1.Lo scenario delle reti MANET

La crescente proliferazione di dispositivi portabili che possono beneficiare di connettività di tipo wireless e la sempre maggiore diffusione di connettività senza fili negli ambienti in cui gli utenti vivono e lavorano aprono nuovi scenari nei quali gli utenti richiedono la possibilità di beneficiare di servizi internet avanzati ovunque si trovino, in ogni momento e possibilmente anche in movimento.
Scenari di E-Care in cui squadre di soccorso, tramite l'utilizzo di PDA, riescono a coordinarsi e tenere sotto controllo le rispettive posizioni in situazioni di emergenza costituiscono un esempio di come le nuove tecnologie possano essere sfruttate al servizio della collettività.
Le nuove tecnologie abilitano inoltre applicazioni automotive, dove punti di accesso alla rete disseminati in autostrada permettono un maggior controllo sul traffico e un conseguente miglioramento delle strategie di smaltimento dello stesso. Le possibilità di comunicazione offerte dai dispositivi wireless montati sulle autovetture permettono inoltre alle stesse lo scambio di informazioni quali velocità di crociera media e misura della distanza di sicurezza; in caso di incidente stradale le vetture sopraggiungenti potrebbero essere notificate in modo tempestivo dell'evento.

1.1 Eterogeneità dei dispositivi

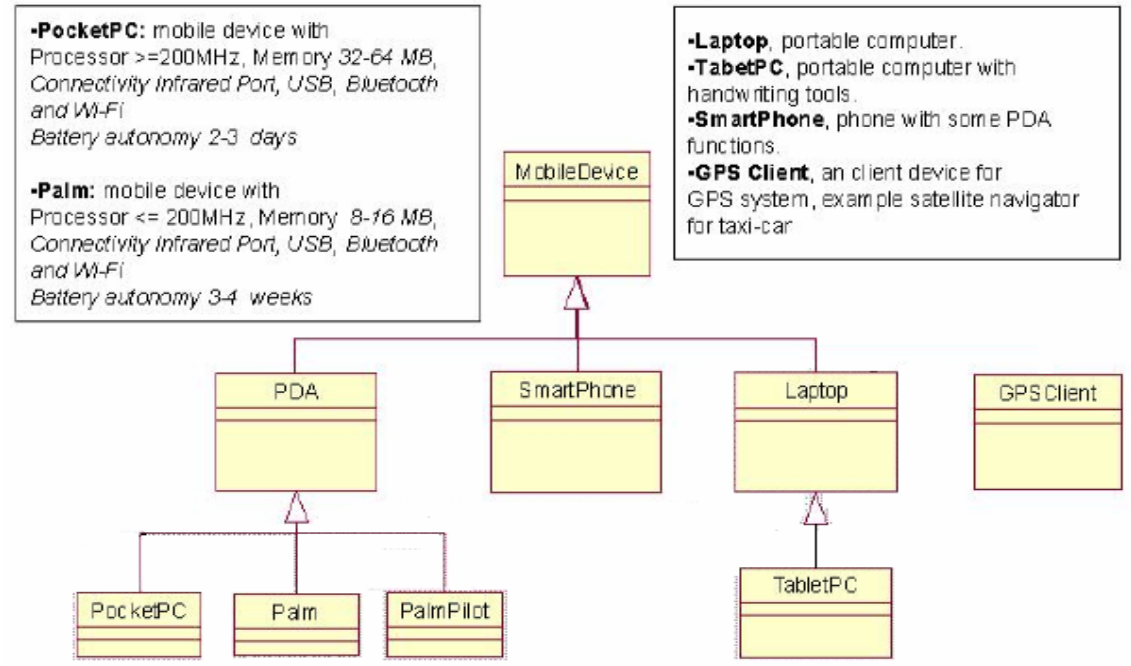


Fig. 1: Tassonomia dei dispositivi mobili attualmente disponibili

Nel nuovo scenario gli utenti richiedono di poter collaborare sfruttando dispositivi eterogenei sia per natura che per prestazioni quali laptop, PDA e telefoni cellulari.
I palmari o Personal Digital Assistant (PDA) sono nati come sostituti delle agende elettroniche; due sono le principali famiglie di PDA disponibili sul mercato: elementi di eterogeneità sono determinati dal sistema operativo supportato, Windows CE, piuttosto che Linux Familiar o PalmOS. I cellulari che integrano le funzioni di un palmare sono definiti SmartPhone e rappresentano la categoria più evoluta oggi disponibile. I Laptop si differenziano dai PC desktop per le ridotte dimensioni, soprattutto del display e dell'hardware. Fino a pochi anni fa i computer da scrivania mantenevano la superiorità tecnologica rispetto ai portatili; oggi questo divario si è notevolmente assottigliato (Fig. 1). I diversi sistemi mostrano eterogeneità sia per processori impiegati che per memoria disponibile, file system, sistema operativo, supporto di rete. La potenza computazionale fornita da CPU performanti e dalla disponibilità di memoria consente a questi dispositivi di avere una gamma di possibilità più o meno vasta a livello applicativo; la maggiore presenza di risorse consente di memorizzare grandi quantità di dati ed eseguire applicazioni pesanti e onerose. Anche la disponibilità di montare un sistema operativo dipende da queste caratteristiche; avere a disposizione uno strato software di gestione del dispositivo significa avere un maggior controllo su ogni componente hardware disponibile e disporre di maggiori possibilità di personalizzazione degli stessi (ad esempio installando software personalizzato o aggiornando quello esistente); in generale la presenza di un sistema operativo come Linux Familiar, Embedix o Windows CE offre la possibilità di rendere un dispositivo programmabile a seconda delle esigenze. Dall'altro lato, possedere minori risorse computazionali può significare aumentare i tempi di funzionamento dei dispositivi (cioè la durata delle batterie) e in generale aumentare la portabilità degli stessi in termini di peso e ingombro. La tecnologia per sistemi mobili sta cercando di offrire al tempo stesso potenza, dinamicità e portabilità ma per trovare il giusto equilibrio è necessario comunque scendere a compromessi nella scelta dell'hardware e del software da montare.
In sostanza, esiste una grande diversità ed eterogeneità dei dispositivi di accesso alla rete sia a livello hardware (eterogeneità a livello di display, tastiera, memoria, processore impiegato, presenza o meno di un File System) che a livello software (Sistemi XP, Linux, Mac OS per Laptop, Windows CE, Linux Familiar, Palm OS per palmari, Symbian per Smartphone) (Fig. 2); inoltre, acquista sempre più rilevanza la presenza sul dispositivo di un supporto di sviluppo per applicazioni: esempi sono forniti dalle versioni "mobile" di Java (JavaSE, J2ME/CDC, J2ME/CLDC) e della piattaforma .NET(.NET compact framework).

Per quanto riguarda le possibilità di connessione con altri dispositivi mobili o con altre reti, esistono soluzioni che utilizzano Wi Fi, GPRS, GSM, Bluetooh e altre tecnologie.

Fig. 2: Eterogeneità dei dispositivi mobili

1.2 Eterogeneità della rete

Le reti wireless non utilizzano connessioni cablate ma la trasmissione dei segnali avviene tipicamente via radiofrequenza o generalmente nello spettro delle microonde o dell'infrarosso.
Uno dei principali vantaggi di una rete wireless è consentire ad un utente di lavorare in qualsiasi posizione ed anche mentre si muove; una tassonomia largamente accettata [1] in ambito di telecomunicazioni è infatti quella di classificare le diverse tipologie di rete a seconda delle esigenze da affrontare e del raggio di comunicazione richiesto dalle stesse: in particolare, si possono distinguere quattro categorie principali (Fig. 4):

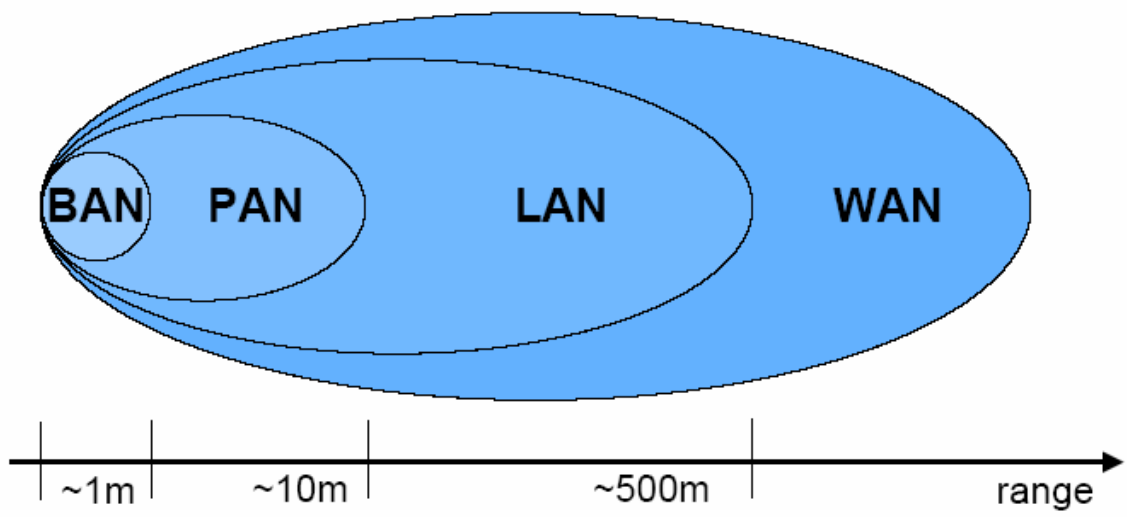


Fig. 3: Tassonomia delle reti in base al range di comunicazione

Body Area Network (BAN): Una BAN serve per connettere dispositivi indossabili; i componenti di un computer indossabile sono distribuiti sul corpo umano (auricolari, computer da polso) e una BAN fornisce il supporto di rete e di auto-configurazione (l'inserimento e la rimozione di un dispositivo da una BAN dovrebbero essere trasparenti per l'utente); servizi di audio e video streaming in tempo reale devono coesistere con trasferimenti di dati non-realtime quali ad esempio il traffico internet.
L'interconnessione con altre reti BAN o PAN costituisce infine una soluzione di interfacciamento col mondo esterno. Il raggio di comunicazione di una BAN corrisponde al raggio di comunicazione di un essere umano, cioè uno o due metri. Le attuali implementazioni non arrivano a bande più alte di un centinaio di Kbps; ci sono

studi e brevetti che utilizzano come mezzo di interconnessione fra i dispositivi anche la pelle [2].

Personal Area Network (PAN): Una PAN nasce dall'esigenza di connettere i dispositivi portabili dell'utente ad altri dispositivi sia fissi che mobili; mentre le BAN sono rivolte all'interconnessione di dispositivi indossabili una PAN è una rete che si colloca nell'ambiente attorno alla persona. Una PAN ha un raggio di comunicazione dell'ordine di un centinaio di metri che abilita ad esempio l'interconnessione di reti PAN di più persone vicine e in generale di una PAN con l'ambiente di rete circostante. Bluetooth è un tipico esempio di tecnologia applicata alle reti PAN; la banda disponibile è di un centinaio di Kbps.

Wireless Local Area Network (WLAN): una rete che consenta la comunicazione di dispositivi in un raggio di qualche centinaio di metri prende il nome di Local Area Network; queste reti sono rivolte a soluzioni di interconnessione in ambienti quali case o uffici e permettono la comunicazione di molti dispositivi eterogenei. In sostanza, sono la trasposizione wireless delle più comuni reti LAN attraverso tecniche di comunicazione senza filo quali quelle fornite dalla famiglia IEEE 802.11. Consentono una banda dell'ordine delle decine di Mbps e lavorano alla frequenza di 2,4 Ghz.

Wide Area Network (WAN): reti di questo genere nascono per permettere la comunicazione fra dispositivi situati a notevoli distanze, nell'ordine delle centinaia e migliaia di km(sono anche chiamate reti geografiche). Esempi di tali reti utilizzano tecnologie UMTS e GPRS. La banda disponibile risulta essere alta: UMTS offre un bit rate reale di circa 256 Kbps a frequenze di 1900/2000 Mhz, mentre la rete GPRS è stata ideata appositamente per poter offrire una banda nominale teorica dalle decine alle centinaia di Kbps, lavorando a frequenze tra i 900 e 1800 Mhz.

Di seguito analizziamo due importanti tecniche di comunicazione wireless che hanno contribuito al successo di tale tecnologia: Bluetooth e l'802.11.

1.2.1 Bluetooth

La tecnologia Bluetooth [1] nasce a metà degli anni novanta con l'obiettivo di semplificare e migliorare le interconnessioni tra apparecchi mobili utilizzando una piccola potenza dell'ordine dei milliwatt. Il dominio applicativo di tale tecnologia risiede nella comunicazione e nell'interazione di apparecchi di natura diversa come telefoni cellulari, PDA, notebook, stampanti e pc tramite onde radio emesse da questi dispositivi.
Bluetooth rappresenta l'evoluzione del protocollo Infrared Data Association (IrDA), nato per la trasmissione dei dati senza fili con banda superiore a 1Mb/s e distanze comprese tra 10 e 100 metri. Bluetooth fa uso di un collegamento radio: caratteristiche salienti di questa tecnologia e linee guida per lo sviluppo delle opportune applicazioni sono sicuramente il basso consumo di energia (nell'ordine del centinaio di mW), il costo modesto (qualche decina di euro per avere connettività Bluetooth) e le dimensioni ridotte (Il fulcro di un dispositivo Bluetooth è costituito da un chip radio delle dimensioni di circa 9x9 mm istallato su un apparecchio elettronico tradizionale, ad esempio un telefono cellulare).
Lo standard di riferimento per la tecnologia Bluetooth è costituito dalle specifiche Special Interest Group (SIG) e stabilisce le linee guida per lo sviluppo di tutti i dispositivi che intendono operare con la tecnologia in questione. Per stabilire connessioni punto punto e multipunto per la trasmissione di voce e di dati, Bluetooth utilizza segnali a radiofrequenza (RF) nella banda Industrial Scientific and Medical Band (ISM).
Ogni dispositivo Bluetooth individua automaticamente un altro dispositivo dello stesso tipo fino ad una distanza di un centinaio di metri, imposta automaticamente una connessione con lui e crea una piccola rete definita piconet, Nell'ambito della piconet, un dispositivo (generalmente colui che ha attivato la connessione) detiene il ruolo di master, finalizzato a verificare il corretto svolgimento della comunicazione tra i diversi dispositivi, mentre fino a sette dispositivi operano da slave (Fig. 4). Ogni dispositivo possiede un indirizzo MAC predefinito assegnatogli dalla casa costruttrice, ricerca automaticamente dispositivi in un raggio di copertura predefinito, ed è in grado di creare la sua piconet privata. Quest'ultima operazione avviene per ogni apparecchio quando, all'accensione, è inviato un segnale richiedente alcune informazioni agli altri

componenti che rientrano nel raggio predefinito. Il dispositivo che risponde a tali richieste è aggiunto automaticamente alla piconet.

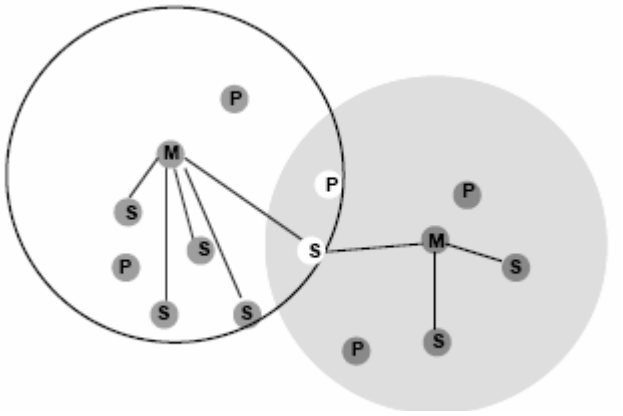


Fig. 4: Struttura di una Scatternet, unione di più piconet: M rappresenta i dispositivi Master, S gli Slave, P rappresenta le entità scollegate dalla rete

Ogni dispositivo scambia dati con i dispositivi all'interno della propria piconet; tutte le operazioni che si verificano all'interno della rete, avvengono senza alcun invio accidentale di dati errati al dispositivo o alla rete. Ci sono poi dispositivi che possono essere configurati per funzionare anche in più di una piconet: in questo modo, possono fungere anche da ponte tra le singole reti. Quando si creano delle piconet che interagiscono tra loro si parla di scatternet. Ogni scatternet può contenere fino a 10 piconet (che equivale ad 80 dispositivi Bluetooth): oltre a questo numero la rete si satura poiché Bluetooth utilizza 79 frequenze. In ogni scatternet le comunicazioni che avvengono tra le diverse piconet sono filtrate dai dispositivi master di ogni piconet.

1.2.2 802.11

La famiglia 802.11 [1], che gestisce l'uniformità delle comunicazioni nelle Wireless LAN (WLAN), è un insieme di standard spesso incompatibili tra loro.
Nel 1997 nasce il primo standard di riferimento che detta le specifiche a livello fisico e datalink per l'implementazione di una rete LAN wireless. Tale standard consente un data rate di 1 o 2 Mbps usando la tecnologia basata su onde radio nella banda 2.4 Ghz o su raggi infrarossi. La limitata velocità ne determina uno scarsa diffusione.

L'evoluzione di tale tecnologia diversi anni dopo porta alla sua evoluzione, l'IEEE 802.11b, denominato anche Wireless Fidelity (Wi-Fi), consentendo una trasmissione dai 5.5 agli 11 Mbit/s oltre a mantenere la compatibilità con lo standard precedente.
Il protocollo 802.11b consente di poter variare la velocità di trasmissione dati per adattarsi al canale; fornisce la possibilità di scelta automatica della banda di trasmissione meno occupata e dell'access point in funzione della potenza del segnale e del traffico di rete. Inoltre permette di creare un numero arbitrario di celle parzialmente sovrapposte permettendo il roaming in modo del tutto trasparente e basse potenze di emissione (nell'ordine di alcune decine di milliWatt). La copertura varia nell'ordine delle centinaia di metri.
Lo standard 802.11b in particolare definisce un insieme di specifiche per il livello fisico e per il livello Medium Access Control (MAC) (Fig. 5).

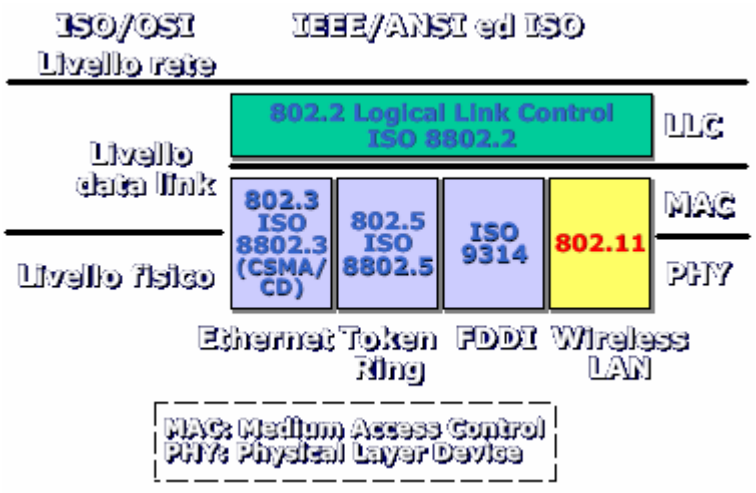


Fig. 5: Architettura IEEE 802.11

Wi-Fi utilizza una tecnologia radio chiamata Spread Spectrum (SS) secondo cui la banda spettrale disponibile viene suddivisa in più canali. La trasmissione avviene utilizzando la tecnica detta Frequency Hopping (FH) per cui la portante effettua dei salti (Hop) di frequenza utilizzando i sottocanali. L'ordine con cui i salti vengono effettuati segue una sequenza che per questo standard è determinato con la tecnica Direct Sequenze (DS). Il cambio di frequenza cioè utilizza una successione lineare dei sottocanali disponibili.
La tecnologia 802.11b inoltre supporta un meccanismo per la sicurezza chiamato WEP (*Wired Equivalent Privacy*) che consiste in un sistema crittografico dei dati affiancato

ad un modello di autenticazione dei nodi connessi. Attualmente la chiave segreta utilizzata è di 40 bit, ma data l'estrema facilità con cui il sistema WEP è violabile, si prevede nell'immediato futuro l'adozione di una chiave di 80 bit;

1.3 Modelli di Networking per Reti Wireless

Esistono vari modelli di rete wireless: a Cella, a Cella Virtuale e Ad-Hoc [].

