SARAH
Shop Assistant in Reti Ad Hoc

Marco Montali

Sommario

La grande diffusione di devices portatili e la crescita dell'interesse per le *Mobile Ad hoc NETworks* (MANET) hanno aperto nuovi orizzonti di comunicazione che prima non erano possibili. Si stanno delineando in quest'ambito scenari applicativi socialmente utili. In questo contesto si inserisce SARAH, un progetto per il coordinamento di un insieme di soggetti nel supportare un disabile durante i suoi acquisti.

La relazione descrive l'architettura dell'applicazione e i principali protocolli di coordinamento su di essa realizzati, evidenziandone le proprietà richieste in considerazione della semantica e delle problematiche della rete sottostante.

1 Scenario applicativo

1.1 Perchè una rete ad hoc

La grande diffusione di devices portatili e la crescita dell'interesse per le *Mobile Ad hoc NETworks* (MANET) hanno aperto nuovi orizzonti di comunicazione che prima non erano possibili. La collaborazione in reti manet pone all'attenzione del progettista una serie di problematiche riguardanti l'interazione e la gestione del concetto di gruppo. La vecchia idea di poter effettuare comunicazioni tra membri del gruppo che mantenessero proprietà di reliability, atomicity e synchronicity deve essere abbandonata per spostarsi verso sistemi maggiormente orientati a dinamiche best effort.

Non ci devono sorprendere quindi frequenti perdite di messaggi, connessioni e disconnessioni continue e problemi ulteriori (citiamo la questione dell'hidden terminal). Nonostante le suddette problematiche le reti ah hoc forniscono caratteristiche non ottenibili con reti "classiche", in quanto ad esempio non richiedono alcun tipo di server centrale, nè cablaggi di cavi o installazione di access point.

Tutto questo, per un utente non di alto profilo e senza particolari esigenze di sicurezza e affidabilità può essere visto come un grosso pregio, soprattutto se i meccanismi di creazione e gestione della manet vengono realizzati in modo trasparente.

1.2 SARAH

In questo contesto si inserisce il nostro progetto: SARAH, *Shop Assistant in Reti Ad Hoc*. Esso permette a persone anziane o disabili di richiedere un aiuto a soggetti al momento nell'ambiente ed a coordinare sia fruitori che fornitori dell'aiuto. Il concetto fondamentale che si trova dietro all'idea di coordinamento è quello di una lista condivisa e sincronizzata. Ovviamente il principio di sincronismo non è citato nel modo classico, ma inteso di tipo più lasco, nel quale non sempre è tutto perfettamente consistente.

Il caso d'uso principale del sistema consiste nell'arrivo di un disabile dotato di un dispositivo portatile all'interno di un ambiente (d'ora in poi supporremo che sia un supermercato) al cui interno è (o sarà) presente una rete ad hoc. Essendo affaticato o non fisicamente abile, e dovendo fare una spesa consistente, della quale abbiamo una rappresentazione all'interno della nostra applicazione, e con prodotti aventi locazioni nel negozio molto distanti tra loro, decide di richiedere aiuto. È predisposto quindi l'apposito bottone per mezzo del quale in modo trasparente all'utente viene creato un nuovo gruppo (individuato da un *gid*) e viene assegnato all'anziano un identificatore (il *pid* relativo). La richiesta di supporto (un particolare messaggio contente il *gid*) verrà inoltrata a tutti gli avventori dotati di dispositivi portatili connessi alla rete ad hoc sui quali è stata avviato SARAH e sui quali è stata espressa la volontà di essere un benefattore. Se i benefattori raggiunti dalla richiesta accettano di aderire al gruppo di aiuto, verrà loro inviata la lista relativa all'anziano che si sia deciso di aiutare. Da questo momento in poi quando qualcuno si troverà a poter prendere un prodotto per il disabile lo segnalerà checkando la corrispettiva entry nella propria lista locale. All'atto di questa azione dovrà essere riportato agli altri collaboratori ed al diretto interessato il cambiamento dello stato dell'elemento della lista (passando da "da prendere" a "preso").

Ovviamente non si vuole escludere la possibilità che un utente "benefattore" possa collaborare con più utenti in difficoltà. In questo caso bisognerà sapere oltre al "cosa" prendere, anche il "per chi" e a questo punto entrerà in gioco una feature di cui tratteremo in seguito: la *Presence Awareness*.

