Alma Mater Studiorum · Università di Bologna

facoltà di scienze matematiche, fisiche e naturali

Corso di Laurea Specialistica in Informatica

Architetture per Strutture Dati Distribuite

Tesi di Laurea in Sistemi Distribuiti

Relatore:
Chiar.mo Prof.
Alessandro Amoroso

Presentata da:
Enrico D'Angelo

Sessione III
Anno Accademico 2009/2010

Indice

1 Introduzione **1**

1.1 Motivazione 1

1.2 Treeds 3

1.3 Treeds e Telex 4

1.4 Struttura della tesi 4

2 Replicazione ottimistica **5**

2.1 Definizione di replicazione ottimistica 5

2.2 Componenti della replicazione ottimistica 7

2.2.1 Oggetti, repliche, siti e operazioni 7

2.2.2 Propagazione delle operazioni 8

2.2.3 Esecuzione provvisoria e scheduling 9

2.2.4 Individuazione e risoluzione dei conflitti 9

2.2.5 Consenso 10

2.3 Scelte di design 10

2.3.1 Numero di scrittori 11

2.3.2 Definizione delle operazioni 12

2.3.3 Scheduling 13

2.3.4 Gestione dei conflitti 13

2.3.5 Strategie di propagazione 15

2.3.6 Garanzie di coerenza 16

2.4 Controllo della concorrenza 16

2.4.1 Scheduling 17

2.4.2 Rilevamento dei conflitti 19
2.4.3 Risoluzione dei conflitti 20
2.4.4 Protocollo di consenso 21
2.5 Sistemi state-transfer 21
2.5.1 Thomas' write rule 22
2.5.2 Algoritmo dei due timestamp 23
2.5.3 Tombstones selettive 23
2.6 Propagazione delle operazioni 24
2.6.1 Propagazione delle operazioni utilizzando i vector clock 24
2.6.2 Propagazione efficiente delle operazioni in sistemi state-transfer 25
2.6.3 Sistemi push 26
2.7 Vincoli di coerenza 27
2.7.1 Garanzie di sessione 27
2.7.2 Vincoli quantitativi 28

3 Stato dell'arte **31**
3.1 Bayou 32
3.2 TACT 36
3.3 IceCube 39
3.4 Confronto 42

4 Telex **47**
4.1 Introduzione a Telex 47
4.2 Componenti fondamentali di Telex 48
4.2.1 Vincoli 49
4.2.2 Conflitti tra azioni 51
4.2.3 Grafo delle azioni e vincoli 52
4.2.4 Protocollo di consenso 53
4.2.5 Viste 54
4.3 L'architettura di Telex 55
4.3.1 Interazioni 56

4.3.2 Scheduler 56
4.3.3 Replica reconciler 58
4.4 Modello di esecuzione di Telex 59
4.5 Costruire un'applicazione utilizzando Telex 61
4.5.1 Azioni e vincoli di sequenzialità 62
4.5.2 Concorrenza e vincoli 62
4.5.3 Vincoli e design dell'applicazione 63

5 Treeds **65**
5.1 Specifiche della struttura dati 66
5.1.1 Operazioni 66
5.2 Invarianti di Treeds 67
5.2.1 Identificativi univoci 68
5.2.2 Semantica di create 69
5.2.3 Semantica di remove 70
5.2.4 Dipendenza causale 70
5.2.5 Precondizioni 72
5.3 Conflitti tra operazioni concorrenti 74

6 Conclusioni **83**
6.1 Esperienza con Telex 83
6.1.1 Definizione di un "processo" per lo sviluppo 85
6.2 Risultati sperimentali 87
6.3 Sviluppi futuri 91

Elenco delle figure

4.1 Architettura di Telex 55
4.2 Modello di esecuzione di Telex 59

5.1 Due azioni *split* concorrenti. (Stessa azione) 79
5.2 Due azioni *move-under* concorrenti. 80
5.3 Azione *move-under* concorrente ad un'azione *split* 81

6.1 Tempo di stabilizzazione delle operazioni (ms). 92
6.2 Azioni stabili per secondo. 93

Elenco delle tabelle

2.1 Scelte di design per un sistema di replicazione ottimistica. . . 11

4.1 Vincoli forniti da Telex. 49

5.1 Vincoli tra operazioni concorrenti (pt. 1) 76
5.2 Vincoli tra operazioni concorrenti (pt. 2) 77

6.1 Risultati sperimentali: Scheduling semantico. 88
6.2 Risultati sperimentali: Scheduling con consenso. 89

Capitolo 1

Introduzione

Questa tesi descrive le tecniche di replicazione che vanno sotto il nome di *replicazione ottimistica* [37], e descrive l'implementazione di una struttura dati ad albero chiamata "Treeds". Treeds è stata implementata utilizzando le funzionalità fornite da Telex [2, 3], un toolkit che supporta lo sviluppo di applicazioni collaborative e nomadiche tramite, appunto, l'utilizzo di tecniche di replicazione ottimistica.

1.1 Motivazione

La replicazione dei dati è una tecnica chiave nei sistemi distribuiti perché permette di raggiungere alti livelli di disponibilità dei dati e di performance. La disponibilità dei dati viene aumentata permettendo l'accesso ai dati anche quando alcune repliche non sono disponibili. Le performance del sistema vengono aumentate riducendo la latenza nell'accesso ai dati permettendo all'utente di accedere a repliche che sono geograficamente a lui più vicine, e tramite un aumento della capacità di trasmissione del sistema permettendo a diversi siti di servire i dati simultaneamente.

Le tecniche di replicazione tradizionali, chiamate pessimistiche proprio in contrapposizione alle tecniche ottimistiche, forniscono un modello di coerenza *single-copy* [4] che dà all'utente l'illusione di interagire con una singola copia

dei dati altamente disponibile. Per mantenere questo livello di consistenza l'accesso alle repliche viene bloccato finché non è provato che siano tutte aggiornate.

Per esempio, nell'algoritmo di *primary-copy* [33] dopo ogni operazione di modifica la replica primaria aggiorna in maniera sincrona tutte le repliche secondarie. Se la replica primaria va in crash, le repliche secondarie si accordano per eleggere una nuova replica primaria. Questo tipo di tecnica funziona bene in una rete locale, in cui la latenza della rete è bassa e i guasti sono poco comuni. Dato il continuo progresso delle tecnologie legate ad Internet, però, si è sempre più tentati ad utilizzare tali tecniche anche in reti geografiche nelle quali non ci si può aspettare lo stesso livello di performance e di disponibilità dei dati.

I motivi chiave che fanno pensare di non riuscire ad ottenere gli stessi livelli di performance e disponibilità dei dati sono tre. Primo, Internet rimane un mezzo di comunicazione lento e inaffidabile; la latenza nelle comunicazioni e la disponibilità dei dati non sembrano migliorare [6, 52]. Inoltre, sistemi mobili caratterizzati da connettività intermittente stanno diventando sempre più popolari, un algoritmo di replicazione pessimistico che cerca di sincronizzarsi con un sito non disponibile può bloccarsi indefinitamente [10]. Infine ci sono anche problemi di corruzione dei dati, per esempio è impossibile raggiungere il consenso con certezza su una nuova replica primaria quando il ritardo nelle comunicazioni di rete non è predicibile [12].

Secondo, gli algoritmi di replicazione pessimistici non scalano in reti geografiche. È molto difficile costruire un sistema distribuito di grandi dimensioni con dati replicati in presenza di frequenti aggiornamenti perché le prestazioni del sistema e la disponibilità dei dati diminuiscono all'aumentare dei siti nel sistema [50, 51]. Proprio per questo motivo molti serivizi Internet utilizzano delle tecniche di replicazione ottimistica, ad esempio Usenet, DNS [26], e file system e database system pensati per sistemi mobili [17, 43].

Terzo, alcune attività umane richiedono la condivisione dei dati in maniera ottimistica. Ad esempio sistemi collaborativi di sviluppo software richie-

dono che gli utenti lavorino relativamente in isolamento tra di loro. È meglio per questi sistemi permettere agli utenti di aggiornare indipendentemente le proprie repliche dei dati e risolvere solo in seguito i conflitti, occasionali, che possono verificarsi, piuttosto che bloccare l'accesso ai dati mentre un utente li sta modificando.

1.2 Treeds

Treeds è un esperimento nel mantenimento in uno stato coerente di una struttura dati replicata in un ambiente altamente collaborativo utilizzando tecniche di replicazione ottimistica, in particolare utilizzando tecniche che garantiscono il modello di coerenza dei dati chiamato *eventual consistency* [47] fornite da Telex [2, 3], un toolkit semantico che supporta lo sviluppo di applicazioni collaborative.

Treeds implementa una struttura dati ad albero, replicata, atta a memorizzare informazioni con una struttura gerarchica. Treeds fornisce le operazioni necessarie per inserire ed eliminare elementi dalla struttura e operazioni per modificare la struttura della gerarchia stessa.

Il mantenimento in uno stato coerente di strutture dati replicate in ambienti in cui le operazioni di modifica sono molto frequenti pone seri problemi di performance. Approcci che forniscono un modello di coerenza single-copy soffrono il problema della sincronizzazione necessaria tra tutti i siti interessati ad ogni operazione di modifica delle repliche. Le tecniche di replicazione ottimistica, invece, promettono di migliorare le performance del sistema distribuito togliendo la sincronizzazione dal critical path dell'applicazione, che avviene ora in background, pur dando garanzie di convergenza dello stato delle repliche.

Il trade off inevitabile viene fatto sul tempo di convergenza dello stato delle repliche, che possono divergere momentaneamente, questo implica che gli utenti possono essere esposti a stati provvisori delle repliche che possono subire rollback.

1.3 Treeds e Telex

Telex è l'infrastruttura scelta per l'implementazione di Treeds perché con Treeds si vuole anche testare il supporto che Telex fornisce alla costruzione di applicazioni collaborative. Telex si fa carico della comunicazione asincrona per la propagazione delle operazioni tra i siti che compongono il sistema, del rilevamento e della risoluzione dei conflitti tra operazioni concorrenti e della creazione di schedule di azioni, cioè un elenco ordinato di operazioni la cui esecuzione determina lo stato delle repliche. Allo sviluppatore vengono fornite delle API necessarie per la definizione delle azioni, per la loro sottomissione a Telex e per indirizzare Telex nella scelta degli schedule.

Con l'implementazione di Treeds ci si vuole accertare che le API fornite da Telex creino il supporto sufficiente allo sviluppatore per la costruzione di un'applicazione collaborativa, e che gli permettano di concentrarsi il più possibile sulla logica dell'applicazione, lasciando a Telex tutta la gestione dello stato delle repliche e della loro convergenza.

1.4 Struttura della tesi

Il Capitolo 2 descrive in dettaglio le tecniche che vanno sotto il nome di replicazione ottimistica e le diverse scelte di design che ne costituiscono le diverse implementazioni pratiche. Nel Capitolo 3 vengono presentati alcuni sistemi software che rappresentano lo stato dell'arte nell'implementazione delle tecniche di replicazine ottimistica. Il Capitolo 4 presenta Telex, il toolkit di replicazione ottimistica utilizzato per l'implementazione di Treeds. Il Capitolo 5 presenta l'implementazione di Treeds e le scelte progettuali prese. Infine il Capitolo 6 presenta le considerazioni derivanti dall'implementazione di Treeds utilizzando le funzionalità fornite da Telex e i risultati sperimentali.

Capitolo 2

Replicazione ottimistica

In questo capitolo vengono descritte in dettaglio le tecniche che vanno sotto il nome di *replicazione ottimistica* [37] e viene introdotta la terminologia che verrà utilizzata in seguito.

2.1 Definizione di replicazione ottimistica

Per *replicazione ottimistica* si intendono un insieme di tecniche per la condivisione di dati in maniera efficiente in reti geografiche o in ambienti mobili.

La caratteristica chiave che distingue gli algoritmi di replicazione ottimistica dagli algoritmi di replicazione pessimistica è il loro approccio al controllo della concorrenza. Gli algoritmi di replicazione pessimistica coordinano in maniera sincrona tutte le repliche ad ogni operazione di modifica dello stato e ne bloccano l'accesso finché l'operazione non è terminata. Gli algoritmi di replicazione ottimistica, invece, permettono di accedere ai dati senza che vi sia sincronizzazione tra tutte le repliche, in base all'assunzione "ottimistica" che i conflitti si verifichino solo raramente, o che non si verifichino affatto. Una volta applicate localmente, le operazioni di modifica vengono propagate in background, e i conflitti occasionali vengono corretti dopo che si sono verificati. Non si tratta di una nuova idea, già in [15] si parla di replicazione

ottimistica, ma il suo uso si sta espandendo da quando le tecnologie legate ad Internet ed ai sistemi mobili stanno diventando di uso sempre più comune.

Gli algoritmi di replicazione ottimistica offrono molti vantaggi rispetto alle loro controparti pessimistiche. Permettono di aumentare la disponibilità dei dati consentendo alle applicazioni di proseguire con l'esecuzione anche quando alcune delle repliche non sono disponibili o quando si verificano problemi sulla rete. Sono flessibili rispetto alla topologia della rete e alla connettività dei siti, infatti, utilizzano tecniche di comunicazione epidemica che garantiscono la propagazione di tutte le operazioni a tutte le repliche anche quando la topologia della rete è sconosciuta e variabile. Offrono una maggiore scalabilità al sistema in presenza di un grande numero di repliche proprio perché è molto minore il livello di sincronizzazione richiesto tra le repliche. Consentono ai diversi siti che compongono il sistema di rimanere autonomi in quanto permettono l'aggiunta e la rimozione di nuove repliche senza la necessità di modifiche alla configurazione delle repliche esistenti. Permettono agli utenti di collaborare in maniera asincrona senza la necessità di essere tutti online nello stesso momento. Infine, forniscono un feedback più immediato in quanto possono applicare le operazioni provvisorie di modifica non appena vengono sottomesse al sistema.

Naturalmente tutti questi benefici hanno un costo. Tutti i sistemi distribuiti devono fare un trade-off tra disponibilità dei dati e coerenza dei dati [5, 13]. Mentre gli algoritmi di replicazione pessimistica aspettano, quelli ottimistici speculano. Gli algoritmi di replicazione ottimistica si trovano dunque a far conto con repliche che presentano uno stato divergente e conflitti tra operazioni concorrenti. Di conseguenza queste tecniche possono essere utilizzate solo se le applicazioni possono tollerare questo genere di conflitti e di divergenza nello stato delle repliche.

2.2 Componenti della replicazione ottimistica

Un algoritmo di replicazione ottimistica consiste dei seguenti cinque elementi:

oggetti, repliche, siti e operazioni: gli utenti sottomettono le operazioni di modifica dello stato delle repliche ad un sito.

propagazione delle operazioni: ogni sito propaga le operazioni *provvisorie* che gli vengono sottomesse dagli utenti agli altri siti e riceve altre operazioni *provvisorie* remote dagli altri siti.

esecuzione provvisoria e scheduling: ogni sito decide un ordinamento per le operazioni di cui è a conoscenza e le esegue.

individuazione e risoluzione dei conflitti: se ci sono dei conflitti tra le operazioni che un sito ha inserito nello schedule prodotto deve intraprendere delle azioni per risolverli.

consenso: tutti i siti raggiungono il consenso sull'ordinamento delle operazioni e sulla risoluzione dei conflitti, dopodiché le operazioni diventano *permanenti*.

2.2.1 Oggetti, repliche, siti e operazioni

Con il termine *oggetto* si intende la più piccola unità di replicazione, una *replica* è una copia di un oggetto memorizzata in un server. Un server che memorizza delle repliche viene chiamato *sito*, un sito può memorizzare repliche di diversi oggetti, ma, dato che le diverse repliche vengono gestite in maniera indipendente, senza perdita di generalità si considerano siti che memorizzano un solo oggetto, in questo modo i termini replica e sito vengono usati in maniera intercambiabile.

Si distinguono inoltre siti che possono aggiornare lo stato della replica, chiamati siti *master*, e siti che vi possono accedere in sola lettura, chiamati

siti *slave*. Si usa il simbolo N per indicare il numero totale delle repliche, ed il simbolo M per indicare il numero di repliche master. Valori comuni per N ed M sono: $M = 1$, sistemi *single-master*, ed $M = N$, *sistemi multi-master*.

Con il termine *operazione* si intende un'operazione che modifica lo stato di un oggetto. Le operazioni vengono propagate in background da un sito all'altro e possono venire applicate alle repliche anche molto dopo la loro sottomissione al sistema da parte dell'utente.

La natura di un'operazione può variare molto da sistema a sistema. Alcuni sistemi, come ad esempio DNS [26], supportano solo operazioni che modificano l'intero oggetto, questi sistemi prendono infatti il nome di sistemi *state-transfer*. Altri sistemi, come ad esempio Bayou [43], supportano operazioni che descrivono in maniera più sofisticata la modifica da applicare alla replica, questi sistemi prendono il nome di sistemi *operation-transfer*.

Per aggiornare un oggetto, gli utenti sottomettono un'operazione ad uno dei siti master. Il sito applica l'operazione localmente e permette poi all'utente di proseguire con la normale esecuzione dell'applicazione. Allo stesso tempo il sito propaga le operazioni locali agli altri siti facenti parte del sistema, dai quali riceve le operazioni remote che vengono applicate alla replica locale. Il modello di coerenza offerto da questi sistemi viene chiamato *eventual consistency* perché garantisce che lo stato delle repliche convergerà ad uno stato coerente, ma non dà alcuna garanzia temporale al riguardo. Alcuni sistemi di replicazione ottimistici offrono però delle garanzie più forti che limitano il livello di divergenza delle repliche (Sezione 2.7.2).

2.2.2 Propagazione delle operazioni

Ogni operazione sottomessa dagli utenti viene memorizzata in maniera persistente in un file di log per essere poi propagata agli altri siti. Questi sistemi utilizzano in genere degli algoritmi epidemici per la propagazione dei messaggi che permettono ai siti che riescono a comunicare direttamente di scambiarsi non solo le operazioni che sono state sottomesse direttamente a

loro, ma anche le operazioni remote ricevute da altri siti. Grazie all'utilizzo di questi algoritmi tutte le operazioni vengono propagate a tutti i siti anche se questi non possono comunicare direttamente tra di loro [8].

Queste tecniche vengono descritte in maggior dettaglio nella Sezione 2.6.

2.2.3 Esecuzione provvisoria e scheduling

Proprio a causa della loro propagazione in background, i diversi siti possono ricevere le operazioni in ordine diverso gli uni dagli altri. Ogni sito deve costruirsi un ordinamento appropriato per le operazioni in modo tale che la loro esecuzione in ongnuno dei siti porti allo stesso stato finale e, allo stesso tempo, che rispetti le intenzioni degli utenti.

Tutte le operazioni sono inizialmente considerate *provvisorie*. Un sito può riordinare o trasformare le operazioni provvisorie ripetutamente prima di raggiungere un accordo con gli altri siti riguardo l'ordinamento finale delle operazioni. Con il termine *scheduling* ci si riferisce alla politica di ordinamento delle operazioni. In genere è necessario raggiungere un consenso tra tutti i siti riguardo l'ordinamento delle operazioni perché l'algoritmo di scheduling potrebbe trovarsi a fare delle scelte arbitrarie riguardo l'ordinamento di certe operazioni, percui è necessario che tutti i siti facciano le stesse decisioni affinché lo stato finale delle repliche converga.

Queste tecniche vengono descritte in maggior dettaglio nella Sezione 2.4.1.

2.2.4 Individuazione e risoluzione dei conflitti

Senza sincronizzazione da parte di tutti i siti per ogni operazione di modifica, può capitare che diversi utenti modifichino lo stesso oggetto allo stesso tempo, introducendo nel sistema delle operazioni concorrenti potenzialmente in conflitto. È possibile ignorare questo tipo di problema scegliendo sempre arbitrariamente una delle due modifiche concorrenti e ignorando l'altra [15], ma ciò darebbe luogo alla perdita di operazioni di modifica (lost update).

La perdita di operazioni di modifica è chiaramente non auspicabile nella maggior parte delle applicazioni. Un modo migliore di trattare il problema dei conflitti tra operazioni concorrenti è quello di rilevarli e in seguito risolverli. Un'operazione genera un conflitto quando, eseguendo l'operazione nello stato derivante dall'esecuzione di tutte le operazioni a lei precedenti nello schedule, questa viola le proprie invarianti.

Le invarianti delle operazioni possono essere inserite implicitamente nell'algoritmo di replicazione, l'esempio più semplice è quello di etichettare come in conflitto tutte le operazioni concorrenti, come avviene nel file system distribuito Coda [19]. Oppure può essere compito dell'utente specificarle escplicitamente, come ad esempio avviene in Bayou [43].

Il processo di risoluzione dei conflitti in genere dipende molto dall'aplicazione. Alcuni sistemi etichettano semplicemente le operazioni in conflitto e lasciano poi all'utente il compito di correggere il conflitto, altri sistemi invece riescono a risolvere i conflitti in maniera automatica.

2.2.5 Consenso

I processi di scheduling e di risoluzione dei conflitti possono trovarsi a dover prendere delle decisioni arbitrarie, inoltre le repliche possono essere a conoscenza di un diverso insieme di operazioni, per questo motivo lo stato delle repliche può divergere. Per far sì che lo stato di tutte le repliche converga occorre che tutte le repliche raggiungano un consenso sull'insieme delle operazioni da eseguire, sul loro ordinamento e sul modo in cui gli eventuali conflitti vengono risolti.

2.3 Scelte di design

Lo scopo ultimo di ogni sistema di replicazione ottimistica è quello di mantenere coerente lo stato delle repliche che gestisce. D'altra parte, come raggiungere tale scopo può variare molto da sistema a sistema.

Sono possibili diverse scelte di design nell'implementazione di un sistema di replicazione ottimistica e queste scelte, come sempre, dipendono dal tipo di applicazione che si intende supportare. In Tabella 2.1 sono riassunte le possibili scelte di design che verranno descritte in dettaglio nelle sezioni successive.

Scelte di design		
Numero di scrittori	Sistemi single-master Sistemi multi-master	
Definizione delle operazioni	Sistemi state-transfer Sistemi operation-transfer	
Scheduling	Sinattico Semantico	
Rilevamento dei conflitti	Sintattico Semantico	
Risoluzione dei conflitti	Manuale	
	Automatica	Sintattica Semanica
Propagazione delle operazioni	Sistemi pull Sistemi push	
Modello di coerenza	Eventual consistency	
	Divergenza vincolata	Garanzie di sessione Vincoli quantitativi

Tabella 2.1: Possibili scelte di design per un sistema di replicazione ottimistica.