1.3.1 Modello a Cella

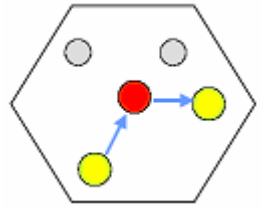


Fig. 6: Comunicazione Intracellulare. Rosso: BS, Giallo: MH

Nel modello a cella individuiamo due tipologie di entità: Base Station (BS) e Mobile Host (MH). Ogni cella consiste di una BS fissa che fornisce copertura per una determinata area e l'insieme delle coperture delle varie BS costituisce l'area totale della rete. Si assume che ogni BS sia connessa alle altre attraverso una infrastruttura cablata. I dispositivi mobili MH hanno la possibilità di muoversi da una cella all'altra in totale libertà e per comunicare fra loro utilizzano la BS della cella che li contiene (comunicazione Base to Remote). Per fare ciò il MH ha bisogno di stabilire un collegamento wireless con la BS di appartenenza. Se il partner della comunicazione è anch'esso nella stessa cella, la BS stessa invia il messaggio al destinatario attraverso un altro canale senza fili (Fig. 6); se il partner è presente in un'altra cella, la BS spedisce il messaggio attraverso l'infrastruttura cablata alla BS che contiene il destinatario, che riceverà il messaggio attraverso un canale wireless che lo collega alla BS di appartenenza (Fig. 7). Questo modello permette una buona dinamicità della rete: i nodi infatti rimangono liberi di spostarsi tra le celle, mantenendo la connessione attiva nel caso vi sia una buona copertura fra le BS (Roaming).

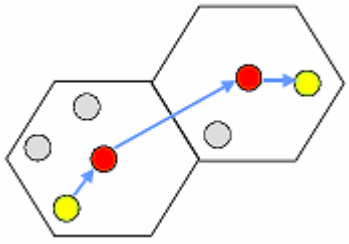


Fig. 7: Comunicazione Intercellulare. Rosso: BS, Giallo: MH

1.3.2 Modello a Cella Virtuale

Il modello a Cella può essere rilassato ammettendo la possibilità di connettere fra di loro le BS tramite connessioni wireless.
In esso le BS sono anch'esse mobili (Fig. 8). Le connessioni tra le BS sono anch'esse wireless. Le BS continuano comunque a funzionare da punti di accesso per l'attività di comunicazione dei MH ma a differenza delle reti a Cella, la topologia delle rete a Cella Virtuale (VC) cambia nel tempo. Si assume comunque che le BS siano in connessione tramite un percorso di routing composto solo da BS.

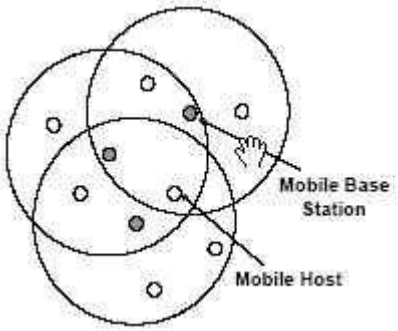


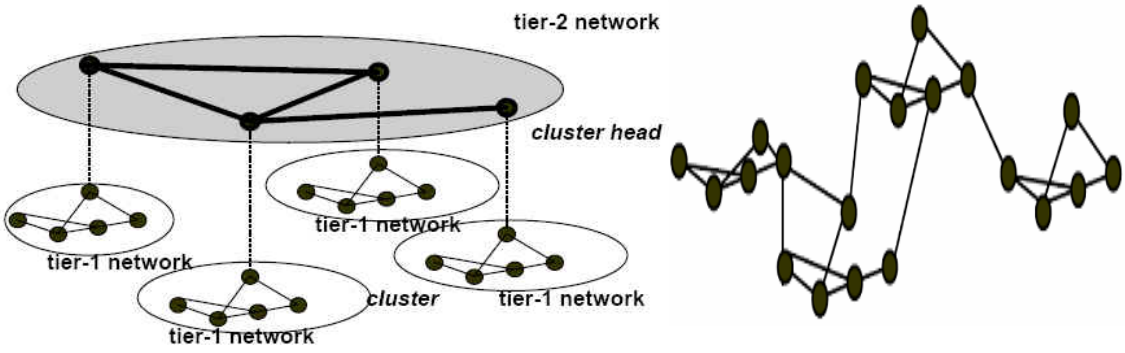
Fig. 8: Modello a Cella Virtuale

E' possibile rilassare ulteriormente il modello VC eliminando la distinzione fra BS e i MH.

1.3.3 Modello Ah-Hoc

In un modello di rete Ad-Hoc tutte le comunicazioni avvengono attraverso canali wireless; tutte le entità del sistema collaborano al fine di instradare i pacchetti nel modo corretto. A causa della mobilità impredicibile dei nodi, la topologia di rete può cambiare costantemente: infatti, la particolarità dei modelli Ad-Hoc è che non hanno alcuna necessità di utilizzare un'infrastruttura fissa. Questo le differenzia totalmente dai modelli distribuiti tradizionali: le reti Ad-Hoc vengono costruite all'occorrenza ed utilizzate in ambienti estremamente dinamici, non necessariamente con l'aiuto di una infrastruttura già esistente.
Si può effettuare una tassonomia delle reti Ad-Hoc in base alla loro topologia, che può essere *gerarchica* (hierarchical) o *piatta* (flat) (Fig. 9).

In una rete *gerarchica* i nodi sono partizionati in gruppi detti cluster. Per ogni cluster è selezionato un *cluster head* attraverso i quali passa il traffico della rete.
In una rete ad hoc *piatta* non è previsto nessun elemento di centralizzazione. Due nodi hanno la possibilità di entrare in comunicazione se la potenza del segnale è tale da permettere al nodo destinazione di "sentire" la trasmissione del vicino (ovvero se due nodi sono in copertura radio). Un vantaggio della rete *piatta* è la possibilità di stabilire più di un percorso tra nodo sorgente e destinazione; questo permette di valutare in modi diversi quale collegamento è da preferire, a seconda delle richieste e dell' utilizzo della rete.
Il vantaggio della rete *gerarchica* è invece quello di minimizzare il numero delle informazioni di routing che vengono scambiate tra i nodi di uno stesso cluster e tra i cluster head.



(1) **(2)**

Fig. 9: Topologia di una rete Ad-Hoc gerarchica (1) e di una rete Ad-Hoc piatta (2)

In tutti i modelli descritti, c'è la possibilità che un nodo mobile possa muoversi fuori dal raggio di copertura degli altri nodi, diventando irraggiungibile e disconnettendosi dalla rete. Anche quando la rete è operativa e tutti i nodi sono raggiungibili, i sistemi mobili pongono comunque interessanti problematiche di gestione in quanto la topologia di rete può cambiare in qualsiasi momento favorendo fusioni, partizioni e merge di insiemi di nodi.
Nell'ambito dell' IETF una rete ad hoc è definita come un sistema autonomo di router mobili connessi mediante link wireless; i nodi formano una rete di comunicazione modellata a grafo arbitrario.

2. Protocolli di Routing

L'alto livello di dinamicità degli ambienti wireless rende possibile la mobilità in modo impredicibile e ad istanti di tempo arbitrario cambiano frequentemente i punti di accesso alla rete.
Se ciò accade i cambiamenti risultanti devono essere resi noti, cosicché le informazioni riguardanti i cambiamenti topologici possano essere sempre aggiornate.
Uno dei principali problemi nella gestione di una rete Ad-Hoc risiede infatti nella scelta delle politiche di routing da utilizzare per adattarsi alla dinamicità dell'ambiente e notificare i frequenti e impredicibili mutamenti della topologia della rete. In generale l'ammontare del traffico di segnalazione necessario ad un algoritmo di instradamento distribuito è molto elevato; la maggior parte degli studi svolti sugli algoritmi di instradamento per reti Ad-Hoc mirano quindi a trovare il modo di diminuire il traffico di segnalazione prodotto dal livello di routing.
I protocolli di routing non hanno ottenuto ancora una standardizzazione. In tempi recenti sono emersi nuovi lavori di ricerca sul routing in scenari MANET [4] [5]: una diffusa classificazione li divide in tre categorie principali: protocolli *proattivi, reattivi e ibridi* (Fig. 10).

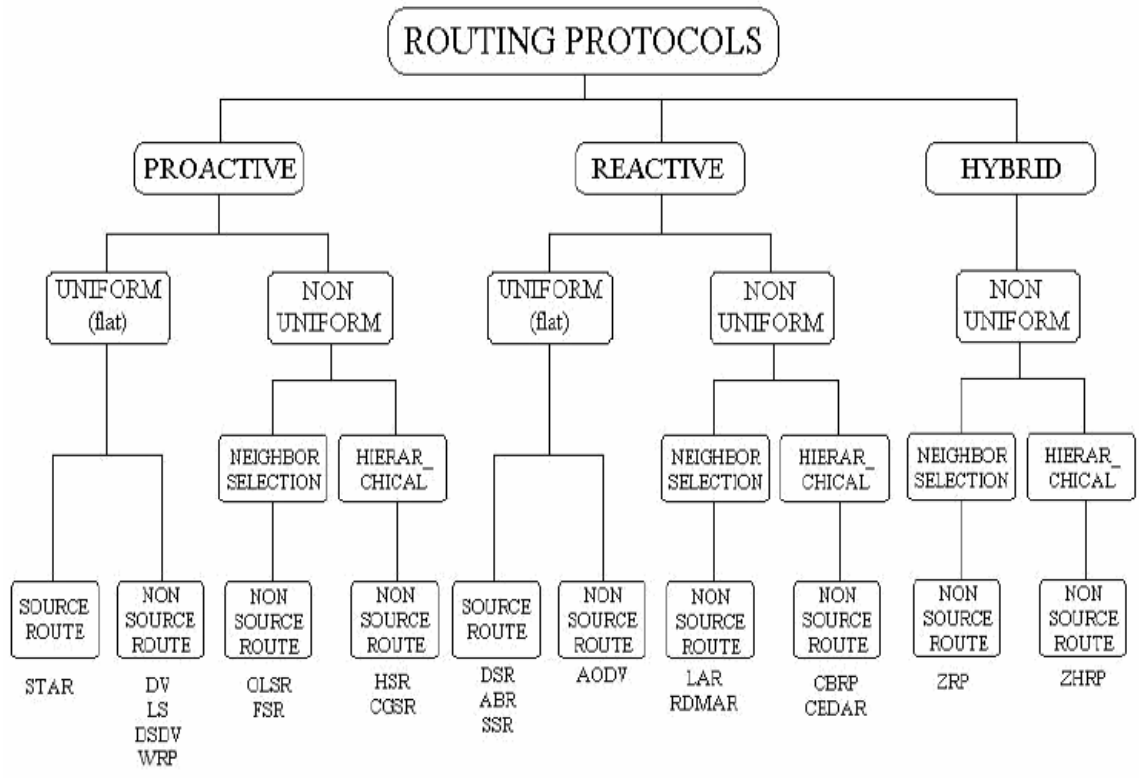


Fig. 10: Tassonomia dei protocolli di routing

2.1 Protocolli Proattivi

I protocolli di tipo *proattivo* mantengono costantemente aggiornate le informazioni di instradamento tramite scambi di informazioni a intervalli temporali fissi. Questo permette di avere l'instradamento immediatamente disponibile ad ogni richiesta di routing. Lo svantaggio è che gli algoritmi proattivi inducono traffico anche quando non viene trasmesso nessun pacchetto dati fra le diverse entità del sistema.

2.1.1 DSR

Il protocollo Dinamic Source Routing (DSR) fa parte della famiglia dei protocolli proattivi, per cui i nodi che intendono conoscere un determinato percorso devono farne esplicitamente richiesta. Ogni dispositivo possiede una route cache nella quale vengono memorizzati una serie di percorsi atti ad arrivare al nodo di destinazione. In questa tabella i percorsi sono mantenuti per un determinato periodo di tempo (scelta che viene gestita dal programmatore): quando questo tempo scade, il percorso, è cancellato se non viene referenziato. La prima fase che un nodo sorgente utilizza nella determinazione di un percorso (se questo non è già memorizzato in route cache) è la scoperta dei vicini. In questa fase si riesce a determinare quali sono i dispositivi che si trovano nel range trasmissivo del nodo che ha richiesto un route. Una volta scoperti i vicini, inizia la fase di Route Discovery (Fig. 11).

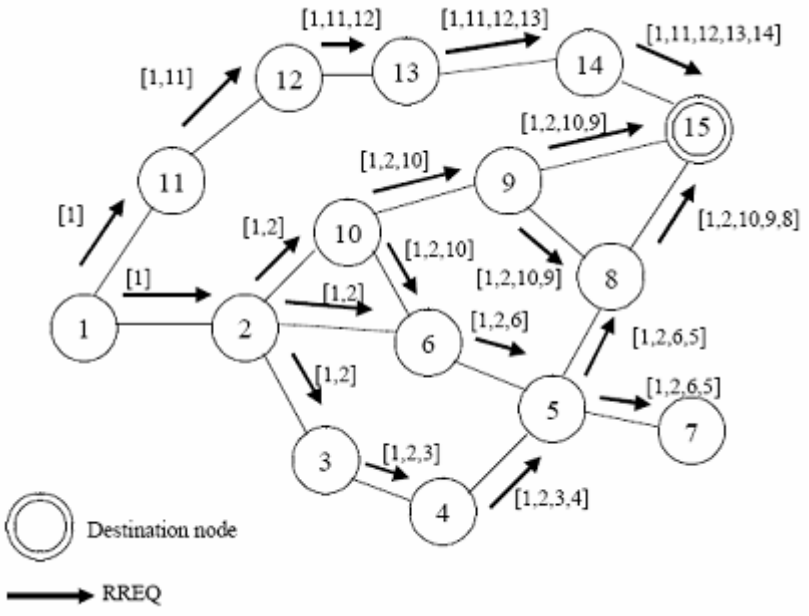


Fig. 11: Esempio di propagazione di una richiesta di routing nel protocollo DSR

In generale un nodo inoltra una richiesta, RR, quando o nella sua route cache non ha un percorso per giungere a destinazione o la scoperta dei vicini non ha dato come esito il nodo destinazione. Questa fase permette di scoprire dinamicamente un percorso verso un altro host, sia se l'host di destinazione è ricercabile direttamente dentro il suo stesso wireless trasmission range, sia se è necessario utilizzare nodi intermedi per ottenere il percorso.
Il pacchetto RR identifica sia il mittente e sia qual è il destinatario della richiesta. Se la RR riesce ad ottenere un percorso, allora viene mandata indietro al mittente una Route Reply, nella quale è memorizzato il percorso completo sorgente-destinazione.

Il pacchetto RR è composto da:

- indirizzo del mittente,
- indirizzo della destinazione,
- request ID che indica il numero generato dal mittente, che identifica univocamente una RR,
- route record che memorizza in un record la sequenza dei salti necessari per raggiungere l'host di destinazione.

Tali dati sono ricavati attraverso l'operazione di Route Discovery.
Per scoprire ed evitare che ci siano RR duplicate nella ricerca di un percorso, ogni host mantiene una lista delle coppie (indirizzo sender, request, id) ricevute dalle varie RR. Quando un device riceve una richiesta si può comportare nei seguenti modi:

- se la coppia indirizzo sender, request ID è nota nella lista, allora l'host scarta la richiesta senza processarla;

- se l'indirizzo dell'host è già presente nel route record di una richiesta, allora scarta la richiesta senza processarla;

- se la richiesta giunge alla destinazione, allora il route record contiene il percorso dal quale la richiesta è giunta.

Infine, la destinazione copia il percorso memorizzato nel route record e lo pone in una RR che invierà al mittente;

- se la richiesta giunge ad un host intermedio, esso aggiunge al route record il suo indirizzo e inoltra la richiesta.

Affinché sia fornita una RR al mittente, la destinazione deve possedere un percorso per la sorgente; se la destinazione ha nella propria route cache un percorso per il mittente, invia la RR su questo percorso. Diversamente, se non si possiede un percorso, può essere utilizzato quello memorizzato nel route record della RR ricevuta. Questo approccio richiede che la comunicazione tra coppie di host sia bidirezionale.
Un ulteriore approccio offerto dal protocollo DSR per mandare un pacchetto di risposta al nodo sorgente, può essere quello di utilizzare la modalità piggyback. Questa tecnica permette di inviare all'interno del pacchetto Route Reply anche quello RR.

2.2 Protocolli Reattivi

Nei protocolli di tipo *reattivo* viene determinato on demand il percorso di routing solo nel momento in cui il pacchetto deve essere trasmesso. Sebbene questo approccio minimizzi il traffico necessario al routing, la comunicazione mostra un aumento dei tempi di consegna.

2.2.1 AODV

Ad esempio il protocollo Ad hoc On-demand Distance Vector (AODV) è un protocollo di routing di tipo reattivo.
Una caratteristica fondamentale di questo protocollo è quella di utilizzare numeri di sequenza, i quali forniscono ai nodi di una rete uno strumento per valutare quanto sia aggiornato un determinato percorso. Un terminale che si trovi a dover scegliere tra più percorsi verso una certa destinazione sceglierà quello caratterizzato dal numero di sequenza maggiore, corrispondente ad un informazione di routing più recente. Inoltre il protocollo supporta l'instradamento multicast, ovvero consente la creazione di gruppi di utenti nella rete, i cui membri possono comunque cambiare in qualunque momento.

Ogni nodo mantiene le informazioni di instradamento per una destinazione all'interno di una routing table. Se un nodo si trova nella condizione di dover trasmettere verso una destinazione per cui non ha informazioni di routing, esso provvede ad inviare un pacchetto broadcast denominato Route Request (RR); quando un nodo riceve un RR ne sfrutta il contenuto per effettuare un *refresh* delle informazioni nella propria routing table: il pacchetto infatti contiene dei campi che forniscono informazioni su come raggiungere il mittente della richiesta, sul numero di hop necessari, e sull'indirizzo del Next Hop (l'ultimo nodo ad aver inviato il pacchetto) (Fig. 12).