1.3 Problematiche di rete e scenario secondario

Le problematiche relative alla comunicazione in manet sono numerose e complesse. Basti citare il problema del routing in reti con una dinamicità molto elevata, nelle quali spesso avvengono partizioni e nodi cadono e ritornano attivi in breve tempo. Per poter lavorare ad un livello più elevato risulta molto utile l'utilizzo di un middleware che fornisca una serie di funzionalità sulle quali ci si possa appoggiare per la realizzazione dei propri obiettivi.

Considerando quanto detto bisogna prevedere una serie di meccanismi che ottengano un gestione quanto migliore possibile. Questo significa prevedere

dei meccanismi di update quando due entità si ritrovano nella stessa rete ma i loro dati non sono congruenti. Basandosi sul meccanismo della Presence Awareness è possibile determinare chi collabora con ciascuno, cosa che rende possibile predisporre dei comportamenti trasparenti all'utente che rendano le liste "sincronizzate". Con questo termine non intendiamo la completa congruenza, ma un tipo di consistenza che privilegi comunque la ridondanza anche a discapito dell'efficienza. Ad esempio prediligiamo una situazione in cui due benefattori prendano lo stesso item piuttosto del caso nel quale nessuno dei due lo faccia. Sicuramente non è una soluzione ottima, ma proprio per le problematiche intrinseche alle manet, a meno di costi proibitivi, qualsiasi tipo di interazione sarà best effort.

1.4 Scelta del middleware: AGAPE

Sono disponibili numerosi sistemi che lavorano su reti ad hoc, ciascuno dei quali è caratterizzato da alcune features peculiari. È responsabilità del progettista scegliere quello più adatto e che meglio fitta con i propri bisogni. Tra le varie opportunità disponibili la nostra scelta è caduta su AGAPE (*Allocation and Group Aware Pervasive Environment*), un sistema sviluppato presso il DEIS, che sebbene sia ancora in versione alfa fornisce un insieme di servizi che sono stati fondamentali nella realizzazione di SARAH.

Le motivazioni che ci hanno spinto ad utilizzare AGAPE sono state molteplici. In primis esso fornisce un meccanismo di routing basato sul gossiping (con probabilità a scelta) e gestisce il concetto di gruppo con primitive già implementate, del tipo *join/leave*. Quindi abbiamo un mezzo per creare gruppi e permettere a coloro che appartengono alla manet di effettuare operazioni di aggregazione o abbandono del gruppo.

La caratteristica per la quale, però, il middleware è unico è l'idea di vista. La vista non è altro che una tabella riassuntiva di coloro che sono nella mia prossimità, inviata a tutti i coloro che sono stati riconosciuti come mebri della rete con attiva una istanza di AGAPE. Il processo di creazione e distribuzione delle liste è gestita da entità centrali definite come LME (*Locality Manager Entity*), che consistono semplicemente in nodi della rete ad hoc con capacità computazionali maggiori. Lavorando sulla vista si può accedere ad informazioni riguardanti i *gid* dei vari gruppi, i *pid* e gli indirizzi dei vari membri e oltre a ciò ad un insieme di dati definito come profilo. Il profilo consiste in una collezione di dati riguardante un'entità ed è semplicemente caricabile da un file xml.

Come si può facilmente notare tutto questo scambio di dati causa un invio di un congruo numero di messaggi di dimensione anche rilevante, ma questo è un prezzo che vale la pena pagare per ottenere tutte le informazioni disponibili con le viste.

2 Architettura

Lo scopo di questo capitolo è quello di presentare i principi architetturali di SARAH analizzando la logica dei vari livelli che compongono l'applicazione.

2.1 Struttura dell'applicazione

Si noti innanzitutto che SARAH è un'applicazione *4-layer*, come mostrato in figura 1.

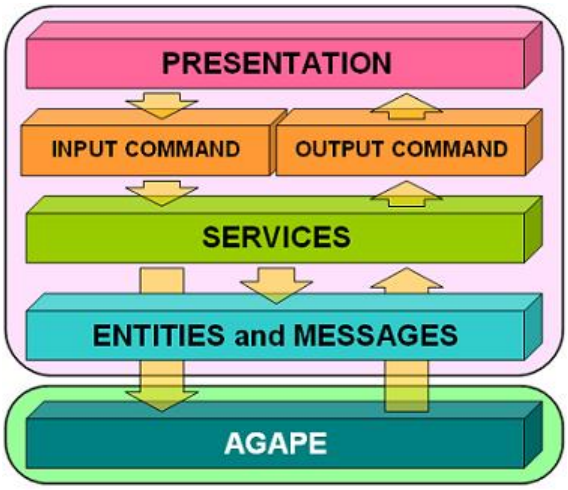

Figura 1: schema generale dell'architettura di SARAH

L'architettura dell'applicazione è stata pensata per renderla il più possibile flessibile e riutilizzabile.