2.3.1 Numero di scrittori

È possibile scegliere il numero di repliche a cui è ammesso sottomettere le operazioni di modifica dello stato.

Nei sistemi *single-master* ($M = 1$) una delle repliche è designata come *master* ed è l'unica replica alla quale è ammesso sottomettere le operazioni di modifica che vengono quindi serializzate. Tutte le operazioni di modifica vengono poi propagate a tutte le altre repliche, dette *slaves*. Questi sistemi sono semplici perché non devono preoccuparsi dello scheduling delle operazioni, né tantomeno del rilevamento e risoluzione dei conflitti tra operazioni concorrenti. D'altra parte forniscono una limitata disponibilità dei dati, soprattutto quando il sistema si trova a sostenere un alto numero di operazioni di modifica.

I sistemi *multi-master* ($M > 1$) invece permettono di sottomettere le operazioni di modifica dello stato a più repliche in maniera indipendente. Le operazioni vengono poi propagate alle altre repliche in background. Questi sistemi forniscono un'elevata disponibilità dei dati ma sono molto più complessi. In particolare si trovano ad affrontare il problema dello scheduling delle operazioni e del rilevamento e risoluzione dei conflitti tra operazioni concorrenti. Inoltre, un altro potenziale problema con questi sistemi è la loro scalabilità a causa dell'aumento del tasso delle operazioni in conflitto.

2.3.2 Definizione delle operazioni

I sistemi *state transfer* limitano le operazioni a letture o modifiche dell'intero oggetto condiviso. Sono più semplici perché il mantenimento della coerenza delle repliche riguarda solo il trasferimento della replica più aggiornata agli altri siti. Questi sistemi offrono però poco aiuto nella risoluzione dei conflitti. Sistemi di questo tipo utilizzano la regola di scrittura di Thomas [15], che però fa sorgere il problema della perdita degli aggiornamenti, o presentano all'utente i due stati in conflitto e lasciano all'utente la risoluzione manuale del conflitto, oppure scelgono arbitrariamente uno dei due stati (ad esempio in base al sito che ha originato l'operazione).

I sistemi *operation transfer* invece permettono una descrizione semantica delle operazioni. Questi sistemi sono più complessi perché devono memorizzare l'insieme di tutte le operazioni sottomesse al sistema e devono fare in

modo che tutti i siti raggiungano il consenso sull'insieme di operazioni da applicare e sul loro ordine. Però questi sistemi offrono la possibilità di implementare un meccanismo di rilevamento e risoluzione dei conflitti più flessibile in quanto possono anche tenere conto della semantica delle operazioni.

2.3.3 Scheduling

Compito dello scheduling è quello di trovare un ordinamento delle operazioni di cui una replica è a conoscenza in modo da rispettare le intenzioni degli utenti e da produrre degli stati equivalenti in tutte le repliche. Le diverse politiche di scheduling si possono classificare in sintattiche o semantiche.

Le tecniche di *scheduling sintattico* riordinano le operazioni con le sole informazioni che riguardano il momento in cui un'operazione è stata sottomessa o quale sito l'ha sottomessa. I sistemi basati sul timestamp delle operazioni ricadono in questa categoria. Questi sistemi sono più semplici ma possono dare luogo ad un numero maggiore di conflitti e di rollback.

Le tecniche di *scheduling semantico*, invece, sfruttano la semantica delle operazioni, come ad esempio la loro commutatività o idempotenza, per ridurre il numero di conflitti e di rollback. Lo scheduling semantico può essere utilizzato solo in sistemi operation-transfer in quanto i sistemi state transfer ignorano completamente, per loro natura, la semantica delle operazioni.

2.3.4 Gestione dei conflitti

L'approccio ideale per la gestione dei conflitti tra operazioni concorrenti è quello di evitarli.

I sistemi di replicazione pessimistica prevengono il verificarsi dei conflitti bloccando l'accesso alle repliche ad ogni operazione di modifica. I sistemi single-master, invece, li evitano permettendo la sottomissione delle operazioni di modifica solo alla replica master, nella quale vengono serializzate. Entrambi gli approcci soffrono però di una bassa disponibilità dei dati.

Per evitare i conflitti in sistemi multi-master che utilizzano le tecniche di replicazione ottimistica l'unico modo è l'utilizzo di strutture dati commutative. Se lo stato delle repliche è costituito da sole strutture dati con operazioni commutative, allora l'unico requisito necessario alla convergenza dello stato delle repliche è che queste ricevano tutte lo stesso insieme di operazioni da eseguire. In questo caso l'ordine di esecuzione non è più importante. Il problema di questo approccio è che non sempre è possibile utilizzare delle strutture dati commutative per rappresentare lo stato delle repliche. Nonostante a volte sia possibile modificare le strutture dati affinché diventino commutative, o affinché alcune delle operazioni su di esse lo diventino, spesso è necessario un redesign completo, e non banale, delle strutture dati affinché diventino commutative [30, 36, 39].

Alcuni sistemi, generalmente sistemi state-transfer, invece, ignorano i conflitti sovrascrivendo ogni potenziale operazione in conflitto con un'operazione più recente [15], si tratta di un approccio non ottimale perché causa la perdita di alcune operazione di modifica.

Un altro approccio consiste nel fare in modo che il sistema possa riconoscere i conflitti e risolverli. Come per le politiche di scheduling, anche le politiche di rilevamento e risoluzione dei conflitti si possono classificare in sintattiche e semantiche.

Nei sistemi con *rilevamento sintattico* dei conflitti le invarianti delle operazioni non sono espresse esplicitamente dall'utente o dall'applicazione, ma dipendono dal tempo di sottomissione delle operazioni stesse. Questi sistemi rilevano un conflitto in maniera conservativa tra ogni coppia di operazioni concorrenti. Le operazione vengono ordinate tramite l'ordinamento parziale imposto dalla relazione *happens-before* [21], se due operazioni non sono comparabili significa che sono concorrenti.

I sistemi con *rilevamento semantico* dei conflitti possono invece sfruttare le informazioni semantiche delle operazioni per ridurre i conflitti. Il modo in cui ciò può essere fatto dipende dal modo in cui vengono descritte le operazioni. Ad esempio, il sistema può accorgersi che nonostante due operazioni

siano concorrenti e vadano a modificare lo stesso oggetto, in realtà sono indipendenti e non creano un reale conflitto, ovvero qualunque sia l'ordine di esecuzione delle due operazioni, lo stato finale dell'oggetto è lo stesso.

Come per le politihe di scheduling, anche per le politiche di rilevamento dei conflitti i sistemi sintattici sono generici e più semplici ma etichettano un maggior numero di operazoni come in conflitto, mentre i sistemi semantici sono più complessi ma anche più accurati. Questo parallelismo tra il sottositema di scheduling e quello del rilevamento dei conflitti è dovuto al fatto che si tratta di due problemi molto legati tra loro.

2.3.5 Strategie di propagazione

I siti che compongono il sistema devono decidere quando e in che modo comunicare le operazioni locali agli altri siti. Le scelte di design in questo caso riguardano la topologia della rete che i siti formano tra di loro ed il grado di sincronia tra i siti che compongono il sistema.

Se i siti formano una rete con *topologia fissa*, come ad esempio a stella o uno spanning tree, l'algoritmo di comunicazione delle operazioni sarà molto efficiente, però non sarà in grado di funzionare in maniera adeguata in reti dinamiche, in cui i siti possono inserirsi o andarsene dal sistema, o in reti soggette a guasti.

Invece, l'uso di *algoritmi di comunicazione epidemici* permette di propagare le operazioni a tutti i siti facenti parte del sistema, non importa quale sia la topologia della rete che formano, e funzionano anche in presenza di cambiamenti dinamici nella topologia [8].

Con "grado di sincronia" tra i siti si intende la velocità e la frequenza con cui i siti si scambiano le operazioni locali.

Nei sistemi *pull-based* ogni sito interroga gli altri siti per sapere se questi ultimi hanno delle nuove operazioni da comunicare. Le interrogazioni possono avvenire sia manualmente, come ad esempio avviene nei PDA, o periodicamente, come avviene in DNS.

Nei sistemi push-based, invece, ogni sito con delle nuove operazioni da trasmettere, le trasmette periodicamente agli altri siti.

In generale, più è veloce la propagazione delle operazione, minore sarà il grado di incoerenza tra le repliche e il tasso delle operazioni in conflitto, ma maggiore sarà la complessità del sistema e l'overhead dovuto alle comunicazioni, specialmente nel caso di applicazioni che producono un elevato numero di operazioni di modifica.

2.3.6 Garanzie di coerenza

Le garanzie di coerenza in un sistema di replicazione ottimistica definiscono il grado di divergenza nello stato delle repliche che un'applicazione client può osservare.

Il modello di *coerenza debole*, che caratterizza i sistemi di replicazione ottimistica, assicura informalmente solo che le repliche si troveranno in uno stato coerente quando il sistema si troverà in uno stato tranquillo, cioè senza che nessun sito immetta nel sistema alcuna operazione di modifica dello stato, a sarà trascorso sufficiente tempo, altrimenti lo stato delle repliche può divergere. Si tratta senza dubbio di una garanzia alquanto debole, ma può essere utile in sistemi in cui la disponibilità dei dati è di primaria importanza.

Esistono altri modelli di coerenza che pongono dei vincoli quantitativi sullo stato delle repliche al fine di dare delle garanzie più forti all'pplicazione [44, 48]. Questi modelli di coerenza in genere operano bloccando l'accesso alle repliche quando determinate condizioni di coerenza del loro stato non sono raggiunte. Questi modelli di coerenza verranno spiegati in maggior dettaglio nella Sezione 2.7

2.4 Controllo della concorrenza

Questa sezione descrive le tecniche che permettono ai siti che fanno parte di un sistema di replicazione ottimistica di raggiungere uno stato coerente. In un sistema di questo tipo i diversi siti raccolgono e ordinano, generando

uno schedule, le operazioni che sono sottomesse localmente alla replica e alle altre repliche remote.

Due schedule vengono definiti *equivalenti* quando, partendo dallo stesso stato iniziale, l'applicazione delle operazioni contenute nei due schedule produce lo stesso stato finale. Ai fini della risoluzione dei conflitti agli schedule è permesso includere operazioni che non vengono eseguite, tali operazioni, chiamate operazioni abortite, vengono denotate con $\overline{A}$, $\overline{B}$, ...

Un oggetto viene detto *eventually consistent* quando rispetta le seguenti condizioni, assumendo che tutte le repliche siano inizializzate con lo stesso stato iniziale:

- in ogni momento, per ogni replica, esiste un prefisso di operazioni nello schedule equivalente ad un prefisso di operazioni nello schedule di ogni altra replica. Chiamiamo questo prefisso di operazioni *prefisso delle operazioni stabili* della replica.
- il prefisso delle operazioni stabili di ogni replica cresce monotonicamente nel tempo.
- tutte le operazioni non abortite facenti parte del prefisso delle operazioni stabili soddisfano le proprie invarianti.
- per ogni operazione A sottomessa al sistema, o A o $\overline{A}$ sarà prima o poi inclusa nel prefisso delle operazioni stabili.

2.4.1 Scheduling

Come già accennato nella Sezione 2.3.3, lo scheduling sintattico definisce un ordinamento totale sulle operazioni basandosi esclusivamente sul tempo di sottomissione delle operazioni o sul sito alle quali sono state sottomesse, mentre lo scheduling semantico sfrutta la semantica delle operazioni.

Scheduling sintattico Gli scheduler sintattici preservano l'ordinamento parziale imposto dalla relazione *happens-before* [21] tra le operazioni e

cercano di ordinare in maniera arbitraria le operazioni concorrenti. I sistemi che utilizzano timestamp scalari sono in grado di ordinare le operazioni in modo deterministico. Questo evita la necessità di utilizzare un algoritmo di consenso per l'ordinamento delle operazioni. Altri sistemi, come Bayou [43], ordinano invece le operazioni in maniera arbitraria e diventa quindi necessario l'utilizzo di un algoritmo di consenso tra i siti.

Scheduling semantico sfruttando la commutatività Questo tipo di scheduler semantici tengono conto delle relazioni semantiche tra le operazioni, in questo modo dispongono di una certa flessibilità nella generazione degli schedule. In generale, questi schedule utilizzano le relazioni semantiche tra le operazioni, come ad esempio la loro commutatività e la loro idempotenza, per ordinarle e non tengono conto della relazione happens-before. Se ad esempio due operazioni sono commutative o idempotenti, possono essere ordinate in qualunque modo l'una rispetto all'altra producendo due schedule equivalenti. In questo caso è quindi meglio ignorare la relazione happens-before che potrebbe esistere tra le due operazioni che altrimenti ne vincolerebbe l'ordine inutilmente.

Scheduling semantico con ordinamento canonico Alcuni sistemi [34] definiscono per ogni coppia di operazioni che può essere eseguita concorrentemente delle regole che specificano come le due operazioni interagiscono tra di loro e come possono essere ordinate. Le operazioni non concorrenti, invece, vengono ordinate secondo l'ordinamento imposto dalla relazione happens-before. Questa tecnica definisce quindi un ordinamento canonico tra le operazioni che possono essere ordinate indipendentemente da ogni sito senza bisogno di alcuna sincronizzazione o comunicazione con gli altri siti. Il problema di questo approccio è la sua complessità: anche in sistemi con un numero ridotto di operazioni, l'insieme delle regole prodotte risulta molto elevato. Rimane quindi da vedere come sia possibile applicare questo approccio a sistemi più

complessi.

Scheduling semantico con trasformazione delle operazioni La trasformazione delle operazioni (Operation Transformation) è una tecnica sviluppata nell'ambito degli editor collaborativi [9, 29, 40, 41, 42, 46]. I siti applicano le operazioni nell'ordine in cui queste vengono ricevute senza riordinare le operazioni già eseguite. Per far sì che lo stato delle repliche converga, questa tecnica definisce per ogni coppia di operazioni che può essere eseguita concorrentemente, una regola di riscrittura delle operazioni che garantisce sia la convergenza dello stato delle repliche che la preservazione dell'intenzione degli utenti. Anche in questo caso l'insieme delle regole di riscrittura è complesso perché dev'essere provato che faccia convergere lo stato delle repliche per ogni possibile coppia di operazioni concorrenti.

Scheduling semantico ottimizzato IceCube [16, 32] è un toolkit generico che supporta lo sviluppo di applicazioni collaborative. È il primo sistema ad avere introdotto il concetto di *vincolo* tra le operazioni. Un vincolo è una relazione semantica tra le operazioni che concretizza sia vincoli di workflow tra le operazioni, sia delle invarianti specifiche dell'applicazione. In IceCube il problema dello scheduling viene ricondotto ad un problema di ottimizzazione il cui scopo è quello di trovare l'ordinamento "migliore" delle operazioni che rispetti i vincoli specificati tra le operazioni stesse. La bontà dello schedule può essere definita dall'utente e dall'applicazione.

2.4.2 Rilevamento dei conflitti

Un'operazione A è in conflitto quando le proprie invariante non sono soddisfatte nello stato in cui si trova la replica dopo aver eseguito tutte le operazioni precedenti ad A nello schedule.

Anche nell'ambito del rilevamento dei conflitti, le tecniche adottabili possono essere classificate in tecniche sintattiche e semantiche.

Le tecniche sintattiche di rilevamento dei conflitti utilizzano la relazione happens-before per rilevare un conflitto tra le operazioni. In pratica due operazioni sono conservativamente dichiarete in conflitto quando sono concorrenti. Queste tecniche vengono illustrate in maggior dettaglio nella Sezione 2.5 e sono tipiche dei sistemi state-transfer che non hanno alcuna nozione della semantica delle operazioni.

Le tecniche semantiche di rilevamento dei conflitti, invece, sfruttano la conoscenza della semantica delle operazioni per rilevare i conflitti tra le operazini. In questo modo il rilevamento dei conflitti può essere più accurato. In alcuni sistemi, come ad esempio i file system distribuiti, la procedura di rilevamento dei conflitti è integrata nel sistema stesso [18, 19, 35]. Altri sistemi invece, in particolare Bayou [43] e IceCube [16, 32], permettono all'applicazione di specificare esplicitamente le invarianti delle operazioni. Questi sistemi sono infatti dei toolkit generici per la costruzione di applicazioni collaborative e devono quindi supportare il maggior numero di applicazioni possibili.

Le tecniche semantiche di rilevamento dei conflitti sono strettamente più espressive delle tecniche sintattiche. È infatti possibile costruire un rilevatore di conflitti semantico che emuli un rilevatore sintattico, ma non è possibile fare l'inverso.

2.4.3 Risoluzione dei conflitti

Il ruolo della risoluzione dei conflitti è quello di riscrivere o di abortire alcune operazioni in modo da rimuovere i conflitti. La risoluzione dei conflitti può essere manuale o automatica:

risoluzione manuale: in questo caso il sistema esclude dallo schedule le operazioni in conflitto e presenta all'utente due versioni dell'oggetto: starà all'utente creare una nuova versione dell'oggetto che incorpori le modifiche presenti nelle operazioni in conflitto e sottomettere al sistema questa nuova operazione. Questo tipo di tecnica è usata ad esempio in CVS.

risoluzione automatica: in questo caso la risoluzione dei conflitti viene affidata ad una procedura automatica. Alcuni sistemi forniscono delle funzioni ad hoc per la risoluzione dei conflitti, come ad esempio i file system distribuiti [18, 19, 35]. Altri sistemi, invece, supportano la costruzione di applicazioni collaborative generiche e quindi lasciano all'applicazione il compito di definire le funzioni necessarie alla risoluzione automatiche dei conflitti [16, 32, 43].

2.4.4 Protocollo di consenso

Quando lo sceduling e la risoluzione dei conflitti non utilizzano algoritmi deterministici, ma si trovano costretti a prendere delle decisioni arbitrarie, diventa necessario che tutti i siti facenti parte del sistema si accordino su tali decisioni. Per accordare tutti i siti sulle scelte non deterministiche da prendere il sistema di replicazione ottimistica utilizza un protocollo di consenso.

Il protocollo di consenso è importante anche perché permette ai siti facenti parte del sistema di sapere con esattezza quali siano le operazioni *stabili*, cioè quelle operazioni che non saranno più soggette a rollback. Quando un sito è a conoscenza di quali operazioni sono diventate stabili può anche eliminare tutte le informazioni relative a tali operazioni perché non saranno più necessarie nei processi di scheduling e di risoluzione dei conflitti. Quindi il protocollo di consenso serve anche come meccanismo per limitare la quantità di memoria usata dal sistema.

2.5 Sistemi state-transfer

Nei sistemi state-transfer le operazioni di modifica consistono nella sovrascrittura dell'intero oggetto: hanno il vantaggio che le repliche convergono applicando solamente l'ultimo aggiornamento, e possono quindi saltare le fasi di rilevamento e risoluzione dei conflitti e di scheduling.

2.5.1 Thomas' write rule

I sistemi state-transfer devono accordarsi solamente su quale replica abbia lo stato più aggiornato la regola di scrittura di Thomas (Thomas' write rule) [45] è un algoritmo molto usato in questo tipo di sistemi per ottenere eventual consistency.

Secondo questo algoritmo ad ogni replica è associato un timestamp che ne indica il livello di aggiornamento. Occasionalmente le repliche copiano il timestamp di un'altra replica: se questo timestamp indica un livello di aggiornamento maggiore, la prima replica copia il contenuto della seconda replica e aggiorna il timestamp. Questo algoritmo non rileva i conflitti tra le operazioni concorrenti, o meglio, risolve tutti i conflitti prendendo come valore corretto per la replica quello più aggiornato, causando la perdita di tutte le operazioni intermedie.

Senza alcuna modifica questo algoritmo ha dei problemi quando il contenuto delle repliche viene eliminato. Se si elimina semplicemente il contenuto della replica e il timestamp ad essa associato, dato che la replica non esiste più, potrebbe verificarsi il seguente caso. Supponiamo che il sito i aggiorni un oggetto, con timestamp T_i, e il sito j elimini lo stesso oggetto, con timestamp T_j, simultaneamente. Ora il sito k contatta il sito j ed esegue l'operazione di eliminazione dell'oggetto. Successivamente, sempre il sito k contatta il sito i. Il sito k dovrebbe ora confrontare i due timestamp T_i, associato all'operazione di aggiornamento, e T_j, associato all'operazione di eliminazione. Se $T_i > T_j$ il sito k deve ricreare l'oggetto eliminato con il contenuto in possesso del sito i, nel caso opposto deve ignorare l'operazione di aggiornamento. Il problema è che il sito k non può eseguire questo confronto perché il timestamp associato all'operazione di eliminazione è stato eliminato.

Due soluzioni sono state proposte per risolvere questo problema. La prima prevede che l'eliminazione degli oggetti possa avvenire solo offline tramite intervento umano, come avviene in DNS e NIS. La seconda soluzione invece prevede l'uso di "*tombstones*", ovvero delle strutture dati che mantengono il timestamp degli oggetti eliminati ma non il contenuto.

2.5.2 Algoritmo dei due timestamp

Questo algoritmo è un'estensione dell'algoritmo Thomas' write rule ed è in grado di rilevare i conflitti tra le operazioni concorrenti. Secondo questo algoritmo, ad ogni replica sono associati due timestamps, uno, come nel caso dell'algoritmo Thomas' write rule, indica il livello di aggiornamento della replica, l'altro indica l'ultima volta che la replica è stata aggiornata.

L'algoritmo rileva un conflitto ogni volta che il timestamp che indica l'ultimo aggiornamento della replica in due siti è differente. Il problema di questo algoritmo è che può rilevare dei falsi positivi in presenza di più di due repliche, per questo motivo è applicabile solo in sistemi con un numero ridotto di repliche e in cui il verificarsi dei conflitti sia raro.

2.5.3 Tombstones selettive

Questo algoritmo è un'estensione della tecnica delle tombstone descritta nella Sezione 2.5.1. L'algoritmo riportato precedentemente non fa alcun garbage collection delle tombstones: e questo può diventare un problema in sistema in cui sono presenti molti oggetti che vengono eliminati.

Alcuni sistemi eliminano le tombstone unilateralmente dopo un certo lasso di tempo [17], una tecnica che funziona bene nella pratica ma che può presentare dei problemi: se ad esempio un sito rimane offline per un lasso di tempo superiore al tempo di vita delle tombstone, può inserire nel sistema delle operazioni errate.