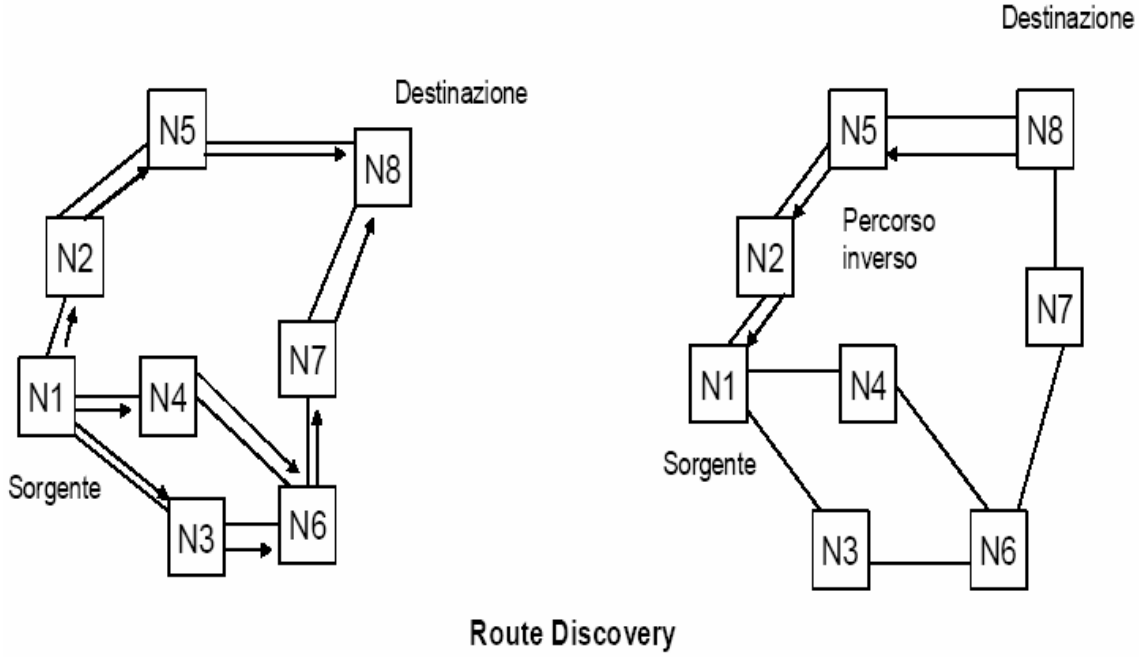


Fig. 12: Route Discovery nel protocollo AODV

Se nella tabella del ricevente non esiste una entry per il mittente del RR, il nodo ne crea una e crea anche un *Reverse Route*, ovvero un pacchetto di ritorno diretto in senso opposto rispetto a quello in cui viaggiano i pacchetti RR. La sua funzione principale è quella di fornire un percorso ai pacchetti di risposta Route Reply di ritorno verso la sorgente, ma potrà essere utilizzato anche per inviare eventuali pacchetti di dati.
Viceversa, se nella tabella esiste già una entry, allora il nodo valuta se sia il caso o meno di fare un aggiornamento: se il numero di sequenza associato al percorso è inferiore al Source Sequence number che compare nel pacchetto RR ciò significa che il percorso in tabella ormai non è più valido.

Nel caso in cui il nodo non possieda alcuna informazione sulla destinazione provvede a ripetere a sua volta il RR, non prima di aver incrementato di una unità il campo Hop Count del pacchetto (per tenere conto del nuovo salto), e aver modificato il campo Source nell'header del pacchetto. Quest'ultima operazione è necessaria per consentire al nodo successivo di sapere quale nodo ha inviato per ultimo il pacchetto. Se, d'altro canto, il nodo ha già informazioni di routing verifica se il valore del campo Destination Sequence Number associato al percorso posseduto è inferiore al valore che compare all'interno del RR. In tale caso significa che l'informazione in possesso del nodo è ormai obsoleta, ed anche in tale caso il pacchetto viene rinviato. In caso contrario l'informazione è sufficientemente aggiornata, e si può quindi procedere all' invio di una risposta verso il nodo Source.
La risposta ad una richiesta di informazioni avviene generando un pacchetto dati unicast chiamato *Route Reply*, ed inviandolo verso il nodo *Source*.
Viene quindi sfruttato proprio il Reverse Route che ciascun nodo ha provveduto a creare al passaggio del RR. Ancora una volta, al passaggio del Route Reply, ogni nodo può sfruttarne le informazioni per aggiornare la propria tabella di routing.
Nel caso in cui un nodo verifichi la rottura di un link, esso provvede a generare un pacchetto unicast chiamato Triggered Reply che informa del problema tutti i precursori. Ogni nodo che riceve tale pacchetto aggiorna quindi la propria tabella di routing, marcando il nodo come inutilizzabile, e provvede a sua volta a ripetere il pacchetto ai propri precursori. L' informazione riguarda quindi solo l'insieme di nodi che stavano sfruttando quel link. A questo punto quindi è necessaria una nuova fase di route discovery allo scopo di trovare un percorso alternativo per la medesima destinazione.

2.3 Protocolli Ibridi

Il terzo tipo di protocolli, *ibridi*, cerca, come dice il nome, di unire i vantaggi di entrambi i protocolli precedenti, restringendo l'applicazione di algoritmi proattivi ai soli vicini del nodo che vuole trasmettere il pacchetto. In sostanza, effettua una ricerca proattiva fra i nodi vicini e reattiva per i nodi lontani.

2.3.1 ZPR

Zone Routine Protocol (ZPR) fa parte di questa categoria: esso usa una routing zone, (letteralmente zona di routing), cioè una zona radio in termini di radio hops (Fig. 13).

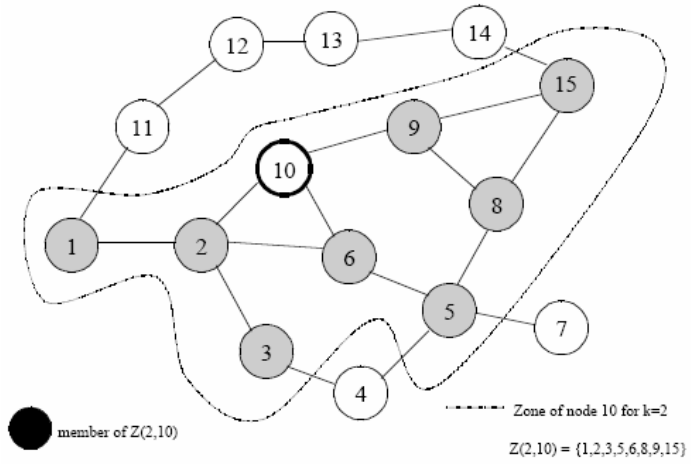


Fig. 13: Esempio di zona di routing in ZRP. k rappresenta il numero di radio hop di riferimento

Le dimensioni di una zona di routing possono andare ad incidere sulle prestazioni della comunicazione della rete ad hoc.
ZPR utilizza nella zona di routing uno dei protocolli di instradamento proattivi. Ogni dispositivo deve avere a disposizione delle tabelle all'interno delle quali vengono memorizzati i percorsi per raggiungere i nodi all'interno della zona.
Gli aggiornamenti dei percorsi (in caso di cambiamenti topologici della rete o in caso di link failure), vengono effettuati dai singoli nodi all'interno della zona di routing; per realizzare invece la ricerca dei nodi (operazione di Route Discovery), che si trovano in zone diverse, viene utilizzato uno dei protocolli reattivi. Il protocollo ZPR si avvale quindi dell'aiuto di tre sotto-protocolli, che sono:

- Un protocollo proattivo denominato InterAzone Routing Protocol (IARP), il cui compito principale è assicurare che ogni nodo dentro una routing zone abbia a disposizione una tabella di routing consistente e che venga aggiornata periodicamente per esprimere informazioni attendibili, atte a determinare la ricerca di tutti i nodi all'interno della zona.

- Un protocollo reattivo denominato IntEr Routing Protocol (IERP), il cui compito è effettuare la ricerca sia di nodi in altre zone, sia di un percorso per giungere a destinazione: la ricerca di un percorso On-Demand viene effettuata dai nodi posizionati sui bordi di una zona (border node), che si preoccupano di reperire le informazione riguardanti i nodi risiedenti in altre zone. Ciò non avviene tramite broadcast, ma tramite l'invio di messaggi dai nodi marginali ai nodi posizionati all'interno di altre zone.
- Bordercast Resolution Protocol (BRP): il cui compito è permettere ad un nodo di conoscere i nodi periferici dei propri nodi periferici; il nodo può così decidere quali dei propri nodi periferici scartare o indicare a questi quali dei loro nodi periferici non utilizzare.

3. Fornire servizi collaborativi in scenari Ad-Hoc

In ambienti Ad-Hoc l'utente ha una maggiore libertà di movimento; l'aumentare di mobilità crea situazioni in cui informazioni ambientali quali la locazione dell'utente e degli oggetti intorno ad esso sono più dinamiche. Un altro obiettivo dei sistemi collaborativi in questi ambienti è quindi quello dare accesso ad informazioni e servizi in qualsiasi situazione di tempo e di spazio.

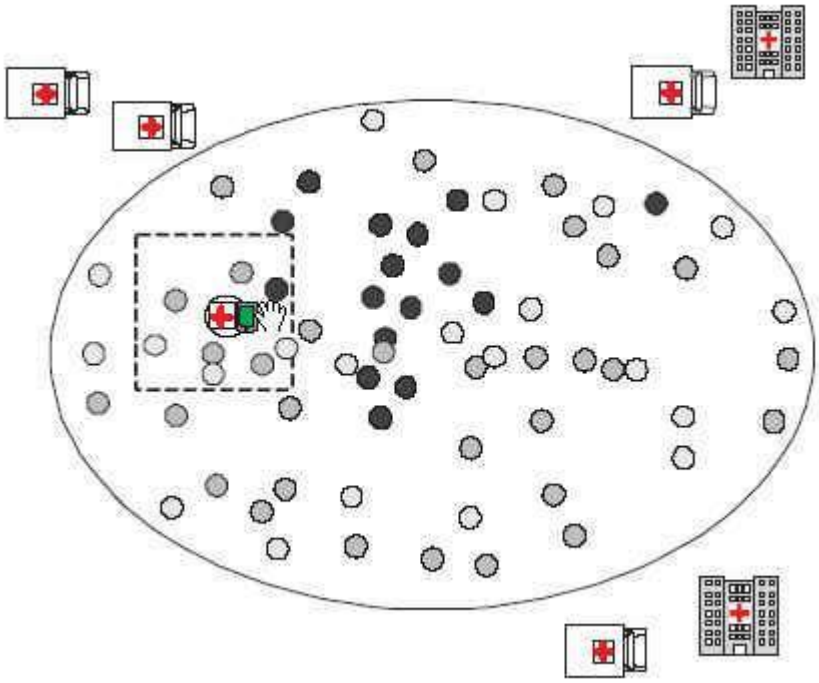


Fig. 14: Fornire soluzioni avanzate nel settore E-Care: attraverso un dispositivo elettronico si possono individuare i feriti più gravi (grigio più scuro) e procedere al soccorso tempestivo.

In questo senso si prospettano scenari di soccorso in cui l'interazione fra soccorritore e ferito passa attraverso dispositivi mobili e diventa "collaborativa"; se ad esempio si dotasse la popolazione di dispositivi in grado di misurare le condizioni di salute della persona che li indossa, in caso di disastri ambientali o incidenti di varia natura si favorirebbe una riduzione dei tempi di intervento delle squadre di soccorso. Il ferito che si trova nel raggio di comunicazione del PDA del soccorritore nel giro di pochi istanti può fornire informazioni sulle proprie condizioni fisiche oltre alla propria locazione, favorendo un soccorso tempestivo (Fig. 14).

Uno dei problemi principali nella realizzazione di sistemi distribuiti per ambienti Ad-Hoc è legato alla comunicazione: la scelta della strategia da usare per comunicare va fatta considerando aspetti quali la necessità o meno di centralizzare alcuni servizi di ausilio alla comunicazione, l'esigenza di prestazioni del sistema, i tipi di interazioni fra le diverse parti del sistema, il tipo di tecnologie a disposizione (quindi i costi) e la strategia di comunicazione che si vuole realizzare.

La complessità introdotta da tali sistemi può essere resa più o meno trasparente da componenti, detti middleware, che forniscono un più alto livello di astrazione rispetto a quello fornito da un sistema operativo tradizionale.

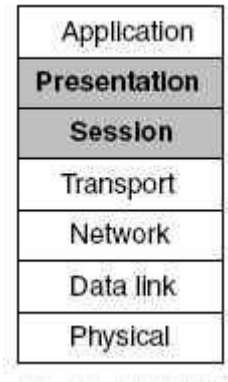


Fig. 15: Livello del modello OSI in cui risiede un middleware.

Esistono differenti definizioni di middleware [6] [7], tutte però hanno in comune la visione del middleware come elemento fondamentale di mascheramento della complessità delle applicazioni distribuite.

Secondo Linthicum, "il middleware è uno strato software che risiede tra il programma applicativo e il livello rete (Fig. 15) delle piattaforme e dei protocolli eterogenei. Disaccoppia le applicazioni da qualsiasi dipendenza riguardante sistemi operativi, piattaforme hardware e protocolli di comunicazione eterogenei".

3.1 Inadeguatezza del supporto fornito dai sistemi tradizionali

La grande diffusione di sistemi distribuiti sviluppati secondo il modello Client/Server ha fornito meccanismi in grado di invocare metodi su oggetti remoti come se questi fossero locali. Questi sistemi si sono presto evoluti implementando piattaforme complete per lo sviluppo di applicazioni ad oggetti distribuiti. Middleware sviluppati in diversi contesti applicativi forniscono funzionalità specifiche, come ad esempio servizi avanzati per la spedizione di messaggi, o servizi avanzati per le transazioni, ad esempio utilizzati nelle basi di dati distribuite. Molte di queste tecnologie sono state progettate per le reti

cablate, ma non sono utilizzabili per reti Ad-Hoc perché le primitive di interazione, come le chiamate di procedure remote, richieste di oggetti, invocazioni di metodi remoti o transazioni distribuite sono basate su paradigmi che assumono un canale di comunicazione tra i componenti sempre disponibile, affidabile e veloce, con una larghezza di banda superiore alle centinaia di Kbps e una conoscenza pressoché totale e statica dell'ambiente di lavoro. Anche il protocollo TCP, basandosi sulle stesse assunzioni, non può essere utilizzato in contesti dinamici come quelli Ad-Hoc. Inoltre in queste reti è sconsigliabile avere data server centralizzati che conoscono ad esempio l'allocazione dei file e delle loro copie, e comunque avere una rigida separazione fra server e client. In caso di disconnessione di uno o piu' server, infatti, verrebbe meno la possibilita' di accedere ai file distribuiti.
I paradigmi di interazione in cui si assumeva che le entità interagenti:

- conoscessero la localizzazione reciproca,
- fossero sempre attive tranne in condizioni di malfunzionamento, e conseguentemente che
- potessero interagire con un semplice protocollo di tipo richiesta/risposta (request/response)

devono essere adattati per essere funzionali ai nuovi ambienti di mobile computing oppure lasciare spazio a nuove soluzioni.

3.2 Middleware per sistemi p2p tradizionali

Nasce così la necessità di orientarsi verso soluzioni totalmente decentralizzate: l'espansione delle reti Ad-Hoc ed i recenti sviluppi dei dispositivi mobili hanno favorito la progettazione di ambienti mobili come ambienti peer to peer [8], dove un sistema o rete consiste di dispositivi che interagiscono in concomitanza di brevi incontri nel mondo reale ingaggiando uno scambio wireless di informazioni.
Con il peer to peer, l'ambiente di elaborazione evolve da un sistema Client/Server ad un nuovo livello di infrastruttura software distribuita creando una rete punto-multipunto in cui tutte le entità sono pari e si forniscono vicendevolmente servizi (Fig. 16).

Nel modello a comunicazione peer to peer, ogni entità comunica con tutte le altre in maniera esplicita e diretta. Sistemi come JXTA o .NET My Services sono esempi di modelli che supportano lo sviluppo di servizi peer to peer.

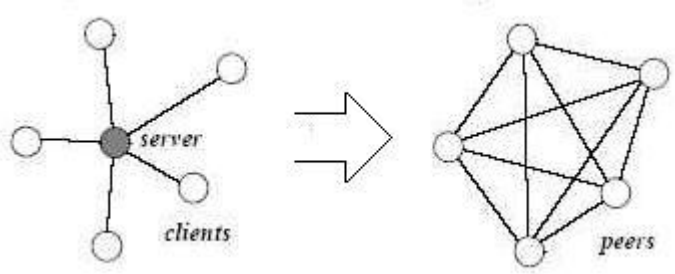


Fig. 16: Evoluzione di sistemi di dispositivi mobili in ambienti peer to peer.

Un sistema peer to peer comprende alcune delle caratteristiche di una rete Ad-Hoc. Il peer to peer tradizionale però si basa sull'assunzione di poter accedere a tutti i nodi di una rete; inoltre uno dei motivi che rende i tradizionali sistemi così potenti sta nel fatto che l'elaborazione si basa su risorse (computazionali, di memorizzazione, umane) disponibili su computer fissi: quando si costruiscono applicazioni distribuite su tali infrastrutture, è utile sfruttare tutte le risorse disponibili (processori veloci, grandi quantità di memoria, etc) in modo da garantire la migliore qualità di servizio all'applicazione.

Infine, come abbiamo visto, il supporto fornito dal protocollo TCP necessita di connessioni di rete stabili e statiche, in contrasto con la natura dinamica delle reti MANET.

I sistemi p2p tradizionali infatti sono spesso permanentemente connessi alla rete attraverso link a banda larga: la posizione di un dispositivo cambia raramente, i punti di accesso alla rete sono statici, la topologia del sistema è conservata nel tempo. Questo significa che il mittente e il destinatario di una richiesta (cioè il componente che richiede un servizio e quello che fornisce il servizio) sono generalmente connessi nello stesso momento: le connessioni permanenti permettono forme accoppiate di comunicazione, sia in tempo che in spazio e le situazioni in cui richiedente e fornitore non sono presenti simultaneamente sono considerate un'eccezione dovuta a fallimenti temporanei del sistema (per esempio di sconnessioni dovute a sovraccarichi della rete).

Nei sistemi Ad-Hoc, invece, l'impredicibilità e la dinamicità della topologia di rete unite alla limitata banda di trasferimento costituiscono parte integrante dell'ambiente di funzionamento; in generale, un ambiente di dispositivi mobili costituisce un sistema più complesso rispetto ad un ambiente di computer fissi. A causa di altre limitazioni quali ad esempio la disponibilità energetica, i dispositivi coinvolti tendono inoltre ad avere CPU meno performanti, meno memoria, minori capacità di memorizzazione, display più piccoli e limitati dispositivi di input rispetto ai più comuni dispositivi desktop;.
Le applicazioni per sistemi peer to peer classici risultano quindi fragili e poco flessibili se trasportate in ambienti Ad-Hoc; anche tenendo conto degli sviluppi recenti di questi modelli (ad esempio, le conversazioni tra agenti intelligenti ed i modelli client-server avanzati [BelPR01, BelCS01]) un tale metodo non può sostenere generalmente l'apertura ed il dinamismo degli scenari identificati.
Un ulteriore elemento di complessità deriva dalla necessità di adattare la fornitura di servizi alle caratteristiche eterogenee dei dispositivi mobili: laptop, PDA e telefoni cellulari sono profondamente diversi sia per natura che per prestazioni ma devono comunque aver la possibilità di usufruire alla stessa offerta di servizi.

3.3 Nuovi requisiti per lo sviluppo di applicazioni collaborative in contesti Mobile Ad-Hoc NETwork

Un approccio a soluzioni peer to peer potrebbe essere adatto allo sviluppo di applicazioni collaborative in contesti Ad-Hoc; esistono delle somiglianze fra i due modelli ma la maggiore dinamicità delle reti Ad-Hoc richiede lo studio di nuove soluzioni.
In particolare, le Mobile Ad-hoc NETwork (MANET) aprono ulteriori possibilità e abilitano la collaborazione fra utenti anche in assenza di una infrastruttura fissa e pre-pianificata: la dinamicità con cui tali reti tendono a formarsi le rende estremamente utili in condizioni in cui l'infrastruttura è assente o inutilizzabile (per esempio in ambito militare o in operazioni di soccorso o di emergenza in zone disastrate).

La caratteristica comune di tutte le reti MANET è l'alto livello di dinamicità delle entità che la compongono; lo scopo fondamentale di una sistema collaborativo in ambiente MANET è quindi quello di fornire un supporto a tale dinamicità attraverso l'analisi dei requisiti derivanti.