In quest'ottica, il layer dei servizi è costituito da un insieme di servizi indipendenti che astraggono ulteriormente il middleware realizzando le funzionalità fondamentali dell'applicazione; le azioni che l'utente esegue su tale layer, nonchè le azioni che il layer esegue verso "l'esterno", sono modellate come un insieme di comandi che separano completamente la logica dell'applicazione dalla sua presentazione.

Di seguito analizzeremo nel dettaglio i livelli che costituiscono l'applicazione.

2.2 Presentation layer

Il layer di presentazione gestisce tutte le interazioni con l'utente. In base alle specifiche, come già accennato precedentemente, ogni utente è, in generale, coinvolto in differenti interazioni a compartimenti stagni, ognuna relativa a un gruppo di lavoro. L'utente fa parte di tanti gruppi quante sono le persone che ha deciso di aiutare, più eventualmente un gruppo in cui è lui

ad assumere il ruolo di persona che chiede aiuto. Ogni gruppo lavora, di conseguenza, su una lista della spesa dedicata.

Abbiamo deciso di realizzare l'interfaccia utente con una semplice GUI, in cui l'attività di ogni gruppo è rappresentata da un TabPanel identificato dal nominativo della persona che ha chiesto aiuto. Ogni pannello mostra la lista della spesa condivisa (la quale indica, oltre ai dati di ogni elemento, se è attualmente preso e da chi) nonchè tutti i nominativi delle persone facenti parte del gruppo e attualmente "visibili". Ovviamente, la lista viene compilata in una fase antecedente alla richiesta di aiuto.

Preso atto della tipologia di target che potrebbe usufruire di tale prodotto, abbiamo pensato di integrare l'area di lavoro relativa alla propria lista (ovvero l'area in cui si gioca il ruolo di chi chiede aiuto, ruolo assunto, di norma, da un disabile) con la tecnologia *JavaSpeech* (implementazione `http://sourceforge.net/projects/freetts/`), per dare anche informazioni di tipo uditivo all'utente su come sta procedendo la spesa di gruppo. Si presuppone, quindi, uno scenario in cui un non vedente si sia fatto aiutare nella fase di inserimento della propria lista.

2.3 Entities and Messages layer

Questo layer contiene le entità fondamentali, nonchè la gerarchia dei messaggi scambiati (vedi figura 2).

SarahEntity è la generica entità manipolata e scambiata; nel nostro caso, è estesa da un'unica classe che rappresenta il singolo elemento della spesa (le liste della spesa, istanze di *ShopList*, sono semplicemente vettori di tali elementi).

L'elemento base della gerarchia dei messaggi è $SarahMessage$, che rappresenta un messaggio scambiato tra i dispositivi nell'ambito di SARH. Ogni messaggio è caratterizzato da un contenuto (di tipo *SarahEntity*) e dal *pid* del mittente. Di seguito elenchiamo i vari messaggi dando una breve descrizione del loro significato:

- **HelpMessage** messaggio utilizzato dal disabile per inviare una richiesta di aiuto a tutti i benefattori presenti nella rete ad-hoc (si noti che un disabile può assumere anche il ruolo di benefattore);
- **AckMessage** messaggio di risposta di un benefattore che accetta la richiesta di aiuto;
- **SarahSingleMessage** messaggio inviato per comunicare l'aggiornamento di un singolo elemento della lista;
- **RequestMessage** messaggio utilizzato da un benefattore per chiedere al disabile il "permesso" di modificare un elemento della lista (si veda il capitolo sui protocolli di coordinamento);

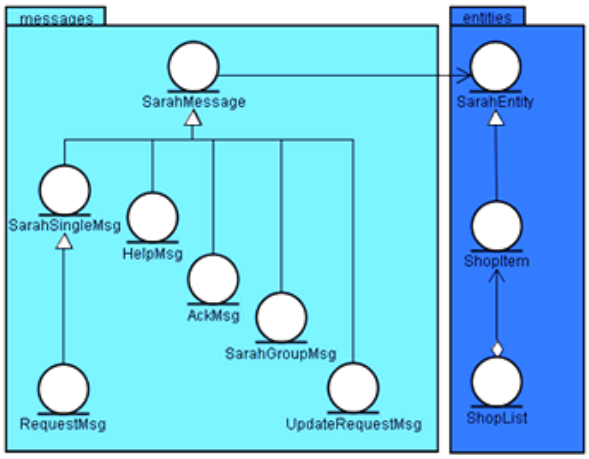

Figura 2: diagramma UML relativo al livello di entità e messaggi

- **SarahGroupMessage** messaggio utilizzato per l'invio di un insieme di elementi (ovvero, di una *ShopList*);
- **UpdateRequestMessage** messaggio utilizzato dal benefattore nella fase di riconciliazione delle liste, conseguente a un partizionamento del gruppo (problematica non trattata in questa relazione).