Altri sistemi [28] estendono questa tecnica fancendo sì che le tombstone rimangano comunque memorizzate indefinitamente, ma solo su alcuni siti designati, mentre vengono eliminate dagli altri dopo il loro tempo di vita. Quando un sito che non ha ricevuto l'operazione di eleminazione di un dato oggetto sottomette al sistema delle operazioni di modifica, se gli altri siti hanno già eliminato la tombstone in questione potrebbero erroneamente reinserire l'oggetto eliminato nel sistema. I siti designati al mantenimento

delle tombstone sottomettono allora al sistema delle operazioni per eliminare l'oggetto reinserito erroneamente.

Altri sistemi si affidano ad algoritmi di consenso [14] per essere sicuri che tutti i siti eliminino le tombstone. Il problema nell'utilizzo di un algoritmo di consenso è che tutti i siti devono essere online perché questo possa terminare.

2.6 Propagazione delle operazioni

In questa sezione vengono descritti degli algoritmi efficienti per la propagazione delle operazioni a tutti i siti facenti parte del sistema di replicazione ottimistica.

2.6.1 Propagazione delle operazioni utilizzando i vector clock

Molti sistemi operation-transfer utilizzano i vector clock [11, 23] per propagare le operazioni in maniera efficiente.

Ogni sito i mantiene il proprio vector clock VC_i, $VC_i[i]$ contiene il numero di operazioni sottomesse al sistema dal sito i, mentre $VC_i[j]$ contiene il timestamp dell'ultima operazione sottomessa al sistema dal sito j di cui i è a conoscenza. Per far sì che due siti si scambino l'intero insieme di operazioni di cui sono a conoscenza, ognuno dei due deve eseguire il seguente algoritmo:

- il i ottieme il vector clock del sito j, VC_j.
- $\forall k$ tale che $VC_i[k] > VC_j[k]$ il sito i invia al sito j tutte le operazioni che sono state sottomesse dal sito k tali che il loro timestamp sia $>$ di $VC_j[k]$.

Quando entrambi i siti hanno eseguito questi due passi, entrambi saranno in possesso dello stesso insieme di operazioni.

2.6.2 Propagazione efficiente delle operazioni in sistemi state-transfer

Nei sistemi state-transfer le operazioni di modifica consistono nella sovrascrittura completa della replica (Sezione 2.3.2). Propagare l'intero stato della replica può diventare inefficiente quando la dimensione dello stato della replica comincia a crescere di dimensione.

Le tecniche usate per ovviare a questo problema sono le seguenti:

sistemi ibridi state e operation transfer: Questi sistemi mantengono un elenco delle ultime operazioni effettuate: per ogni operazione mantengono le differenze tra lo stato prima della sua applicazione e dopo e il timestamp associato al momento in cui l'operazione è stata applicata. Quando una replica deve aggiornarne un'altra il cui timestamp è registrato nell'elenco delle operazioni, può semplicemente spedire le differenze degli stati dal momento indicato dal timestamp in poi. Se invece il timestamp è troppo vecchio e non è memorizzato nell'elenco, allora dovrà spedire l'intero stato della replica.

divisione gerarchica degli oggetti: questa tecnica viene applicata agli oggetti che presentano una naturale suddivisione gerarchica. Gli oggetti vengono strutturati in alberi e ogni nodo intermedio memorizza il timestamp dell'aggiornamento più recente applicato ai propri nodi figli. In questo modo è possibile discendere l'albero dalla radice alle foglie individuando quali parti dell'oggetto hanno subito delle modifiche e trasmettere solo quelle.

collision resistant hash functions: anche questa tecnica divide gli oggetti in oggetti più piccoli, ma viene applicata agli oggetti che non presentano una naturale struttura gerarchica. In questo caso l'oggetto viene diviso in più chunck. Quando un sito vuole trasmettere il proprio stato ad un altro sito, applica una funzione hash ad ogni suo chunck e trasmette al sito ricevente questi valori calcolati. Il sito ricevente poi

richiede al sito mittente ogni chunk che risulta a lui mancante. Questa tecnica però funziona efficientemente solo se l'unica operazione di modifica degli oggetti consiste nell'aggiunta di nuovi dati in coda agli oggetti.

Per ovviare a questo problema, l'utility di sincronizzazione di file *rsync* spedisce i valori calcolati dalla funzione hash nella direzione inversa. Prima la replica che deve aggiornarsi spedisce il risultato della funzione hash applicata a tutti i chunk della sua replica. Poi la replica che deve aggiornare la prima applica la funzione hash ad ogni possibile chunk partendo da ogni byte nell'oggetto. In questo modo riesce a scoprire tutti i chunck mancanti all'altra replica.

2.6.3 Sistemi push

Nei sistemi *push*, quando un sito ha una nuova operazione la spedisce proattivamente agli altri siti. In questo modo si riduce il ritardo nella propagazione delle operazioni e si riduce l'overhead del sistema dovuto all'interrogazione delle repliche per sapere se hanno degli aggiornamenti da propagare.

flooding: è la tecnica più semplice di pushing. Quando un sito possiede un'operazione nuova, la propaga a tutti i siti con cui può comunicare direttamente. Quando un sito riceve un'operazione ne controlla il timestamp per evitare di propagare duplicati nel sistema. Questa tecnica assicura che tutte le operazioni vengano propagate a tutti i nodi.

tecniche di monitoraggio dello stato dei link: il *rumor mongering* [8] e *gossiping direzionale* [22] sono due tecniche che migliorano la tecnica di flooding immettendo nel sistema un minor numero di operazioni duplicate.

Il rumor mongering inizia come il flooding, ma memorizza il numero di duplicati ricevuti per ogni operazione e quando il numero di duplicati eccede una data soglia smette di propagare quell'operazione.

Nel gossiping direzionale, invece, ogni sito monitora i percorsi distinti che le operazioni hanno seguito nel corso della loro propagazione. I collegamenti diretti tra siti condivisi da pochi di questi percorsi distinti vengono favoriti con la propagazione di un numero maggiore di operazioni rispetto agli altri. Questo perché i collegamenti poco condivisi sono probabilmente uno dei pochi collegamenti esistenti tra quei due siti. Dato che entrambe le tecniche presentate sono delle tecniche euristiche, possono entrambe sbagliarsi e propagare le operazioni in maniera errata, per questo motivo entrambe le tecniche, periodicamente, ritornano ad operare come la tecnica di flooding semplice, per far sì che tutte le operazioni vengano propagate a tutti i siti.

2.7 Vincoli di coerenza

Queste tecniche servono per garantire che una replica rispetti un dato livello di coerenza specificato. Con l'utilizzo di queste tecniche è quindi possibile vincolare la qualità dello stato delle repliche alle quali si accede.

Queste tecniche funzionano controllando che la replica alla quale si sta indirizzando un'operazione rispetti i vincoli imposti. Se i vincoli sono rispettati allora l'operazione viene sottomessa alla replica, se non sono rispettati l'accesso alla replica viene bloccato finché questa non raggiunge lo stato di aggiornamento necessario. Si tratta quindi di tecniche che aumentano la coerenza delle repliche diminuendo la disponibilità dei dati.

2.7.1 Garanzie di sessione

Le garanzie di sessione [44] generano delle dipendenze causali tra le operazioni al fine di garantire uno dei seguenti livelli di coerenza:

Read your writes: (RYW) garantisce che lo stato letto da una replica includa tutte le operazioni di modifica sottomesse dallo stesso utente.

Monotonic reads: (MR) garantisce che operazioni di lettura sucessive eseguite dallo stesso utente restituiscano valori sempre più aggiornati (non strettamente).

Writes follow reads: (WFR) garantisce che ogni operazione di modifica venga eseguita su una data replica solo dopo che tutte le operazioni di modifica osservate da una precedente lettura, dallo stesso utente, siano state eseguite sulla stessa replica.

Monotonic writes: (MW) garantisce che un'operazione di modifica sia eseguita su una data replica solo dopo che tutte le precedenti operazioni di modifica sottomesse dallo stesso utente siano state eseguite dalla stessa replica.

Tramite l'utilizzo delle garanzie di sessione, l'applicazione è in grado di avere una vista dei dati condivisi che sia coerente con le operazioni effettuate durante la sessione corrente, anche se l'applicazione accede, durante la sessione, a repliche differenti, che possono quindi trovarsi in stati inizialmente incoerenti tra loro.

2.7.2 Vincoli quantitativi

Queste tecniche vincolano l'accesso alle repliche ad una misura quantitativa della divergenza tra il loro stato.

La metrica più semplice da utilizzare è l'obsolescenza dello stato delle repliche [1]. In questo caso l'accesso ad una replica viene bloccato se il suo stato non è stato aggiornato per un tempo superiore a quello definito.

Un'altra metrica usata, chiamata *order error*, misura il numero di operazioni provvisorie che una replica incorpora. In questo caso l'accesso ad una replica viene bloccato se lo stato di questa replica è stato generato da un numero di operazioni provvisorie superiore a quello specificato.

Un'altra metrica quantitativa molto interessante, utilizzata insieme alle altre in [48], è chiamata *numerical error*. Questa metrica misura la divergenza

tra il valore dello stato della replica e una stima del valore che la replica avrebbe avuto se le repliche avessero rispettato delle garanzie di coerenza forti (coerenza single-copy). In questo caso quindi l'accesso alla replica viene bloccato se viene stimato che il valore del suo stato si discosta dal valore "reale" oltre una certa soglia definita. Ovviamente il valore "reale" che le repliche avrebbe se venissero rispettate delle garanzie di coerenza forte viene stimato dal sistema. Questa metrica è la più costosa da ottenere perché richiede una certa comunicazione tra le repliche per poter essere mantenuta.

Capitolo 3

Stato dell'arte

La replicazione è una tecnica chiave per aumentare la scalabilità di sistemi distribuiti che devono operare in una rete geografica. Allo stesso tempo, l'overhead associato al mantenimento di garanzie di coerenza forte tra lo stato delle repliche (ad esempio coerenza single-copy [4]) rende questo approccio proibitivo molto velocemente all'aumentare del numero delle repliche coinvolte. Una soluzione effettiva a questo problema si ha rilassando le garanzie di coerenza con l'utilizzo di tecniche di replicazione otimistica (Capitolo 2), che permettono di aumentare sia le performance del sistema che la disponibilità dei dati.

Diversi sistemi distribuiti utlizzano le tecniche di replicazione ottmistica per rilassare le garanzie di coerenza tra le repliche [15, 18, 19, 26] al fine di aumentare le performance e la disponibilità dei dati, ma tutti questi sistemi distribuiti mancano di generalità, nel senso che utilizzano tutti tecniche specifiche per una certa classe di applicazioni.

Le diverse classi di applicazioni che operano su reti geografiche impongono invece due requisiti fondamentali per un sistema distribuito che voglia fornire supporto per lo sviluppo di tali applicazioni fornendo un modello di coerenza debole:

Generalità: la semantica della coerenza delle repliche è specifica per ogni classe di applicazione. Per esempio un editor collaborativo ha una

semantica ben definita, ma totalmente differente da un file system distribuito per quanto riguarda la coerenza delle repliche. Il modello di coerenza fornito da questi sistemi distribuiti dev'essere abbastanza generale e astratto da riuscire a catturare un ampio spettro di semantiche di coerenza.

Praticità: il modello fornito da tali sistemi distribuiti per specificare la semantica della coerenza delle repliche deve essere pratico da poter essere utilizzato senza problemi nella fase di design delle applicazioni. In particolare il protocollo di coerenza fornito deve essere efficiente, indipendente dall'applicazione e deve fornire un modo semplice per esprimere la semantica dell'applicazione.

Questi due requisiti sono solitamente in conflitto tra di loro. Per esempio, per ottenere un sistema generale occorre evitare di definire un modello di coerenza uniforme e fare in modo che le applicazioni possano specificare la propria semantica di coerenza. Dovendo catturare un ampio spettro di semantiche di coerenza, il modello di coerenza fornito da questi sistemi deve essere astratto, il che a volte non fornisce un modo naturale per gli sviluppatori di utilizzare tale modello. e In questo capitolo vengono illustrati alcuni sistemi distribuiti che forniscono supporto per lo sviluppo di applicazioni collaborative in una rete geografica che cercano di rispettare entrambi i requisiti di generalità e di praticità.

3.1 Bayou

Bayou [7, 43] è un toolkit generico sviluppato principalmente per supportare lo sviluppo di applicazioni collaborative in sistemi mobili. Bayou non fa alcuna assunzione sulla connettività dei nodi che compongono il sistema, o sulla topologia della rete. L'unico modo per mantenere coerenti lo stato delle repliche in un ambiente simile è l'utilizzo di tecniche di replicazione ottimistica.

Bayou replica *database*, il termine viene usato per indicare una collezione di dati, non è importante per Bayou che i dati siano memorizzati tramite un database relazionale o in un file system come semplici file. Non viene supportata replicazione parziale, quindi ogni database viene replicato interamente. Bayou utilizza un'architettura client-server. Un server è un computer che mantiene una replica di un databse, mentre i client accedono ai server per le operazioni di lettura e scrittura. Lo stesso computer può essere server per alcuni database e client per altri.

Bayou utilizza tecniche di *fluid replication* [27] per gestire le copie dei database: questo significa che tali copie possono "fluire" nel sistema cambiando grado di replicazione, ovvero creare nuove repliche, e cambiando locazione, ovvero spostandosi in un altro server.

Bayou fornisce due operazioni fondamentali: *read* e *write*. Utilizzando un sottoinsieme del linguaggio SQL l'operazione di read permette di effettuare delle query ad una replica, mentre l'operazione di write permette di inserire, modificare o eliminare degli elementi da una replica. Le singole operazioni di read e write vengono sempre indirizzate ad un solo server, ma un client può connettersi a diversi server durante l'esecuzione.

La comunicazione con diversi server può causare dei problemi di coerenza: ad esempio i diversi server con cui un client entra in comunicazione possono trovarsi in stati differenti, perciò dati a cui si era acceduto in precedenza potrebbero non essere più disponibili, almeno finché il nuovo server non raggiunge lo stesso livello di aggiornamento del precedente. Per risolvere questo problema Bayou fornisce la possibilità di specificare per ogni client delle garanzie di sessione (Capitolo 2.7.1): in questo modo ogni client avrà una vista della replica coerente con le operazioni che ha sottomesso al sistema durante la sessione corrente, indipendentemente da quali server acceda. Naturalmente questo riduce la disponibilità dei dati perché ci si potrebbe trovare a comunicare con un server che non dispone dei dati desiderati: in questo caso l'operazione verrebbe bloccata finché il server non si trova in uno stato tale da poter soddisfare le garanzie di sessione.

Bayou è stato il primo sistema a fornire delle tecniche semantiche di rilevamento e risoluzione dei conflitti (Sezioni 2.4.2 e 2.4.3) basate sulla semantica dell'applicazione. Altri sistemi in precedenza hanno fornito una funzionalità simile, ad esempio [34, 38], ma si tratta di sistemi che gestiscono solo un tipo di dato condiviso (file system distribuiti nei casi citati), mentre Bayou è il primo in grado di gestire diversi tipi di dati condivisi lasciando allo sviluppatore la definizione dei metodi per il rilevamento e la risoluzione dei conflitti. Per fare ciò, in Bayou, ogni operazioni di scrittura incorpora anche due funzioni, chiamate *dependency check* e *merge procedure*.

La funzione di *dependency check* consiste in una serie di query da effettuare sullo stato attuale della replica e un insieme di risultati attesi. Se le query non restituiscono i risultati attesi, allora significa che l'operazione è in conflitto con qualche altra operazione di scrittura precedente. Una volta che viene rilevato un conflitto viene eseguita la funzione *merge procedure* dell'azione corrispondente. Questa funzione risolve il conflitto proponendo degli aggiornamenti alternativi a quello fornito con l'operazione stessa. L'esecuzione della funzione dependency check e dell'operazione di aggiornamento della replica, o alternativamente della funzione merge procedure, avviene atomicamente.

Per raggiungere uno stato coerente, le repliche devono non solo ricevere lo stesso insieme di operazioni, ma anche eseguirle nello stesso ordine. Per la propagazione delle operazioni Bayou impiega un protocollo di propagazione epidemica [8]. Il protocollo funziona mettendo in comunicazione i server a coppie. Ad ogni round una coppia di server si scambia le operazioni di cui l'altro server non è a conoscenza. Questo protocollo assicura che ogni server verrà prima o poi a conoscenza dell'intero insieme di operazioni, anche se alcuni server non possono comunicare direttamente tra loro. Bayou non dàlcuna garanzia sul tempo di propagazione delle operazioni a tutte le repliche del sistema perché il tempo di propagazione dipende dalla connettività delle repliche.

L'ordinamento delle operazioni in Bayou avviene tramite tecniche di sche-

duling sintattico (Sezione 2.4.1). Le operazioni vengono ordinate tramite timestamo, in particolare sono ordinate secondo il momento in cui ogni operazione viene ricevuta dalla prima replica. In questo modo ogni server può ordinare in maniera deterministica le operazioni senza il bisogno di coordinazione tra tutti i server.

Ogni operazione in Bayou può trovarsi in uno di due stati: *provvisoria* o *stabile*. Ogni operazione è inizialmente provvisoria e può quindi essere soggetta a rollback a causa della ricezione di un'altra operazione che deve essere ordinata prima di lei. Bayou utilizza la tecnica di *primary commit* per rendere stabile un'operazione. Per ogni database condiviso esiste un server, designato *primary*, che è responsabile dell'ordinamento delle operazioni stabili. Non appena un'operazione raggiunge il server primario, questa viene dichiarata stabile e ordinata secondo l'ordine di ricezione al server primario. In seguito tramite il protocollo di propagazione epidemica le operazioni stabili vengono propagate dal server primario a quelli secondari. In ogni server le operazioni vengono ordinate ponendo prima le operazioni stabili, secondo l'ordine definito dal server primario, e in seguito le operazioni provvisorie secondo l'ordinamento definito precedentemente.

Bayou cerca, quando possibile, di generare un ordinamento stabile che sia consistente con gli ordinamenti provvisori delle operazioni ad ogni server. Quando un server secondario propaga delle operazioni al server primario, queste vengono in genere ordinate in maniera stabile secondo l'ordine che avevano nel server secondario, in quanto il server primario le riceve ordinate in tal modo. Però l'ordine con cui i diversi server secondari comunicano con il server primario e tra di loro può far sì che alcuni ordinamenti provvisori non vengano rispettati e che le operazioni siano quindi soggette a rollback e rieseguite nel nuovo ordine.

Per lo stesso motivo, Bayou permette ad un insieme di server secondari di lavorare in modo collaborativo tra di loro pur trovandosi scollegati dal server primario. Infatti, quando un insieme di server secondari si trova scollegato dal resto della rete, ed in particolare dal server primario, può continuare con

l'esecuzione dell'applicazione aspettandosi che le modifiche provvisorie che applicano siano poi rese stabili nello stesso ordine. L'ordinamento raggiunto da quest'insieme di server è provvisorio nel senso che può essere differente da quello che verrà effettivamente scelto dal server primario. Tuttavia, se nessun server esterno a questo insieme propaga al server primario delle operazioni in conflitto con quelle generate dal gruppo offline, quando un server del gruppo propagherà le operazioni al server primario, queste verranno ordinate in maniera stabile nello stesso ordine in cui sono state ordinate provvisoriamente dai server secondari.

3.2 TACT

TACT [48] è un middleware che permette alle applicazioni di specificare il livello di coerenza desiderato nell'accesso ad una replica. TACT si infrappone tra la replica e l'applicazione mediandone l'accesso. Se l'operazione richiesta non viola i requisiti di coerenza specificati dall'applicazione, allora viene inoltrata alla replica per l'secuzione, altrimenti viene bloccata finché TACT non ha ristabilito il giusto livello di coerenza.

È un sistema operation-transfer (Sezione 2.3.2) che propaga le operazione tra le repliche utilizzando un protocollo epidemico, una versione ottimizzata del protocollo anti-entropy [8], chiamato *compulsory anti-entropy*. TACT replica *database*, anche in questo caso il termine database viene usato in maniera generica per indicare una collezione di dati. Ogni database viene replicato interamente.

TACT utilizza tecniche sintattiche di scheduling (Sezione 2.4.1), infatti ad ogni operazione quando viene propagata è associato un timestamp composto da un orologio logico [21] e l'identificativo della replica che per prima ha ricevuto l'operazione. Grazie a questo timestamp viene definito un ordinamento completo e ogni replica può ordinare tutte le operazioni in maniera deterministica senza bisogno di comunicare con le altre.

Per il rilevamento e la risoluzione dei conflitti vengono utilizzate tecniche semantiche (Sezioni 2.4.2 e 2.4.3) in una maniera molto simile a quanto accade in Bayou. Anche in TACT, infatti, ogni operazine di modifica incorpora una funzione che controlla se l'esecuzione dell'operazione causa un conflitto.

Le operazioni vengono applicate secondo l'ordine di ricezione e riordinate, secondo l'ordinamento definito precedentemente, alla ricezione di nuove operazioni. Ogni operazione è provvisoria finché la replica non è in gardo di determinare la sua posizione finale nell'ordinamento, dopodiché diventa stabile e non sarà più soggetta a rollback. Anche la determinazione di quali operazioni siano stabili avviene tramite l'uso dei timestamp. Ogni replica mantiene un *logical time clock*, un vettore alla cui i-esima posizione viene memorizzato l'orologio logico dell'ultima operazione ricevuta dall'i-esima replica. La *proprietà di copertura* (coverage property) ci assicura che una replica abbia visto tutte le operazioni con orologio logico minore o uguale al valore più piccolo presente nel logical time clock. Quindi tutte le operazioni con orologio logico minore o uguale al valore più piccolo presente nel logical time clock sono considerate stabili. Il logical time clock viene aggiornato durante le fasi di propagazione delle operazione tramite il protocollo anti-entropy

L'innovazione fondamentale di TACT è quella di fornire la possibilità alle applicazioni di selezionare il proprio livello di divergenza fornendo dei vincoli quantitativi (Sezione 2.7.2): le applicazioni possono così decidere il proprio livello di coerenza in uno spettro che vede ai due estremi: il modello di coerenza single-copy [4] da un lato, e il modello di eventual consistency dall'altro. Il livello di coerenza viene misurato tramite l'uso di alcune metriche, definite in seguito, che sostanzialmente misurano la discrepanza che si osserva tra lo stato di una data replica e lo stato che si avrebbe se fosse stato adottato il modello di coerenza single-copy, chiamato *immagine finale*. L'immagine finale, per ovvi motivi, non viene calcolata veramente ma viene stimata.