Una MANET è un ambiente in cui i punti di accesso non sono statici; le entità cambiano spesso il punto di attacco alla rete o si muovono durante la connessione, di conseguenza partizioni e merge sono eventi frequenti che possono causare cambiamenti topologici anche significativi (Fig. 17).
Inoltre essendo queste reti inserite anche in contesti urbani, in presenza di muri o

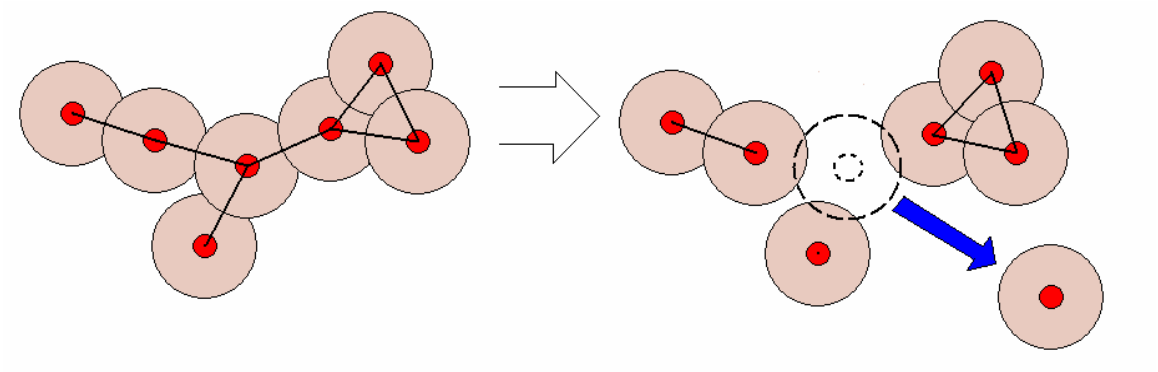


Fig. 17: Lo spostamento di una entità può causare modifiche nella topologia di rete

ostacoli di varia natura i link di comunicazione possono essere soggetti a disconnessioni inaspettate; conseguentemente, i dispositivi mobili devono prevedere ed eventualmente anticipare queste problematiche attuando meccanismi di risoluzione opportuni.
La comunicazione tra entità arbitrarie in una rete MANET può richiedere un processo di routing basato su lunghi percorsi wireless: per quanto riguarda l'instradamento di pacchetti di dati le difficoltà sorgono in quanto, senza l'ausilio di una infrastruttura fissa, i percorsi consistono di collegamenti in cui sia i partner che i nodi intermediari di una comunicazione sono portati a muoversi indipendentemente l'uno dall'altro: la mobilità dei nodi causa frequenti cambiamenti della topologia di rete che portano ad una minore consistenza delle informazioni di routing necessarie ad un pacchetto per raggiungere la destinazione (Fig. 18).

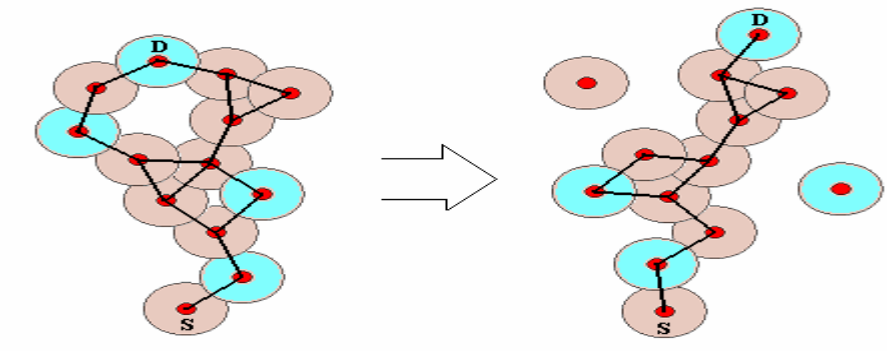


Fig. 18: Se alcuni nodi si spostano il percorso di routing può divenire inconsistente.

I sistemi collaborativi applicati a scenari MANET consentono di beneficiare di risorse fornite da entità che risiedono su dispositivi eterogenei nelle immediate vicinanze; a causa della mobilità impredicibile di questi dispositivi, anche il processo di discovery delle risorse diventa un problema.
L'impossibilità di sfruttare una conoscenza a priori della rete, dell'identità, dei nomi e delle caratteristiche dei nodi rende necessaria l'adozione di meccanismi di discovery dinamici: si necessita di protocolli attraverso i quali un dispositivo possa rilevare la presenza di altri dispositivi nelle vicinanze, condividere informazioni di configurazione e di servizi offerti da ciascuna entità e notificare quando un dispositivo risulti disponibile o meno. Il processo di ricerca delle risorse deve essere in qualche modo continuo, per rilevare le modifiche topologiche significative della rete, ed efficiente, in modo da non sovraccaricare il traffico di rete. A tale proposito, un primo esempio di gestione delle risorse potrebbe essere quello di valutare su quali basi poggiare i protocolli di discovery: si potrebbe pensare di trattare ogni entità del sistema in maniera anonima; oppure si potrebbero considerare le risorse condivise dai vicini prossimi di ciascuna entità, restringendo il raggio di azione dei protocolli di discovery.
Poiché i dispositivi mobili possono migrare da un luogo all'altro, è necessario suddividere il dominio di mobilità in diversi sottospazi. Ciascun sottospazio creato dinamicamente, rappresenta un insieme di interazione popolato da dispositivi mobili; questi ultimi non sono vincolati nei loro movimenti e possono quindi formare altri insiemi oppure rimanere isolati. I sottospazi (gruppi) non sono però necessariamente isolati fra loro: in generale possono interagire scambiandosi informazioni estendendo o restringendo l'insieme di interazione.

Riassumendo, l'alta dinamicità degli ambienti MANET si riflette pesantemente sulle scelte progettuali dei supporti middleware per applicazioni collaborative; è necessario adattarsi all'ambiente di lavoro per far fronte a nuovi requisiti non funzionali quali la dinamicità dei punti di accesso alla rete e delle risorse, le difficoltà di comunicazione in presenza di infrastrutture civili o di percorsi wireless multi-hop, l'eterogeneità delle entità coinvolte e l'elasticità nel raggruppamento delle stesse.

3.4 Approcci ai problemi di sviluppo di applicazioni in scenari MANET

Negli ultimi anni la ricerca ha identificato diverse linee guida per lo sviluppo di middleware a supporto di applicazioni collaborative: in sistemi altamente dinamici e decentralizzati infatti, per fornire un corretto supporto di comunicazione risulta di fondamentale importanza l'approccio indirizzato al reperimento di informazioni. Le sorgenti di informazioni e servizi per sistemi MANET sono infatti non persistenti e si adattano ai cambiamenti dell'ambiente di lavoro e delle risorse distribuite; in generale si modificano assieme al contesto delle entità che formano il sistema di riferimento.

3.4.1 Il contesto

Nella pubblicazione che per prima introdusse il termine context awareness (consapevolezza del contesto), Schilit e Theimer definiscono il contesto come l'insieme di informazioni sulla locazione e sull'identità di persone e oggetti nelle immediate vicinanze dell'utente.
Analogamente, Brown definisce il contesto come la locazione e l'identità delle persone attorno all'utente, ma include proprietà quali l'ora del giorno, la temperatura, la stagione, etc;
Dey descrive il contesto come sintesi di svariati fattori quali lo stato emotivo dell'utente, il suo livello di attenzione, la sua locazione e il suo orientamento, le persone e l'ambiente che lo circondano.
Questi ricercatori definiscono il contesto come un ambiente di esecuzione in continuo mutamento, classificandolo in:

- Ambiente computazionale: processori disponibili, dispositivi accessibili, capacità della rete, connettività, costo computazionale;
- Ambiente utente: insieme dei vicini, situazione sociale;
- Ambiente fisico: locazione, livello di illuminazione, temperatura e disturbi.

in sostanza, il contesto rappresenta l'insieme delle informazioni rilevanti un'applicazione e il suo gruppo di utenti.

Intendiamo per contesto:
“qualsiasi informazione che può essere usata per caratterizzare la situazione di una entità. Un’entità è definita come persona, luogo o oggetto considerato rilevante nell’interazione utente-applicazione, inclusi l’utente e l’applicazione stessi”[9].

La natura dinamica del contesto fa sì che questo vari nel tempo in modo impredicibile e ad istanti di tempo arbitrari; lo sviluppatore può usare le continue informazioni di contesto per adattare il comportamento dell’applicazione alle variazioni dell’ambiente di lavoro.
Esistono certe informazioni di contesto che, nella pratica, si considerano più importanti di altre: queste sono la locazione, l’identità dell’utente, l’attività in corso e il tempo.
Tali informazioni si comportano da sorgenti per la determinazione di altre caratteristiche contestuali. Per esempio, dall’identità di una persona si possono acquisire informazioni come numero di telefono, indirizzo, data di nascita, relazioni con altre persone dell’ambiente, ecc.
Attraverso la locazione di un’entità si possono individuare altri oggetti o persone nelle vicinanze o in generale l’attività che si sta consumando attorno all’entità stessa.
Esistono inoltre diverse rappresentazioni della locazione.

3.4.2 La locazione

La locazione è importante perché consente di erogare servizi sulla base della posizione dell’utente.
E’ stata studiata una tassonomia dei sistemi di localizzazione per aiutare gli sviluppatori di applicazioni location-aware ad effettuare la scelta delle tecnologie di cui avvalersi. Tale tassonomia è stata realizzata considerando vari aspetti dei principali sistemi di localizzazione: il tipo di informazione richiesta, il tipo di rappresentazione dello spazio, il tipo di computazione (locale o remota), il livello di precisione, la scalabilità del sistema da realizzare, la necessità di riconoscere l’identità delle entità nello spazio, i costi e le limitazioni. In generale un sistema location-aware può trattare diversi tipi di informazione.
Una prima distinzione va fatta fra locazione fisica e locazione simbolica. La locazione fisica è data rispetto a un sistema di riferimento che fornisce coordinate geografiche come ad esempio il Global Positioning System (GPS), mentre quella simbolica si basa

su idee astratte della locazione di un'entità nello spazio: "nella stanza di fianco", "vicino alla porta", ecc…
Data una locazione fisica abbastanza accurata, è possibile in generale derivarne una simbolica (si pensi al caso di un database che associa insiemi di locazioni fisiche a locazioni simboliche). La scelta relativa al tipo di localizzazione da usare in un'applicazione riguarda, oltre ad aspetti non funzionali quali i costi, anche (e soprattutto) l'uso che l'applicazione fa di queste informazioni.
Ragionare sulla locazione in uno scenario MANET significa fornire una valida astrazione che consente la collaborazione di una entità con i suoi vicini.
Inoltre combinare informazioni di località fisica e simbolica dei dispositivi coinvolti può essere utile per trarre altre informazioni contestuali utili; ad esempio se due o più dispositivi sono "vicini" fra loro, potrebbero far parte della stessa rete oppure no; il concetto di località costituisce uno degli elementi principali per la creazione di una rete Ad-Hoc in quanto in presenza di collegamenti wireless, la topologia di una rete ha come requisito fondamentale la raggiungibilità dei dispositivi coinvolti. Una volta appurato ciò, le politiche di appartenenza ad un sistema o gruppo di entità possono essere decise analizzando le altre informazioni di contesto primarie e le informazioni secondarie derivanti.

3.4.3 Il gruppo

In ambienti MANET vi è la necessità di un supporto che assicuri una corretta gestione delle entità presenti nella rete dal punto di vista dell'accesso alle risorse condivise: le entità che fanno parte di una stessa rete Ad-Hoc infatti potrebbero avere la necessità di partizionarsi in insiemi diversi o di condividere dinamicamente solo determinate categorie di informazioni.
Mentre i meccanismi di discovery per sistemi desktop possono basarsi su approcci centralizzati (come in Napster) per il reperimento delle risorse, un dispositivo mobile inserito in un ambiente MANET può individuare altri dispositivi nelle vicinanze basandosi sul proprio raggio di comunicazione e su questa base definire risorse e gruppi di entità.
L'astrazione di gruppo risulta utile in questo senso e consente, tramite lo scambio di informazioni, di raggruppare entità che condividono ad esempio medesimi obiettivi (appartenenza ad uno stesso progetto), interessi (stesso topic di un forum) o che sono

prossime (Fig. 19). Non solo: l'implementazione del concetto di gruppo può influenzare anche la scelta dei percorsi di routing da utilizzare: ad esempio due entità vicine potrebbero non potersi scambiare direttamente alcuna informazione proprio perché appartenenti a gruppi differenti; in questo caso il percorso di routing potrebbe diventare significativamente più lungo.

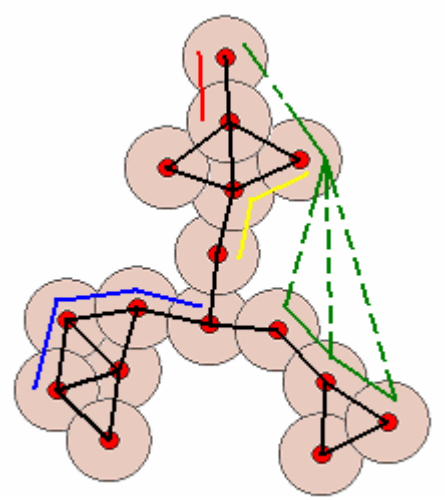


Fig. 19: Suddivisione di una rete in gruppi. Blu, Giallo: per stesso topic; Verde: per stesso progetto; Rosso: per prossimità.

Da questa analisi appare sempre più chiaro che disponendo di uno strato software in grado di tradurre le informazioni fornite dai dispositivi in informazioni direttamente utilizzabili dal resto del sistema (ad esempio la posizione di un'entità rispetto ad un'altra), è possibile rendere l'applicativo maggiormente indipendente dai dettagli delle sorgenti in uso.

Per questo motivo i componenti non possono essere legati da pattern d'interazione stabiliti staticamente in fase di sviluppo dell'applicazione, ma devono essere stabiliti dinamicamente in fase di esecuzione.

3.4.4 Modelli applicativi in un sistema MANET

Al fine di assicurare l'interazione tra i dispositivi, esistono diversi modelli che utilizzano comunicazioni asincrone supportate dallo scambio di particolari tipi di dati.

3.4.5 Modello ad eventi

Nel modello basato ad eventi, un'applicazione distribuita è modellata attraverso un insieme di entità che interagiscono tra di loro attraverso la generazione di informazioni (eventi) e la reazione a quelle di interesse. In generale le componenti di un sistema generano eventi che contengono informazioni su cosa è successo in un certo istante: l'evento vero e proprio dovrà essere costituito da un qualche tipo di struttura dati contenente informazioni.
Un evento è di norma asincrono, cioè viene generato nel momento stesso dell'accadimento, senza attendere alcuna sincronizzazione dei potenziali destinatari.
Questo modo implicito di comunicare libera le entità dall'esplicitare nell'interazione le altre entità o le risorse presenti nell'ambiente.
Le componenti di un sistema Ad-Hoc possono dichiarare tramite una sottoscrizione di essere interessate ad un certo evento o ad una certa classe di eventi ed il modello provvede attraverso una notifica a far pervenire loro eventuali eventi compatibili generati da altre componenti. Quindi ogni entità pubblica gli eventi affidandoli al dispatcher, il quale si occupa di distribuirli alle entità che sono interessate ad essi.
Un modello basato sugli eventi promuove sia un forte disaccoppiamento tra i componenti (le interazioni avvengono in modo asincrono e anonimo senza contare sulla presenza di uno spazio di dati condiviso), sia maggiore *context-awareness* (i componenti possono essere considerati come inseriti in un ambiente attivo in grado di informarli su cosa sta accadendo).
Il modello ad eventi presenta dunque delle caratteristiche interessanti per il coordinamento in un contesto Ad-Hoc, in quanto è un modello asincrono e non necessita di un accoppiamento temporale; in più, essendo stateless non richiede la memorizzazione di molti dati.
Per poter usufruire di queste importanti caratteristiche anche in ambienti MANET, si richiede che ogni nodo della rete possa agire da dispatcher, conseguenza del fatto che in una rete Ad-Hoc ogni nodo deve agire da router. Questo aumenta la complessità di

sviluppo e di gestione della rete in quanto diventano necessari meccanismi di sincronizzazione tra i vari dispatcher al fine di mantenere consistenza nelle informazioni scambiate e per evitare la perdita di eventi non notificati.
Un approccio incentrato sul coordinamento ad eventi fra le diverse entità di un sistema Ad-Hoc promuove quindi un certo livello di disaccoppiamento spazio-temporale tra i processi coinvolti in una comunicazione, ma nascono comunque nuovi problemi e di conseguenza aumenta la complessità delle soluzioni.

3.4.6 Modello a memoria condivisa

I modelli basati su spazi di dati condivisi utilizzano memorie condivise al fine di permettere ai componenti di un'applicazione di raccogliere informazioni ed interagire e coordinarsi a vicenda. Queste strutture di dati possono essere ospitate in un certo spazio dati centralizzato (per esempio, uno spazio di tuple), come in JavaSpaces, o possono essere completamente distribuite sui nodi della rete, come in MARS. In questi casi, le interazioni dei componenti non sono più accoppiate rigorosamente, perché sono mediate dagli spazi di tuple, i quali possono essere usati efficacemente come depositi per informazioni locali e contestuali. Eppure, tali informazioni contestuali possono rappresentare soltanto una descrizione rigorosamente locale del contesto che difficilmente può sostenere compiti complessi, coordinati su scala globale. Il modello a memoria condivisa è stato ampiamente utilizzato per lo sviluppo di applicazioni distribuite in contesti wireline e ha caratteristiche interessanti, in particolare il livello di disaccoppiamento spazio-temporale che lo vede adatto anche in contesti Ad-Hoc.
Il concetto di memoria condivisa può essere trasferito in ambienti MANET perché anch'esso è un ambiente distribuito; tuttavia, disconnessioni e conseguente inaccessibilità dei dati rendono l'applicazione diretta di questo modello praticamente impossibile.
Nei modelli di coordinamento a memoria condivisa tradizionalmente adottati la struttura dati che rappresenta il mezzo di coordinazione è infatti assunta come persistente e globalmente disponibile a ciascun componente. Chiaramente, queste assunzioni discordano con l'alta dinamicità dello scenari Ad-Hoc che stiamo trattando: occorre considerare infatti che cambiamenti nella connettività si riflettono inevitabilmente sulla raggiungibilità e sulla integrità dei dati.

Considerando ciò, la visibilità sui dati condivisi può essere raggiunta ignorando la topologia della rete ma considerando la totalità dei dati che rientrano nella comunità raggiungibile dal dispositivo.

3.4.7 Modello a scambio di messaggi

Questo modello supporta la comunicazione attraverso lo scambio di messaggi: una entità invia un messaggio contenente informazioni di contesto o la richiesta di un servizio alle entità che si sono precedentemente iscritte alla ricezione di quel tipo di messaggio, che lo riceveranno e potranno eventualmente inviare una risposta contenente il risultato dell'esecuzione del servizio o altre informazioni utili. A questo punto è opportuno chiarire la differenza tra messaggio ed evento: un messaggio ha sempre un mittente e almeno un destinatario, tipicamente ha lo scopo di trasferire una o più informazioni. Un evento non ha sempre un mittente e non è detto che abbia almeno un destinatario; un evento indica l'accadimento di un certo avvenimento ed eventualmente porta con sé alcune informazioni.
Sfruttando questo paradigma, è possibile disaccoppiare l'entità richiedente da quella destinataria. Questo tipo di modello può supportare anche il multicasting, può cioè distribuire lo stesso messaggio a più destinatari in modo trasparente; La comunicazione così realizzata, offre disaccoppiamento nello spazio tra "cliente" e "servitore" ma, poiché le entità possono ricevere i messaggi solo dopo essersi iscritti alla ricezione, il disaccoppiamento nel tempo e nel sincronismo non viene del tutto superato. Inoltre talvolta sono necessarie risorse aggiuntive, per esempio in termini di memoria, a causa dell'esigenza di conservare le code di messaggi ricevuti ma non ancora processati. È possibile applicare comunque questi modelli in semplici ambienti mobili; ad esempio è stato sviluppato un adattamento del Java Message Service per dispositivi mobili.

