i bindings prima e dopo la migrazione.

- **Strong Mobility:** si permette migrazione di codice e stato d'esecuzione EU su un differente computational environment (**migration & remote cloning**, in modalità *proattiva o reattiva*)
- **Weak Mobility:** abilità del sistema di permettere mobilità del codice tra differenti computational environment (**code shipping/fetching**, **stand-alone/fragment code** in modalità sincrona/asincrona e immediata/differita)

Quando avviene una migrazione, si ha alterazione del data space. Bisogna cambiare i bindings delle risorse e portare alcune risorse assieme all'executing unit.

Le risorse possono essere **trasferibili** oppure **non trasferibili** (a seconda del loro tipo). Istanze di risorse trasferibili possono essere *libere* oppure *fisse*. I binding possono essere caratterizzati (in livello decrescente di restrizioni):

1. per identificatore
2. per valore
3. per tipo

Questo rende possibili diverse soluzioni

	<b>Resource Type</b>		
<b>Binding Type</b>	<b>Free Transferrable</b>	<b>Fixed Transferrable</b>	<b>Non Transferrable</b>
<b>by Identifier</b>	<i>by Move (Network Reference)</i>	<i>Network Reference</i>	<i>Network Reference</i>
<b>by Value</b>	<i>by Copy (Network Reference, by Move)</i>	<i>by Copy (Network Reference)</i>	<i>Network Reference</i>
<b>by Type</b>	<i>Re-binding (by Copy, by Move, Network Reference)</i>	<i>Re-binding (by Copy, Network Reference)</i>	<i>Re-binding (Network Reference)</i>

**Come gestire la sicurezza?** Le Execution unit appartengono a diversi utenti. Come stabilire relazioni valide? Come gestire computational environments appartenenti a differenti organizzazioni? Come gestire la comunicazione insicura?

Il codice mobile deve essere

- **Portable:** piattaforme d'esecuzione eterogenee
- **Sicuro:** prevenire danneggiamenti accidentali e maliziosi all'environment

Si adottano soluzioni ibride con codice compilato in un linguaggio intermedio di basso livello che viene quindi interpretato. La traduzione può essere effettuata alla sorgente oppure alla destinazione; compilazione JIT.

## Paradigmi di Design e Tecnologie

		Design Paradigms		
		CS	REV	MA
Technologies	Non Mobile	Appropriate	Code represented as data Code receipt and execution must be programmed explicitly	Code and state represented as data Execution and state restoring must be programmed explicitly
	Weakly Mobile	Degenerated code Unnecessary EUs are created	Appropriate	State represented as data State restoring must be programmed explicitly
	Strongly Mobile	Degenerated code Unnecessary EUs are created Unnecessary state migration	Unnecessary overhead for migration Unnecessary state migration	Appropriate

L'utilizzo del codice mobile permette customizzazione del servizio, autonomia, fault-tolerance migliorata, flessibilità di trattamento dati e incapsulamento dei protocolli.

### Case Studies

**Network Management:** permette autonomia (processing distribuito e minimizzazione del traffico), flessibilità (si usano più agenti di management solo dove effettivamente servono), comprensione semantica (locale e globale). Vedere slides per analisi dettagliata

### Peer-to-Peer

E' un paradigma architetturale: tutti i nodi sono potenziali utenti e provider del servizio. Ogni nodo è indipendente dagli altri, non c'è amministrazione centralizzata. I nodi entrano ed escono dalla rete in modo fortemente dinamico, e le loro capacità sono fortemente variabili. Il sistema è internet-wide: le risorse sono geograficamente distribuite e non c'è una vista globale del sistema.

**Problema principale:** trovare le risorse nel modo più efficiente possibile.

**Primitive comuni:** *Join, Publish, Search, Fetch*

Il lookup e la search possono ricercare un certo *pattern*, oppure si cerca un *item specifico*

Il fetching può interessare i dati attuali oppure una reference su dove trovare i dati cercati.

Principali implementazioni

- **Centralized Database:** Napster , DC, Emule, WinMX
- **Query Flooding:** Gnutella
- **Intelligent Query Flooding:** Kazaa
- **Swarming:** BitTorrent
- **Unstructured Overlay Routing:** Freenet
- **Structured Overlay Routing:** Distributed Hash Tables

Per i dettagli vedere le slides.