2.4 Processori dei messaggi

Per separare la logica operazionale relativa alle azioni da intraprendere in conseguenza della ricezione di una specifica tipologia di messaggio dai messaggi stessi, abbiamo inserito un terzo package ($ProcessMsg$), contenente un insieme di classi la cui responsabilità è l'interpretazione di essi. I vari servizi, alla ricezione di un messaggio, richiedono a uno di tali processori di interpretarlo ed eseguire di conseguenza una serie di azioni (nell'ambito dei protocolli di coordinamento). Il legame tra i messaggi e i vari processori è simile a quello utilizzato per il pattern *Visitor* (meccanismo di double-dispatch).

2.5 Servizi

I servizi costituiscono, insieme ai processori dei messaggi, il cuore di SARAH; vanno infatti a formare la parte dell'applicazione che si interfaccia con i servizi di AGAPE per usufruire delle primitive di comunicazione e dell'astrazione del concetto di gruppo. La congiunzione di servizi e processori realizza i protocolli di coordinamento.

Abbiamo tre servizi legati ai ruoli di SARAH: un servizio per il disabile ($HelpService$) e una coppia di servizi per il benefattore ($CharityService$ e $ServantService$). I principali compiti di questi servizi possono essere riassunti in tre punti:

1. ricevere i messaggi e invocare il corrispondente processore
2. gestire la spedizione dei messaggi
3. mostrare al livello dei comandi un'ulteriore astrazione della rete (ovvero solo i concetti di cui i layer superiori necessitano).

Andiamo quindi a specificare più nel dettaglio le responsabilità di tali servizi; per avere un quadro più completo, si faccia riferimento al capitolo sui protocolli di coordinamento.

- **HelpService** L'*HelpService* gestisce le interazioni lato disabile. É quindi il componente che inizialmente si prende carico di creare un gruppo di aiuto per il disabile (invitando chi è presente in rete ad unirsi a tale gruppo quando il disabile lo richiede). Ad inizializzazione avvenuta, i compiti dell'*HelpService* sono quelli di comunicare ai membri del gruppo gli aggiornamenti che avvengono sulla lista condivisa.
- **CharityService** Il *CharityService* ha il compito di gestire la fase che va dalla ricezione di una richiesta di aiuto ad una sua eventuale accettazione, permettendo al benefattore di ottenere le informazioni utili per lavorare. Lascia poi il posto al *ServantService*.
- **ServantService**. Il *ServantService* gestisce le interazioni del benefattore "a regime" (ovvero quando è già inserito in un gruppo di aiuto). I suoi compiti sono quindi quelli di gestire l'invio e la ricezione di messaggi relativi a modifiche sulla lista della spesa.

L'ambito applicativo di SARAH necessita anche delle informazioni relative alla "presenza" dei vari partecipanti. La possibilità di capire chi è visibile attualmente nel gruppo, resa disponibile da AGAPE tramite la distribuzione delle viste, risulta estremamente utile sia a livello applicativo (è importante sapere "chi ha fatto cosa", ovvero chi deve prendere un certo item) che da un punto di vista protocollare, per poter cambiare protocollo di coordinamento a seconda del contesto.

A tale scopo, è presente un quarto servizio, il *PresenceService*. Ogni *PresenceService* è un componente attivo/passivo che monitora la visibilità degli altri utenti in un certo gruppo, interrogando periodicamente il middleware e notificando i livelli superiori ogni qual volta la lista degli utenti visibili cambia (ovvero quando si perde o riottiene visibilità di qualcuno).

Per coordinare l'insieme dei servizi di presenza, è presente un *ServicePooler* (entità singleton), che restituisce, dato il *group identifier* o il *pid* del disabile, il corrispondente *PresenceService*.

