MOBILITY MANAGEMENT IN PCS NETWORKS

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Qing Tian
May 2002
I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

______________________________
Donald C. Cox
(Principal Advisor)

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

______________________________
Jennifer Widom

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

______________________________
Joseph W. Goodman

Approved for the University Committee on Graduate Studies:
Abstract

We take the stance that a Universal Personal Identification (UPI) will become an integral part of future personal communication systems so as to better serve the ever increasing communication needs across heterogeneous networks and through a variety of diverse communication devices. Under such context, the relevant problem to be solved is how to locate a user given his UPI, i.e., the problem of efficient tracking and retrieving of a mobile user’s location information using his UPI.

In wireless communication systems, the network keeps track of a user’s location through an up-to-date user profile stored in various databases. A user profile contains not only a user’s current location information, but also service information, such as billing and authentication. The coverage area of an access network is divided into registration areas (RAs), and each RA is associated with a location database. The two basic operations in mobility management are location update and location lookup. When a user moves across the boundaries of these RAs, the network updates his location information in the pertinent databases. When a caller places a call using the callee’s identification, the network queries the relevant database(s) to obtain the current location and other service information of the callee. Current standards (IS-41 and GSM) use a geographical phone number as the callee’s identification. This geographical phone number directly points to a home location register (HLR) that serves as a location database in these standards. Geographical numbering, however, ties a user’s identification to a particular network and geographical location, therefore it can not support UPI.

Hierarchical mobility management techniques (MMTs) have been proposed to support UPI. A hierarchical structure scales better because location information is
distributed, and there is no bottleneck in the system. Hierarchical structures can also exploit locality in user calling and mobility patterns. In addition to using a hierarchical structure, the performance of MMTs can be further enhanced by profile replication, which makes profile information more readily available, thus reducing lookup cost and latency. But to keep these replicas consistent and fresh, they must be updated whenever the user profile is updated. The principle of replication is to replicate if the benefit of replication is greater than its overhead.

A replication algorithm can be either off-line or on-line. While an off-line algorithm assumes complete knowledge of user calling and mobility statistics, an on-line algorithm does not make any assumption about user traffic patterns. Instead, it decides whether to distribute new replicas or delete existing replicas after serving each request, all based on the input sequence seen so far. Previously proposed profile replication algorithms are threshold-based. These algorithms treat requests as isolated events, and do not consider the network structure and communication link costs. Our optimal off-line replication algorithms minimize the network messaging cost based on the network structure, communication link costs, and user calling and mobility statistics. We develop optimal off-line replication algorithms for both unicast and multicast replica updates. Our on-line replication algorithms dynamically adjust replica placement according to user calling and mobility patterns. We not only generalize previously proposed on-line algorithms with a unified framework, but also effectively reduce the overhead incurred by these algorithms, hence making a family of on-line algorithms feasible in practice.

We also propose two MMTs to support integrated personal communication services from heterogeneous networks using UPI. We first augment the HLR/VLR with Meta-HLR databases to map a UPI to multiple HLR addresses. To solve the scalability problem in this centralized approach, we propose a shared hierarchical MMT that not only scales better but also takes advantage of locality in user calling and mobility patterns.

The performance of our proposed MMTs is studied via large scale computer simulations. Both our off-line and on-line replication algorithms are optimal and perform better than previously proposed threshold-based algorithms.
Acknowledgments

This dissertation would not have been possible without the help of many individuals. First and foremost has been the support from my thesis advisor Professor Donald C. Cox. He allowed me into his group, has given me the encouragement to continue in my thesis topic and pushed me along when I came upon stumbling blocks, and has made the ride much more enjoyable through his devotion to trains, sporadic group outings to soda fountains, and gentle introduction to Americana. And even now, at the time of writing, has been nudging me to find a job in industry.

As with all dissertations, the unsung heroes are the ones who actually have to read through and listen to my thesis. These men and women of valor are Professors Nick Bambos, Joseph W. Goodman, Fouad A. Tobagi, and Jennifer Widom.

And as with all dissertations, there are those who have planted the seed and given it water to grow. For breaking the ground, I would like to thank the students and faculties who started the Pleiades project. And for keeping it alive, I would like to thank NSF for funding this project.

Beyond these immediate factors contributing to the completion of this dissertation, there are those factors which have made my presence here at Stanford possible. I am very grateful to Dr. Rick Bogart and the Solar Physics Group at Stanford for their generous support during my master study.