### Wireless Sensor Networks

Il focus è sul distributed embedded computing. In futuro ci saranno molti più nodi computazionali che utenti loro associati. I nodi saranno più poveri di risorse rispetto a quelli attuali ma lavoreranno in cooperazione

## **MICA2 Motes**

- 2 Batterie AA
- Processore a 8Mhz
- 5Mbit di memoria flash (128KB codice, 512KB di dati)
- 4KB di RAM
- Comunicazione radio: range 300m, varie frequenze, 38.4 Kbit/s di banda
- Realizzazione modulare: è possibile montare diverse board di sensoristica (magnetici, luce, temperatura, acustici, umidità, pressione, GPS...)

**Wireless Sensor Net:** abbiamo set di nodi (sensori) che si organizzano in **Patch Networks**. Ogni PN ha uno o più gateway che la collegano alla **Transit Network** (una rete di comunicazione con le altre PN). La Transit Network è poi collegata a una **basestation** che si interconnette a internet (o a nodi di rete classici per permettere il collegamento con data services e clients).

Una variante delle WSN sono le **WSAN** (Wireless Sensor and Actor Network): i nodi sono sensibili all'ambiente ma lo possono anche modificare (provvisi di attuatori). Le architetture sono più decentralizzate. Altre varianti: sensori subacquei (ultrasuoni), BAN (Body Area Network, sensori disposti sul corpo umano)

## **Features**

- **Applicazioni:** non interattive; data & event-centric. Non è importante l'identità di un nodo, interessa l'informazione complessiva.
- **Network:** diversamente dalle reti tradizionali, è application-specific.
- **Deployment:** spesso avviene in ambienti non controllati, inaccessibili o ostili. Necessita di auto-organizzare la topologia di rete, bisogna gestire i malfunzionamenti e il recovery, spesso è impossibile rimpiazzare la sorgente energetica
- **Energy saving:** bisogna salvare energia per preservare il tempo di vita dell'intera rete. I sensori rimangono in *sleep* e hanno duty-cycles molto ridotti. Uso di data-aggregation per ridurre il traffico di rete.
- **Ambienti chiusi:** in generale la posizione dei sensori una volta stabilita rimane fissata (non in tutte le applicazioni). La topologia invece è sempre dinamica (per via del duty cycle ridotto: bisogna garantire percorsi di rete verso il gateway sincronizzando i cicli di sleep!)

## **Sistemi operativi:**

- Molto ridotti come consumo di memoria
- Operazioni intrinsecamente reattive
- Altamente modulari
- I layer software non sono marcati (spesso sono integrati come libreria in un programma)

**TinyOS:** Approccio component-based. Le interfacce dei componenti specificano i comandi possibili e gli eventi che possono essere lanciati. **Split-phase:** i comandi vengono lanciati, le risposte e il valore di ritorno sono prodotti in modo asincrono tramite un'evento. Le unità di concorrenza sono i **task** (interrompibili). Componenti definiti e programmati con nesC.

**Matè:** è una virtual machine (approccio che permette descrizioni compatte, ridotti overhead di comunicazione, facilità di fare security & safety checks, update/relocation del codice).

## Approcci di programmazione middleware:

1. **Programmare la sensor network:** non si guarda al nodo singolo, ma agli effetti aggregati sull'intera rete. Adattamento di astrazioni ben conosciute (es. Repository). Le problematiche di distribuzione (es. routing) non sono esplicite al programmatore
2. **Programmare il sensor node:** si rappresenta il comportamento di un nodo singolo. La distribuzione è evidente al programmatore, che è fornito di astrazioni di alto livello per risolverla.

**TinyDB:** si considera la sensor network come un database che viene interrogato con una versione modificata di SQL. I valori delle "tuple" sono creati dai nodi che li mantengono per un breve periodo di tempo, solo quando servono. Si usa un overlay network ad albero.

**Direct Diffusion:** ogni nodo identifica con uno o più attributi i dati che genera. Gli altri nodi esprimono **interessi** basandosi sugli attributi. Gli interessi stabiliscono **gradienti** su cui vengono diretti i dati diffusi (tipicamente *reverse paths*). Questi possono essere rinforzati basandosi sulla qualità dei dati ricevuti. Simile al content-based Publish Subscribe

**Ottimizzazione utilizzo dei sensori:** si massimizza la vita dell'applicazione pur rientrando nel QoS richiesto. Si organizzano diversi schemi di copertura da utilizzare nei vari cicli dell'applicazione. Si può incrementare parecchio la vita dello schema.

**Regioni Astratte:** si permette al programmatore di descrivere facilmente l'interazione tra set di nodi dotati di alcune proprietà d'insieme. Si offre un'interfaccia di accesso comune a uno "spazio di tuple" di una regione.