4. Supporto per applicazioni collaborative in Ad-Hoc

Sono attualmente in fase di studio diverse proposte per offrire strumenti più sofisticati a supporto della coordinazione tra i componenti distribuiti di un'applicazione.
Alcuni approcci proposti di recente propongono modelli per consentire l'interazione tra i componenti dinamici di una MANET attraverso strutture dati condivise.
Un altro modello è quello definito a Profili, dove non esiste uno spazio dati condiviso ma le interazioni fra entità avvengono attraverso oggetti, i profili appunto, che contengono le informazioni necessarie al sistema per coordinare l'attività dei vari nodi di una rete Ad-Hoc.

4.1 Global Virtual Data Structure

I modelli basati su una Global Virtual Data Structure (GVDS) [10] mantengono una prospettiva di coordinamento incentrata sempre sul disaccoppiamento tra comunicazione e interazione, considerando anche requisiti di dinamicità.
In una GVDS, la struttura dati utilizzata come intermediario di interazione fra processi è distribuita tra i componenti coordinati seguendo un set di regole ben precise. A ciascun componente è così associato un frammento della struttura dati globale, che può essere manipolata attraverso le operazioni definite su di essa.
In questo modo, lo spazio dei dati non è letteralmente indivisibile, ma costituito da parti separate; lo spazio di coordinamento è invece percepito come un'entità singola, dato che i dati locali risiedenti su ciascun componente sono dinamicamente condivisi in una singola struttura dati globale, la GVDS: la condivisione avviene solamente tra quei componenti che sono mutuamente raggiungibili. Ignorando le regole di connettività, la combinazione dei dati di tutte le parti forma una singola struttura dati globale.

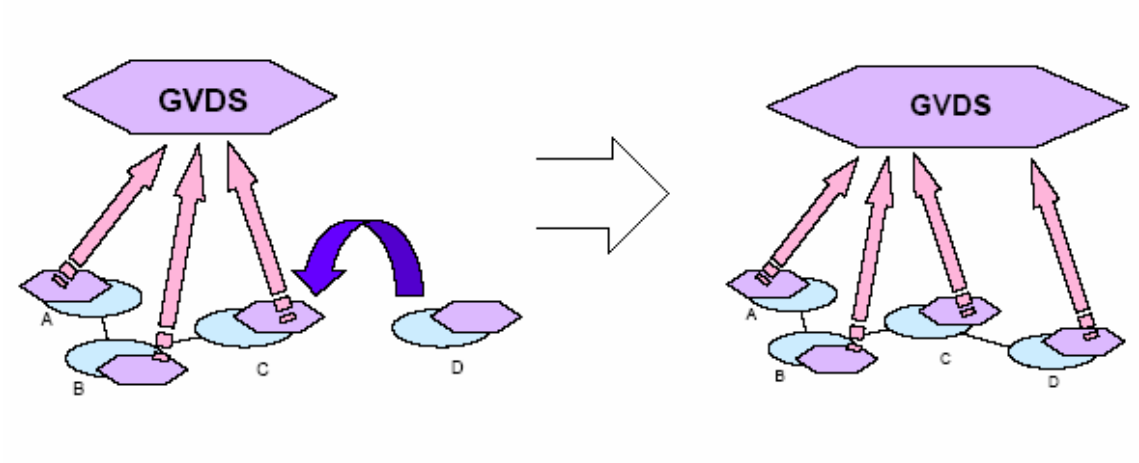


Fig. 20: Arrivo di una nuova entità nel sistema.

Comunque sia, a causa della costante dinamicità dell'ambiente, non tutte le parti saranno simultaneamente disponibili in ogni momento. Tipicamente ogni entità del sistema percepisce solo una parte della GVDS, definita dai dati contenuti dalle singole entità che sono raggiungibili (Fig. 20).

In genere viene assunta come regola principale per determinare la vista locale sui dati la connettività fisica, che risulta ragionevolmente una precondizione all'applicazione di più sofisticate regole di condivisione. In ogni caso, la struttura dati sviluppata per il coordinamento è generalmente disponibile solo attraverso partizioni della stessa. A maggior ragione risulta quindi non persistente, dato che le partizioni visibili possono cambiare dinamicamente.

L'accesso a questa struttura dati condivisa è garantito generalmente attraverso una serie di operazioni definite su ciascuna struttura dati locale. In questo modo, dal punto di vista dei componenti, la distinzione fra dati locali e globali risulta sfumata: è consentito un unico metodo di interazione con il resto del sistema. E' da notare infatti come la nozione di GVDS definisca un modello di coordinamento intrinsecamente peer to peer incentrato su ciascun componente facente parte il sistema; in altri modelli ad esempio, le informazioni di coordinamento sono totalmente esterne ai componenti coordinati. Inoltre, il modello GVDS non solo offre una netta distinzione fra il comportamento dei componenti e la loro interazione, ma separa anche i dati forniti da ciascun componente da quelli presenti nel resto del sistema.

La nozione di GVDS formalmente definita può così essere caratterizzata da alcune proprietà fondamentali: una GVDS è distribuita, nel senso che le componenti della struttura dati risiedono localmente su diverse entità; inoltre può essere considerata

costruttiva perché la visione locale di ciascun componente della GVDS è fornita attraverso combinazioni dei dati dei componenti raggiungibili; risulta uniformemente accessibile in quanto tutte le entità accedono alla GVDS attraverso lo stesso set di operazioni; infine, le operazioni appaiono eseguite localmente anche se possono avere accesso a dati distribuiti o avere effetti globali ("thinking locally but acting globally").
La nozione di GVDS rappresenta quindi una meta-modello per il coordinamento: fornisce una sorta di framework concettuale che può essere interpretato per definire implementazioni indirizzate al soddisfacimento di bisogni specifici.
Specifiche implementazioni del concetto di GVDS possono infatti differenziarsi in diversi aspetti: per esempio, non solo la struttura dati utilizzata e la scelta delle corrispondenti operazioni può variare, ma anche le regole di partizione e di condivisione della stessa. Scelte appropriate nella selezione di queste ed altre alternative costituiscono la chiave per un corretto uso di questo modello.

4.2 Scambio di profili

Fra le tecniche e i metodi per la descrizione e l'accesso agli oggetti digitali, i profili di applicazione rappresentano uno strumento importante, tanto per i fornitori di informazioni quanto per gli utilizzatori di queste informazioni [11].
Grazie alla definizione e documentazione di un profilo infatti è possibile conoscere e sfruttare adeguatamente i servizi in una applicazione o progetto e accedere quindi agli oggetti digitali in modo preciso, integrando fra loro risorse e applicazioni diverse.
Un profilo non è altro che un'interfaccia fornita da una applicazione distribuita verso il mondo esterno: questa interfaccia contiene le regole e il linguaggio di accesso ai servizi offerti dall'applicazione. L'insieme dei profili posseduti da ciascuna entità del sistema costituisce il set di risorse potenzialmente disponibili per la comunità. Un sistema a scambio di profili ha la responsabilità di acquisire e diffondere profili (che rappresentano le informazioni di contesto) rendendoli disponibili a livello applicativo (Fig. 21).
Anche se questo modello si propone di rispondere a problematiche di coordinamento, l'approccio a tali soluzioni risulta differente dal modello GVDS, in primo luogo perché non esiste una struttura dati globalmente condivisa, in secondo luogo perché lo scambio di profili ha una visione molto più semplicistica della realtà che si vuole modellare.

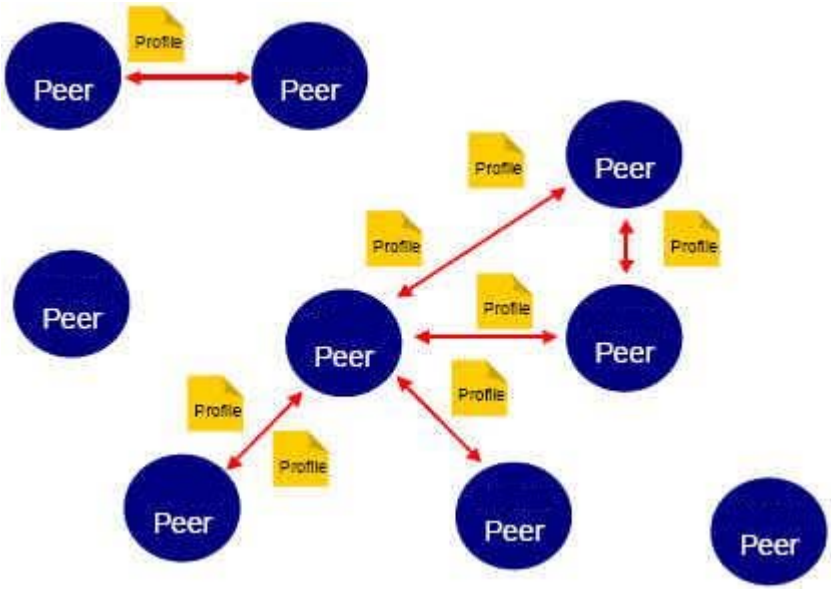


Fig. 21: Scambio di profili tra entità

In questo modello infatti c'è un forte accoppiamento tra i dati e le entità che li contengono (peer). Non è possibile accedere direttamente alle informazioni di interesse ma è necessario utilizzare le interfacce fornite dal peer che contiene le informazioni e che decide di condividerle.

La comunicazione fra entità avviene a scambio di messaggi unicast, broadcast o multicast: se tali messaggi rispettano un dato profilo, comune fra mittente e destinatari/io, vengono processati seguendo le regole proprie di livello applicazione. E' possibile consentire una sorta di profilo generale, comune a qualsiasi peer connesso alla rete, che consente la notifica di cambi di contesto quali ad esempio situazioni di visibilità con nuovi peer o link failure dei canali di comunicazione; lo scambio di messaggi corrisponde quindi al verificarsi di eventi, notificati e sottoscritti attraverso i profili propri di ciascun peer.

La condivisione di profili abilita inoltre una naturale politica di aggregazione in gruppi. peer che condividono lo stesso profilo (ad esempio lo stesso set di messaggi da ricevere/inviare) possono essere automaticamente pensati come facenti parte di uno stesso gruppo; l'appartenenza ad un gruppo può comunque prevedere sistemi di gestione più complessi.

5 Sistemi al lavoro

Al fine di approfondire l'analisi dei requisiti per il supporto di sistemi collaborativi in ambienti MANET consideriamo tre sistemi che costituiscono lo stato dell'arte della ricerca: questi middleware sono nati per il supporto alla programmazione in ambienti MANET e sono implementati in Java, ma le soluzioni adottate si differenziano sia per il modello di sviluppo considerato che per la logica di accesso ai dati.
Il particolare, utilizzando i diversi sistemi abbiamo implementato un semplice servizio di chat per evidenziare come le scelte progettuali di un middleware influenzino di fatto lo sviluppo e le meccaniche di interazione di applicazioni collaborative basate su di essi.

5.1 Specifiche

Il servizio di chat deve poter permettere ad una molteplicità di utenti la comunicazione attraverso messaggi di testo; esiste un canale unico per i messaggi, quindi ogni utente ha la possibilità di leggere tutti quelli inviati dagli altri utenti.
Un messaggio consiste di due campi: il primo è l'UserID del mittente, che permette di associare a ciascun messaggio l'utente che l'ha generato; il secondo contiene il testo vero e proprio; entrambi sono rappresentati da una stringa.

```
public class Message implements java.io.Serializable {

   String id,message;

   /** Crea un nuovo messaggio Message **/
   public Message(String id, String message) {
      this.id=id; //UserID
      this.message=message; //testo del messaggio
   }
   …

   //metodo che consente di stampare a video
   public String toString()
```

```
    {
        String msg="";
        msg=msg+id+": "+message;
        msg += "<br>";
        return msg;
    }
}
```

La connessione al sistema di chat avviene automaticamente all'avvio dell'applicazione. Attraverso l'immissione di un UserID non nullo, inserito attraverso un JOptionPane, si accede all'interfaccia grafica che permette la scrittura e la ricezione di messaggi da parte di altri utenti (Fig. 22).

Fig. 22: Scenario nell'utilizzo del programma di chat

L'interfaccia grafica è composta essenzialmente da un JEditorPane per la scrittura dei messaggi di input e da un JEditorPane per la visualizzazione di tutti i messaggi ricevuti/inviati; una volta scritto il messaggio di testo, alla pressione del tasto Invio esso viene automaticamente inviato a tutti i nodi connessi alla rete e visualizzato nella finestra di output di ciascuno. Ogni nodo connesso alla rete può partecipare alla chat dal momento in cui inserisce l'userID al momento in cui decide di chiudere l'applicazione.

Il primo ambiente che andremo ad analizzare si basa sul modello GVDS e utilizza spazi di tuple condivisi.

5.2 LIME: Linda In a Mobile Environment

LIME [12] è un middleware basato su Java che supporta sia la mobilità fisica che la mobilità logica. Per mobilità fisica si intende la capacità degli host di muoversi nello spazio fisico continuando a mantenere connessioni con altri host. Per mobilità logica intendiamo uno stile architetturale che rompe i legami tra i componenti software e gli host, e permette le migrazioni a run-time delle applicazioni per migliorare la flessibilità e le prestazioni.

LIME è nato specificamente per supportare la complessità degli ambienti MANET ereditando e adattando il modello di comunicazione a tuple proposto da Linda [14].
Uno spazio di tuple aderisce al modello di memoria condivisa ed è usato dalle entità per comunicare. Uno spazio di tuple può essere realizzato come una collezione di strutture dati chiamate appunto tuple, fondamentalmente dei vettori di valori tipati.
La tupla costituisce l'elemento base dello spazio. Le entità creano una tupla e la inseriscono nello spazio tramite un'operazione di out. È poi possibile, da parte di più entità, prelevarla in maniera concorrente usando le primitive di rd e in, che possono essere bloccanti o non bloccanti per il chiamante (Fig. 23). Il modello a spazio di tuple è implementato dalle primitive del sistema LighTS.

```
public void out(lights.interfaces.ITuple tuple)
                        throws TupleSpaceEngineException
```

public lights.interfaces.ITuple rd(lights.interfaces.ITuple template) throws TupleSpaceEngineException
public lights.interfaces.ITuple in(lights.interfaces.ITuple template) throws TupleSpaceEngineException

La comunicazione in Linda è disaccoppiata nel tempo e nello spazio. Il disaccoppiamento nel tempo si riferisce al fatto che mittenti e destinatari non devono essere in comunicazione nello stesso momento per scambiarsi informazioni. Il disaccoppiamento nello spazio si riferisce invece al fatto che le tuple sono disponibili per entità distribuite su più nodi, anche fisicamente lontani. Gli spazi di tuple supportano anche la comunicazione multicast tra diversi gruppi di dispositivi e applicazioni, poiché più applicazioni possono ricevere una copia della stessa tupla.

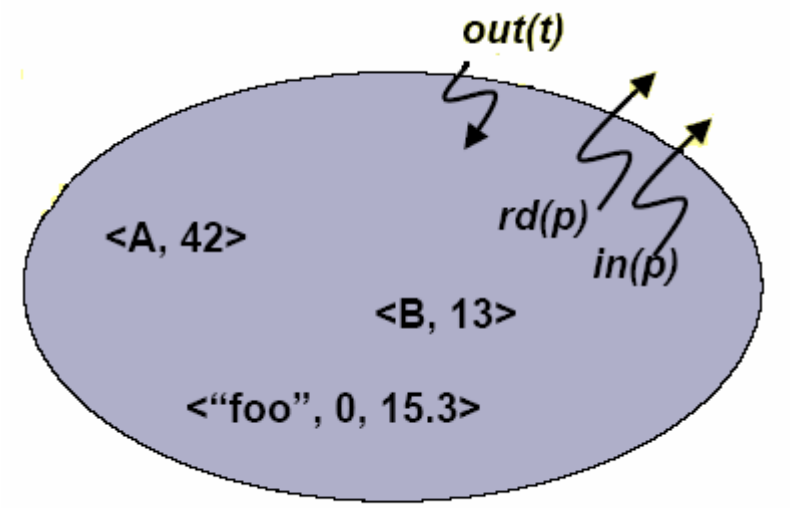


Fig. 23: Struttura di uno spazio di tuple.

In pratica, lo spazio di tuple di Linda viene suddiviso in molti spazi e ciascuno di essi viene permanentemente agganciato ad un host mobile rappresentato da un agente.
Un agente in LIME può essere descritto sostanzialmente come un thread: può essere stazionario, ossia permanentemente agganciato all’host che rappresenta, oppure mobile, cioè con la capacità di migrare da un host all’altro.

Interface ILimeAgent

All Superinterfaces:
java.io.Serializable
All Known Implementing Classes:
StationaryAgent, MobileAgent

```
public class chatl extends StationaryAgent{
...
//creazione della ITS
private LimeTupleSpace chatTS;

public chatl(){
      super();
 }
...
}
```

Ciascun dispositivo mobile può accedere al contesto globale solo attraverso una Interface Tuple Space (ITS), uno spazio di tuple di interfaccia, che è permanentemente unito all'agente e che segue le sue migrazioni.

La ITS contiene le tuple che devono essere condivise con gli altri dispositivi mobili e tutti i metodi necessari sia per effettuare inserimenti, letture e prelievi, che per altre operazioni. Ecco una sintesi delle funzionalità offerte da una ITS:

Metodi principali della classe LimeTupleSpace	
`RegisteredReaction[]`	**`addStrongReaction`**`(LocalizedReaction[] reactions)` Registra un gruppo di reazioni da eseguire al verificarsi di una o più classi di eventi.
`RegisteredReaction[]`	**`addWeakReaction`**`(Reaction[] reactions)` Registra un gruppo di reazioni da eseguire al verificarsi di una o più classi di eventi.
`java.lang.String`	**`getName`**`()`

	Ritorna il nome della ITS.
lights.interfaces.ITuple	**in**(lights.interfaces.ITuple template) Restituisce una tupla che matcha con il parametro template e la rimuove dallo spazio delle tuple.
lights.interfaces.ITuple[]	**ing**(Location current, AgentLocation destination, lights.interfaces.ITuple template) Restituisce tutte le tuple che matchano con il parametro template.
lights.interfaces.ITuple	**inp**(Location current, AgentLocation destination, lights.interfaces.ITuple template) Restituisce una tupla che matcha con il parametro template e la rimuove dallo spazio delle tuple, oppure null se non c'è match.
boolean	**isShared**() Restituisce vero se questo spazio delle tuple è correntemente condiviso, falso altrimenti.
void	**out**(lights.interfaces.ITuple tuple) Inserisce una tupla nello spazio di tuple condiviso.
void	**outg**(AgentLocation destination, lights.interfaces.ITuple[] tuples) Scrive un insieme di tuple nel tuple space.
void	**print**() Stampa il contenuto dello spazio di tuple.
lights.interfaces.ITuple	**rd**(lights.interfaces.ITuple template) Restituisce una copia della tupla che soddisfa il parametro template.
lights.interfaces.ITuple[]	**rdg**(Location current, AgentLocation destination, lights.interfaces.ITuple template) Copia tutte le tuple che soddisfano il template fornito come parametro.
lights.interfaces.ITuple	**rdp**(Location current, AgentLocation destination, lights.interfaces.ITuple template) Restituisce una copia della tupla che soddisfa il parametro template, oppure null se la tupla non esiste.
boolean	**setShared**(boolean isShared)

	Abilita o disabilita la condivisione di questo spazio di tuple con altri aventi lo stesso nome.
`static boolean`	**`setShared`**`(LimeTupleSpace[] lts, boolean isShared)` Abilita o disabilita la condivisione di un insieme di spazi di tuple.

Applicazioni risiedenti sullo stesso dispositivo creano uno spazio di tuple locale detto host-level tuple space. I vari dispositivi poi creano il federated tuple space, il cui contenuto è condiviso in rete (Fig. 24).