And there are those factors which affect our mental stability in the workplace. I would like to thank my team of psychiatrists, psychologists, and psychopath at the wireless research group: Bora Akyol, Shin-Shiuan Cheng, Sung Chun, Kerstin B. Johnsson, Byoung-Jo Kim, Persefoni Kyrtsi, Derek Lam, Andy Lee, Yumin Lee, Angel Lozano, Yasamin C. Mostofi, Ravi K. Narasimhan, Hichan Moon, Dana Porrat,
Tim Schmidl, W. Mark Smith, Mehdi Soltan, Jeff Stribling, Qinfang Sun, Raymond Wang, Bill Wong, and Daniel Wong.

And there are those factors which get the tedious jobs done. I would like to thank our Administrative Assistants Marli Williams and Jenny Zheng Xu for taking care of endless piles of paperwork.

And there are those factors which show us how to have fun: Liying Tian for being my very first friend at Stanford, Ming Yan Zhu for teaching me how to read stereogram pictures, and Kevin Yu for showing me there is an alternative to PBS (mind-altering TV).

And there are those factors which get our blood circulating: aerobics instructors Mary K. O’Connell and Keren Friedman.

And there are those factors which gives us a hint of a greater purpose: friends from Chinese Christian Fellowship at Stanford, David Banjerdponchaisai, Gary Chan, Phoebe Chen, Puay Guan Goh, Joshua Hui, David Hsu, Shi-Wei Liao, Lai-Chee Man, Tina Seto, Alice Tull, Mark and Libby Verber, Linda Wong, and Liya Yu.

And those factors which have to take blunt of our odorous selves: my roommates at Stanford, Eunjin Oh and Sumeet Sandhu.

Finally, there are those which cannot be pigeonholed into any category, because their influence crosses many boundaries: my parents for never giving me up and for their love toward my sister and me, my hubby, Frankie, for being BB for me, and God for His loving kindness.
# Contents

Abstract iv

Acknowledgments vi

1 *Introduction* 1

1.1 Dissertation Outline 3

1.2 Contributions 4

2 *Mobility Management Techniques* 7

2.1 Basics 7

2.1.1 Wireless Network Architecture 7

2.1.2 User Profile 9

2.2 Mobility Management Techniques 9

2.2.1 A Perspective 9

2.2.2 HLR/VLR 15

2.2.3 Hierarchical MMT 19

2.3 Profile Replication 24

2.3.1 Motivations 24

2.3.2 System Model 25

2.3.3 Issues 27

2.4 Related Work 30

2.5 Simulation Environment 31

2.5.1 Topology 31

2.5.2 Traffic model 31

viii


3 Off-line Profile Replication

3.1 Related Work

3.1.1 Replication Algorithms for Mobility Management

3.1.2 General Location Problems for Tree Networks

3.2 Optimization Objectives

3.3 Replication for Unicast Replica Update (UR)

3.3.1 1-median Problem

3.3.2 Uncapacitated Facility Location Problem

3.3.3 p-median Problem

3.3.4 Discussion

3.4 Replication for Multicast Replica Update (MR)

3.4.1 General MR Algorithm

3.4.2 Modified MR Algorithm

3.4.3 Discussion

3.5 Comparison Between UR and MR

3.6 Simulations

3.6.1 Network Cost

3.6.2 Database Access

3.6.3 Replica Distribution and Access

3.6.4 Discussion

3.7 Summary

4 On-line Profile Replication

4.1 Related Work

4.1.1 A Replication Algorithm for Mobility Management

4.1.2 General File Allocation Problems for Tree Networks

4.2 Edge Algorithms

4.2.1 Problem Description
4.2.2 Framework ............................................. 87
4.2.3 Replica State Assignments ............................... 94
4.2.4 Discussion ............................................. 117

4.3 Tree Algorithms ........................................... 121
  4.3.1 Problem Description .................................. 121
  4.3.2 Factoring ............................................. 122
  4.3.3 Offset Distribution ................................. 123
  4.3.4 Sufficient Conditions ............................... 125
  4.3.5 Propagation Cost Reduction ......................... 139
  4.3.6 Discussion ............................................. 155

4.4 Implementation Issues .................................... 156

4.5 Simulations ............................................... 159
  4.5.1 Network Cost ........................................ 160
  4.5.2 Database Access .................................... 163
  4.5.3 Storage ............................................. 170
  4.5.4 Discussion ............................................. 171

4.6 Summary .................................................. 173

5 Mobility Management For Heterogeneous Networks 174
  5.1 Access Network Model ................................... 174
    5.1.1 Multiple Registration Scheme ....................... 175
    5.1.2 Access Control of Location Information ........... 176
    5.1.3 System Model ...................................... 177
  5.2 Related Work ........................................... 177
  5.3 Mobility Management Techniques ......................... 179
    5.3.1 Meta-HLR MMT .................................... 179
    5.3.2 Shared Hierarchical MMT .......................... 180
  5.4 Simulations ............................................. 184
  5.5 Summary ................................................ 188