L'applicazione specifica il livello di coerenza desiderato tramite l'uso dei cosiddetti *conit* [49]. Un conit (consistency unit) è un unità di dato definita dall'applicazione sulla quale viene misurato il livello di coerenza. Un conit

può essere ad esempio il bilancio di un conto bancario in un'applicazione di contabilità. Il livello di coerenza in TACT viene misurato tramite l'uso di tre metriche:

numerical error: misura la differenza tra il valore del conit nella replica ed il valore del conit nell'immagine finale.

order error: misura la differenza nell'ordine in cui le operazioni sono applicate al conit nella replica locale rispetto all'ordine in cui sarebbero state applicate per ottenere l'immagine finale.

staleness: misura la differenza tra l'orologio reale della replica locale e il tempo reale in cui l'operazione più vecchia nel sistema riguardante un conit presente nella replica, e non ancora ricevuta, è stata sottomessa alla prima replica. La misurazione di questa metrica richiede che gli orologi delle repliche siano lascamente sincronizzati [25] (maggiore il livello di sincronia, migliore la stima della metrica).

Quando il limite di tolleranza su tutte queste metriche è posto a zero, il modello di coerenza ottenuto è quello di coerenza single-copy, quando invece non viene posto alcun limite di tolleranza, il modello di coerenza ottenuto è quello di eventual consistency.

Per stimare il *numerical error*, ogni replica mantiene delle informazioni relative a tutte le altre repliche ricavate durante il protocollo di propagazione delle operazioni. Tramite tali informazioni ogni replica calcola una stima conservativa dell'errore numerico delle altre repliche. Quando una di queste stime supera il livello fissato, la replica invia al server in questione (pushing) le operazioni di cui non è a conoscenza (ricavate tramite il logical time clock memorizzato dalla ogni replica).

Per stimare l'*order error* ogni replica controlla il numero di operazioni provvisorie che possiede. Se queste superano il limite fissato la replica inizia il "protocollo di commitment" per ridurre il numero di tali operazioni. Il protocollo richiede alle altre repliche l'invio delle operazioni di cui la replica locale non è a conoscenza (pulling) tramite un round del protocollo di

propagazione compulsory anti-entropy. In questo modo il logical time clock memorizzato nella replica può avanzare e grazie alla proprietà di copertura la replica diventa in grado di determinare come stabili un maggior numero di operazioni.

La stima della *staleness* di ogni replica avviene in modo simile alla stima dell'order error. Ogni replica mantiene un *real time vector*, un vettore che mantiene nell'i-esima posizione il tempo reale dell'ultima operazione eseguita dall'i-esima replica, di cui la replica locale è a conoscenza. Per limitare l'obsolescenza di una replica ad un dato valore v, ogni replica i controlla che $\textit{current time} - \textit{real time vector}[j] < v$, per ogni $j \neq i$. Se ciò non è vero per alcune repliche j, allora la replica locale i esegue un round del protocollo di propagazione compulsory anti-entropy con tali repliche. In questo modo le repliche si scambiano le operazioni di cui non sono a conoscenza e il real time vector viene aggiornato.

3.3 IceCube

IceCube [16, 32] è un toolkit che supporta lo sviluppo di applicazioni collaborative tra sistemi mobili fornendo il modello di coerenza eventual consistency. È un toolkit di natura generica che permette di sviluppare applicazioni che condividono dati di qualunque natura, siano ad esempio file sorgenti, un file system o un calendario condiviso.

IceCube è un sistema operation-transfer (Sezione 2.3.2). Lo scheduling, il rilevamento e la risoluzione dei conflitti utilizzano tutti tecniche semantiche per ridurre il numero di conflitti e di rollback delle operazioni (Sezioni 2.4.1, 2.4.2 e 2.4.3). Anche IceCube utilizza la semantica dell'applicazione per rilevare e risolvere i conflitti tra le operazioni concorrenti, ma a differenza di Bayou e TACT, non definisce alcun ordinamento sintattico tra le operazioni. La propagazione delle operazioni avviene proattivamente: ogni replica propaga alle altre tutte le operazioni di cui è a conoscenza tramite tecniche di pushing (Sezione 2.6.3).

L'innovazione fondamentale fornita da IceCube è la definizione di vincoli tra le operazioni per determinarne l'ordinamento. IceCube fornisce dei metodi per catturare i *vincoli statici* e i *vincoli dinamici* tra le operazioni. Un vincolo statico mette in relazione una coppia di operazioni e non dipende dallo stato della replica, mentre un vincolo dinamico dipende dallo stato attuale della replica nella quale l'operazione in questione dev'essere eseguita.

Invece di ordinare le operazioni secondo un ordine prestabilito, IceCube cerca un ordinamento delle operazioni che rispetti i vincoli che sono stati imposti tra di esse, in questo modo IceCube può tenere conto della commutatività delle operazioni per limitare il numero di conflitti tra le operazioni e di rollback delle operazioni provvisorie già eseguite.

I vincoli statici tra le operazioni sono costruiti facendo uso di due primitive, `BEFORE`, indicata con $\rightarrow$, e `MUSTHAVE`, indicata con $\rhd$. Dato un insieme di operazioni e vincoli che mettono in relazione tali operazioni, IceCube genera degli schedule di operazioni, ovvero degli insiemi ordinati di operazioni che soddisfano tali vincoli. Ogni schedule s che IceCube produce soddisfa le seguenti condizioni di correttezza:

1. $\forall$ operazione $\alpha\ \beta\ \in\ s$, se $\alpha\ \rightarrow\ \beta$ è un vincolo, allora α si trova prima di β nello schedule, ma non necessariamente immediatamente prima.

2. $\forall$ operazione $\alpha\ \in\ s$ tale che $\exists$ il vincolo $\alpha\ \rhd\ \beta$, allora $\beta\ \in\ s$, in un qualunque ordine e non necessariamente contigue.

Tramite l'uso di queste primitive in IceCube vengono defini i seguenti vincoli, chiamati *log constraint* in IceCube, che mettono in relazione operazioni locali, ovvero operazioni generate dallo stesso client:

predSucc($\alpha,\ \beta$): è equivalente a $\alpha\ \rightarrow\ \beta\ \wedge\ \beta\ \rhd\ \alpha$. Impone che l'operazione β sia inserita nello schedule dopo l'operazione α se e solo se α fa parte dello schedule. Impone la dipendenza causale tra l'operazione α e l'operazione β.

parcel($\alpha,\ \beta$): è equivalente a $\alpha\ \rhd\ \beta\ \wedge\ \beta\ \rhd\ \alpha$. Impone che lo schedule contenga entrambe le operazioni o nessuna.

alternative(α, β): è equivalente a $\alpha \rightarrow \beta \wedge \beta \rightarrow \alpha$. Impone che lo schedule contenga solo una delle due operazioni.

IceCube definisce inoltre i vincoli che mettono in relazione le operazioni concorrenti, ovvero le operazioni generate da client diversi:

mutuallyExclusive(α, β): è una funzione che restituisce il valore `true` se le due operazioni sono mutualmente esclusive. Nel caso la funzione restituisca il valore `true`, tra le operazioni vegono inseriti i seguenti vincoli: $\alpha \rightarrow \beta \wedge \beta \rightarrow \alpha$.

bestOrder(α, β): è una funzione che restituisce il valore `true` se l'operazione α deve essere inserita nello schedule prima dell'operazione β. Nel caso la funzione restituisca il valore `true`, tra le operazioni viene inserito il seguente vincolo: $\alpha \rightarrow \beta$.

Tutte le operazioni che non sono in relazione tramite il vincolo $\rightarrow$ possono essere inserite nello schedule in un qualsiasi ordine. IceCube però definisce dei metodi per dichiarare due operazioni semanticamente commutative, ovvero due operazioni che non hanno side effect tra di loro. In questo modo se un'operazione β deve essere ordinata prima di un'altra operazione α, e queste due operazioni sono semanticamente commutative, non è necessario eseguire il rollback di α. Se due operazioni modificano oggetti differenti, allora sono semanticamente commutative, se invece modificano lo stesso oggetto, la funzione **commute(α, β)** restituisce il valore `true` se le due operazioni sono semanticamente commutative.

Tramite l'uso di questi vincoli le operazioni vengono ordinate dalle singole repliche. Nella fase di generazione degli schedule, quando un'operazione soddisfa i vincoli statici e viene inserita in uno schedule, passa attraverso uno stage di simulazione che ne controlla i vincoli dinamici. Durante questa fase le operazioni vengono eseguite su una copia dello stato della replica.

I vincoli dinamici di un'operazione consistono nelle precondizioni e postcondizioni dell'operazione, cioè delle invarianti che devono essere vere prima e dopo l'esecuzione dell'operazione. Ogni operazione definisce una funzione,

preCondition(), senza side effect, che controlla che le precondizioni dell'operazione siano rispettate nello stato risultante dall'applicazione di tutte le operazioni che precedono quella corrente nello schedule. Se questa funzione restituisce il valore `true`, allora viene eseguita l'operazione nella fase di simulazione, altrimenti l'operazione viene eliminata dallo schedule. Anche l'esecuzione dell'operazione restituisce un valore booleano che indica se le postcondizioni dell'operazione sono valide. Se l'esecuzione dell'operazione non restituisce il valore `true`, allora l'operazione viene eliminata dallo schedule. Quando un'operazione α viene eliminata dallo schedule, oltre ad eliminare ogni side effect che può aver causato, tramite rollback, occorre eliminare dallo schedule anche ogni altra operazione β tale che esista il vincolo $\beta \rhd \alpha$.

I vincoli statici restringono quindi lo spazio di ricerca degli schedule, mentre i vincoli dinamici controllano che l'operazione rispetti le invarianti dell'applicazione nello stato della replica in cui questa dev'essere eseguita.

3.4 Confronto

Tutti e tre i sistemi descritti in questo capitolo sono dei toolkit generici per il supporto allo sviluppo di applicazioni collaborative che operano su reti geografiche e che utilizzano tecniche di replicazione ottimistica. Tutti e tre cercano di rispettare i requisiti di generalità e di praticità, definiti all'inizio del capitolo, fornendo allo sviluppatore delle API per la definizione della semantica della coerenza dell'applicazione in modo semplice e prendendosi carico di tutto ciò che è invece indipendente dall'applicazione. Le parti del sistema che sono indipendenti dall'applicazione sono la propagazione delle operazioni in maniera asincrona malgrado la dinamicità della connettività dei nodi che compongono il sistema e un framework per il rilevamento e la risoluzione dei conflitti tra le operazioni concorrenti, nonché la definizione di un ordine totale, finale, sulle operazioni che faccia convergere lo stato delle repliche.

I sistemi presentati hanno dei punti in comune e delle particolarità che li

distinguono gli uni dagli altri, perché ognuno di questi sistemi pone un'enfasi particolare su un diverso aspetto della replicazione.

Tutti e tre, ad esempio, sono sistemi operation-transfer (Sezione 2.2.1), ovvero le repliche si comunicano una descrizione semantica delle operazioni da applicare piuttosto che un nuovo stato risultante dall'applicazione dell'operazione. Dato che questi sistemi usano una descrizione semantica delle operazioni, anche la fase di rilevamento e risoluzione dei conflitti utilizza tecniche semantiche. IceCube in particolare utilizza tecniche semantiche anche per la fase di scheduling delle operazioni.

Altra caratteristica comune a questi sistemi è il fatto che nessuno di essi fa delle assunzioni particolari sulla connettività dei nodi che compongono il sistema e sulla topologia di rete che formano, e quindi utilizzano tecniche di propagazione delle operazioni epidemiche che assicurano che tutte le repliche ricevano tutte le operazioni immesse nel sistema.

Come già detto, ognuno di questi sistemi pone un'enfasi particolare su un aspetto particolare della replicazione. Bayou, ad esempio, pone l'enfasi sul supporto a sistemi mobili e sulla disponibilità dei dati. Per questo motivo fornisce supporto per delle repliche sempre disponibili che non rifiutano mai nessuna operazione, neanche se completamente scollegate dal resto del sistema, e lo stato delle repliche avanza verso uno stato stabile senza bisogno di coordinazione. Bayou utilizza la tecnica di *primary commit*, secondo cui un server viene designato come "primario", ed è compito di questo server decidere l'ordinamento delle operazioni stabili. Bayou offre la possibilità di definire delle garanzie di sessione, per client, che permettono anche nel caso in cui un client acceda a diversi server nel corso di una sessione di avere una vista comunque coerente dello stato delle repliche alle quali accede. Questa è l'unica caratteristica di Bayou che può limitare la disponibilità dei dati nel caso si acceda ad un server non sufficientemente aggiornato, altrimenti ogni client può sempre continuare la propria esecuzione, ammesso che abbia accesso ad almeno un server, che può comunque risiedere sullo stesso device su cui risiede il client, indipendentemente dalla presenza o meno di altri server

nella rete.

TACT invece pone l'enfasi sulla possibilità per le applicazioni di definire le proprie garanzie di coerenza desiderate, garanzie che possono posizionarsi in uno spettro che vede ai due estremi, il modello di coerenza single-copy da un lato, e il modello di eventual consistency dall'altro. L'idea fondamentale alla base di TACT è che nonostante molte classi di applicazioni tollerino il rilassamento delle garanzie di coerenza, queste possano comunque beneficiare molto dalla possibilità di definire le proprie garanzie ottimali di coerenza. Tutti i sistemi simili sviluppati in precedenza, invece, forniscono delle garanzie di coerenza che stanno ad uno dei due estremi dello spettro di possibilità offerto da TACT. Naturalmente, variando il livello di coerenza anche il livello di disponibilità dei dati varia di conseguenza.

IceCube, invece, pone l'enfasi sull'utilizzo pervasivo della semantica dell'applicazione. Utilizzando la semantica dell'applicazione anche per la fase di scheduling, IceCube è in grado di rilevare solo i conflitti tra le operazioni concorrenti che rispecchiano dei reali conflitti a livello semantico per l'applicazione. IceCube è infatti in grado di riordinare le operazioni in maniera più flessibile rispetto a Bayou e TACT: questo gli permette di cambiare l'ordine di alcune operazioni in modo tale che non generino più un conflitto. In IceCube le operazioni vengono ordinate secondo dei vincoli che l'applicazione pone tra di esse e che definiscono la commutatività tra operazioni, la loro dipendenza causale e preferenze di ordinamento. In definitiva, IceCube è in grado di massimizzare il numero di operazioni eseguite grazie alla possibilità di riordinarle secondo i vincoli semantici imposti. Bayou e TACT, per contro, ordinano le operazioni tramite tecniche sintattiche basate sul timestamp delle operazioni utilizzando la relazione *happens-before* [21]. Tale relazione rispetta la dipendenza causale tra le operazioni in maniera conservativa: ciò significa che viene imposta dipendenza causale anche tra operazioni che non sono semanticamente dipendenti le une dalle altre. In IceCube invece la dipendenza causale tra le operazioni viene inserita sotto forma di vincolo dall'applicazione solo quando due operazioni sono realmente semanticamente

dipendenti. In questo modo IceCube ha molta più liberta nell'ordinamento delle operazioni. Come al solito esiste un tradeoff: questa capacità di IceCube nel riordinare le operazioni viene a scapito di una minor stabilità delle operazioni. L'algoritmo di scheduling si trova infatti, a volte, a dover fare delle scelte arbitrarie per ordinare certe operazioni: per esempio certe operazioni possono essere ordinate arbitrariamente tra di loro, ma il loro ordinamento influenza lo stato finale della replica. Per questo motivo le singole repliche non sono in grado da sole di generare un ordinamento di operazioni stabili, ma occorre un protocollo di consenso tra le repliche che le faccia accordare su queste scelte arbitrarie.

Capitolo 4

Telex

Questo capitolo introduce Telex [2, 3] [1], un toolkit semantico che supporta lo sviluppo di applicazioni collaborative e nomadiche sviluppato nel centro di ricerca INRIA di Paris Roquencourt presso il laboratorio informatico dell'università di Parigi 6 "Pierre et Marie Curie".

Telex è ispirato ad IceCube (Sezione 3.3), dal quale prende molte idee. Utilizzando la terminologia definita nel Capitolo 2, Telex è un sistema di replicazione ottimistica multi-master, permette cioè la sottomissione di operazioni di modifica a più siti, è un sistema operation-transfer, le cui operazioni descrivono quindi la semantica della modifica che devono applicare, sia lo scheduling che il rilevamento a la risoluzione dei conflitti utilizzano tecniche semantiche e le operazioni vengono propagate proattivamente agli altri siti tramite tecniche di pushing utilizzando un algoritmo di comunicazione epidemico. Telex garantisce il modello di coerenza *eventual consistency*.

4.1 Introduzione a Telex

L'implementazione di un'applicazione collaborativa è molto complessa. Lo sviluppatore deve tener conto della comunicazione asincrona tra tutti i siti, della replicazione dei dati, del rilevamento dei conflitti tra operazioni

[1] Il progetto Telex è reperibile all'indirizzo `https://gforge.inria.fr/projects/telex2/`

concorrenti e della loro risoluzione, e del consenso tra tutti i siti. Inoltre l'applicazione deve essere costruita in modo che gli utenti finali non siano confusi dal rollback delle operazioni, che la risoluzione dei conflitti abbia senso rispetto alla semantica dell'applicazione evitando rollback inutili, che gli utenti siano in grado di specificare delle preferenze riguardo la risoluzione dei conflitti stessi e che ci siano delle garanzie sulla coerenza dei dati condivisi.

Le attuali applicazioni collaborative (wiki, editor di testi collaborativi, Version Control Systems) utilizzano tutte degli approcci ad-hoc che non danno garanzie di correttezza nel caso generale e lasciano gran parte del lavoro riguardante la risoluzione dei conflitti all'utente finale.

Telex è un toolkit progettato per supportare la costruzione di applicazioni collaborative che utilizza il modello di replicazione ottimistica [37].

Telex supporta la replicazione ottimistica dei dati in maniera decentralizzata in un sistema distribuito su larga scala. In particolare Telex si prende carico della distribuzione delle operazioni di modifica dello stato, della replicazione dei dati, della persistenza dei dati e garantisce la convergenza dello stato delle repliche. Le API di Telex permettono di specificare parte di questo processo utilizzando la semantica dell'applicazione.

L'approccio di Telex alla replicazione ottimistica è quello classico: le operazioni di modifica dello stato sono memorizzate in maniera persistente sullo storage, vengono in seguito propagate agli altri siti e una volta ricevute eseguite da ciascun sito.

4.2 Componenti fondamentali di Telex

La costruzione di un'applicazione utilizzando il supporto fornito da Telex richiede che lo sviluppatore conosca alcuni concetti e componenti fondamentali di base utilizzati da Telex.

Documento: Un *documento* in Telex rappresenta la più piccola unità di informazione condivisa tra i diversi siti di un sistema distribuito. L'ap-

plicazione interagisce con Telex sottomettendo delle "azioni" ad un determinato documento.

Azione: Un'*azione* è un'operazione atomica che modifica lo stato di un documento. Un'azione è l'astrazione di un'operazione a livello dell'applicazione: è ciò che le diverse istanze di Telex si comunicano per mantenere coerente lo stato delle diverse repliche, perciò solo le operazioni che modificano lo stato delle repliche devono avere una corrispettiva azione Telex.

Vincolo: Un *vincolo* è una relazione semantica tra due azioni. I vincoli concretizzano sia dei vincoli di workflow tra le azioni, sia delle invarianti specifiche dell'applicazione. La Tabella 4.1 illustra nei dettagli i vincoli forniti da Telex e la loro semantica.

Schedule: Uno *schedule* è una sequenza di azioni, locali e remote, che i siti Telex eseguono. Uno schedule può contenere sia azioni provvisorie che azioni stabili.

4.2.1 Vincoli

Nome	Notazione	Significato
`NOT AFTER`	$A \rightarrow B$	A non è mai dopo B in qualunque schedule
`ENABLES`	$A \lhd B$	B in uno schedule implica A nello stesso schedule
`NON COMMUTING`	$A \between B$	tutti i siti devono concordare sul loro ordine
`ATOMIC`	$A \overset{\lhd}{\rhd} B$	A e B eseguono entrambe o nessuna delle due
`CAUSAL`	$A \overset{\lhd}{\rightarrow} B$	B esegue dopo A, sse A esegue
`ANTAGONISM`	$A \overset{\leftarrow}{\rightarrow} B$	A e B non sono mai nello stesso schedule

Tabella 4.1: Vincoli forniti da Telex. (A e B sono azioni arbitrarie)

Telex dispone di 3 vincoli primitivi che possono essere combinati tra loro per ottenere una semantica più ricca.

I vincoli primitivi che Telex fornisce sono:

`NOT AFTER:` $A \rightarrow B$ significa che l'azione A non sarà mai inserita dopo l'azione B in qualunque schedule prodotto da Telex. Questo vincolo non implica che le due azioni saranno consecutive, né implica che saranno presenti entrambe in ogni schedule. Questo vincolo impone solamente che, se uno schedule prodotto da Telex contiene entrambe le azioni A e B, allora l'azione A non sarà dopo l'azione B.
Gli schedule possibili dunque sono: $\{\ldots A \ldots\}$, $\{\ldots B \ldots\}$, $\{\ldots A \ldots B \ldots\}$.

`ENABLES:` $A \lhd B$ significa che la presenza dell'azione B in uno schedule implica la presenza dell'azione A nello stesso schedule. In questo caso non vi è alcun vincolo sull'ordine relativo delle due azioni.
Gli schedule possibili dunque sono: $\{\ldots A \ldots\}$, $\{\ldots A \ldots B \ldots\}$, $\{\ldots B \ldots A \ldots\}$.

`NON COMMUTING:` $A \between B$ (equivalente a $B \between A$) significa che le due azioni non sono commutative, ovvero l'ordine in cui vengono applicate è importante ai fini della convergenza dello stato delle repliche. Questo vincolo dice a Telex di scegliere un ordinamento tra le due azione, non importa quale, e di mantenere lo stesso ordinamento in tutti i siti Telex. Anche in questo caso non vi è alcun vincolo sulla presenza di entrambe le azioni negli schedule.
Gli schedule possibili dunque sono: $\{\ldots A \ldots\}$, $\{\ldots B \ldots\}$, $\{\ldots A \ldots B \ldots\}$, $\{\ldots B \ldots A \ldots\}$.