2.6 Comandi

Il layer dei comandi ha il compito di separare completamente la logica dell'applicazione dalla sua interfaccia utente (si faccia riferimento allo schema in figura 3). Si suddivide in due package:

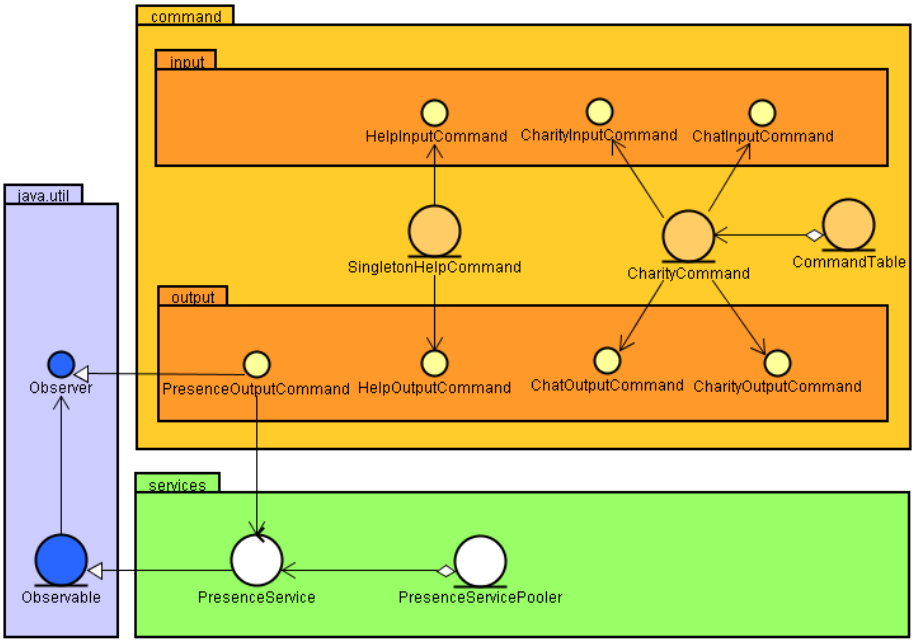

Figura 3: diagramma UML relativo al livello dei comandi

- **input command** Comandi che l'interfaccia utente utilizza per scatenare delle azioni nei layer sottostanti (ad esempio: quando il disabile decide di chiedere aiuto, l'interfaccia utente consegna la richiesta all'*HelpInputCommand*, che richiama i servizi sottostanti per inviare effettivamente la richiesta);
- **output command** Comandi che i servizi (o anche i processori di messaggi) utilizzano per consegnare delle informazioni all'utente (ad esempio, quando il *CharityService* riceve una richiesta di aiuto, chiede all'utente, tramite l'opportuno comando, se vuole accettare o meno la richiesta).

Possiamo quindi mappare i vari servizi in una coppia di comandi (uno di input e uno di output) che appunto fanno da intermediari tra servizi e interfaccia utente. Estendendo tali comandi, abbiamo realizzato dei comandi concreti che operano sull'interfaccia grafica (ad esempio, quando un benefattore accetta una richiesta di aiuto, viene chiamato un comando che aggiunge all'interfaccia un *TabPanel* relativo al gruppo di aiuto corrispondente).

Un'eccezione rispetto a questi meccanismi è costituita dal comando di presenza, che notifica i livelli superiori solo quando l'insieme dei componenti visibili del gruppo cambia.

3 Protocolli di coordinamento

In questo capitolo evidenziamo esplicitamente i protocolli di interazione impiegati in SARAH per

1. coordinare la fase iniziale (richiesta di aiuto e join al gruppo da parte dei benefattori);

2. coordinare i partecipanti di un gruppo al fine di condividere le modifiche apportate da ognuno alla lista della spesa condivisa

Inizieremo riprendendo velocemente quali componenti dell'applicazione realizzano tali protocolli, per poi passare ad analizzare la logica di essi.

3.1 architettura e protocolli di coordinamento

Nel capitolo sull'architettura del sistema abbiamo evidenziato alcune entità che concorrono alla realizzazione dei protocolli di coordinamento. Al centro abbiamo il layer dei servizi, che gestiscono la comunicazione, delegando le azioni da fare alla ricezione di un messaggio ai vari processori di messaggi. Abbiamo poi, appunto, la gerarchia dei messaggi: la tipologia di un messaggio permette di identificare il protocollo nell'ambito del quale è stato spedito. Abbiamo infine il layer dei comandi, chiamato in causa per tutti i protocolli che prendono il via da specifiche azioni dell'utente o che, comunque, lo chiamano in causa (il protocollo di richiesta d'aiuto, ad esempio, parte su decisione del disabile).