6 Conclusions ............................................ 193
  6.1 Summary ............................................... 193
6.2 Future Work .................................................. 195

Bibliography ....................................................... 197
List of Tables

3.1 Network cost in Examples I and II. ................................. 67
4.1 Cost functions $\text{serv}(s, \sigma_i)$ and $\text{tran}(s', s)$ .... 86
4.2 Work function $W_i$, where $W_{i-1}(b) = W_{i-1}(a) + k$ and $W_{i-1}(ab) = W_{i-1}(a) + l$. The $\text{opt}_i$’s are highlighted. ........................................... 88
4.3 Offset change upon $\sigma_i$, where $w_{i-1}(a) = 0, w_{i-1}(b) = k, w_{i-1}(ab) = l$. 92
4.4 $\Delta\text{opt}_i$ values upon $\sigma_i$, where $w_{i-1}(a) = 0, w_{i-1}(b) = k, w_{i-1}(ab) = l$. 95
4.5 State transitions for algorithm $A_L$. ................................. 102
4.6 Inequality (4.1) holds for all possible replica state transitions prescribed by algorithm $A_L$. ................................. 102
4.7 Inequality (4.1) holds for all combinations of input requests, offsets, and replica states in algorithm $A_L$. ................................. 103
4.8 State transitions for algorithm $A_T$. ................................. 103
4.9 Inequality (4.1) holds for all possible replica state transitions prescribed by algorithm $A_T$. This table is identical to Table 4.6 except for the replica state transition for $w_i = (0kl)$, where $0 < k < D$. ................................. 104
4.10 Inequality (4.1) holds for all combinations of input requests, offsets, and replica states in algorithm $A_T$. This table is identical to Table 4.7 except for the forbidden state $b$ when $w_i = (0kl)$, with $k > 0$. ................................. 105
4.11 State transitions for algorithm $A_H$. ................................. 105
4.12 Inequality (4.1) holds for all possible replica state transitions prescribed by algorithm $A_H$. The additional state transitions are made possible by the potential function defined in (4.5) such that $\Phi(a) - \Phi(ab) = 0$ for $l = D$. ................................. 107
4.13 Inequality (4.1) holds for all combinations of input requests, offsets, and replica states in algorithm $A_R$. This table is similar to Table 4.7 except for the underlined entries. .......................... 108
4.14 Example 4.4. .......................... 108
4.15 Trans(·) function .......................... 111
4.16 Offset change upon $\sigma_i$, where $0 < d < D$, $w_{i-1}(a) = k_a$, $w_{i-1}(b) = k_b$, $w_{i-1}(ab) = l$. .......................... 111
4.17 State transitions for algorithm $A_d$ when deletion cost $d \neq 0$. .......................... 114
4.18 Inequality (4.1) holds for all possible replica state transitions prescribed by algorithm $A_d$. .......................... 116
4.19 Inequality (4.1) holds for all combinations of input requests, offsets, and replica states in algorithm $A_d$. An offset can be in any of the four possible forms: (1) $(0k_bl)$, where $1 \leq k_b \leq l + d$ and $1 \leq l \leq D$. (2) $(00l)$, where $0 \leq l \leq D$. (3) $(k_a0)$, where $1 \leq k_a \leq k_b \leq d$. (4) $(0k_b0)$, where $1 \leq k_b \leq d$. .......................... 116
4.20 Theorem 4.9: Algorithm $A_L$ is 3-competitive with the cost of request propagation added. .......................... 141
4.21 Theorem 4.9: Algorithm $A_T$ is 3-competitive with the cost of request propagation added. .......................... 142
4.22 Theorem 4.9: Algorithm $A_d$ is 3-competitive with the cost of request propagation added. $(00l)$, where $0 \leq l \leq D$. .......................... 143
4.23 Theorem 4.9: Algorithm $A_d$ is 3-competitive with the cost of request propagation added. $(k_a0)$, where $1 \leq k_a \leq k_b \leq d$. .......................... 143
4.24 Theorem 4.9: Algorithm $A_d$ is 3-competitive with the cost of request propagation added. $(0k_b0)$, where $1 \leq k_b \leq d$. .......................... 144
List of Figures