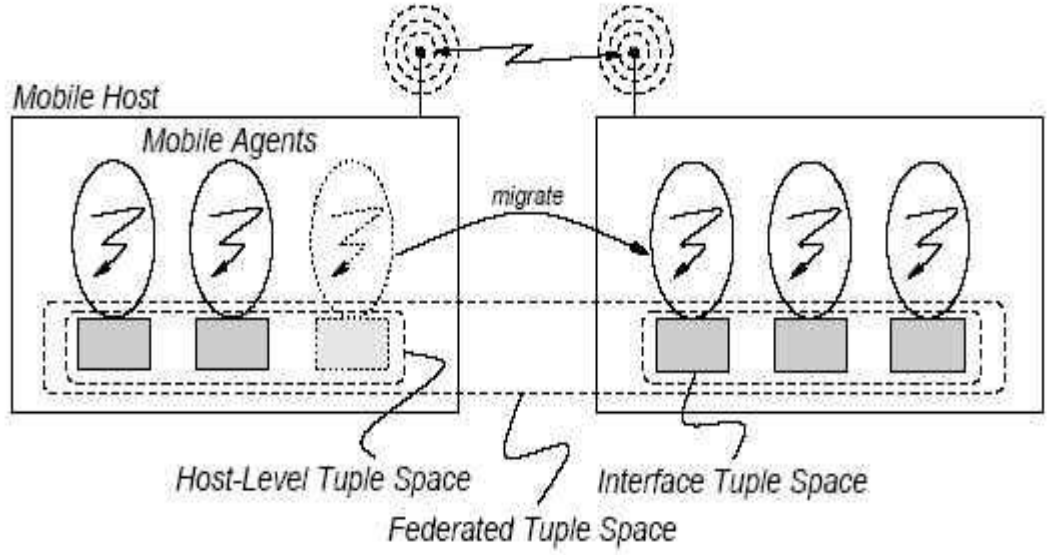


Fig. 24: Vari livelli di condivione di tuple

L'insieme di tuple a cui si può accedere tramite la ITS è ricomputato di volta in volta in base agli altri dispositivi mobili che sono in quel momento attivi. In questo modo, ciascun dispositivo beneficia della visibilità dello stesso spazio di tuple temporaneo presentato agli altri dispositivi. Le operazioni fatte sulla ITS sono effettivamente fatte sullo spazio condiviso. Per esempio, se due agenti, A e B, sono attivi contemporaneamente nello stesso spazio e A inserisce una nuova tupla nella sua ITS con un'operazione out(t), la tupla inserita da A risulta poi disponibile per B, che può prelevarla con una in(p).

I dispositivi possono avere più ITS, alcune anche private, cioè non condivisibili. Il contenuto delle ITS cambia in base alle migrazioni dei dispositivi. All'arrivo di un

nuovo dispositivo, lo spazio di tuple viene ricalcolato dal supporto run-time, sommando il contenuto delle nuove ITS giunte. Questo insieme di operazioni è chiamato engagement dello spazio di tuple e si realizza in un'unica operazione atomica. Considerazioni simili valgono quando un dispositivo lascia lo spazio, con un'operazione di disengagement. Il contenuto delle sue ITS è automaticamente rimosso dallo spazio (Fig. 27).
Nel caso di studio proposto, ogni messaggio può essere considerato come una tupla costituita da due campi: userid e message:

```
public void sendMsg() {

...
//generazione della tupla
ITuple temp = new Tuple().addActual(m.getID()).addActual(m.toString());
...
```

Una tupla, può contenere combinazioni di parametri attuali e parametri formali; in questo caso è necessario creare una tupla contenente due parametri attuali (ID utente e testo del messaggio); per condividerla attraverso la ITS si effettua un'operazione di scrittura (out) sullo spazio di tuple condiviso:

```
…
//recupero l'userID dell'utente che ha scritto un nuovo msg
IField[] fields = temp.getFields();

ITuple template = new Tuple();

template.add(fields[0]).addFormal(Message.class);
      try{
          //l'operazione inp serve per eliminare eventuali msg vecchi                    //appartenenti all'utente
che decide di inviare un nuovo messaggio
          chatTS.inp(new HostLocation(new
LimeServerID(InetAddress.getLocalHost())),
                  new AgentLocation(chatl.this.getMgr().getID()),template);

//scrittura effettiva della nuova tupla
          chatTS.out(temp);
      }
      catch(LimeException le){
      le.printStackTrace();
      }
      catch(UnknownHostException uhe){
      uhe.printStackTrace();
      }
```

```
        jEditorPane1.setText("");
...

}
```

Osserviamo però questa situazione: supponiamo di avere due servizi di chat A e B che condividono lo stesso spazio: A esegue una scrittura di una tupla nella sua ITS. Poiché condividono lo stesso spazio transitorio, la tupla è visibile a B. Se però A parte prima che B prelevi la tupla, A porta il contenuto della sua ITS con sé, compresa quindi la tupla per B. Quando B vorrà leggerla, non la troverà e potrà rimanere bloccato in attesa indefinita se nessuna altra tupla soddisfa il template richiesto.
Per evitare queste situazioni e continuare a godere del disaccoppiamento nel tempo e nello spazio, LIME estende le primitive di Linda con la nozione di location. La location di una tupla è uno spazio. Nel caso di una ITS, la location di una tupla è identificata univocamente dal nome dello spazio e dall'identificatore del dispositivo mobile che la possiede in una delle sue ITS. Con questa informazione, attraverso il meccanismo delle primitive annotate, una tupla può essere posizionata nella ITS T di un'applicazione λ semplicemente utilizzando l'operazione T.out[λ](t), una nuova versione dell'operazione out. La nuova semantica di out[λ](t) si realizza in due passi. Il primo passo, out(t) inserisce la tupla t nel ITS dell'applicazione ω che invoca l'operazione. La tupla t ha due posizioni, una posizione corrente ω e una posizione di destinazione λ. Se l'applicazione λ è attualmente connessa, la tupla t viene trasferita immediatamente nella posizione di destinazione. Le operazioni di inserimento e migrazione sono effettuate in una singola operazione atomica. Altrimenti, se λ non risulta attiva, la tupla t rimane nella posizione corrente ω. Sono necessarie anche versioni diverse di in e rd che fanno uso del parametro di location. La notazione di queste nuove funzioni è in[ω,λ] e rd[ω,λ].

```
public void out(AgentLocation destination,lights.interfaces.ITuple tuple)
throws TupleSpaceEngineException
```

public lights.interfaces.ITuple rdp(Location current,AgentLocation destination, lights.interfaces.ITuple template) throws TupleSpaceEngineException
public lights.interfaces.ITuple inp(Location current, AgentLocation destination, lights.interfaces.ITuple template) throwsTupleSpaceEngineException

Le tuple create dall'utente sono automaticamente incrementate dal supporto a run-time con i campi che memorizzano le posizioni corrente e di destinazione.
Gli spazi di tuple hanno un nome; il nome effettivamente definisce una nozione di tipo per lo spazio e, nel caso delle ITS, determina le regole di condivisione. Per esempio, se il dispositivo A ha le interfacce chiamate S, T e U e il dispositivo B ha le interfacce T, U, V, solo T e U sono condivise tra A e B.

LIME estende lo spazio di tuple aggiungendo la nozione di reaction ad un evento. Un evento può riguardare cambiamenti dell'ambiente o delle applicazioni, per esempio disconnessioni e qualità del servizio. Gli eventi sono naturalmente rappresentati con delle tuple. Il supporto a run-time monitorizza continuamente lo spazio per controllare se avviene un evento. Quando se ne verifica uno, viene immediatamente creata una tupla che lo rappresenta e viene inserita nella LimeSystem ITS. Esiste inoltre il concetto di reactive statement, esprimibile tramite l'operazione T.reactsTo(s,p), in cui s è un frammento di codice che deve essere eseguito se nello spazio è presente una tupla che corrisponde al pattern p specificato. Dopo ogni operazione, viene selezionata, in modo non deterministico, una reazione tra quelle registrate e viene valutata. Se il matching tra una tupla dello spazio e il template è verificato, viene eseguito il codice s. Questo meccanismo offre la possibilità di reagire alla disponibilità di tuple in ITS remote.
Due tipi di reazione sono disponibili per venire incontro a requisiti di atomicità che possono presentarsi:

Strong Reaction
public RegisteredReaction[] addStrongReaction(LocalizedReaction[] reactions) throws TupleSpaceEngineException

L’operazione è eseguita in modo atomico; la reazione deve avvenire nella locazione in cui si presenta la tupla che la scatena.

Weak Reaction
public RegisteredReaction[] addWeakReaction(Reaction[] reactions) throws TupleSpaceEngineException
L’operazione non è eseguita in modo atomico e il fatto che non ci siano vincoli sulla locazione della tupla coinvolta può indurre a una transazione distribuita.

Nel nostro caso gli agenti sono interessati alla notifica di nuovi messaggi inviati da utenti nello spazio di tuple condiviso; si crea così un template che definisce la base di confronto per le nuove tuple inserite e si registra una reazione a qualsiasi tupla che soddisfa tale template.

```
//metodo run dell’agente
public void run(){
...

notifyReceivedMsg();

…
}

//definisco la volontà di reagire ad ogni inserimento di tuple che soddisfano un
//template
private void notifyReceivedMsg(){

    ITuple template = new Tuple();

//template che definisce le tuple che scatenano la reazione
    template.addFormal(String.class).addFormal(String.class);

    try{
      chatTS.addWeakReaction(new Reaction[] {
        new UbiquitousReaction(template, new MessageListener(chatl.this),
                    Reaction.ONCEPERTUPLE)});
    }
```

```
      catch(TupleSpaceEngineException ex){
        ex.printStackTrace(System.err);
      }
    }


…
//un'istanza di questa classe viene creata e viene utilizzato il metodo reactsTo
di //gestione dell'evento
class MessageListener implements ReactionListener{
  private chatl agent = null;
  public MessageListener(chatl agent){
    this.agent = agent;
  }

  public void reactsTo(ReactionEvent e){
    System.out.println("nuovo messaggio arrivato");
    agent.processMsg((ITuple)e.getEventTuple());
  }
}
…

public void processMsg(ITuple t){
        IField[] ifields = t.getFields();
        String id = (String)ifields[0].getValue();
        String m = (String) ifields[1].getValue();
        outputMsg += m;
        jEditorPane2.setText(outputMsg);
        System.out.println("messaggio scritto a video");
    }
```

Le weak reaction vengono usate principalmente per rilevare cambiamenti di contesto dei dati in porzioni della GVDS; in questo caso l'host che contiene la tupla che matcha con il pattern inserito e l'host nel quale l'operazione viene eseguita sono in genere differenti. La possibilità di specificare delle weak reaction sull'intero spazio delle tuple costituisce così un mezzo per reagire agli eventi in maniera indipendente dai cambiamenti del sistema.

5.3 PeerWare

Peerware [13] fornisce un set di primitive atte a supportare architetture di tipo peer to peer.
La struttura dati scelta per l'implementazione del modello GVDS in PeerWare è ad albero, o più precisamente a foresta di alberi. Ciascun componente del sistema, chiamato peer, ospita al suo interno diversi alberi con radici distinte. I nodi di ciascun albero sono essenzialmente contenitori per documenti, i quali forniscono i dati appartenenti al contesto di applicazione (Fig. 25).

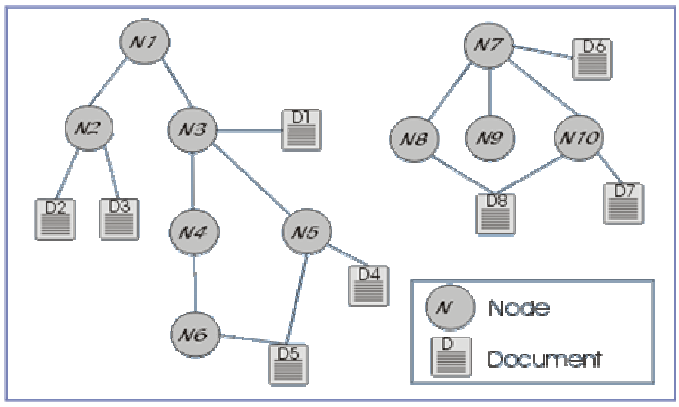
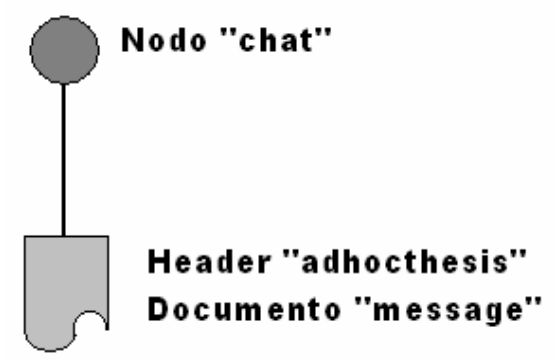


Fig. 25: Struttura dati di PeerWare

Per l'applicazione chat si è pensato di utilizzare una struttura dati formata da un nodo chiamato "chat" e un documento figlio di nome "message", avente come Header "adhocthesis" (Fig. 26).




```
public class header implements
Header{

    private String info;
    /** Creates a new instance of
header */
    public header(String info) {
        this.info=info;
    }

    public String getDocInfo()
    {
      return info;
    }
}
```

Fig. 26: Struttura dati della chat di PeerWare

Quando viene stabilita una connessione fra più peer, la GVDS viene dinamicamente ricostruita nel modo seguente: tutti i nodi omologhi, ad esempio i nodi con lo stesso nome e che hanno la stessa posizione nell'albero del peer di riferimento, sono

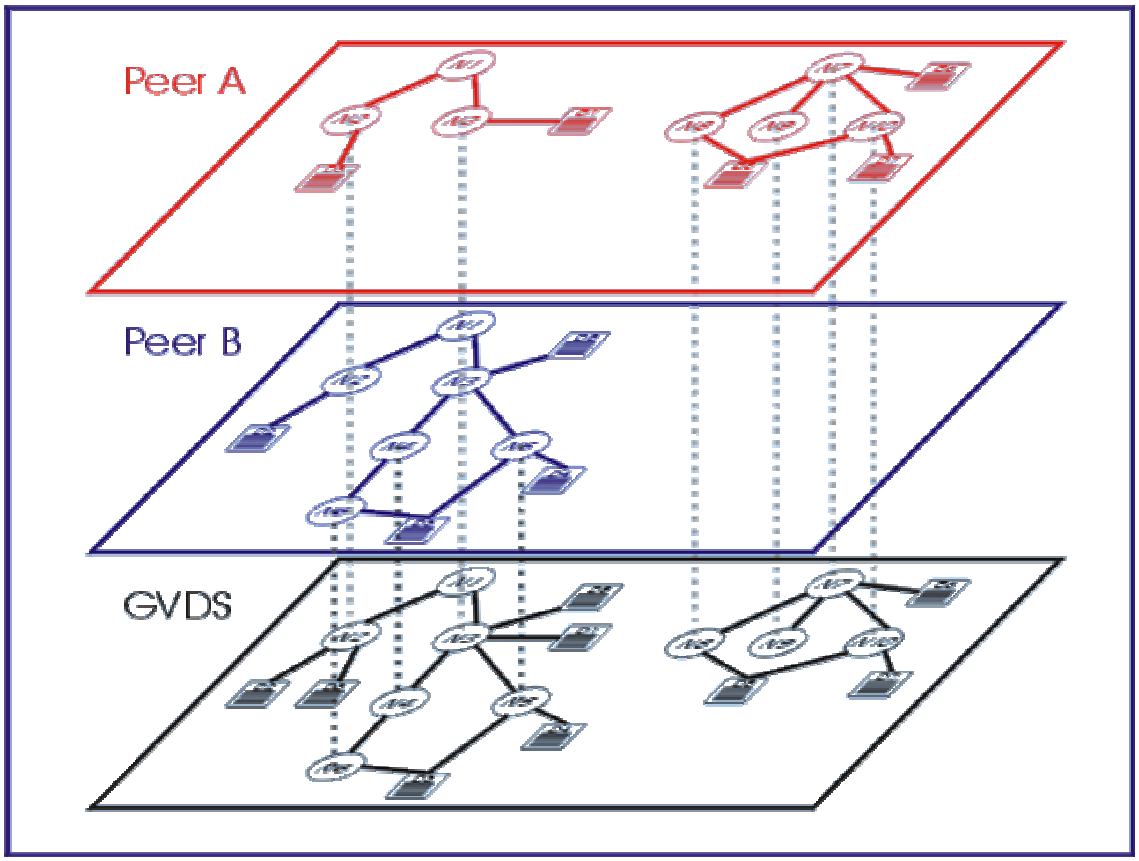


Fig. 27: Composizione della GVDS in PeerWare

rappresentati da un nodo della GVDS che ha lo stesso nome e la stessa posizione di quelli concreti. Il contenuto di questo nodo nella GVDS corrisponde all'unione dei documenti contenuti nei nodi omologhi. In sostanza, la GVDS è ottenuta sovrapponendo tutte le strutture dati locali dei peer correntemente connessi alla rete PeerWare (Fig. 27).

Nell'applicazione di chat sviluppata, la struttura dati (repository) dedicata a contenere nodi e documenti deve riferirsi all'interfaccia IRepository, che fornisce tutte le operazioni da implementare necessarie per una corretta gestione delle informazioni.
In particolare la struttura ad albero viene ricreata utilizzando una hashtable per i nodi e un vettore per i documenti: ogni elemento della hashtable è formato da una chiave, il nome del nodo, e da un'altra hashtable interna che rappresenta l'eventuale contenuto del nodo (Fig. 28).