Si noti che non tutti i protocolli richiedono un intervento esplicito ai livelli di astrazione più alti; un caso di questo tipo è il protocollo per la riconciliazione delle liste, non trattato in questa relazione.

3.2 proprietà fondamentali dei protocolli

Andiamo ora ad elencare due proprietà fondamentali riguardanti i protocolli di coordinamento "centrali", ovvero quelli impiegati quando il gruppo agisce sulla lista della spesa.

Il nostro obiettivo ideale è quello di fare in modo che *istante per istante, tutti i componenti del gruppo vedano la stessa lista*. Ovviamente, tale obiettivo è praticamente irrealizzabile, soprattutto ricordandoci che SARAH lavora in un ambiente ad-hoc. In un ambiente di questo tipo, che è tipicamente best-effort, possono verificarsi problemi quali ad esempio la perdita di

messaggi o il partizionamento del gruppo (fenomeno per cui alcuni componenti perdono visibilità di altri, con la conseguenza che il gruppo originario si frammenta).

Fenomeni di continue entrate/uscite dalla rete (come quando un dispositivo è ai limiti del raggio di copertura), vengono fortunatamente gestite dal middleware, che si preoccupa di eliminare un utente dalla vista solo dopo che manca da un certo tempo (tale utente, ovviamente, perderà comunque dei messaggi), ossia dopo che per due volte consecutive il suo *Proximity Service* non invia all'*LME* del gruppo il *Beacon* (inteso come testimonianza di presenza).

Dobbiamo in ogni caso considerare l'impatto di queste problematiche sulla nostra applicazione, cercando di capire in che modo possiamo rilassare il nostro obiettivo ideale. Preso atto del fatto che le liste dei vari componenti del gruppo possono essere inconsistenti fra loro, diciamo che:

Proprietà 1 *Le liste dei partecipanti possono divergere in positivo, ovvero può capitare che più utenti prendano lo stesso elemento.*

Tale proprietà va comunque "limitata"; non possiamo ammettere la situazione al limite in cui tutti i partecipanti prendono tutti gli elementi. Il nostro obiettivo diventa quindi quello di *minimizzare il numero di queste inconsistenze*!

A tal proposito dobbiamo individuare, tra i partecipanti del gruppo, un componente che abbia più potere decisionale degli altri. Per quanto possibile, infatti, vogliamo che tale partecipante, nel caso in cui più persone abbiano lo stesso elemento checkato, ne scelga uno. Anche in questo caso, considerando lo scopo della nostra applicazione, possiamo dire che:

Proprietà 2 *Se più utenti entrano in collisione relativamente ad un elemento della lista, nel sceglierne uno non ci sono mai criteri di preferenza.*

Tale proprietà evidenzia che, in SARAH, tutti i componenti di un gruppo sono allo stesso livello: ci importa che un item "conteso" venga preso da uno dei contendenti, ma non abbiamo criteri per capire chi fra i contendenti sia il migliore.

Dobbiamo quindi scegliere un componente del gruppo che si accolli queso onere decisionale, ma il middleware non fornisce meccanismi di elezione. Possiamo però, senza alcuna forzatura a livello logico, decidere a priori che tale ruolo sia rivestito dal disabile: è infatti l'unico componente "irrinunciabile" del gruppo (se dovesse definitivamente mancare, non avrebbe più senso proseguire).

Di seguito faremo l'assunzione forte che il disabile sia sempre comunque visibile a tutti i benefattori facenti parte del gruppo.

3.3 handshake iniziale

La situazione iniziale del sistema è quella in cui nel supermercato è presente un certo numero di benefattori attivi.

Supponiamo ora che arrivi un disabile con una lista della spesa inserita nell'applicativo. Quando il disabile chiede aiuto, vogliamo che tutti i benefattori presenti in rete vengano contattati, al fine di chiedere loro se sono disponibili ad aiutarlo (si ricordi che un benefattore può contemporaneamente far parte di più gruppi di lavoro, e quindi deve avere la possibilità di rifiutare un'ulteriore richiesta d'aiuto).

Il disabile manda quindi un messaggio di tipo $HelpMessage$ in broadcast. Alla ricezione di tale messaggio, l'applicazione lato benefattore chiede esplicitamente se egli vuole accettare la richiesta. In tal caso, ha bisogno di ricevere anche la lista su cui lavorare. Si instaura quindi un piccolo protocollo di handshake in cui ogni benefattore che accetta esegue un $join$ sul gruppo (precedentemente creato dal disabile) e invia al disabile un acknowledgement. Ad ogni messaggio $AckMessage$ il disabile risponde con un unicast contenente la lista della spesa.