2.1 Wireless network architecture. ............................................. 8
2.2 Hierarchy based on regional matching. ................................. 14
2.3 Location update in HLR/VLR. ............................................. 17
2.4 Location lookup in HLR/VLR. ............................................. 18
2.5 Database hierarchy. ......................................................... 21
2.6 Location update in Hierarchical MMT. User $i$ moves from RA $X$ to RA $Y$. The dotted lines delineate the propagation paths of an update request. ................................................................. 22
2.7 Location lookup in Hierarchical MMT. Caller $j$ in DB($j$)'s coverage area attempts a call to callee $i$ in DB($i$)'s coverage area. The dotted lines delineate the propagation paths of a lookup request. ............ 23
2.8 System model. ..................................................................... 26
2.9 Lookup the closest replica. $\hat{v}$ is the closest replica to $v$, such that $e(v, \hat{v}) = \min_{r \in R} (e(v, r))$. ................................................. 27
2.10 Update all replicas. ........................................................... 28
2.11 Simulation topologies and database hierarchies. ..................... 32
2.12 Traffic volume. .................................................................. 34
2.13 Traffic volume variations. ................................................... 35
2.14 Off-line and on-line simulations. ......................................... 37

3.1 Lemma 3.1. ........................................................................ 45
3.2 Lemma 3.2. ........................................................................ 47
3.3 UFL: Minimum cost algorithm. ............................................ 49
3.4 Example 3.1. ...................................................................... 52
3.5 Example 3.2, replicate at node $v_1$ ................................. 55
3.6 Lemma 3.4 ............................................................... 58
3.7 Theorem 3.2. ............................................................ 59
3.8 Multicast replication algorithm: $u \in R$, replicate at node $v$ if $\mathcal{L}(T_v) > U(T) - U(T_v)$ .................................................. 60
3.9 Example 3.3. ............................................................. 61
3.10 Example 3.5. MR has smaller lookup network cost than UR. Section 3.4.2 shows that we can remove replicas at $v_1$ and $v_2$ without changing the overall network cost. ........................................... 66
3.11 Example 3.6. UR has smaller lookup network cost than MR. .... 67
3.12 Lookup network cost. ..................................................... 69
3.13 Update network cost. .................................................... 70
3.14 Total network cost. ....................................................... 71
3.15 Local lookup percentage. ................................................. 72
3.16 Database lookups. ....................................................... 73
3.17 Database updates. ....................................................... 74
3.18 Comparison among the three algorithms: HIPER, UR, and MR. The number of database access and the network cost of UR and MR are normalized to those of HIPER. The following performance measures are considered: database lookups (DBL), lookup network cost (LNC), database updates (DBU), update network cost (UNC), replica database updates (RDBU), replica update network cost (RUNC), and total network cost (TNC). .................................................. 75
3.19 Replica distribution histogram. ........................................... 76
3.20 Replica lookup access statistics, normalized by the total number of replicas. .......................................................... 77
3.21 Comparison among different sets of traffic parameters. Users in $T P_2$
move 10 times faster than users in $T P_1$. The local callee percentage
in $T P_3$ is $\frac{1}{4}$ of that in $T P_1$. The network cost of both UR and MR is
normalized to that of HIPER. The following performance measures are
considered: lookup network cost (LNC), update network cost (UNC),
and total network cost (TNC). ............................................... 79

3.22 Comparison among different sets of network link costs. The network
cost of both UR and MR is normalized to that of HIPER. The difference
between $L_1$ and $L_{10}$ is that the communication link cost is unity in $L_1$
throughout the hierarchy, and uniformly distributed between 1 and 10
in $L_{10}$. The following performance measures are considered: database
lookups (DBL), lookup network cost (LNC), database updates (DBU),
update network cost (UNC), and total network cost (TNC). ......... 80

4.1 Example 4.1: a sample path of replica states. The square box contains
the work function value upon request $\sigma_r$. Replica states for $W_i(a)$ are
$\{a, a, a, a, a, a, a\}$, replica states for $W_i(b)$ are $\{b, b, b, b, b, b, b\}$, and
replica states for $W_i(ab)$ are $\{b, b, b, b, a, b, a b, a b\}$. ................. 88

4.2 Example 4.1: a sample path of work function values. ............... 89

4.3 Offset transition diagrams. ................................................. 91

4.4 Framework for the edge problem: $f$ calculates the offset vector, $g$ as-
signs an appropriate replica state according to the offset vector. .... 93

4.5 $\Delta opt_i = 1$ for the transitions shown, and $\Delta opt_i = 0$ for the transitions
not shown. $D = 2$. ............................................................. 96

4.6 Minimum requirements. Next to each offset are the permitted replica
states. ................................................................. 97

4.7 Offset transition diagrams with input requests replaced by $\Delta opt_i$ and
$\Delta \Phi_i$ constraints. $D = 3$. .............................................. 101