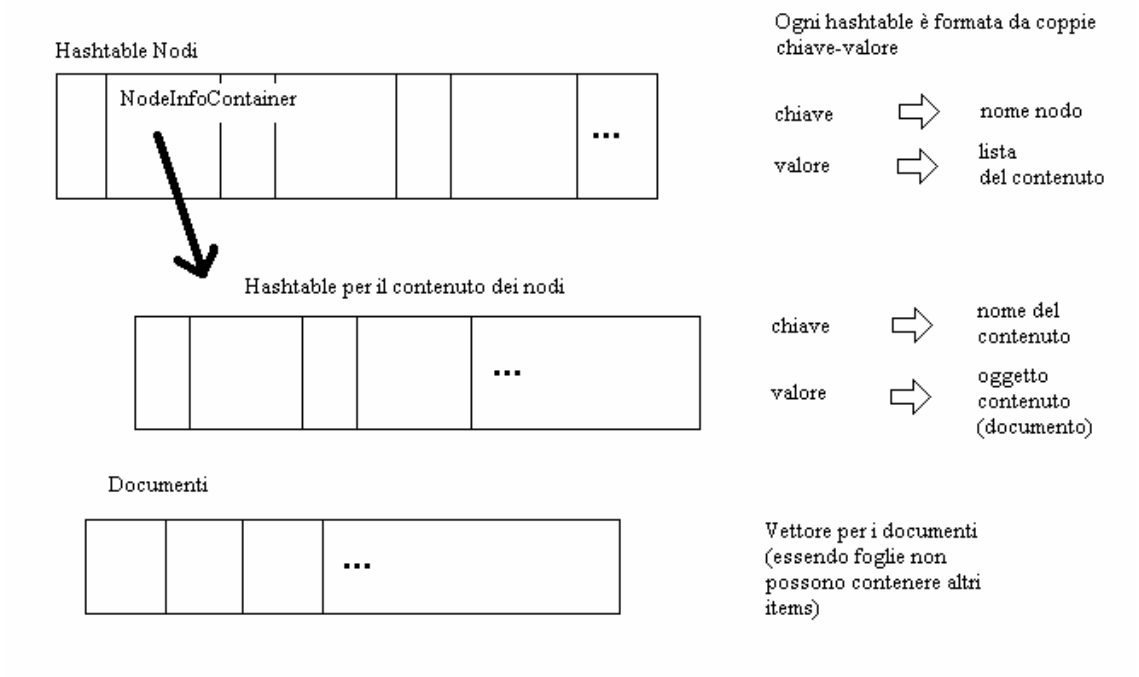


Fig. 28: Implementazione della struttura dati attraverso hashtable

```
public class SimpleRepository
    implements IRepository
{

    private Hashtable allNodesList;
    private Vector allDocumentsList;
    private boolean debug;


//classe che funge da contenitore per eventuali oggetti presenti in un nodo
     private class NodeInfoContainer
```

```
   {

      public Node getNode()
      {
         return nodeReference;
      }

      public Hashtable getChildrenDocsList()
      {
         return childrenDocs;
      }

      public String toString()
      {
         String messageToReturn = " Node: " +
nodeReference.getFullName();
         if(alreadyTaken)
            messageToReturn = messageToReturn + ", alreadyTaken=true, ";
         else
            messageToReturn = messageToReturn + ", alreadyTaken=false,
";
         messageToReturn = messageToReturn + "docs: \n" +
childrenDocs.toString() + "\n";
         return messageToReturn;
      }

      public Node nodeReference;

//hashtable che mantiene i possibili figli di un nodo
      public Hashtable childrenDocs;
      public boolean alreadyTaken;

      public NodeInfoContainer(Node nodeReference, Hashtable
childrenDocs)
      {
         alreadyTaken = false;
         this.nodeReference = nodeReference;
         this.childrenDocs = childrenDocs;
      }
   }

...

public SimpleRepository()
```

```
   {
      debug = Peer.debug;

//hashtable primaria per i nodi
      allNodesList = new Hashtable();

//vettore per i documenti
      allDocumentsList = new Vector();
   }

...

public synchronized void insertNode(Node node, Node nodeFather)
      throws DuplicateItemException, NotExistingItemException,
RepositoryException
   {
      if(node == null)
         throw new IllegalArgumentException();
      if(nodeFather != null)
      {
         if(!belongsToNodesList(nodeFather.getFullName()))
            throw new NotExistingItemException();
         if(belongsToNodesList(node.getFullName()))
            throw new DuplicateItemException();
         if(node.getFullName().compareTo(nodeFather.getFullName() + "/" +
node.getLabel()) != 0)
            throw new RepositoryException();

//aggiorno la lista dei nodi
         allNodesList.put(node.getFullName(), new NodeInfoContainer(node,
new Hashtable()));
      } else
      {
         if(node.getFullName().compareTo("/" + node.getLabel()) != 0)
            throw new RepositoryException();
         if(belongsToNodesList(node.getFullName()))
            throw new DuplicateItemException();
         allNodesList.put(node.getFullName(), new NodeInfoContainer(node,
new Hashtable()));
      }
   }

...
```

```
public synchronized void placeIn(Document doc, Node node)
      throws DuplicateItemException, NotExistingItemException,
RepositoryException
   {
      if(doc == null || node == null)
         throw new IllegalArgumentException();
      boolean exception = false;
      if(exception)
         throw new RepositoryException();
      NodeInfoContainer nodeInfoContainer =
(NodeInfoContainer)allNodesList.get(node.getFullName());
      if(nodeInfoContainer == null)
         throw new NotExistingItemException();
      Hashtable childrenDocsList = nodeInfoContainer.getChildrenDocsList();
      if(childrenDocsList.containsKey(doc.getLabel()))
         throw new DuplicateItemException();
      String existingPaths[] = doc.getPaths();
      DocumentInfoContainer docInfoContainer = null;
      for(int i = 0; i < existingPaths.length && docInfoContainer == null; i++)
      {
         NodeInfoContainer tempNodeInfoContainer =
(NodeInfoContainer)allNodesList.get(existingPaths[i]);
         if(tempNodeInfoContainer != null)
         {
            Hashtable docList =
tempNodeInfoContainer.getChildrenDocsList();
            docInfoContainer =
(DocumentInfoContainer)docList.get(doc.getLabel());
         }
      }

      if(docInfoContainer == null)
         try
         {
            Document copyDoc = (Document)doc.clone();
            docInfoContainer = new DocumentInfoContainer(copyDoc);
            allDocumentsList.addElement(docInfoContainer);
         }
         catch(CloneNotSupportedException ex)
         {
            if(debug)
               ex.printStackTrace();
            throw new RepositoryException();
         }
```

```
        else
            try
            {
                Document copyDoc = (Document)doc.clone();
                docInfoContainer.setDocument(copyDoc);
            }
            catch(CloneNotSupportedException ex)
            {
                if(debug)
                    ex.printStackTrace();
                throw new RepositoryException();
            }

//creo una corrispondenza tra nodo e documento
        childrenDocsList.put(doc.getLabel(), docInfoContainer);
    }

...

public Node getNode(String nodeLabel)
        throws NotExistingItemException
    {
        if(nodeLabel == null)
            throw new IllegalArgumentException();
        NodeInfoContainer nodeInfoContainer =
(NodeInfoContainer)allNodesList.get(nodeLabel);
        if(nodeInfoContainer != null)
            return nodeInfoContainer.getNode();
        else
            throw new NotExistingItemException();
    }

...

public synchronized Document getDocument(String docLabel)
        throws NotExistingItemException
    {
        if(docLabel == null)
            throw new IllegalArgumentException();
        String fatherFullName = Item.parseFatherFullName(docLabel);
        String nodeLabel = Item.parseLabel(docLabel);
        if(fatherFullName == null || fatherFullName == "" || nodeLabel == "" ||
nodeLabel == null)
            throw new NotExistingItemException();
```

```
        NodeInfoContainer nodeInfoContainer =
(NodeInfoContainer)allNodesList.get(fatherFullName);
        if(nodeInfoContainer == null)
            throw new NotExistingItemException();
        Hashtable docList = nodeInfoContainer.getChildrenDocsList();
        DocumentInfoContainer docInfoContainer =
(DocumentInfoContainer)docList.get(nodeLabel);
        if(docInfoContainer == null)
            throw new NotExistingItemException();
        else
            return docInfoContainer.getDocument();
    }

...

public synchronized void removeNode(Node node)
        throws NotExistingItemException, NodeNotEmptyException,
RepositoryException
    {
        if(node == null)
            throw new IllegalArgumentException();
        boolean exception = false;
        if(exception)
            throw new RepositoryException();
        if(node == null)
            throw new NotExistingItemException();
        NodeInfoContainer nodeInfoContainer =
(NodeInfoContainer)allNodesList.get(node.getFullName());
        if(nodeInfoContainer == null)
            throw new NotExistingItemException();
        Hashtable docList = nodeInfoContainer.getChildrenDocsList();
        if(!docList.isEmpty() || readChildrenNodes(node.getFullName()) != null)
        {
            throw new NodeNotEmptyException();
        } else
        {
            allNodesList.remove(node.getFullName());
            return;
        }
    }

…

public synchronized void removeFrom(Document doc, Node node)
```

```
        throws NotExistingItemException, RepositoryException
    {
        boolean exception = false;
        if(exception)
            throw new RepositoryException();
        if(doc == null || node == null)
            throw new IllegalArgumentException();
        NodeInfoContainer nodeInfo =
(NodeInfoContainer)allNodesList.get(node.getFullName());
        if(nodeInfo == null)
            throw new NotExistingItemException();
        Hashtable docList = nodeInfo.getChildrenDocsList();
        if(docList == null)
            throw new NotExistingItemException();
        if(!docList.containsKey(doc.getLabel()))
        {
            throw new NotExistingItemException();
        } else
        {
            DocumentInfoContainer docInfoContainer =
(DocumentInfoContainer)docList.get(doc.getLabel());
            docInfoContainer.setDocument(doc);
            docList.remove(doc.getLabel());
            return;
        }
    }

…

}
```

Il middleware distingue tra struttura dati locale e globale fornendo due differenti set di operazioni: la scelta di distinguere fra accesso locale e accesso globale ai dati, con le implicazioni derivanti, è esplicita per il programmatore.
Quindi, è definito un set di operazioni solo per la struttura dati locale; queste hanno a che fare con la modifica dei dati memorizzati localmente, come creazioni e rimozioni di nodi e documenti.

Primitive di contesto locale	
`Item[]`	**`execute`**`(NodeFilter nf, ItemFilter itf, ActionDesc ad)`

	Esegue in maniera bloccante l'azione ad sulla proiezione della struttura dati identificata dai filtri nf e itf per produrre la sequenza di item I che sarà poi rispedita al chiamante.
ResultWrapper	**executeAndSubscribe**(NodeFilter nf, ItemFilter itf, EventFilter ef, ActionDesc ad, EventCallback ec) Esegue l'azione ad sulla proiezione della struttura dati identificata dai filtri nf e itf. Inoltre, allo stesso tempo, sottoscrivere la notifica all'occorrenza di una particolare classe di eventi selezionata da ef, e pubblicata sul set di item precedentemente selezionati. Quando si presenta l'evento, ec è eseguito localmente al chiamante.
void	**executeAsynch**(NodeFilter nf, ItemFilter itf, ActionDesc a, ItemCallback icb) Esegue in maniera non bloccante l'azione a sulla proiezione della struttura dati identificata dai filtri nf e itf .
void	**insertDoc**(Document doc, Node n) Inserisce una copia del documento doc nel nodo n.
void	**insertNode**(Node n, Node nf) Inserisce il nodo n nel nodo padre nf (nf deve esistere).
void	**placeIn**(java.lang.String docFullName, Node n) Inserisce un collegamento fra un documento esistente docFullName e il nodo fornito come parametro.
void	**publish**(PeerEvent evt, Item item) Questo metodo permette di informare l'event dispatcher dell'occorrenza di un evento evt su un item che è passato come parametro. Il sistema di gestione eventi provvederà poi a notificare i peer che hanno sottoscritto la sua notifica.
void	**removeFrom**(Document d, Node n) Rimuove il documento d dal nodo n.
void	**removeNode**(Node n) Rimuove il nodo n.
SubscriptionID	**subscribe**(NodeFilter nf,ItemFilter itf,EventFilter ef, EventCallback evc) Permette ad un peer di sottoscrivere la notifica all'occorrenza di una particolare classe di eventi selezionata da ef, e pubblicata sul set di item identificati dalla combinazione dei filtri nf e itf. Quando si presenta l'evento, evc è eseguito localmente al chiamante.
void	**unsubscribe**(SubscriptionID subID) Permette ad un peer di eliminare la sottoscrizione identificata

	da subID.

PeerWare, al contrario di Lime non fornisce nessun meccanismo per modificare direttamente dati remoti. In PeerWare è presente una primitiva chiamata *publish* che permette la generazione di classi di eventi riguardanti nodi o documenti.
Un'altra tipologia di operazioni è definita per la GVDS: le operazioni permesse sulla GVDS sono in numero molto limitato ma posseggono un buon potere espressivo.

Primitive di contesto globale	
`Item[]`	**`execute`**`(NodeFilter nf, ItemFilter itf, ActionDesc a)` Esegue in maniera bloccante l'azione ad sulla proiezione della struttura dati identificata dai filtri nf e itf per produrre la sequenza di item I che sarà poi rispedita al chiamante.
`ResultWrapper`	**`executeAndSubscribe`**`(NodeFilter nf, ItemFilter itf, EventFilter ef, ActionDesc a, EventCallback evc)` Esegue, attraverso ciascuna struttura dati locale, l'azione a sulla proiezione della struttura dati identificata dai filtri nf e itf. Inoltre, allo stesso tempo, sottoscrivere la notifica all'occorrenza di una particolare classe di eventi selezionata da ef, e pubblicata sul set di item precedentemente selezionati. Quando si presenta l'evento, evc è eseguito localmente al chiamante.
`void`	**`executeAsynch`**`(NodeFilter nf, ItemFilter itf, ActionDesc a, ItemCallback itcb)` Esegue in maniera non bloccante l'azione a sulla proiezione della struttura dati identificata dai filtri nf e itf .
`SubscriptionID`	**`subscribe`**`(NodeFilter nf, ItemFilter itf, EventFilter ef, EventCallback evc)` Permette ad un peer di sottoscrivere la notifica all'occorrenza di una particolare classe di eventi selezionata da ef, e pubblicata sul set di item identificati dalla combinazione dei filtri nf e itf. Quando si presenta l'evento, evc è eseguito localmente al chiamante.
`void`	**`unsubscribe`**`(SubscriptionID subID)` Permette ad un peer di eliminare la sottoscrizione identificata da subID.

La distinzione fra esecuzione delle operazioni in un contesto locale o globale consente di decidere esplicitamente su quali dati lavorare; in definitiva però l'esecuzione di una

operazione globale si riassume nella sua traduzione in locale sui peer correntemente connessi.
La principale primitiva di accesso ai dati è la execute, che accetta come parametri un set di filtri e una azione.
I filtri sono espressioni template che permettono di restringere l'insieme di documenti e nodi (generalmente identificati come items) nel quale opera la execute.
Nell'applicazione di chat, un filtro di nodi (NodeFilter) deve selezionare tutti i nodi identificati dalla label "chat".

```
public class nodeFilter implements NodeFilter{

//l'oggetto filtro selezionerà tutti i nodi che matchano con la stringa nodeFilter
    private String nodeFilter;

    public nodeFilter(String nodeFilter) {
        this.nodeFilter=nodeFilter;
    }

    public String getFilterInfo(){
      return nodeFilter;
    }

    public boolean match(String nodeFullName) {
        if(nodeFilter.equals(nodeFullName))
            return true;
        if(nodeFilter.equals("*"))
            return true;
        if(nodeFilter.endsWith("*") && nodeFilter.regionMatches(0,
nodeFullName, 0, nodeFilter.length() - 1))
            return true;
        return nodeFilter.startsWith("*") && nodeFilter.regionMatches(1,
nodeFullName, (nodeFullName.length() - 1 - (nodeFilter.length() - 1)) + 1,
nodeFilter.length() - 1);
    }

     public boolean equals(Object obj)
    {
        if(!(obj instanceof nodeFilter))
            return false;
        else
            return nodeFilter.equals(((nodeFilter)obj).getFilterInfo());
```

```
    }
}
```

Analogamente, un filtro per gli item deve restringere il dominio della execute ai documenti di label “message”.

```
public class itemFilter implements ItemFilter{

    /** Creates a new instance of itemFilter */
    public itemFilter(String nodeFilter,header headerFilter) {

        if(nodeFilter == null || headerFilter == null)
        {
            throw new IllegalArgumentException();
        } else
        {
            this.nodeFilter = nodeFilter;
            this.headerFilter = headerFilter;
            return;
        }

    }

    public header getHeaderFilter()
    {
        return headerFilter;
    }

    public String getNodeFilter()
    {
        return nodeFilter;
    }


    //match per eventuali altri nodi
    public boolean match(String nodeFullName)
    {
        if(nodeFilter.equals(NOMATCH))
            return false;
        else
            return valueMatch(nodeFilter, nodeFullName);
```

```
    }


    public boolean match(Header head) {

        if(headerFilter.getDocInfo().equals(NOMATCH))
            return false;
        if(headerFilter.getDocInfo().equals("*"))
            return true;
        if(!(head instanceof header))
            return false;
        String docInfo = ((header)head).getDocInfo();

        return valueMatch(headerFilter.getDocInfo(), docInfo);



    }

    public boolean equals(Object obj)
    {
        if(!(obj instanceof itemFilter))
            return false;
        return headerFilter.equals(((itemFilter)obj).getHeaderFilter()) &&
nodeFilter.equals(((itemFilter)obj).getNodeFilter());
    }

    private boolean valueMatch(String filter, String value)
    {
        if(value == null || value.equals(""))
            return true;
        if(filter.equals(value))
            return true;
        if(filter.equals("*"))
            return true;
        if(filter.endsWith("*") && filter.regionMatches(0, value, 0, filter.length() -
1))
            return true;
        return filter.startsWith("*") && filter.regionMatches(1, value,
(value.length() - 1 - (filter.length() - 1)) + 1, filter.length() - 1);
    }

    private String nodeFilter;
    private header headerFilter;
```

```
    private static String NOMATCH="//NOMATCH//";
}
```

Le azioni specificano una sequenza di operazioni da eseguirsi sugli items prima di passarli al chiamante dell'operazione di execute. L'azione più semplice risulta chiaramente quella vuota: in questo caso l'effetto dell'azione è semplicemente quello di ritornare il set di items selezionato dal filtro, senza modifiche. La scelta delle operazioni da eseguire durante un'azione permette di ridefinire comunque la semantica di accesso alla struttura dati: per esempio è possibile costruire azioni che eseguono ulteriori operazioni di filtering sugli items e poi accedono alla struttura dati (per rimuovere un determinato elemento).
In sostanza, la primitiva execute fornisce ai peer un meccanismo per disseminare sui peer correntemente connessi una richiesta di esecuzione di codice che accede alle strutture dati che popolano la GVDS. L'esecuzione di una singola azione è atomica solo per ogni peer che la riceve.
Assieme alla execute, PeerWare fornisce anche una primitiva chiamata subscribe: anch'essa richiama al modello publish subscribe ed opera sullo stato del sistema invece che su un generico evento. Infatti, subscribe permette ai peer di registrarsi alla notifica di una o più classi di eventi generati su un sottoinsieme dei dati contenuti nella GVDS identificati da uno specifico filtro.
Infine l'ultima primitiva, chiamata executeAndSubscribe combina in un'unica espressione atomica le due precedenti: permette al programmatore di "agganciarsi" ad un determinato insieme di dati tramite una execute ed allo stesso tempo, esegue una sottoscrizione alla notifica di eventi sugli stessi, consentendo di implementare schemi aventi un livello di consistenza più forte rispetto all'esecuzione successiva di una execute e di una subscribe separate. In quest'ultimo caso infatti è garantita l'atomicità di ogni singola operazione ma non delle due insieme.
Questa primitiva viene utilizzata dall'applicazione chat: infatti la prima fase (execute) consente di modificare il contenuto del documento che contiene il messaggio nella GVDS di PeerWare aggiornandolo con il nuovo messaggio inviato a livello utente.
Nel nostro caso l'azione deve eseguire la modifica del contenuto del documento corrente sovrascrivendo il vecchio messaggio e inserendo quello nuovo.

```
public class changeMessageAction extends peerware.Action {

    /** Creates a new instance of changeMessageAction */

    //recupero la stringa in ingresso dalla text area e la metto in newMessage
    public changeMessageAction(Message m) {
        newMessage=m;
        //associo il pannello all'output della GUI
        outputView=chatp.jEditorPane2;

    }

    //metodo da chiamare in presenza di una execute
    public peerware.Item[] processItem(Item[] item, IRepository iRepository,
MuServer muServer) {
        if(item==null)  return null;
        for(int i=0;i<item.length;i++)
        if(item[i] instanceof Document)
        {
         System.out.println("procedo alla modifica del messaggio");
         Object o=((Doc)item[i]).getContent();
         if(o instanceof Message)
         {
            //se o e' un messaggio provvedo a modificarlo
            //modifica id
            ((Message)o).setID(newMessage.getID());
            //modifica testo
            ((Message)o).setMessage(newMessage.getMessage());

            //scrittura del nuovo messaggio a video
           // outputView.setText(((Message)o).toString());

            //notifica evento di modifica messaggio
            Hashtable info = new Hashtable();
            eventDefined evt = new eventDefined(info);
            evt.setEventType("MESSAGE_CHANGED");
            enqueueEvent(evt, item[i]);
         }

        }
        return item;

    }
```

```
    private Message newMessage;

    //pannello che fa riferimento alla GUI
    //precisamente al pannello di uscita
    JEditorPane outputView;

}
```

Nello stesso momento si richiede al runtime la notifica di eventuali cambiamenti negli item selezionati dai filtri. In questo modo, inserendo un nuovo messaggio vengono notificati i peer connessi e si procede alla stampa a video.

```
//la classe, implementando l'interfaccia eventCallBack, puo' rispondere
//all'evento di modifica del messaggio
public class eventCallBack implements EventCallback {

    /** Creates a new instance of eventCallBack */
    public eventCallBack() {

    }

    //il metodo risponde all'evento "messaggio modificato"
    //non fa altro che visualizzare il nuovo messaggio
    public void processEvent(PeerEvent peerEvent) {
        Doc d=null;
        try{
            //prendo il documento di label "message"
            d=(Doc)rep.getDocument("message");
        }
        catch(NotExistingItemException e)
        {
            e.printStackTrace();
            System.exit(1);
        }
        //devo stampare a video il documento
        outputView.setText(((Message)d.getContent()).toString());

    }

    //pannello che fa riferimento alla GUI
    //precisamente al pannello di uscita
    private JEditorPane outputView=chatp.jEditorPane2;


    private SimpleRepository rep=chatp.rep;
}
```

La pubblicazione di classi di eventi è riservata a livello locale; in sostanza, può essere creato qualsiasi tipo di evento; sta al programmatore decidere la politica di gestione opportuna e sottoscrivere i peer solo alla notifica degli eventi interessanti.