La figura 4 mostra il protocollo di inizializzazione. Alla sua conclusione, tutti i benefattori che hanno accettato la richiesta appartengono al gruppo di lavoro e hanno ottenuto la lista; si passa quindi ai veri e propri protocolli di coordinamento.

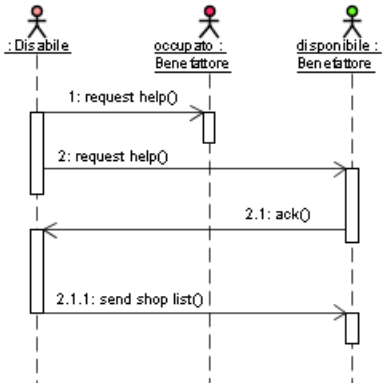

Figura 4: protocollo di coordinamento iniziale

3.4 condivisione delle modifiche

Andiamo ora a ragionare sul protocollo che viene utilizzato dai componenti del gruppo di lavoro per comunicare le modifiche fatte sulla lista agli altri (e tenere quindi le varie liste congruenti). Supponiamo che tutti i componenti del gruppo siano effettivamente on-line e visibili l'un l'altro.

Lo scopo del protocollo è il seguente: quando un utente spunta un elemento della lista, tutti gli altri componenti del gruppo devono accorgersi

che l'elemento è stato spuntato, e da chi. Abbiamo precedentemente spiegato che il disabile ha, all'interno del gruppo, una posizione di predominanza decisionale, ovvero è il suo applicativo, quando possibile, a dare il "lasciapassare" su una qualsiasi modifica effettuata sulla lista. Isoliamo quindi il caso in cui a modificare lo stato di un elemento della lista sia proprio il disabile, poichè in tal caso il protocollo deve semplicemente spedire a tutti un messaggio che avverta della modifica. La figura 5 mostra appunto questa situazione.

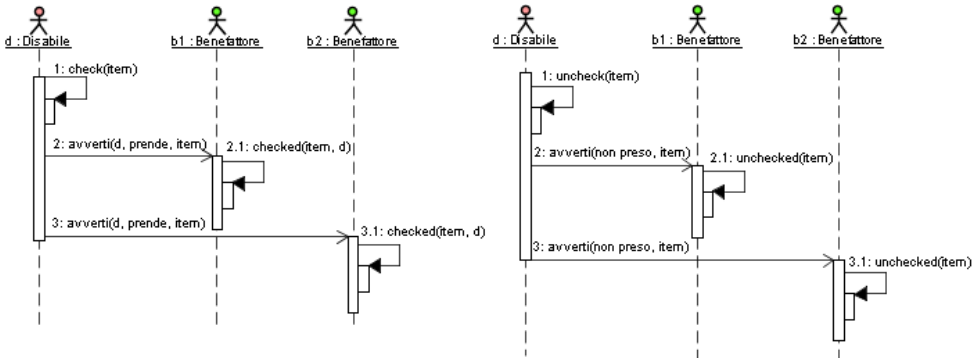

Figura 5: protocollo di aggiornamento nel caso di modifica del disabile

Una cosa importante da mettere in evidenza è che quando un utente spunta un elemento (e il disabile dà il suo lasciapassare) solo lui può (ragionevolmente) avere la possibilità di rinunciare alla scelta decheckandolo. Di conseguenza, quando un item è assegnato ad un certo utente, per gli altri diventa una entry di sola lettura.

Supponiamo ora che a spuntare un elemento sia un benefattore. In questo caso il protocollo impone che, al check di un elemento, venga mandata una richiesta di conferma dall'applicazione del disabile, e sia il disabile, nel caso venga accettata la modifica, a rimandare l'aggiornamento a tutti.

In questo modo si risolvono facilmente problematiche relative a richieste "contemporanee" di modifica di uno stesso elemento: per la *Proprietà 2*, viene semplicemente accettata la richiesta che arriva prima e scartata la seconda (dopodichè viene rispedito a tutti un messaggio contentente l'elemento con l'indicazione di chi è effettivamente il benefattore che lo deve prendere, in modo che tutti i componenti del gruppo riallineino correttamente la propria lista). La figura 6 chiarifica tale protocollo.