Componendo i vincoli appena descritti è possibile ottenerne di altri con una semantica più ricca:

`ATOMIC:` $A \overset{\lhd}{\rhd} B$ significa che devono essere eseguite entrambe le azioni o nessuna. L'ordine in cui devono essere eseguite non viene specificato. È la combinazione del vincolo $A \lhd B$ con $B \lhd A$.
Gli schedule possibili dunque sono: $\{\ldots A \ldots B \ldots\}$, $\{\ldots B \ldots A \ldots\}$.

`CAUSAL:` $A \overset{\lhd}{\rightarrow} B$ significa che l'azione B esegue dopo l'azione A se e solo se l'azione A viene eseguita. È la combinazione del vincolo $A \rightarrow B$ con

$A \lhd B$

Gli schedule possibili dunque sono: $\{\ldots A \ldots B \ldots\}$, $\{\ldots A \ldots\}$.

`ANTAGONISM:` $A \overset{\leftarrow}{\rightarrow} B$ significa che le due azioni non possono essere contenute entrambe nello stesso schedule. È la combinazione del vincolo $A \rightarrow B$ con $B \rightarrow A$.

Gli schedule possibili dunque sono: $\{\ldots A \ldots\}$, $\{\ldots B \ldots\}$.

In Tabella 4.1 vengono riportati tutti i vincoli che Telex fornisce, insieme alla loro notazione, che verrà utilizzata in seguito, ed al loro significato.

4.2.2 Conflitti tra azioni

Telex sospetta ogni coppia di azione concorrente di essere in conflitto, ma utilizza la semantica dell'applicazione per affinare il rilevamento dei conflitti.

Per prima cosa, è possibile escludere con sicurezza che certe coppie di operazioni concorrenti generino un conflitto se queste operazioni modificano parti differenti e indipendenti di un documento.

Per riconoscere questi casi le azioni Telex utilizzano l'attributo *keys*. Una chiave (*key*) indica quale parte di un documento l'azione va a modificare. Ogni azione possiede quindi un insieme di chiavi. Solo se l'intersezione delle chiavi di due azioni non è vuota le due azioni possono effettivamente causare un conflitto e solo in questo caso diventa necessario interpellare l'applicazione per sapere se davvero viene generato un conflitto. È compito dello sviluppatore assegnare il giusto insieme di chiavi alle azioni in maniera tale da evitare controlli inutili, ma controllando comunque ogni possibile conflitto reale.

Una volta che una coppia di azioni concorrenti è sospettata di generare un conflitto, Telex usa la semantica dell'applicazione per decidere se effettivamente le due azioni danno luogo ad un conflitto. La semantica viene specificata dallo sviluppatore e Telex la sfrutta invocando la funzione *getConstraint(Action A, Action B)*, definita in una delle interfacce che l'applicazione deve implementare. È compito di questa funzione esaminare le due azioni che gli vengono passate come parametro e decidere se danno luogo o

meno ad un conflitto. In caso di conflitto, la funzione inserisce un vincolo tra le due azioni che servirà a guidare Telex nella risoluzione del conflitto stesso.

4.2.3 Grafo delle azioni e vincoli

Il *grafo di azioni-vincoli* (Action-Constraint Graph, ACG) è la principale struttura dati mantenuta da Telex. Le azioni costituiscono i nodi del grafo, mentre i vincoli ne costituiscono gli archi. Ogni sito mantiene il proprio ACG, che contiene tutte le azioni e i vincoli a lui noti. Ogni documento viene rappresentato con una parte del grafo; generalmente ogni documento dà origine ad una componente sconnessa del grafo, ma a volte queste componenti possono essere collegate tra loro in quanto in Telex è possibile esprimere anche vincoli tra azioni che modificano documenti diversi.

L'ACG è una struttura dati strettamente crescente, ma parti obsolete (che si riferiscono ad operazioni già stabili) possono essere recuperate tramite un algoritmo di garbage collection.

Gli archi di un ACG sono etichettati secondo il vincolo che rappresentano ($\rightarrow$, $\lhd$ o $\nleftrightarrow$), i nodi invece possono essere colorati, bianchi o neri, o non avere nessun colore. I nodi vengono colorati quando le azioni che rappresentano sono operazioni stabili; un nodo è bianco se l'azione che rappresenta deve essere eseguita, è nero se non deve essere eseguita.

Un *taglio solido* è un sotto-grafo dell'ACG privo di conflitti che contiene solo nodi colorati. Tutti i nodi che rappresentano azioni commutative, ovvero prive di vincoli, sono colorati di bianco e possono far parte di ogni taglio solido. Tutte i nodi collegati con archi etichettati `NON COMMUTATIVE` ($\nleftrightarrow$) sono colorati di bianco e possono fare parte di ogni taglio solido perché non possono introdurre conflitti. Se un nodo B, di colore bianco fa parte del taglio, ed esiste un arco etichettato `ENABLES` ($\lhd$) che collega un nodo A col nodo B, allora anche il nodo A fa parte del taglio ed è bianco. I conflitti possono essere introdotti solo da cicli di nodi bianchi collegati tra loro da archi etichettati `NOT AFTER` ($\rightarrow$). Per questo motivo in ogni ciclo di questo tipo almeno un nodo è colorato di nero.

Uno *schedule solido* è un taglio solido del grafo totalmente ordinato. I nodi collegati con archi etichettati `NOT AFTER` seguono l'ordinamento imposto da tale vincolo. I nodi collegati con archi etichettati `NON COMMUTATIVE` vengono ordinati arbitrariamente utilizzando un protocollo di consenso. I nodi collegati con archi etichettati `ENABLES` devono rispettare il vincolo che l'arco rappresenta. I nodi non collegati da alcun arco (azioni commutative) sono ordinati arbitrariamente. Dato un taglio solido sono possibili più schedule solidi equivalenti; questi schedule differiscono per l'ordine delle azioni che li compongono, e per il numero di azioni stabili.

Telex fornisce all'applicazione tutti gli schedule solidi possibili dato il grafo ACG. Il primo schedule fornito è quello che contiene il maggior numero di azioni stabili, i successivi schedule, invece, ne contengono un numero sempre minore. Gli schedule successivi al primo, inoltre, vengono calcolati solo su richiesta dell'applicazione.

4.2.4 Protocollo di consenso

Dato che l'esecuzione dell'applicazione è ottimistica, lo stato della replica locale rimane provvisorio finché tutti i siti interessati allo stesso documento condiviso non hanno raggiunto un accordo sull'insieme di azioni da eseguire e sul loro ordine.

Il protocollo di consenso viene costantemente eseguito in background da tutti i siti Telex. Tale protocollo procede definendo un *prefisso di azioni stabili* monotonicamente crescente, le quali rispettano tutti i vincoli esistenti tra loro. Questo processo può causare il rollback di alcune azioni in alcuni siti, ma le azioni che fanno parte del prefisso di azioni stabili non saranno mai soggette a rollback in alcun sito.

Il protocollo di consenso definisce anche un insieme di azioni che non possono essere eseguite perché violerebbero almeno uno dei vincoli esistenti tra loro.

Il protocollo si basa sulle *proposte*, se ce ne sono, fatte dai siti Telex, che non sono altro che schedule approvati dall'applicazione. Tramite le proposte

gli utenti possono dare indicazioni a Telex sulle azioni da eseguire e quelle da non eseguire. Infatti, a meno di conflitti con altre proposte, Telex calcola i nuovi schedule a partire dalle proposte fornite, cercando quindi di rispettare il più possibile la volontà degli utenti.

Telex calcola il più lungo prefisso comune tra tutti gli schedule proposti. Quando non è possibile trovare degli elementi comuni, Telex compie delle scelte arbitrarie per terminare il protocollo. Lo schedule risultante viene trasmesso a tutti i siti Telex, i quali lo rendono persistente su disco e poi lo trasmettono all'applicazione che si occuperà di eseguirne le azioni contenute.

Questo protocollo è ciò che rende il sistema "eventually consistent", ovvero prima o poi tutte le repliche si accorderanno su uno stato comune, corretto, per ogni documenti condiviso. Gli schedule contengono anche una parte di azioni provvisorie, che possono quindi divergere nei diversi siti. Nel momento in cui, però, ogni sito Telex possiede lo stesso insieme di azioni e vincoli, lo schedule calcolato grazie al protocollo di consenso conterrà soltanto azioni stabili, quindi lo stato delle repliche, che risulta dall'esecuzione delle azioni contenute nello schedule, convergerà.

4.2.5 Viste

Telex offre la possibilità agli utenti di scegliere una particolare *vista* dei documenti che condividono tramite l'utilizzo degli *action filters*.

Le viste sono una proprietà auspicabile per le applicazioni collaborative. Permettono ad esempio ad un utente di ignorare temporaneamente le modifiche remote o quelle ancora provvisorie.

Gli action filters definiscono quali azioni dell'ACG Telex deve escludere quando calcola gli schedule. L'esclusione delle azioni viene fatta sempre rispettando i vincoli esistenti tra le azioni, quindi l'esclusione di un azione causa anche l'esclusione di tutte le azioni che dipendono da essa, ovvero tutte le azioni legate alla prima tramite il vincolo `ENABLES`.

I filtri possono riguardare qualunque attributo di un'azione, si possono definire diversi filtri per ogni documento e possono anche essere aggiunti e

rimossi dinamicamente durante l'esecuzione. I filtri correntemente attivi per ogni documento vengono salvati in maniera persistente da Telex come parte dello stato del documento stesso.

I filtri forniscono anche un modo per escludere permanentemente tutte le azioni generate da un utente. Questa proprietà è utile per escludere gli utenti che risultano avere un comportamento dannoso che comprometterebbe la normale esecuzione dell'applicazione.

4.3 L'architettura di Telex

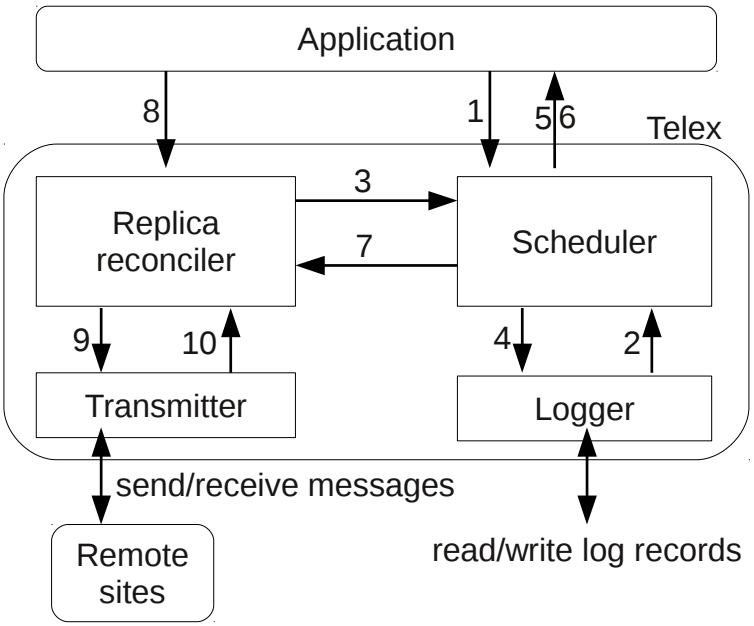

Figura 4.1: Architettura di Telex

In Figura 4.1 viene riportato uno schema dell'architettura di Telex.

In alto sono rappresentate le applicazioni che utilizzano i servizi che Telex fornisce: diverse applicazioni possono essere attive simultaneamente nello stesso sito.

Al centro c'e' Telex, composto di 4 componenti: *replica reconciler* e *scheduler* che sono i due moduli principali; *transmitter* e *logger* che sono invece i due moduli ausiliari.

Ogni applicazione può aprire uno o più documenti; per ogni documento aperto, Telex crea un'istanza di ciascun modulo.

4.3.1 Interazioni

L'interazione tra Telex e l'applicazione consiste nello scambio di parti dell'ACG sotto forma di insiemi di azioni e vincoli dall'applicazione verso Telex o di schedule da Telex verso l'applicazione.

Il ciclo di interazione è il seguente: l'utente interagisce con l'applicazione che traduce le operazioni in azioni e vincoli che trasmette poi a Telex. Telex, dato l'insieme di azioni e vincoli conosciuti localmente, calcola uno schedule e lo trasmette all'applicazione. L'applicazione esegue le azioni presenti nello schedule e presenta lo stato risultante all'utente. Se alcune azioni sono in conflitto tra di loro, allora diversi schedule sono possibili, ognuno corrispondente ad una possibile soluzione dei conflitti. Se l'utente non è soddisfatto dal primo schedule presentato, può richiederne altri.

Il modulo *transmitter* implementa il servizio di multicast atomico tra i siti Telex. I siti Telex utilizzano tale servizio per scambiarsi azioni e vincoli e per l'esecuzione del protocollo di consenso.

Il modulo *logger* si occupa della persistenza delle azioni e dei vincoli sottomessi localmente, che vengono registrati in un log.

4.3.2 Scheduler

Lo scheduler calcola periodicamente gli schedule risultanti dall'insieme di azioni e vincoli conosciuti localmente e li propone per l'esecuzione all'applicazione.

Azioni e vincoli vengono aggiunti all'ACG dalle seguenti componenti:

- dall'applicazione (Figura 4.1, freccia #1), quando l'utente locale esegue delle operazioni che modificano il documento.
- dal modulo logger (Figura 4.1, freccia #2), quando riceve un aggiornamento da un sito remoto.
- dal modulo replica reconciler (Figura 4.1, freccia #3), quando uno schedule viene concordato da tutti i siti Telex (commit).

  Lo scheduler passa le azioni e i vincoli sottomessi localmente al logger (Figura 4.1, freccia #4) che si occupa della loro memorizzazione in maniera stabile sullo storage.

Generazione di vincoli cross-site

Azioni inserite indipendentemente da due diversi utenti possono creare conflitti tra di loro. Quando Telex riceve un'azione da un sito remoto la confronta con tutte le azioni che modificano lo stesso documento per sapere se possono dare origine ad un conflitto tramite il confronto delle chiavi delle azioni (Sezione 4.2.2). Se le azioni possono dare origine ad un conflitto, allora Telex invoca il metodo *getConstraint()* dell'applicazione. Ora l'applicazione si prende carico di controllare se effettivamente le azioni sono in conflitto tra di loro e, in caso positivo, di sottomerre a Telex i vincoli opportuni.

Generazione degli schedule

Generalmente sono possibili diversi schedule dato un ACG. Per non utilizzare troppe risorse, Telex non genera subito tutti gli schedule, ma genera solo il miglior schedule. La metrica di qualità utilizzata è il numero di azioni stabili incluse. Tutti gli altri schedule sono generati dinamicamente su richiesta dell'applicazione.

La generazione dello schedule ottimo è un problema NP-completo, per questo motivo Telex utilizza un'euristica, ispirata da IceCube [31].

L'euristica divide lo spazio di ricerca in sottoproblemi, in questo modo la complessità totale dei sottoproblemi è minore della complessità del pro-

blema originale. Lo spazio di ricerca viene diviso in sottoproblemi disgiunti tali che le azioni in ogni sottoproblema sono commutative con ogni azione in ogni altro sottoproblema, e non esistono vincoli che connettono sottoproblemi differenti. In questo modo lo schedule viene generato cercando in maniera indipendente il miglior schedule in ogni sottoproblema. Il costo computazionale dell'euristica risulta quadratico nel numero di sottoproblemi.

4.3.3 Replica reconciler

Il modulo replica reconciler è il modulo preposto al raggiungimento del consenso, da parte di tutti i siti, sul prefisso di azioni stabili che porterà alla convergenza dello stato delle repliche. Il consenso avviene in background senza interrompere l'esecuzione dell'applicazione.

Ogni sito Telex *propone* uno schedule agli altri siti remoti. Le diverse *proposte* possono differire perché l'insieme di azioni e vincoli conosciuti localmente da ciascun sito può differire. Inoltre, gli utenti stessi possono dare delle preferenze su quali schedule proporre per il consenso.

Questo modulo raggiunge il consenso in quattro fasi:

1. ogni sito genera una proposta in base alle informazioni locali e alle preferenze dell'utente (Figura 4.1, freccia #8).

2. il modulo transmitter manda la proposta a tutti i siti direttamente interessati al consenso tramite un multicast atomico. (Figura 4.1, freccia #9). Il protocollo di multicast atomico utilizzato ha proprietà di liveness in caso di crash o congestione della rete.

3. sempre il modulo transmitter inoltra al modulo replica reconciler le proposte ricevute dai siti remoti (Figura 4.1, freccia #10).

4. il modulo replica reconciler sceglie la proposta vincitrice, informa il modulo scheduler (Figura 4.1, freccia #3) che, oltre a generare nuovi schedule per l'applicazione, informa il modulo logger (Figura 4.1,

freccia #4) che si occupa di memorizzare in maniera persistente la scelta.

4.4 Modello di esecuzione di Telex

Astraendo ad alto livello, il modello di esecuzione di Telex è quello illustrato in figura 4.4.

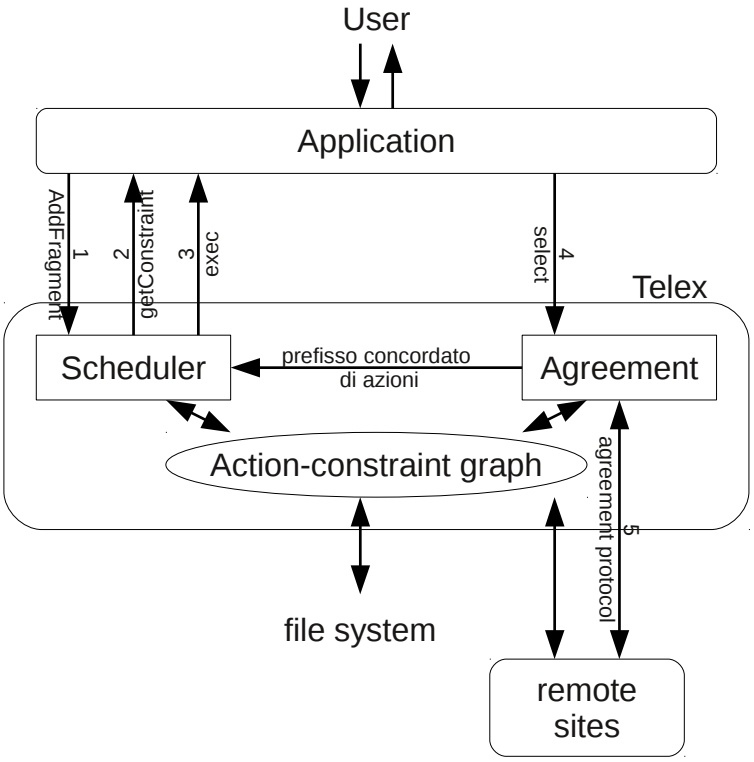

Figura 4.2: Modello di esecuzione di Telex

- L'applicazione apre un documento utilizzando la primitiva *open()* di Telex. Se tale documento è già presente su disco, Telex apre tale copia, altrimenti crea un nuovo documento. Dopo l'apertura del documento, Telex si mette in comunicazione con gli altri siti che hanno aperto lo

stesso documento per ricevere eventuali aggiornamenti avvenuti mentre il sito era offline.

- L'utente esegue un'operazione dell'applicazione. L'applicazione traduce l'operazione nella corrispettiva azione Telex che può essere vincolata ad altre azioni. L'applicazione, tramite la primitiva Telex *addFragment()*, fornisce a Telex un insieme di azioni e vincoli. Telex memorizza tali azioni e vincoli in maniera persistente su disco, li aggiunge al proprio grafo ACG, ed infine le propaga ai siti che replicano lo stesso (o gli stessi) documento a cui si riferiscono.

- Per rilevare eventuali conflitti con un'azione remota, Telex invoca il metodo *getConstraint()* presente nell'interfaccia implementata dall'applicazione. Il metodo viene invocato con una coppia di azioni come input e viene chiamato per ogni coppia di azioni concorrenti sospettate di generare un conflitto.

- Per ogni documento aperto dall'applicazione, Telex calcola uno schedule tale da:

  – includere il prefisso di azioni stabili,

  – contenere tutte le azioni ricevute fino a quel momento,

  – soddisfare tutti i vincoli tra le azioni presenti. In presenza di conflitti Telex fornisce schedule alternativi.

  Telex invoca il metodo *exec()* presente nell'interfaccia implementata dall'applicazione per fornire lo stato di ogni documento condiviso e lo schedule da applicare allo stato appena fornito al fine di ottenere un documento aggiornato con le ultime azioni. Al momento Telex non supporta ancora gli *snapshot* dei documenti, perciò lo stato fornito è sempre lo stato iniziale, ovvero un documento vuoto. Il meccanismo degli snapshot servirà per aumentare le performance delle applicazioni costruite con Telex. Infatti, al momento attuale è necessario rieseguire

tutte le azioni sottomesse dall'apertura del documento ogni volta che si riceve uno schedule; in presenza di snapshot, invece, sarà necessario eseguire solo un sottoinsieme di tutte le azioni.

- Se l'utente finale dell'applicazione è soddisfatto dello stato risultante dall'esecuzione delle azioni presenti nello schedule, l'applicazione invoca la primitiva *select()* di Telex per spingere il sistema ad estendere tale schedule in futuro, piuttosto che generarne di alternativi, e a proporlo come schedule candidato nel protocollo di consenso.

- I siti Telex eseguono il protocollo di consenso iterativamente, estendendo monotonicamente il prefisso di azioni stabili.

4.5 Costruire un'applicazione utilizzando Telex

Per costruire un'applicazione utilizzando Telex occorre specificare l'applicazione secondo il modello di esecuzione ottimistico fornito da Telex. Per fare ciò i passi da seguire sono i seguenti:

1. specificare le strutture dati condivise che l'applicazione utilizzerà.

2. specificare le operazioni che modificano lo stato di tali strutture dati.

3. specificare le relazioni che intercorrono tra le diverse operazioni utilizzando i vincoli forniti da Telex.

Naturalmente si tratta di un processo iterativo; si partirà con una specifica generica delle strutture dati e delle loro operazioni e da qui si potrà iniziare a pensare ai vincoli che intercorrono tra loro. Una volta definiti i vincoli tra le operazioni sarà possibile modificare le strutture dati e le loro operazioni al fine di diminuire il numero di tali vincoli e di aumentare la commutatività tra le operazioni.