L'atomicità di esecuzione delle operazioni è garantita solo a livello locale ma, vista la natura della struttura dati e dell'interazione della stessa con la GVDS, garantisce la consistenza del sistema.

5.4 Proem

L'architettura di Proem [11] è implementata seguendo il modello Context Aware e definisce quattro entità fondamentali:

- **Peer:** host autonomo o dispositivo mobile facente parte della rete Proem

- **Individual:** un Individual è una persona o un utente che utilizza uno o più Peer; ciascun Individual può usare uno o più Peer, ma ciascun Peer è usato solamente da un Individual.

- **Data Space:** è una collezione di strutture dati possedute e gestite da un Peer. Un Data Space può essere condiviso e ciascun Peer che lo possiede contiene una sua copia.

- **Community:** è un insieme di entità (incluse altre Community). Ciascuna entità può essere membro di diverse Community e ciascuna Community può avere membri di diverso tipo; in generale, le Community vengono usate per identificare dei gruppi e per definire gli accessi a dati o servizi. L'appartenenza ad una Community non è controllata da una entità in particolare ma è conferita attraverso lo scambio continuo fra i partecipanti di un token crittografato, univoco per ciascuna Community. Ogni Peer controlla il token ricevuto e verifica il rispetto dei criteri di appartenenza (ad esempio se il numero di segnature del token è maggiore o uguale di un certo numero) (Fig. 29).

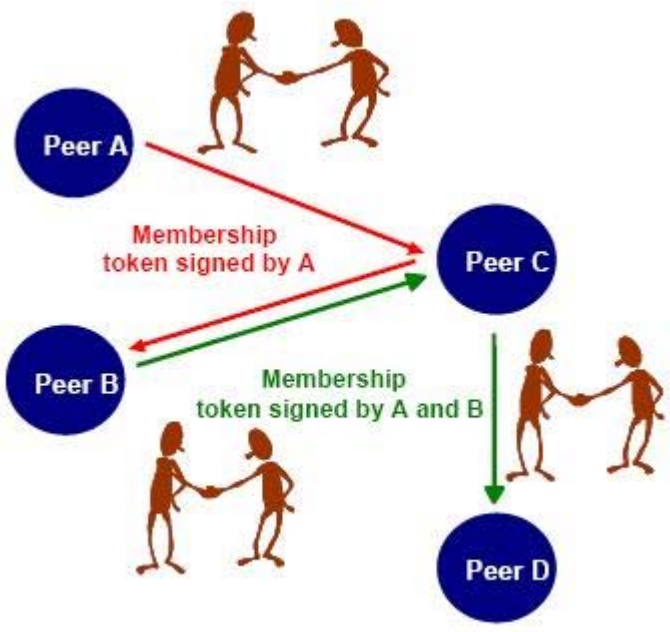


Fig. 29: Passaggio del membership token in Proem

Ciascuna singola entità è caratterizzata da un nome, espresso secondo lo standard URI; ogni entità può avere più di un nome ma ogni nome è univocamente associato ad una sola entità.
Esiste però un altro metodo per identificare le entità; queste posso essere indirettamente referenziate attraverso un profilo.
Un profilo in Proem non è altro che una struttura dati usata per descrivere entità. Ad esempio, il profilo per un Individual può contenere nome e indirizzo e-mail, mentre il profilo di un Peer può includere la lista dei Data Spaces che condivide o ai quali può avere accesso. In sostanza, i profili lavorano sullo stesso principio dei nomi, ma permettono di informare le entità connesse alla rete della presenza di determinati servizi anche senza conoscere direttamente chi li offre (anonimità).

Concettualmente, Proem può essere visto come un gestore di profili, i quali definiscono la semantica dei messaggi che i Peer connessi possono scambiarsi e i protocolli di livello utente concessi.

Proem implementa il concetto di profilo attraverso delle classi wrapper; ogni peerlet definisce un proprio set di profili che descrive le regole di comunicazione e di accesso alle risorse possedute dalle varie entità. Esistono due classi principali di profili:

- **A livello di protocolli supportati:** la classe più importante definisce il protocollo applicativo di comunicazione; se avviene un incontro fra peer e questi parlano lo stesso protocollo, allora può sussistere la comunicazione.

  ```
  public ProtocolType(java.lang.String _name)
  ```

- **A livello di messaggi supportati:** questo profilo definisce la tipologia di messaggi consentiti dallo sviluppatore e permette ai Peer di attuare tecniche di reazione agli eventi; infatti, ogni messaggio rappresenta un'informazione sul cambiamento del contesto di applicazione e conseguentemente secondo il paradigma ad eventi possono essere attuate politiche di gestione opportune.

  ```
  public MessageType(java.lang.String _name)
  ```

L'ambiente di sviluppo Proem è costituito da una collezione di tools, API e applicazioni runtime: esso si basa su due importanti nozioni:

Peerlet

Le Peerlet sono applicazioni peer to peer che seguono un paradigma di programmazione basato sugli eventi e costituiscono gli end points di comunicazione fra pari. Lo sviluppatore attraverso le Peerlet può gestire Data Spaces, Individuals e ogni singola entità dell'applicazione.

Class AbstractPeerlet

All implemented interfaces:

java.util.EventListener, Peerlet, ProemEventListener, java.lang.Runnable

Metodi principali della classe AbstractPeerlet	
abstract UserProfile	**createProfile**(ProfileAttribute[] pa) Permette di creare un profilo da un set di attributi passati come parametro
abstract void	**destroy**() Questo metodo viene invocato appena prima che il PeerletEngine rimuova la peerlet.
abstract java.lang.String	**getName**() Ritorna il nome della peerlet.
abstract ProtocolType[]	**getSupportedProtocols**() Fornisce la lista dei protocolli supportati dalla peerlet.
abstract void	**handleProemEvent**(ProemEvent event) Questo metodo viene chiamato dal Peerlet Engine quando viene notificato un evento di sistema.
abstract void	**init**() Il metodo inizializza il peerlet e lo mette in servizio.
void	**kill**() Metodo che arresta il peerlet.
void	**run**() Il metodo run contiene il lavoro che deve svolgere il peer.
void	**startup**() Metodo per lanciare il thread peerlet.
java.lang.String	**toString**() Restituisce una rappresentazione della peerlet sottoforma di stringa.

Le peerlet sono contenute nel Peerlet Engine, che è responsabile della loro istanziazione, esecuzione e terminazione.

L’engine indirizza in modo asincrono gli eventi alle Peerlet in relazione ai cambiamenti del contesto di ambiente o in reazione ai messaggi ricevuti da altri Peer (Fig. 30).

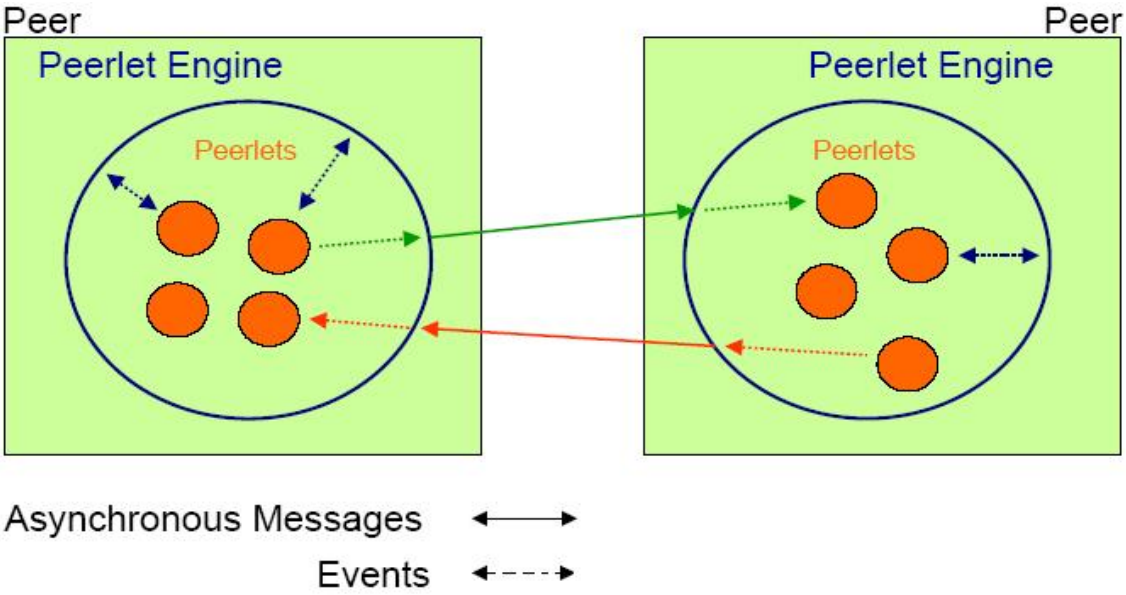


Fig. 10: Relazione tra Peerlet e Peerlet Engine in Proem

Proem Runtime System (PRS)

Il PRS consiste nell’implementazione di un sistema che permette ai peer di comunicare attraverso i protocolli di Proem (Fig. 31).
I componenti del PRS sono il Protocol Stack, che implementa i 4 protocolli built-in di Proem (descritti in seguito); il Peerlet Engine, che gestisce il ciclo di vita delle Peerlet. In più esistono servizi per la gestione delle Peerlet (naming, event logging, gestione delle strutture dati, etc), implementati attraverso componenti di sistema (Service API).

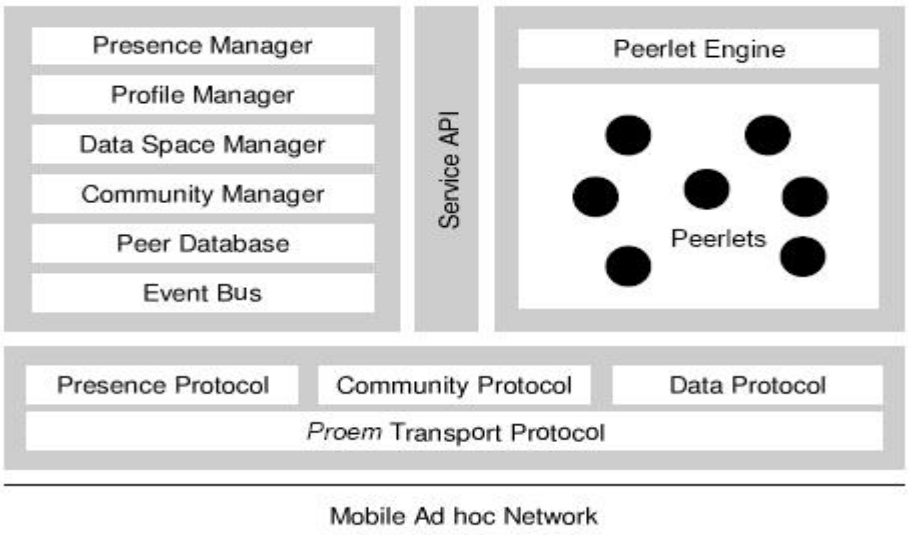


Fig. 31: Componenti del sistema Proem

La definizione e l'utilizzo di protocolli built-in da parte dei Peer garantisce l'interoperabilità delle applicazioni su differenti piattaforme hardware e software.

Protocolli built-in

- **Transport protocol:** è un protocollo asincrono e connectionless; i dati vengono scambiati da un peer all'altro in unità atomiche. Questo protocollo può utilizzare XML per codificare i dati e si basa su protocolli già esistenti come TCP/IP, UDP o HTTP.

- **Presence protocol:** contiene messaggi che permettono ai peer di annunciare la loro presenza e la disponibilità di nuove entità nella rete.

- **Data Protocol:** consente ai Peer di comunicare e sincronizzare lo scambio di dati attraverso messaggi.

- **Community protocol:** contiene messaggi per applicare, garantire e modificare l'appartenenza a una community.

In aggiunta a questi protocolli standard, gli sviluppatori possono definire un proprio protocollo di applicazione, in maniera tale da adattare ed estendere le funzionalità offerte dall'ambiente in risposta alle proprie esigenze.

Nell'applicazione di chat si deve creare ed inizializzare la peerlet creando un nuovo profilo (protocollo) di livello applicazione chiamato "easychat" e definendo i possibili messaggi supportati dallo stesso:

```
public class chatProem extends AbstractPeerlet{

// protocolli supportati dal peerlet definibili dal programmatore
    //protocoltype e' un wrapper per una stringa
    //che rappresenta il tipo di protocollo supportato
    //in questo caso easychat
    private ProtocolType  protocol=new ProtocolType("easychat");

//definizione dei tipi possibili di messaggi
    //uno solo,il messaggio da inviare alla chat
    private MessageType SEND_MESSAGE=new
MessageType("Send_message");

    //definizione del profilo utente
    //viene istanziato dal metodo createProfile
    //provvede il runtime a lanciarlo
    public UserProfile myProfile;

    //variabile di supporto
    private Peer peer;

    //implementa la lista dei peer connessi
    //molto utile perche' ha il metodo send che permette di
    //mandare un messaggio a tutti i peers del gruppo
    //all'inizio e' vuoto
    private PeerGroup peers=new PeerGroup();

…

public class chatProem extends AbstractPeerlet{

...
```

```
//inizializzo il peerlet e lo metto in servizio
    public void init() {
        //visualizzazione GUI
        initComponents();

        //richiamo informazioni sul peer...
        peer=Peer.getLocalPeer();
        //per generare il mio profilo
        //si usa l'if else per eliminare eventuali null pointer exception
        if(peer.supports(protocol))
            myProfile=peer.getProfile(protocol);
        else
            myProfile=new UserProfile("EasychatUser:" +peer.getURI());

       ...

    }
}
```

Si è già visto come una classe generale di eventi possa essere gestita mediante l'utilizzo di profili; esistono anche classi di eventi predefinite che consentono alle Peerlet di aggiornare i profili descriventi le entità Community: la classe EncounterEvent permette di notificare la presenza di nuovi peer all'interno della rete; ContextChangeEvent consente la notifica di cambiamenti di contesto (normalmente disconnessioni dalla community). Infine, quando viene ricevuto un messaggio mandato da un altro peer si istanzia un oggetto evento di tipo IncomingMessageEvent, che incapsula il messaggio.

L'applicazione deve reagire a due tipologie di evento: nuove connessioni di Peer e ricezione di messaggi: si termina la scrittura del metodo init sottoscrivendo la peerlet alla notifica di entrambe le classi di eventi.

```
public class chatProem extends AbstractPeerlet{
...
public void init() {
    //sottoscrizione all'evento ENCOUNTER_EVENT
        //per poter essere notificati dal runtime degli incontri
        Proem.getServices().EventBus_Subscribe(this,
             EventTypes.ENCOUNTER_EVENT);
        //sottoscrizione all'evento CONTEXT_CHANGE_EVENT
        //per poter essere notificati dal runtime del cambio di contesto
```

```
        Proem.getServices().EventBus_Subscribe(this,
            EventTypes.CONTEXT_CHANGE_EVENT);
    }
}
```

Si procede alla definizione del metodo handleProemEvent che consente di reagire alla notifica di tali eventi:

```
public class chatProem extends AbstractPeerlet{
    //metodo che risponde agli eventi
    //sono due gli eventi da intercettare:
    //1)incontro di un nuovo peer
    //2)messaggio in arrivo
    //entrambi gli eventi sono implementati dal PDK
    public void handleProemEvent(ProemEvent proemEvent) {

        //1)
        //aggiungiamo il peer al gruppo
        if(proemEvent instanceof EncounterEvent)
        {
            System.out.println("nuovo peer aggiunto al gruppo...");
            Encounter
encounter=((EncounterEvent)proemEvent).getEncounter();
            peer=encounter.getPeer();
            peers.add(peer);
        }
        //2)
        //gestione del nuovo messaggio arrivato
        if(proemEvent instanceof IncomingMessageEvent)
        {
            ProemMessage
message=((IncomingMessageEvent)proemEvent).getMessage();
            //se il messaggio e' rivolto a noi lo stampo a video
            if (message.getMessageType().equals(SEND_MESSAGE))
            {
                System.out.println("recupero messaggio");
                Message m=((Message)message.getBody());
                System.out.println("scrittura messaggio a video");
                jEditorPane2.setText(m.toString());
            }
        }
}
```

Conclusioni

Lo sviluppo di servizi collaborativi avanzati in scenari MANET solleva problemi complessi che richiedono l'adozione di nuove linee guida per il design e l'implementazione dei servizi. In questa tesi abbiamo mostrato i principali sistemi che costituiscono lo stato dell'arte della ricerca per lo sviluppo di applicazioni collaborative in scenari MANET. Al fine di confrontare i diversi approcci abbiamo progettato ed implementato un servizio di chat utilizzando tre diversi sistemi di supporto: Lime, PeerWare e Proem. Il progetto del servizio di chat ci ha consentito di effettuare un confronto fra i diversi sistemi studiati identificandone il modello di riferimento, le caratteristiche comuni e le peculiarità.

I primi risultati ottenuti promuovono ulteriori ricerche al fine di approfondire lo studio condotto. In particolare sarebbe interessante implementare altre applicazioni per arricchire il confronto fra i diversi sistemi.

Bibliografia

[1] Marco Conti
"Body, Personal, and Local Ad Hoc Wireless Networks"
2003

[2] T. G. Zimmerman
"Personal Area Networks: Near-field Intrabody Communication"
1996, IBM Systems Journal Vol. 35, No.3&4

[3] Ravi Prakash, Roberto Baldoni
"Architecture for Group Comunication in Mobile Systems"
1999

[4] Roberto Berardi, Roberto Baldoni
"Unicast Routing Techniques for Mobile Ad Hoc Networks"
2003

[5] Elizabeth M. Royer, Chai-Keong Toh
"A Review of Current Routing Protocols for Ad Hoc Mobile Wireless Networks"
1999, IEEE Personal Communications

[6] Cecilia Mascolo, Licia Capra, Wolfgang Emmerich
"Mobile Computing Middleware"
2002

[7] Cecilia Mascolo, Licia Capra, Wolfgang Emmerich
"Chapter 12: Principles of Mobile Computing Middleware "
2002

[8] Dejan S. Milojicic, Vana Kalogeraki, Rajan Lukose, Kiran Nagaraja, Jim Pruyne, Bruno Richard, Sami Rollins, Zhichen Xu
"Peer to Peer Computing"
2002

[9] Anind K. Dey, Gregory D. Abowd
"Towards a Better Understanding of Context and Context-Awareness"
2000

[10] Dan C. Marinescu, Craig Lee
"Process Coordination and Ubiquitous Computing"
2003, Cap. 2

[11] Gerd Kortuem, Jay Schneider, Dustin Preuitt, Thaddeus G. C. Thompson, Stephen Fickas, Zary Segall
"When Peer-to-Peer comes Face-to-Face: Collaborative Peer-to-Peer Computing in Mobile Ad-Hoc Networks"
2002
http://www.cs.uoregon.edu/research/werables/proem/

[12] Amy L. Murphy, Gianpietro Picco, Gruia-Catalin Roman
"Lime: A Middleware for Physical and Logical Mobility"
2000
http://www.lime.sourceforge.net

[13] Giampaolo Cugola, Giampietro Picco
"Peer to Peer for Collaborative Applications"
2002
http://www.peerware.sourceforge.net

[14] D. Gelernter
"Generative Communication in Linda"
1985