Riprendiamo ora in considerazione il medesimo protocollo, ma in uno scenario in cui $b1$ non riceve alcun messaggio di risposta dal disabile (figura 7). $b1$ non sa quindi se il suo messaggio non è arrivato, se è il messaggio di risposta ad essere stato perso, o se si trova nella fase di transizione che porta ad un partizionamento del gruppo. Cautelativamente, seguendo quanto detto nella *Proprietà 1*, $b1$ decide comunque di checkare l'elemento dopo un *timeout* (parametro di progetto).

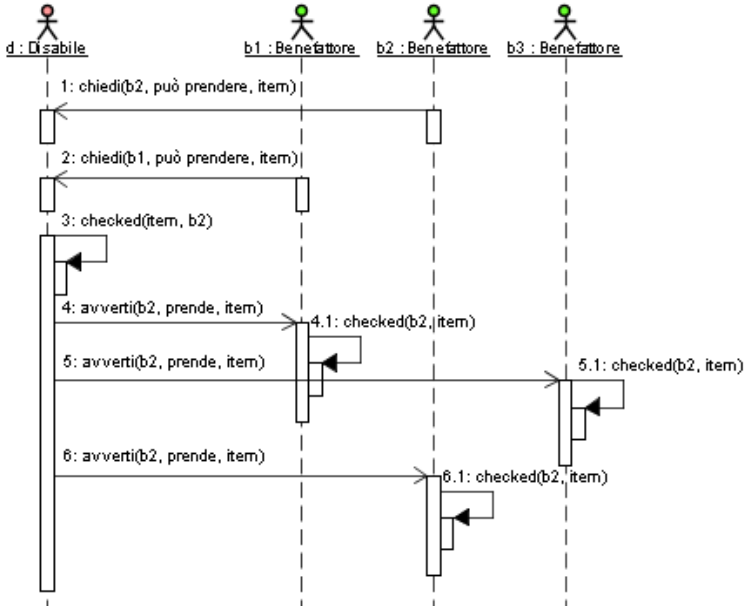

Figura 6: aggiornamento in presenza di modifiche "contemporanee"

Il quadro definitivo è quindi quello di un insieme di protocolli di coordinamento abbastanza snelli (come è consigliato in un ambito best-effort come quello delle reti ad-hoc) ma ragionevolmente robusti; ciò anche in virtù della possibilità di considerare il disabile come elemento parzialmente accentratore dell'attività del gruppo, elemento che ha sempre l'ultima parola sull'aggiornamento di un elemento della lista. Ponendoci nel caso fortunato in cui ad ostacolare l'attività del gruppo sia semplicemente la perdita di messaggi (ovvero tralasciando fenomeni più problematici come quello del partizionamento del gruppo) la verifica della *Proprietà 1* è garantita introducendo semplicemente un timeout, che induca ad un'azione di recovery strettamente locale.

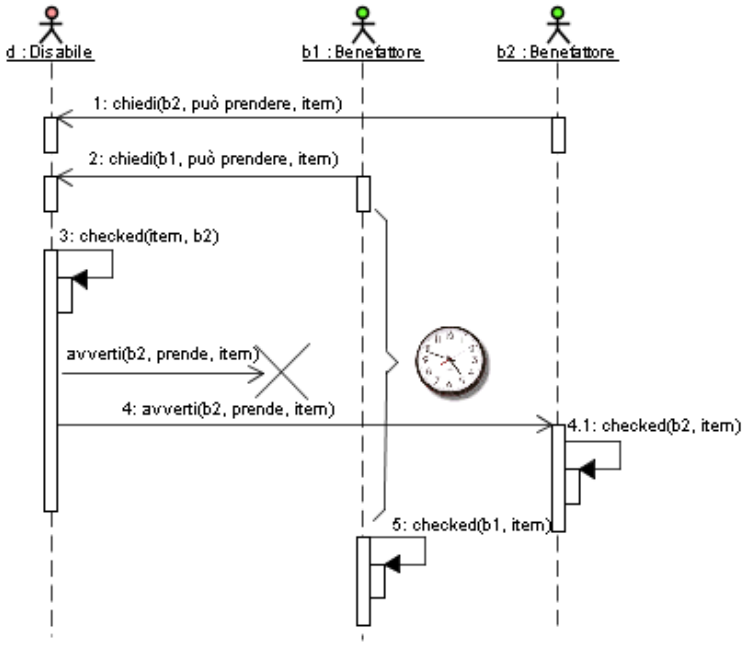

Figura 7: aggiornamento con perdita di messaggi

Indice