4.5.1 Azioni e vincoli di sequenzialità

Consideriamo un'esecuzione sequenziale dell'applicazione, ignorando per il momento la concorrenza e i conflitti tra azioni concorrenti.

La dipendenza causale tra due operazioni si traduce nel vincolo `CAUSAL` ($\overset{\triangleleft}{\rightarrow}$) tra le due azioni che rappresentano le operazioni in questione. Se ad esempio l'azione B legge un valore prodotto dall'azione A, allora occorre introdurre il vincolo $A \overset{\triangleleft}{\rightarrow} B$: in questo modo Telex produrrà uno schedule che contiene l'azione B se e solo se lo stesso schedule contiene anche l'azione A.

In maniera analoga è possibile dire a Telex quando due azioni devono essere considerate come un'unica entità. Se, cioè, le azioni A e B devono essere eseguite entrambe o nessuna delle due: allora occorre introdurre il vincolo `ATOMIC` ($\overset{\triangleleft}{\triangleright}$). In questo caso i vincoli $A \overset{\triangleleft}{\triangleright} B$ e $B \overset{\triangleleft}{\triangleright} A$ sono equivalenti. Il vincolo `ATOMIC` non assicura la proprietà di isolamento nell'esecuzione delle azioni: al momento infatti se delle operazioni devono essere eseguite in isolamento occorre tradurle in un'unica azione Telex.

4.5.2 Concorrenza e vincoli

Come già accennato (Sezione 4.2.2), quando Telex sospetta che due azioni possano generare un conflitto, invoca il metodo *getConstraint()* implementato dall'applicazione. È quindi compito dello sviluppatore definire questo metodo in modo da inserire gli opportuni vincoli tra le azioni secondo la semantica dell'applicazione.

Se due azioni possono essere eseguite in qualunque ordine portando allo stesso risultato allora sono dette commutative. Se due azioni sono commutative non è necessario inserire alcun vincolo tra loro e in questo caso la chiamata del metodo *getConstraint()* non restituirà alcun valore di ritorno.

Se due azioni non possono essere eseguite insieme si dice che sono antagoniste. Questo significa che Telex non deve produrre alcuno schedule che le contenga entrambe. Per fare ciò si utilizza il vincolo `ANTAGONIM` ($\overset{\leftarrow}{\rightarrow}$).

Se due azioni possono essere eseguite entrambe, ma il loro ordine di esecuzione è importante perché cambia lo stato finale dell'applicazione, sono dette non commutative. In questo caso si utilizza il vincolo `NON COMMUTING` (⋔) che fa sì che Telex scelga un ordine di esecuzione arbitrario per le due azioni che verrà mantenuto in tutti i siti Telex.

4.5.3 Vincoli e design dell'applicazione

La scelta dei vincoli giusti tra le azioni è cruciale per la convergenza dello stato delle repliche. Telex fornisce sempre degli schedule corretti rispetto ai vincoli inseriti tra le azioni. Se però la definizione dei vincoli non è corretta rispetto alla semantica dell'applicazione, le repliche possono manifestare degli stati incoerenti tra loro, o addirittura errati.

La corretta definizione dei vincoli tra le azioni è inoltre importante per le performance dell'applicazione e per la percezione da parte dell'utente della responsività dell'applicazione.

Ovviamente le azioni commutative sono da preferire perché non hanno bisogno di alcun controllo da parte di Telex. Anche se non tutte le azioni possono però essere commutative, a volte è possibile farle diventare commutative modificando le strutture dati utilizzate internamente all'applicazione. A volte è sufficiente dividere una struttura dati in strutture dati più piccole in modo che le diverse operazioni che le modificano, e quindi le corrispondenti azioni Telex, non interferiscano tra di loro. Altre volte è invece necessario un redesign completo, e non banale, delle strutture dati [30, 36, 39].

Ogni applicazione ha delle invarianti da rispettare, alcune delle quali non possono essere fatte rispettare tramite l'uso dei vincoli forniti da Telex perché dipendono dallo stato della replica locale al momento della sottomissione dell'azione o della sua esecuzione. Questo significa che prima di eseguire ogni azione l'applicazione deve controllare se la sua esecuzione violerà una delle invarianti. Quando la verifica delle invarianti fallisce, lo schedule che l'ha causata non è sicuro. L'applicazione deve allora chiedere a Telex di

calcolare un altro schedule finché non ne viene generato uno che rispetta tutte le invarianti dell'applicazione.

Il controllo delle invarianti è un meccanismo costoso perché deve essere effettuato dinamicamente al tempo di esecuzione e perché può causare la computazione di nuovi schedule da parte di Telex. Per questo motivo, quando possibile, è meglio definire le invarianti dell'applicazione tramite i vincoli forniti da Telex: in questo modo le azioni che violano tali invarianti vengono escluse dagli schedule tramite un controllo statico.

Capitolo 5

Treeds

Treeds [1] implementa una struttura dati ad albero, replicata, atta a memorizzare informazioni con una struttura gerarchica. Treeds fornisce le operazioni necessarie per inserire ed eliminare elementi dalla struttura e operazioni per modificare la struttura della gerarchia stessa.

Treeds è un esprimento nell'uso delle tecniche di replicazione ottimistica che forniscono il modello di coerenza *eventual consistency* per mantenere in uno stato coerente una struttura dati replicata in un ambiente altamente collaborativo. Il mantenimento in uno stato coerente di strutture dati replicate in ambienti in cui le operazioni di modifica sono molto frequenti pone seri problemi di performance. Approcci che forniscono un modello di coerenza single-copy soffrono il problema della sincronizzazione necessaria tra tutti i siti interessati ad ogni operazione di modifica delle repliche. Le tecniche di replicazione ottimistica, invece, promettono di migliorare le performance del sistema distribuito togliendo la sincronizzazione dal critical path dell'applicazione, che avviene ora in background, pur dando garanzie di convergenza dello stato delle repliche. Il trade off inevitabile viene fatto sul tempo di convergenza dello stato delle repliche, che possono divergere momentaneamente, questo implica che gli utenti possono essere esposti a stati provvisori delle

[1] Treeds fa parte del progetto Telex, tutti i sorgenti sono disponibili all'indirizzo `https://gforge.inria.fr/scm/viewvc.php/trunk/applications/treeds/?root=telex2`

repliche che possono subire rollback.

5.1 Specifiche della struttura dati

Treeds implementa una foresta, ovvero una collezione di strutture dati ad albero. Un albero è un grafo diretto in cui ogni nodo ha al più un arco entrante e un numero arbitrario di archi uscenti. Ogni coppia di nodi è connessa da uno e un solo cammino semplice, ovvero un cammino che non contiene nodi ripetuti.
Una foresta è un grafo diretto le cui componenti connesse sono alberi.

In Treeds ogni albero ha un nodo designato radice, un nodo senza archi entranti. Il *padre* di un nodo è il nodo a lui connesso tramite l'arco entrante; ogni nodo, eccetto la radice, ha un unico padre. Il *figlio* di un nodo è un nodo a lui connesso tramite un arco uscente. Un nodo *foglia* è un nodo senza archi uscenti.

Non esiste nessun ordinamento tra i nodi che compongono un albero in Treeds.

5.1.1 Operazioni

Treeds fornisce quattro operazioni:

create(NodeID x): Crea un nuovo nodo etichettato "x".

remove(NodeID x): Rimuove il nodo etichettato "x".

move-under(NodeID x, NodeID y): Inserisce il nodo etichettato "x" nell'insieme dei figli del nodo etichettato "y".

split(NodeID x): Rimuove il nodo etichettato "x" dall'insieme dei figli del proprio nodo padre. Il nodo etichettato "x" diventa un nodo radice.

read(NodeID x): Restituisce il valore associato al nodo etichettato "x".

Le operazioni *create*, *remove*, *move-under* e *split* sono tradotte in altrettante azioni Telex, l'operazione *read* invece non viene tradotta in un'azione Telex perché non modifica lo stato delle repliche. L'operazione read viene trattata, quindi, come un operazione locale che legge il valore attuale della replica locale al momento dell'esecuzione dell'operazione.

L'operazione *create* crea un nuovo albero composto da un solo nodo. L'etichetta del nodo dev'essere univoca all'interno del sistema.

L'operazione *remove* elimina un nodo. Il nodo non deve avere figli per poter essere eliminato. Se si vuole eliminare un intero sottoalbero occorre invocare quest'operazione su ogni nodo che compone il sottoalbero partendo dalle foglie e salendo fino alla radice del sottoalbero.

L'operazione *move-under* è la prima operazione che permette di modificare la struttura della gerarchia rappresentata da Treeds. Con questa operazione due alberi vengono fusi insieme inserendo la radice di uno dei due alberi come nodo figlio di uno dei nodi che compongono l'altro albero.

L'operazione *split* è la seconda operazione che permette di modificare la struttura della gerarchia rappresentata da Treeds. Quest'operazione divide un albero in due alberi separati.

L'operazione *read* non fornisce alcuna garanzia sul valore che restituisce, non viene cioè assicurato alcuna garanzia di sessione (Sezione 2.7.1). Questo significa che l'operazione *read* può leggere anche valori provvisori che possono quindi essere soggetti a rollback. Se si vogliono fornire delle garanzie di sessione è necessario definire nel modo adeguato le operazioni e imporre i giusti vincoli.

5.2 Invarianti di Treeds

Le invarianti sono un insieme di regole che devono essere sempre rispettate affinchè Treeds operi correttamente. Esistono diversi modi per controllare che le invarianti siano rispettate:

- possono essere assunte come sempre vere.

- possono venire imposte da vincoli.
- possono essere controllate dinamicamente al tempo di esecuzione.

In Treeds la validità delle invarianti viene assicurata da una combinazione di questi tre metodi.

In Treeds sono state individuate le seguenti invarianti:

Identificativi univoci: ogni nodo in Treeds deve avere un identificatore univoco.

Semantica di create: data una sequenza di azioni σ, se l'azione A è parte di σ ed ha effetti sul nodo n, allora l'azione $create(n)$ è inclusa in σ e $create(n) <_\sigma A$.

Semantica di remove: dato un nodo n e una sequenza di azioni σ, se $remove(n)$ è inclusa in σ, allora $\nexists$ un'azione A che abbia effetti sul nodo n tale che $remove(n) <_\sigma A$.

Dipendenza causale: le azioni locali devono essere ordinate se esiste dipendenza causale tra di loro: $\text{action}^1(a) \overset{\lhd}{\rightarrow} \text{action}^2(a) \overset{\lhd}{\rightarrow} \ldots \overset{\lhd}{\rightarrow} \text{action}^n(a)$

Precondizioni: sono state individuate altre invarianti, specifiche per ogni operazione, che ne costituiscono le precondizioni e vengono controllate dinamicamente al tempo di esecuzione, sia quando un'azione viene sottomessa a Telex, sia quando un'azione viene eseguita.

5.2.1 Identificativi univoci

Gli identificativi dei nodi sono assunti univoci. È compito di ogni sito che replica la struttura dati di Treeds quello di generare degli identificativi che siano univoci all'interno del sistema quando un nuovo nodo viene creato.

In un sistema distribuito con garanzie di coerenza debole, quale Treeds, non c'è modo di assicurare questa invariante affidandosi allo stato locale delle

repliche. Ogni replica può divergere momentaneamente, perciò non c'è la garanzia che la replica locale sia a conoscenza di tutti gli identificativi presenti all'interno del sistema. Inoltre, potrebbero esserci delle repliche offline: tali repliche possono continuare la loro esecuzione, quindi anche generare nuovi identificativi, e non c'è modo di saperlo finché tali repliche non ritornano online e le loro azioni sono propagate a tutti i siti.

Per questo motivo è l'applicazione a prendersi carico della generazione degli identificativi univoci. Gli identificativi vengono creati giustapponendo l'identificativo del sito al valore locale del tempo corrente in millisecondi. L'uso del valore del tempo corrente garantisce che ogni sito non generi mai due identificativi uguali, mentre, l'uso dell'identificativo del sito garantisce che due siti differenti non generino mai due identificativi uguali tra di loro. Non occorre che gli orologi dei diversi siti siano sincronizzati perché il tempo corrente viene utilizzato solo per assicurare l'univocità degli identificativi generati localmente.

5.2.2 Semantica di create

Tutte le operazioni che modificano un dato oggetto, nel caso di Treeds un nodo, devono essere eseguite dopo che tale oggetto sia stato creato: questa è la semantica naturale che viene seguita da ogni applicazione. Può però accadere con Telex che un sito riceva un'azione che modifica un dato nodo prima di ricevere l'azione che crea tale nodo.

Dato che in Telex tutte le azioni sono commutative se non diversamente specificato, occorre imporre, tramite l'uso di vincoli, il giusto ordinamento tra le azioni. Per fare ciò, ogni azione viene dichiarata *causalmente dipendente* (`CAUSAL`, $\overset{\triangleleft}{\rightarrow}$) dall'azione *create* che ha creato il nodo (o i nodi) a cui si riferisce.

In termini di schedule generati da Telex, questo significa che se uno schedule contiene un'azione che si riferisce ad un dato nodo n , allora lo stesso schedule contiene anche l'azione di tipo *create* che crea lo stesso nodo n, e l'azione di tipo create si trova prima dell'azione in questione.

Quindi, se un'azione *create* viene abortita, allora anche tutte le altre azioni che si riferiscono allo stesso nodo vengono automaticamente abortite.

5.2.3 Semantica di remove

In maniera simile alla semantica di *create*, la semantica di *remove* richiede che ogni operazione che si riferisce ad un dato oggetto sia eseguita prima che quest'oggetto venga eliminato.

Per imporre questa semantica vengono utilizzati ancora una volta i vincoli forniti a Telex. Ogni azione che si riferisce ad un dato nodo deve essere ordinata *non dopo* (`NOT AFTER`, $\rightarrow$) l'azione di tipo *remove* che elimina tale nodo.

In termini di schedule generati da Telex, questo significa che se uno schedule contiene un'azione di tipo *remove* che elimina un dato nodo n, allora ogni altra azione che si riferisce allo stesso nodo n contenuta nello schedule si trova prima di tale azione remove.

In questo caso l'aborto di una qualunque azione che si riferisce ad un dato nodo n, ad eccezione dell'azione *create*, non causa l'aborto di nessun'altra azione che si riferisce allo stesso nodo n.

5.2.4 Dipendenza causale

Come già detto, in Telex le azioni sono considerate tutte commutative se non diversamente specificato. Questo significa che anche le azioni sottomesse localmente vengono considerate commutative e possono quindi essere riordinate se non viene imposto alcun ordine tra di loro.

A volte questo non è il comportamente che ci aspettiamo perché un'azione può dipendere causalmente da un'altra e quindi deve essere eseguita dopo, e solo se anche la prima azione viene eseguita. Per imporre l'ordinamento di dipendenza causale in Telex si utilizzano ancora una volta i vincoli che vengono messi a disposizione.

Data un'aziona B che dipende causalmente dall'azione A occorre inserire il vincolo $A \overset{\lhd}{\rightarrow} B$. In questo modo le due azioni saranno ordinate nel modo giusto. È importante notare che il vincolo `CAUSAL` non impone la sequenzialità tra le due azioni, che possono quindi essere intervallate da altre azioni.

È anche vero che non tutte le azioni sono causalmente dipendenti tra loro: a volte molte azioni possono essere riordinate senza che lo stato finale dell'applicazione ne risenta. Se due azioni non sono causalmente dipendenti è bene non inserire alcun vincolo tra di loro: in questo modo la computazione degli schedule da parte di Telex risulterà più veloce perché il grafo ACG risulterà più sparso e sarà quindi più facile calcolare un taglio solido del grafo.

Un modo facile per imporre un ordinamento tra le azioni che rispetti la dipendenza causale è quello di dichiarare ogni azione locale causalmente dipendente dalla precedente. In questo modo le azioni locali formano una sequenza totalmente ordinata di azioni secondo il loro tempo di sottomissione. Questo ordinamento rispetta la dipendenza causale tra le azioni, ma allo stesso tempo introduce moltissimi vincoli tra le azioni che possono anche risultare deleteri. Se infatti un'azione venisse abortita, automaticamente anche tutte le azioni successive verrebbero abortite, e se tali azioni non sono realmente causalmente dipendenti dalla prima azione abortita, sarebbero in realtà potute essere eseguite.

Per l'implementazione di Treeds ho scelto un algoritmo abbastanza semplice per imporre l'ordinamento di dipendenza causale tra le azioni (algoritmo 5.1). L'algoritmo implementato rispetta la dipendenza causale tra le azioni introducendo solo pochi vincoli superflui tra le azioni.

L'algoritmo tiene traccia dell'ultima azione locale che ha modificato ogni nodo in Treeds. Quando l'applicazione sottomette una nuova azione l'algoritmo controlla se esiste un'azione che ha già modificato il nodo a cui si riferisce la nuova azione. Se esiste, allora viene introdotto un vincolo `CAUSAL` ($\overset{\lhd}{\rightarrow}$) tra l'azione precedente e quella appena creata, e l'azione appena creata diventa

Algorithm 5.1 *setCausalDependency_Smart(action)*

```
String[] modifiedObjects ← action.getModifiedObjects()
for all modifiedObject ∈ modifiedObjects do
  latestAction ← getLatestAction(modifiedObject)
  if latestAction ≠ null then
    new Constraint(latestAction → action)
  end if
end for
```

l'ultima azione che ha modificato quel dato nodo.

L'algoritmo può essere ulteriormente raffinato. Per esempio diverse azioni di tipo *split* consecutive possono essere riordinate in maniera sicura. Anche diverse operazioni consecutive di tipo *move-under* che inseriscono nodi diversi come figli dello stesso nodo padre possono essere riordinate in maniera sicura. Non è quindi necessario che vengano dichiarate causalmente dipendenti.

Nonostante questo non sia il miglior algoritmo possibile, perché come appena spiegato introduce dei vincoli non necessari, esso è corretto perché rispetta la dipendenza causale tra le operazioni quando questa esiste. Infatti questo algoritmo introduce dipendenza causale tra due azioni, in maniera conservativa, ogni volta che queste azioni vanno a modificare lo stesso nodo, che è l'unico caso in cui due azioni possono essere causalmente dipendenti.

5.2.5 Precondizioni

Qui di seguito vengono elencate le precondizioni specifiche per ognuna delle operazioni fornite da Treeds.

Queste precondizioni sono controllate dinamicamente durante l'esecuzione dell'applicazione e vengono controllate in due diversi momenti: una prima volta nel momento in cui l'azione sta per essere sottomessa a Telex, una seconda volta quando l'azione sta per essere eseguita. In entrambi i casi se il controllo per l'invariante fallisce, l'azione non viene sottomessa a Telex o non viene eseguita.

Il primo controllo viene fatto per evitare che un sito sottometta delle azioni illegali nel sistema. Il secondo controllo invece viene fatto per assicurarsi che l'azione che si sta per eseguire abbia significato nello stato attuale della replica locale. Questo perché anche se un'azione passa il test sull'invariante al momento della sottomissione a Telex, lo stato della replica nel momento in cui l'azione sta per essere eseguita può essere differente dallo stato locale della replica che l'ha sottomessa.

È fondamentale che la funzione che implementa il controllo di queste precondizioni sia una funzione deterministica che prende in input soltanto lo stato locale della replica e che non causi side effects.

Dato che lo stato delle repliche può divergere momentaneamente, la stessa azione può essere abilitata, e quindi eseguita, da alcune repliche e non essere abilitata da altre. Questo, ovviamente, porta ad un ulteriore divergenza nello stato delle repliche. Ciononostante, Telex garantisce che lo stato delle repliche convergerà prima o poi e tutti i siti avranno lo stesso insieme di azioni e vincoli. Quindi prima o poi tutte le repliche abiliteranno lo stesso insieme di azioni.

Le precondizioni che devono essere controllate dinamicamente sono:

remove(NodeID x)

1. il nodo "x" deve esistere nella replica locale.

2. il nodo "x" non deve avere figli.

Il nodo "x" deve esistere nella replica locale e non deve avere nodi figli. Se l'azione *remove* potesse eliminare nodi con figli, allora sarebbe impossibile rilevare conflitti con azioni concorrenti che vanno a modificare uno qualunque dei nodi che compongono il sottoalbero radicato in "x". Come è già stato detto, se un utente desidera eliminare un intero sottoalbero, deve esplicitamente eliminare ogni singolo nodo partendo dalle foglie.

move-under(NodeID x, NodeID y)

1. il nodo "x" deve esistere nella replica locale.
2. il nodo "y" deve esistere nella replica locale.
3. il nodo "x" non deve avere un nodo padre.
4. il nodo "y" non deve far parte del sottoalbero radicato in "x".

Entrambi i nodi "x" e "y" devono esistere nella replica locale. Inoltre il nodo "x" deve essere un nodo radice, cioè senza nodo padre, e il nodo "y" non deve essere nel sottoalbero radicato in "x". Le ultime due invarianti servono ad evitare che si vengano a creare dei cicli nella struttura di Treeds.

split(NodeID x)

1. il nodo "x" deve esistere nella replica locale.
2. il nodo "x" deve avere un nodo padre.

In questo caso le uniche invarianti necessarie sono che il nodo "x" esista nella replica locale e che abbia il nodo padre.

5.3 Conflitti tra operazioni concorrenti

Quando Telex sospetta che due azioni concorrenti possano generare un conflitto chiede all'applicazione, tramite la chiamata alla funzione *getConstraint()*, di controllare se effettivamente le due azioni generano un conflitto e, in caso positivo, di inserire gli opportuni vincoli per risolverlo. Per sospettare che due azioni possano generare un conflitto, Telex controlla l'attributo *keys* delle azioni.

L'attributo key, come già spiegato nella sezione 4.2.2, indica a Telex quale parte del documento l'azione va a modificare: solo se due azioni posseggono una chiave uguale significa che potenzialmente possono generare un conflitto

perché vanno a modificare la stessa parte del documento. Nel caso di Treeds le "chiavi" di un'azione indicano quali nodi l'azione va a modificare, infatti le chiavi sono ricavate applicando una funzione hash agli ID dei nodi che costituiscono i parametri delle azioni.

Lo sviluppatore deve quindi controllare, per ogni possibile coppia di azioni, se queste possono generare un conflitto (nel caso in cui vadano a modificare lo stesso nodo naturalmennte), e in caso positivo deve definire qualli vincoli introdurre per risolvere tale conflitto.

Il risultato di questa fase di design dell'applicazione è una tabella per ogni documento Telex, le cui righe e colonne rappresentano tutte le azioni disponibili e le cui celle contengono i vincoli da introdurre nel caso la coppia di azioni sia eseguita concorrentemente. Dato che le azioni sono concorrenti, non è importante il loro ordine, perciò le tabelle che risultano da questa fase del design saranno in realtà delle tabelle triangolari.

La tabella dei vincoli concorrenti di Treeds è illustrata nelle tabelle 5.1 e 5.2. La tabella è stata divisa in due solo per questioni di spazio.

Come già detto nella sezione 5.2.2, ogni azione concorrente ad un'azione di tipo *create* che vada a modificare lo stesso nodo, inserisce un vincolo di tipo `CAUSAL` ($\overset{\lhd}{\rightarrow}$) tra l'azione di tipo *create* e l'altra.

Come già detto nella sezione 5.2.3 invece, ogni azione concorrente ad un'azione di tipo *remove*, che vada a modificare lo stesso nodo, inserisce un vincolo di tipo `NOT AFTER` ($\rightarrow$) tra l'azione e l'azione di tipo *remove*.

I casi rimanenti sono:

- due azioni concorrenti di tipo *split*.
- due azioni concorrenti di tipo *move-under*.
- un'azione di tipo *move-under* concorrente ad una di tipo *split*.

Due azioni concorrenti di tipo split

Nel caso di due azioni concorrenti di tipo *split* non c'è alcun bisogno di introdurre vincoli perché le azioni sono commutative. Nel caso le due azioni

	create(a^1)	remove(a^1)
create(a^2)	impossible	se $a^2 = a^1$ create(a^1) $\overset{\lhd}{\rightarrow}$ remove(a^2)
remove(a^2)	se $a^2 = a^1$ create(a^2) $\overset{\lhd}{\rightarrow}$ remove(a^1)	commutative
move-under(a^2, b^2)	se $a^2 = a^1 \vee a^2 = b^1$ create(a^2) $\overset{\lhd}{\rightarrow}$ move-under(a^1, b^1)	se $a^2 = a^1 \vee a^2 = b^1$ move-under(a^1, b^1) $\rightarrow$ remove(a^2)
split(a^2)	se $a^2 = a^1$ create(a^2) $\overset{\lhd}{\rightarrow}$ split(a^1)	se $a^2 = a^1$ split(a^1) $\rightarrow$ remove(a^2)

Tabella 5.1: Vincoli tra operazioni concorrenti (pt. 1)

	move-under(a^2, b^2)	split(a^2)
create(a^1)	se $a^1 = a^2 \vee a^1 = b^2$ create(a^1) $\overset{\triangleleft}{\rightarrow}$ move-under(a^2, b^2)	se $a^1 = a^2$ create(a^1) $\overset{\triangleleft}{\rightarrow}$ split(a^2)
remove(a^1)	se $a^1 = a^2 \vee a^1 = b^2$ move-under(a^2, b^2) $\rightarrow$ remove(a^1)	se $a^1 = a^2$ split(a^2) $\rightarrow$ remove(a^1)
move-under(a^1, b^1)	move-under(a^1, b^1) $\asymp$ move-under(a^2, b^2)	se $a^1 = a^2$ move-under(a^1, $b1$) $\asymp$ split(a^2)
split(a^1)	se $a^1 = a^2$ split(a^1) $\asymp$ move-under(a^2, $b2$)	commutative

Tabella 5.2: Vincoli tra operazioni concorrenti (pt. 2)

modifichino lo stesso nodo, e quindi nel caso in cui Telex sospetti un conflitto, possono verificarsi due casi:

le due azioni sono azioni "diverse": questo significa che le due operazioni sono state sottomesse a Telex da repliche con lo stato locale differente; in particolare il nodo passato come parametro dell'azione ha un padre diverso in ognuna delle due repliche.

La divergenza è causata dal fatto che le repliche hanno ricevuto un insieme diverso di azioni al momento della sottomissione dell'azione; in particolare una delle due repliche ha ricevuto un'azione di tipo *split* ed una di tipo *move-under* che hanno fatto sì che il nodo passato come parametro delle due azioni concorrenti abbia un padre differente in ognuna delle repliche.

Il risultato finale dell'esecuzione delle due azioni in entrambe le repliche, sarà quello di far diventare il nodo passato come parametro delle azioni in questione un nodo radice. L'ordine in cui vengono eseguite le due operazioni di tipo *split* non ha importanza: Telex garantisce che prima o poi tutte le repliche riceveranno lo stesso insieme di azioni, perciò prima o poi lo stato delle repliche convergerà.

le due azioni sono la "stessa" azione: questo significa che le due azioni sono state sottomesse da due repliche che avevano lo stesso stato locale, almeno per quanto riguarda il nodo in questione. In questo caso entrambe le repliche vogliono fare la stessa cosa, ovvero rendere il nodo passato come parametro all'azione un nodo radice. La Figura 5.1 mostra questo caso.

In realtà non sarebbe necessario eseguire entrambe le azioni: si potrebbe inserire tra le due azioni un vincolo di tipo `ANTAGONISM` ($\overleftrightarrow{\rightarrow}$). Il problema è che non è possibile distinguere tra questo caso in cui è sufficiente eseguire una sola delle due azioni e il precedente in cui occorre eseguirle entrambe. Per questo motivo non viene inserito alcun vincolo: in questo modo le azioni sono commutative e verranno inserite

entrambe in ogni schedule (almeno che non vengano abortite a causa di altri vincoli).

A questo punto saranno i controlli per le precondizioni delle operazioni che faranno sì che la seconda azione nello schedule, indipendentemente da quale delle due sia, non venga eseguita perché il nodo passato come parametro all'azione sarà già un nodo radice.

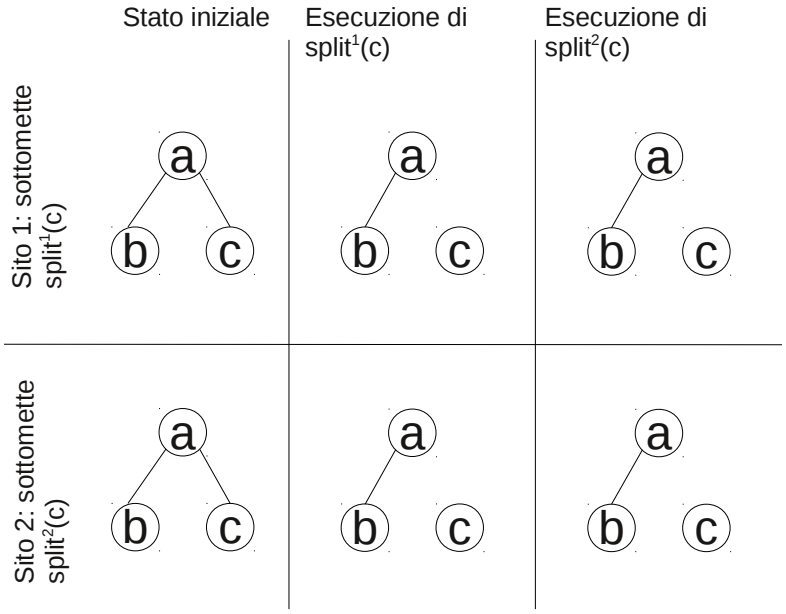

Figura 5.1: Due azioni *split* concorrenti. (Stessa azione)

Due azioni concorrenti di tipo move-under

Nel caso di due azioni concorrenti di tipo *move-under*, le azioni non sono commutative, indipendentemente da quali siano i nodi passati come parametri. Occorre quindi inserire un vincolo di tipo `NON COMMUTING` (⋔) tra le due azioni.

Questo perché l'operazione *move-under* può introdurre cicli all'interno della struttura di Treeds e i cicli che possono così crearsi possono anche essere complessi, non soltanto tra i nodi parametri delle due operazioni.

La Figura 5.2 mostra il più semplice caso in cui due azioni di tipo *move-under* possono creare un ciclo all'interno di Treeds.

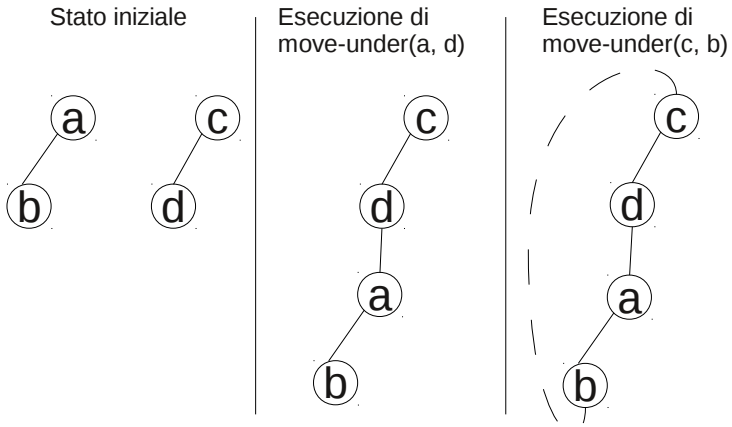

Figura 5.2: Due azioni *move-under* concorrenti. L'esecuzione della seconda azione creerebbe un ciclo in Treeds, per questo motivo i controlli sulle precondizioni dell'operazione falliranno facendo sì che l'azione non venga eseguita.

Inserendo il vincolo di non commutatività tra le azioni, nel caso in cui queste diano luogo ad un ciclo, la seconda azione nello schedule non verrà eseguita perché farà fallire il controllo sulle precondizioni dell'operazione. Dato che tutti i siti Telex avranno le due azioni nello stesso ordine, grazie al vincolo inserito, tutti i siti Telex eseguiranno la stessa azione.

Un'azione di tipo move-under concorrente ad una di tipo split

Nel caso di due azioni concorrenti, una di tipo *move-under* e l'altra di tipo *split*, occorre inserire un vincolo di tipo `NON COMMUTING` ($\not\asymp$) tra le due azioni solo se l'azione di tipo *split* viene applicata allo stesso nodo che compare come primo parametro dell'azione di tipo *move-under*. Come già spiegato nella sezione 5.2.5, quando un'azione viene sottomessa a Telex, le sue precondizioni sono controllate dinamicamente. Dato che un'invariante per l'azione *split* è $parent(x) \neq \emptyset$ e un'invariante per l'azione *move-under* è $parent(x) = \emptyset$, le

due azioni non possono essere state sottomesse dallo stesso sito, ma per forza da due siti diversi con diversi stati locali della replica.

In questo caso entrambi gli ordinamenti possibili tra le due azioni portano alla convergenza dello stato delle repliche, l'importante è che tutte le repliche seguano le azioni nello stesso ordine. L'esempio è illustrato in Figura 5.3.

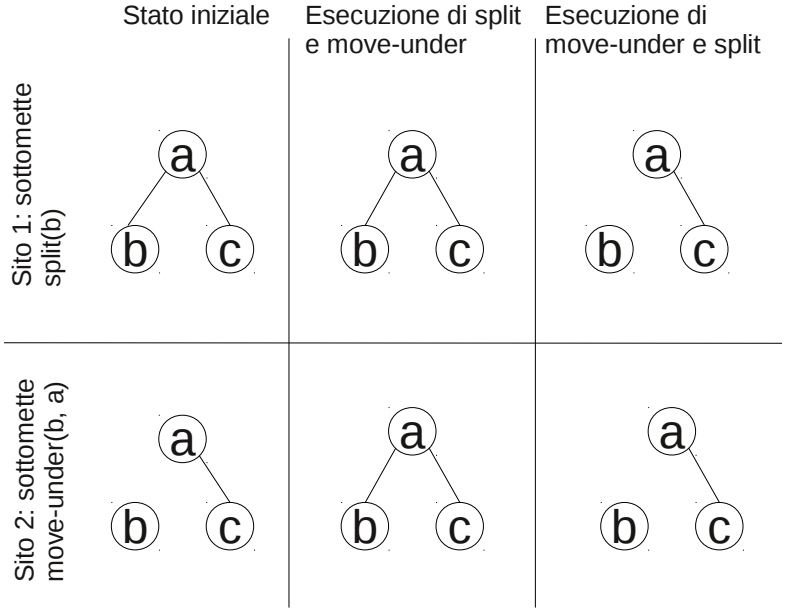

Figura 5.3: Azione *move-under* concorrente ad un'azione *split*. La prima colonna mostra lo stato iniziale dei due siti. La seconda e la terza colonna mostrano invece lo stato finale dovuto all'esecuzione delle due azioni nei due ordini possibili. In entrambi i casi entrambi i siti presentano lo stesso stato finale.

Capitolo 6

Conclusioni

6.1 Esperienza con Telex

Telex è un toolkit che gestisce lo stato delle repliche fornendo il modello di coerenza chiamato "eventual consistency" [47]. Telex supporta lo sviluppo di applicazioni collaborative permettendo allo sviluppatore di concentrarsi sulla logica dell'applicazione, mentre la comunicazione asincrona tra i siti, il rilevamente e la riconciliazione dei conflitti tra operazioni concorrenti viene gestita in background da Telex e l'applicazione può continuare la sua esecuzione normale.

Questo semplifica molto il lavoro dello sviluppatore, ma crea qualche problema iniziale. L'ostacolo fondamentale da superare consiste nel nuovo modello di esecuzione fornito. In un sistema che utilizza un modello di coerenza forte, tutte le azioni sono serializzate ed è possibile pensare all'esecuzione dell'applicazione distribuita come se fosse sequenziale. Utilizzando Telex, invece, ogni operazione è commutativa con tutte le altre operazioni, a meno che non sia propriamente vincolata. Questo porta a due conseguenze fondamentali alle quali lo sviluppatore deve prestare particolare attenzione:

- ogni azione può essere eseguita concorrentemente con ogni altra operazione,
- non esiste un ordinamento prestabilito tra le operazioni locali.

Per questo motivo la definizione delle precondizioni delle operazioni e la definizione dei vincoli tra le operazioni stesse assumono un ruolo centrale nello sviluppo di un'applicazione utilizzando Telex, in quanto dalla loro corretta definizione dipende il corretto comportamento dell'applicazione.

Per Telex le azioni definite dallo sviluppatore sono degli oggetti opachi. Durante la fase di scheduling, Telex prende in input un insieme di azioni e vincoli tra di esse e restituisce in output un elenco ordinato di azioni che rispetta tutti i vincoli forniti in input. Per far sì che le azioni vengano ordinate nel modo corretto è quindi cruciale che i vincoli rispecchino la semantica dell'applicazione: se i vincoli non sono definiti nel modo corretto, l'applicazione potrebbe produrre un comportamento inaspettato. Nel migliore dei casi, un'errata definizione dei vincoli tra le operazioni può causare una maggiore computazione, ad esempio inserendo un vincolo per l'ordinamento di due azioni che in realtà sono commutative, in questo caso lo stato delle repliche si troverà comunque in uno stato coerente, l'unico inconveniente sarà una minore reattività dell'applicazione, proporzionale al numero di vincoli superflui presenti tra le azioni. Nel peggiora dei casi, invece, le repliche potrebbero non convergere ad uno stato comune, ad esempio definendo come commutative due operazioni che in realtà non lo sono.

Occorre tenere a mente il fatto che lo stato delle applicazioni che utilizzano tecniche di replicazione ottimistica, e quindi anche applicazioni sviluppate utilizzando Telex, è anche il risultato dell'esecuzione di operazioni provvisorie, che possono essere soggette a rollback e replay, ovvero rieseguite in uno stato differente dell'applicazione dovuto all'esecuzione di un diverso insieme di operazioni. Per questo motivo, anche considerando solamente l'esecuzione nella replica locale di operazioni generate localmente, può accadere che lo stato in cui si trova la replica al momento in cui l'operazione deve essere eseguita sia diverso dallo stato in cui si trovava nel momento in cui l'operazione è stata sottomessa al sistema. Questa differenza tra lo stato dell'applicazione al momento della sottomissione dell'operazione e al momento della sua esecuzione è ancora più facile da immaginare quando si pensa all'esecuzione

distribuita dell'applicazione in un modello di coerenza in cui è permesso allo stato delle repliche di divergere momentaneamente.

Le precondizioni delle operazioni servono appunto ad evitare che in un sistema come quello appena descritto vengano eseguite delle operazione che non hanno senso nello stato attuale della replica. Anche in questo caso, nella migliore delle ipotesi l'esecuzione di queste operazioni può non comportare alcun problema, ma nella peggiore delle ipotesi può compromettere la corretta esecuzione dell'applicazione.

6.1.1 Definizione di un "processo" per lo sviluppo

Tramite l'esperienza maturata con lo sviluppo di Treeds e i suggerimenti forniti dagli sviluppatori del progetto Telex, ho definito un "processo" per lo sviluppo di applicazioni utilizzando Telex. Si tratta di un workflow iterativo che aiuta nella definizione delle operazioni, delle loro precondizioni e dei vincoli tra di esse.

1. **Identificazione delle risorse** In questa fase lo sviluppatore identifica quelli che sono gli oggetti importanti a livello dell'applicazione.

2. **Definizione delle operazioni** In questa fase lo sviluppatore identifica tutte le operazioni che modificano gli oggetti identificati nella fase precedente in maniera significativa per l'applicazione.

3. **Creazione di una corrispondenza tra operazioni e azioni** Non tutte le operazioni devono necessariamente diventare delle azioni Telex. Le azioni Telex sono ciò che le repliche si comunicano per mantenere il proprio stato coerente, quindi solo le operazioni che modificano lo stato condiviso dell'applicazione devono essere tradotte in un'azione Telex.
   Lo sviluppatore deve inoltre definire con cura i parametri formali di tali azioni: a volte, infatti, potrebbe servire fornire maggiori informazioni all'azione Telex rispetto alla corrispondente operazione perché potrebbero essere utili nella fase di risoluzione dei conflitti.

4. **Identificazione delle precondizioni delle azioni** In questa fase lo sviluppatore identifica tutte le precondizioni delle azioni Telex, anche le più ovvie. Una volta definite lo sviluppatore deve anche decidere il modo in cui queste verranno fatte rispettare, cioè quali verranno supposte sempre vere, quali saranno fatte rispettare tramite dei vincoli e infine quali tramite un controllo dinamico al tempo di esecuzione.

5. **Definizione dei vincoli tra azioni concorrenti** In questa fase lo sviluppatore deve definire i vincoli che vanno imposti per ogni coppia di azioni che potrebbero essere eseguite concorrentemente. In questa fase occorre in genere considerare tutte le coppie possibili tra le azioni che modificano gli stessi oggetti, ma potrebbero esistere anche altri casi in base all'applicazione a e come sono state definite le strutture dati.

   Il modo migliore per la definizione dei vincoli tra le azioni concorrenti è quello della definizione della "tabella dei vincoli concorrenti" (Sezione 5.3). Anche in questo caso è consigliato un approccio iterativo:

   (a) la tabella viene inizialmente costruita inserendo vincoli di non commutatività tra tutte le azioni. In questa maniera si impone a Telex di raggiungere il consenso per l'ordinamento di ogni operazione concorrente, ovvero le operazioni vengono seriliazzate.

   (b) iterativamente si procede al miglioramento dei vincoli definiti, sostituendo i vincoli di non commutatività tra le operazioni, con altri, quando possibile.

Come già detto, questo è un processo iterativo che aiuta lo sviluppatore a definire le giuste strutture dati e le operazioni su di esse. Ad ogni ciclo, lo sviluppatore può affinare le strutture dati definite in precedenza in modo da rendere commutative il maggior numero di operazioni possibili. Infatti, maggiore sarà il numero di operazioni commutative, maggiore sarà la facilità della fase di definizione dei vincoli, ma soprattutto, maggiore sarà la libertà per Telex nell'ordinare le azioni, permettendo così una migliore risoluzione dei conflitti.

6.2 Risultati sperimentali

Treeds è stato testato in due diverse versioni: una versione "*semantica*", in cui viene tenuto conto della semantica delle operazioni per il loro ordinamento, e una versione "*con consenso*", in cui ogni azione concorrente deve essere ordinata tramite il protocollo di consenso tra tutti i siti coinvolti. In questo modo si vogliono valutare i benefici derivanti dall'uso della semantica per la generazione degli schedule di azioni, rispetto a metodi classici che richiedono il consenso da parte di tutti i siti interessati.

La versione "con consenso" di Treeds è stata implementata sempre utilizzando Telex, ma cercando di emulare un comportamento pessimistico nella gestione delle repliche. In questa versione ogni operazione locale è causalmente dipendente dalla precedente operazione locale, e tutte le operazioni concorrenti sono dichiarate non commutative, forzando così Telex al raggiungimento di un consenso da parte di tutti i siti per l'ordinamento di ogni operazione.

Entrambe le versioni di Treeds sono state testate in diverse configurazioni, con 2, 3, 4 e 5 siti. Per ognuna delle configurazioni, inoltre, i siti facenti parte del sistema hanno generato un diverso carico di operazioni (50, 75, 100 e 125 operazioni per ogni sito). Per ogni test, tutti i siti generano lo stesso numero di operazioni in maniera casuale, una ogni decimo di secondo. Quando un sito ha sottomesso tutte le operazioni attende un tempo sufficiente per fa sì che tutte le azioni sottomesse al sistema si propaghino a tutti i siti.

I computer su cui sono stati eseguiti i test hanno tutti processore Intel® Core™ 2 Duo CPU E7500 (2.93GHz), 2GB di RAM e scheda di rete Intel® 82567LM-3 GigaBit Ethernet Controller.

I risultati dei test sono riportati nelle Tabelle 6.1 e 6.2.

I test hanno misurato quattro aspetti dell'esecuzione di Treeds:

- la convergenza dello stato delle repliche,
- il numero di operazioni abortite,

Algoritmo	**Siti**	**Azioni generate**	**Tempo di stabilizzazione (ms)**	**Azioni stabili per secondo**
semantico	2	50	34527	4.86
semantico	2	75	63490	3.93
semantico	2	100	21375	1.48
semantico	2	125	96741	2.58
semantico	3	50	26838	5.94
semantico	3	75	36926	6.83
semantico	3	100	67979	5.06
semantico	3	125	60322	7.27
semantico	4	50	17683	5.4
semantico	4	75	20428	5.38
semantico	4	100	37528	3.27
semantico	4	125	89046	4.93
semantico	5	50	48442	3.73
semantico	5	75	83092	3.74
semantico	5	100	97907	2.4
semantico	5	125	137392	3.59

Tabella 6.1: Risultati sperimentali: Scheduling semantico.

Algoritmo	**Siti**	**Azioni generate**	**Tempo di stabilizzazione (ms)**	**Azioni stabili per secondo**
con consenso	2	50	35062	3.51
con consenso	2	75	45342	3.27
con consenso	2	100	63509	2.53
con consenso	2	125	75325	2.47
con consenso	3	50	36890	5.23
con consenso	3	75	59011	5.31
con consenso	3	100	166614	4.94
con consenso	3	125	65838	4.83
con consenso	4	50	84642	4.37
con consenso	4	75	41372	4.29
con consenso	4	100	59718	3.74
con consenso	4	125	71209	3.65
con consenso	5	50	41515	4.04
con consenso	5	75	42663	3.68
con consenso	5	100	88619	2.89
con consenso	5	125	73923	2.68

Tabella 6.2: Risultati sperimentali: Scheduling con consenso.

- il tempo medio, in millisecondi, impiegato da ogni operazione per diventare stabile,

- il numero di operazioni dichiarate stabili per secondo.

I test eseguiti su Treeds hanno dimostrato sia la correttezza del funzionamento di Telex, sia la correttezza del design e dell'implementazione di Treeds; in particolare i vincoli definiti tra le operazioni sono stati definiti in maniera corretta. Tutti i test hanno infatti portato al convergere dello stato delle repliche.

Data la natura di Treeds, nessuno dei vincoli inseriti tra le operazioni causa l'aborto di altre azioni. Infatti in tutti i test effettuati il numero di operazioni abortite è stato pari a 0, per questo motivo tale risultato è stato omesso dalle tabelle riassuntive.

Il tempo medio impiegato da ogni operazione per diventare stabile, misurato come la differenza tra il tempo in cui un'operazione viene fornita come stabile da uno schedule e il tempo in cui un'operazione viene sottomessa al sistema, non presenta, in genere, un andamento lineare. Questo comportamento può essere causato da due fattori. In primo luogo, i computer utilizzati non erano dedicati esclusivamnte all'esecuzione dei test, quindi il carico di ogni computer era differente e fuori dal controllo dei test, questo può quindi aver causato rallentamenti nell'esecuzione dei test e nelle comunicazioni di rete. In secondo luogo, i diversi carichi di operazioni generati non erano molto diversi tra loro, ed essendo di piccola entità molto probabilmente non hanno forzato l'architettura di Telex verso i proprio limiti e non si nota quindi una differenza sostanziale nei tempi. Ciononostante, si nota in genere che i tempi impiegati dalle operazioni per diventare stabili sono maggiori nelle configurazioni che utilizzano la versione di Treeds "con consenso", proprio perché in questo caso è necessario il consenso tra tutti i siti per decidere l'ordinamento di ogni operazione, forzando quindi la comunicazione via rete. In Figura 6.1 viene mostrato un grafico di tutti i tempi di stabilizzazione dei test effettuati.

Il numero di operazioni dichiarate stabili per secondo è calcolato come il quoziente tra il numero totale di operazioni stabili al termine dell'esecuzione e il tempo totale di esecuzione (wall clock time). Anche in questo caso l'andamento di questa misurazione non è lineare, sempre per gli stessi motivi esposti nel paragrafo precedente. Anche in questo caso si nota una differenza tra le configurazioni che usano la versione di Treeds "semanica" rispetto a quella "con consenso". La versione "con consenso" impiega, infatti, più tempo per generare un'ordinamento delle operazioni dovendo ogni volta interpellare tutti i siti per il raggiungimento del consenso, e questo risulta in un quoziente minore rispetto alla rispettiva configurazione del test eseguita con la versione "semantica" di Treeds. In Figura 6.2 viene mostrato un grafico del numero di azioni stabili per secondo dei test effettuati.

6.3 Sviluppi futuri

Telex è ancora un progetto di ricerca, non è ancora pronto all'uso in produzione e non è ancora del tutto stabile. Per questo motivo i test effettuati utilizzano pochi siti e generano un numero di operazioni relativamente basso. Sono attualmente in corso diversi sforzi per renderlo più stabile e dare quindi la possibilità di eseguire test su larga scala che permettano di stressarne l'architettura fino ai suoi limiti.

Treeds, invece, è stato semplicemente un esperimento per testare Telex oltre i limiti esplorati finora; i precedenti test, infatti, si sono concentrati soprattutto nel dimostrare la capacità di Telex di mantenere le repliche in uno stato coerente e di definire le operazioni di modifica con una semantica sufficientemente ricca da permettere di esprimere diverse classi di applicazioni. Precedenti esperimenti con Telex comprendono: l'implementazione di un *calendario condiviso* [3] e di un *framework per la modellazione condivisa* che supporta lo standard UML 1 [24].

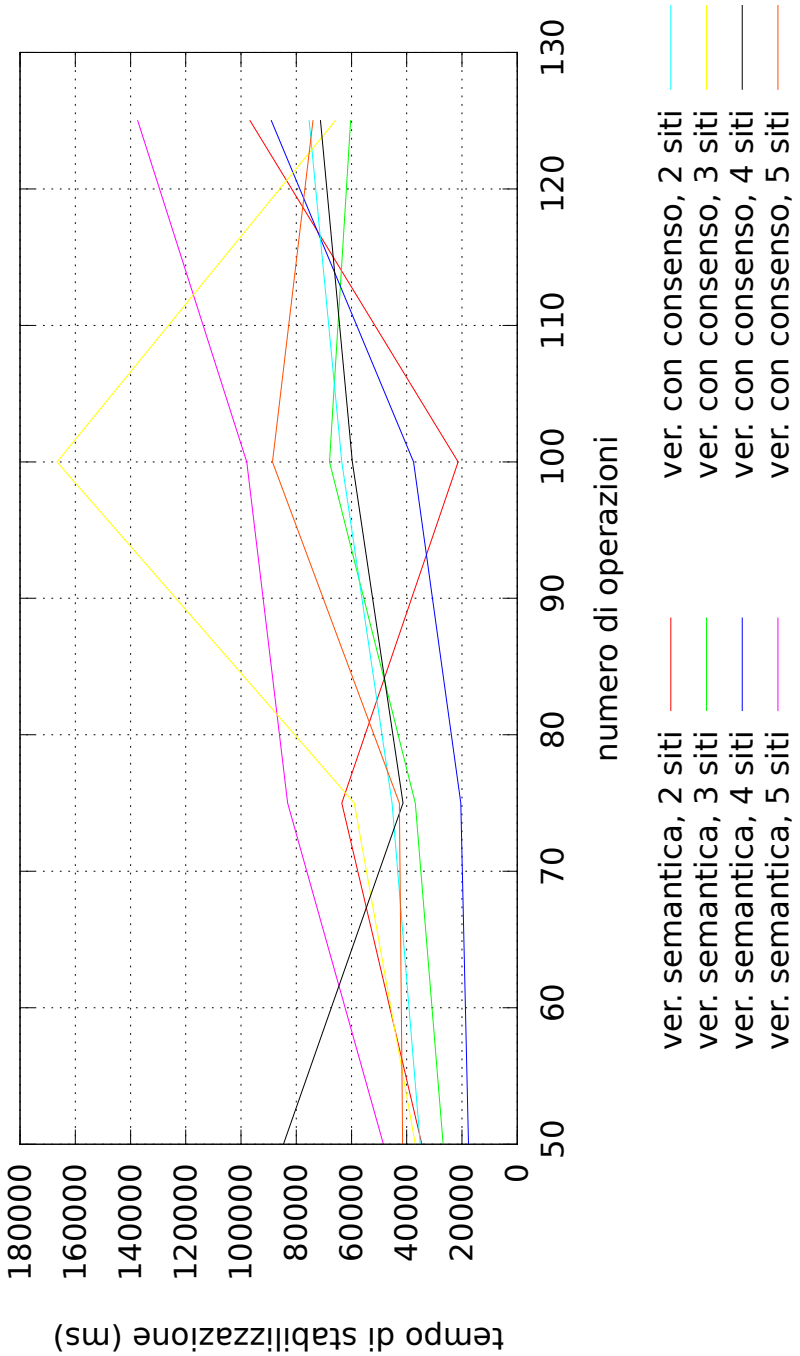

Figura 6.1: Tempo di stabilizzazione delle operazioni (ms).

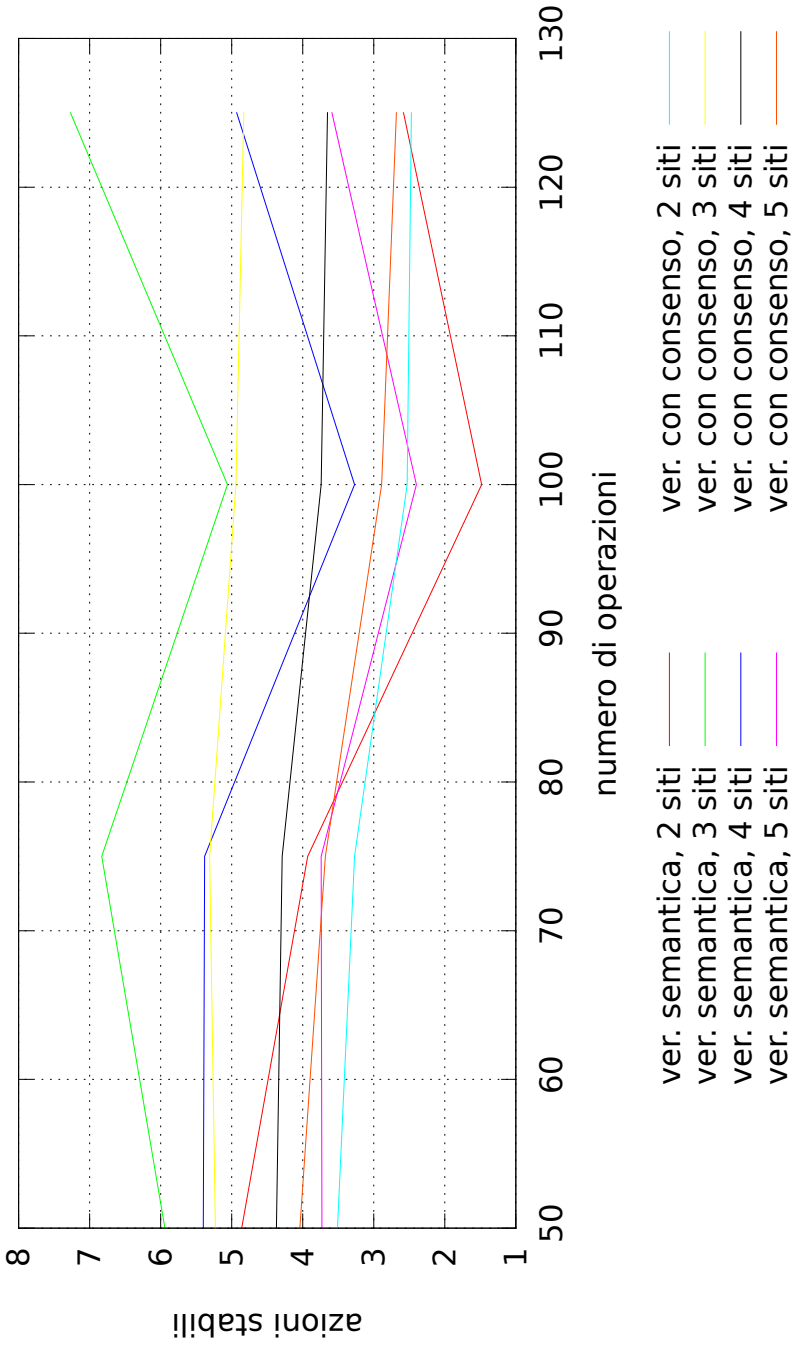

Figura 6.2: Azioni stabili per secondo.

Bibliografia

[1] Rafael Alonso, Daniel Barbara, and Hector Garcia-Molina. Data caching issues in an information retrieval system. *ACM Trans. Database Syst.*, 15:359–384, September 1990.

[2] Lamia Benmouffok, Jean-Michel Busca, Joan Manuel Marquès, Marc Shapiro, Pierre Sutra, and Georgios Tsoukalas. Telex: Principled System Support for Write-Sharing in Collaborative Applications. rr 6546, INRIA, rocq, may 2008.

[3] Lamia Benmouffok, Jean-Michel Busca, Joan Manuel Marquès, Marc Shapiro, Pierre Sutra, and Georgios Tsoukalas. Telex: A Semantic Platform for Cooperative Application Development. In *cfse*, Toulouse, France, sep 2009.

[4] Philip A. Bernstein and Nathan Goodman. The failure and recovery problem for replicated databases. In *Proceedings of the second annual ACM symposium on Principles of distributed computing*, PODC '83, pages 114–122, New York, NY, USA, 1983. ACM.

[5] Eric A. Brewer. Towards robust distributed systems. In *PODC '00: Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*, pages 7+, New York, NY, USA, 2000. ACM.

[6] Michael Dahlin, Bharat Baddepudi V. Chandra, Lei Gao, and Amol Nayate. End-to-end WAN service availability. *IEEE/ACM Trans. Netw.*, 11:300–313, April 2003.

[7] A. J. Demers, K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and B. B. Welch. The Bayou architecture: Support for data sharing among mobile users. In *Proceedings IEEE Workshop on Mobile Computing Systems & Applications*, pages 2–7, Santa Cruz, California, August-September 1994.

[8] Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. Epidemic algorithms for replicated database maintenance. In *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, PODC '87, pages 1–12, New York, NY, USA, 1987. ACM.

[9] C. A. Ellis and S. J. Gibbs. Concurrency control in groupware systems. *SIGMOD Rec.*, 18:399–407, June 1989.

[10] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Commun. ACM*, 19:624–633, November 1976.

[11] C. Fidge. Logical time in distributed computing systems. *Computer*, 24(8):28–33, 1991.

[12] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32:374–382, April 1985.

[13] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33:51–59, June 2002.

[14] R. G. Guy, G. J. Popek, and T. W. Page. Consistency Algorithms for Optimisic Replication. In *Proceedings of the First International Conference on Network Protocols*, 1993.

[15] Paul Johnson and Robert Thomas. Maintenance of duplicate databases, January 1975.

[16] Anne-Marie Kermarrec, Antony Rowstron, Marc Shapiro, and Peter Druschel. The IceCube approach to the reconciliation of divergent replicas. In *PODC '01: Proceedings of the twentieth annual ACM symposium on Principles of distributed computing*, pages 210–218, New York, NY, USA, 2001. ACM.

[17] James J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda File System. *ACM Trans. Comput. Syst.*, 10:3–25, February 1992.

[18] Puneet Kumar and M. Satyanarayanan. Log-based directory resolution in the coda file system. In *Proceedings of the second international conference on Parallel and distributed information systems*, PDIS '93, pages 202–213, Los Alamitos, CA, USA, 1993. IEEE Computer Society Press.

[19] Puneet Kumar and M. Satyanarayanan. Flexible and safe resolution of file conflicts. In *Proceedings of the USENIX 1995 Technical Conference Proceedings*, TCON'95, pages 8–8, Berkeley, CA, USA, 1995. USENIX Association.

[20] Leslie Lamport. The implementation of reliable distributed multiprocess systems. *Computer Networks (1976)*, 2(2):95–114, May 1978.

[21] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21:558–565, July 1978.

[22] Meng Lin and Keith Marzullo. Directional Gossip: Gossip in a Wide Area Network. Technical report, University of California at San Diego, La Jolla, CA, USA, 1999.

[23] Friedemann Mattern. Virtual Time and Global States of Distributed Systems. In M. Cosnard et. al., editor, *Parallel and Distributed Algorithms: proceedings of the International Workshop on Parallel and Distributed Algorithms*, pages 215–226. Elsevier Science Publishers B. V., 1989.

[24] Jonathan Michaux, Xavier Blanc, Pierre Sutra, and Marc Shapiro. A Semantically Rich Approach for Collaborative Model Edition. In *Symp.*

on Applied Computing, volume 26, TaiChung, Taiwan, Mar 2011. ACM SIGAPP, ACM.

[25] David L. Mills. Improved algorithms for synchronizing computer network clocks. *IEEE/ACM Trans. Netw.*, 3:245–254, June 1995.

[26] P. Mockapetris and K. J. Dunlap. Development of the domain name system. *SIGCOMM Comput. Commun. Rev.*, 18:123–133, August 1988.

[27] Brian Noble, Ben Fleis, and Minkyong Kim. A Case for Fluid Replication. In *Proceedings of the 1999 Network Storage Symposium (Netstore)*, Seattle, Washington, USA, October 1999.

[28] Derek C. Oppen and Yogen K. Dalal. The clearinghouse: a decentralized agent for locating named objects in a distributed environment. *ACM Trans. Inf. Syst.*, 1:230–253, July 1983.

[29] Gérald Oster, Pascal Urso, Pascal Molli, and Abdessamad Imine. Data consistency for P2P collaborative editing. In *Proceedings of the 2006 20th anniversary conference on Computer supported cooperative work*, CSCW '06, pages 259–268, New York, NY, USA, 2006. ACM.

[30] Nuno Preguiça, Joan Manuel Marques, Marc Shapiro, and Mihai Letia. A Commutative Replicated Data Type for Cooperative Editing. In *Proceedings of the 2009 29th IEEE International Conference on Distributed Computing Systems*, ICDCS '09, pages 395–403, Washington, DC, USA, 2009. IEEE Computer Society.

[31] Nuno Preguiça, Marc Shapiro, and Caroline Matheson. Efficient semantics-aware reconciliation for optimistic write sharing. Technical Report MSR-TR-2002-52, Microsoft Research, Cambridge, UK, May 2002.

[32] Nuno Preguiça, Marc Shapiro, and Caroline Matheson. Semantics-based reconciliation for collaborative and mobile environments. In *coo-*

pis, volume 2888 of *lncs*, pages 38–55, Catania, Sicily, Italy, nov 2003. springer.

[33] Erhard Rahm. Primary copy synchronization for DB-sharing. *Inf. Syst.*, 11:275–286, October 1986.

[34] Norman Ramsey and Elöd Csirmaz. An algebraic approach to file synchronization. *SIGSOFT Softw. Eng. Notes*, 26:175–185, September 2001.

[35] Peter Reiher, John Heidemann, David Ratner, Greg Skinner, and Gerald Popek. Resolving file conflicts in the Ficus file system. In *Proceedings of the USENIX Summer 1994 Technical Conference on USENIX Summer 1994 Technical Conference - Volume 1*, USTC'94, pages 12–12, Berkeley, CA, USA, 1994. USENIX Association.

[36] Hyun-Gul Roh, Jinsoo Kim, and Joonwon Lee. How to Design Optimistic Operations for Peer-to-Peer Replication. In *Proceedings of the 9th International Conference on Computer Science and Informatics (JCIS/CSI 2006)*, Kaohsiung, Taiwan, October 2006.

[37] Yasushi Saito and Marc Shapiro. Optimistic Replication. *Computing Surveys*, 37(1):42–81, March 2005.

[38] M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel, and D. C. Steere. Coda: a highly available file system for a distributed workstation environment. *Transactions on Computers*, 39(4):447–459, 1990.

[39] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. A comprehensive study of Convergent and Commutative Replicated Data Types. rr 7506, INRIA, rocq, jan 2011. `http://hal.archives-ouvertes.fr/inria-00555588/`.

[40] Chengzheng Sun and Clarence Ellis. Operational transformation in real-time group editors: issues, algorithms, and achievements. In *Proceedings*

of the 1998 ACM conference on Computer supported cooperative work, CSCW '98, pages 59–68, New York, NY, USA, 1998. ACM.

[41] Chengzheng Sun, Xiaohua Jia, Yanchun Zhang, Yun Yang, and David Chen. Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems. *ACM Trans. Comput.-Hum. Interact.*, 5:63–108, March 1998.

[42] Chengzheng Sun, Yanchun Zhang, Yun Yang, and David Chen. A consistency model and supporting schemes for real-time cooperative editing systems. In *Proceedings of the 19th Australian Computer Science Conference*, pages 582–591, 1996.

[43] Douglas Terry, Marvin Theimer, Karin Petersen, Alan Demers, Mike Spreitzer, and Carl Hauser. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *In Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 172–183, 1995.

[44] Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike Spreitzer, Marvin Theimer, and Brent W. Welch. Session Guarantees for Weakly Consistent Replicated Data. In *Proceedings of the Third International Conference on Parallel and Distributed Information Systems*, PDIS '94, pages 140–149, Washington, DC, USA, 1994. IEEE Computer Society.

[45] Robert H. Thomas. A Majority consensus approach to concurrency control for multiple copy databases. *ACM Trans. Database Syst.*, 4(2):180–209, June 1979.

[46] Nicolas Vidot, Michelle Cart, Jean Ferrié, and Maher Suleiman. Copies convergence in a distributed real-time collaborative environment. In *Proceedings of the 2000 ACM conference on Computer supported cooperative work*, CSCW '00, pages 171–180, New York, NY, USA, 2000. ACM.

[47] Werner Vogels. Eventually consistent. *Communications of the ACM*, 52(1):40–44, January 2009.

[48] Haifeng Yu and Amin Vahdat. Design and evaluation of a continuous consistency model for replicated services. In *Proceedings of the 4th conference on Symposium on Operating System Design & Implementation - Volume 4*, OSDI'00, pages 21–21, Berkeley, CA, USA, 2000. USENIX Association.

[49] Haifeng Yu and Amin Vahdat. Combining Generality and Practicality in a Conit-Based Continuous Consistency Model for Wide-Area Replication. In *Proceedings of the The 21st International Conference on Distributed Computing Systems*, pages 429–, Washington, DC, USA, 2001. IEEE Computer Society.

[50] Haifeng Yu and Amin Vahdat. The costs and limits of availability for replicated services. *SIGOPS Oper. Syst. Rev.*, 35:29–42, October 2001.

[51] Haifeng Yu and Amin Vahdat. Minimal replication cost for availability. In *Proceedings of the twenty-first annual symposium on Principles of distributed computing*, PODC '02, pages 98–107, New York, NY, USA, 2002. ACM.

[52] Y. Zhang, V. Paxson, and S. Shenker. The Stationarity of Internet Path Properties: Routing, Loss, and Throughput. Technical report, ACIRI Technical Report, 2000.