4.8 Comparison between the potentials defined in (4.4) and (4.5). The
numbers on top of the offsets are the potentials. $D = 3$ .............. 107
4.9 Correspondence between algorithms $A_L$ and $A_H$. An offset $(0kl)$ is equivalent to a counter pair $(D - (l - k), D - l)$. Next to each offset are the permitted replica states. The differences in replica state assignments are underlined. Input requests are not shown in the transition diagrams for clarity. ................................. 110

4.10 Offset transition diagram, $D = 2$, $d = 1$. The dashed lines outlines the $d$-counting region. .................................................. 112

4.11 $d$-counting regions in the offset transition diagrams, $d = 0, 1, \text{and } 2$. 113

4.12 Offset transition diagrams with input requests replaced by $\Delta opt_i$ and $\Delta \Phi_i$ constraints, $D = 3$, $d = 2$ ........................................... 115

4.13 The migration problem, $D = 3$ ........................................ 117

4.14 Offset transition diagram with input requests replaced by transition probabilities. All requests are equally likely, i.e., $p\{a^\lambda\} = p\{a^\delta\} = p\{b^\lambda\} = p\{b^\delta\} = 0.25$. Steady state probabilities are calculated for $D = 2$. .................................................. 118

4.15 Example 4.5. Transitions relevant to the input sequence $\sigma = \{(b^\delta b^\delta a^\lambda a^\lambda)^+\}$ are highlighted. ................................. 119

4.16 Examples 4.7 and 4.8, step-by-step vs one-shot processing. ....... 121

4.17 Tree algorithm. ........................................................... 122

4.18 Factoring and request propagation. In (a), a request from subtree $T_a(a, b)$ ($T_b(a, b)$) is factored to a request from node $a$ ($b$) in the induced edge problem. If the replica set $R$ falls entirely in subtree $T_a(a, b)$ ($T_b(a, b)$), then the replica state of edge $(a, b)$ is $a$ ($b$) in the induced edge problem. If $R$ spans both subtrees $T_a(a, b)$ and $T_b(a, b)$, then the replica state of edge $(a, b)$ is $ab$ in the induced edge problem. In (b), a request from node $v$ reaches node $a$ before node $b$, therefore it is factored to a request from node $a$ for edge $(a, b)$. ............... 123

4.19 Offset distribution along directions $d_1$ and $d_2$, where $\uparrow$ means non-decreasing and $\downarrow$ means non-increasing. Given $D = 3$ and $w(a, b) = (012)$, the valid offsets for its adjacent edges are $(012)$, $(022)$, $(023)$, and $(033)$. .................. 126

xvii
4.20 Edge algorithm $A_Q$ replicates for offsets in gray. Since it results in a disconnected replica set $R$ for $D \geq 4$, it does not lead to a valid tree algorithm. .................................................. 129
4.21 Sufficient conditions if $\rho_z \in T_a(a, b)$. ............................................. 131
4.22 Sufficient conditions if $\rho_z \in T_b(a, b)$ and $\rho_z \notin P_{b \rightarrow R_{t-1}}$. .............. 132
4.23 Sufficient conditions if $\rho_z \in T_b(a, b)$ and $\rho_z \in P_{b \rightarrow R_{t-1}}$. ................. 133
4.24 Examples 4.9 and 4.10. $D = 3$. .................................................. 134
4.25 Theorem 4.5. .................................................. 137
4.26 Explicit request propagation in the distributed edge model. The filled circles correspond to replicas and the arrows mark explicit request propagation, assuming the offset of edge $(a, b)$ changes upon the current request. .................................................. 140
4.27 Normal propagation vs. smart propagation. In (b), request propagation does not pass edge $(a, b)$ because its offset does not change upon the current request. .................................................. 146
4.28 Lookup-only requests, $D = 2$. The counting processes for $a^\lambda$ and $b^\lambda$ are independent. .................................................. 148
4.29 Theorem 4.11. Possible replications are along path $P_{v \rightarrow v'}$. ................. 153
4.30 Cases when the order of requests in a sequence does not affect the offset value. .................................................. 156
4.31 The hybrid implementation of an on-line replication algorithm. The replica set $R$ is the set of filled nodes. Each boundary replica node $u$ covers a subtree $T(u)$, and computes offsets for all the edges in $T(u)$, as denoted by dashed lines. .................................................. 158
4.32 Total network cost. .................................................. 162
4.33 Total network cost. The network cost of both smart and delayed propagation is lower than that of the threshold-based algorithm. .... 163
4.34 Total network cost. With or without initial placement of replicas, the network costs of our on-line algorithms are similar. ............ 164
4.35 Lookup network cost. .................................................. 165
4.36 Update network cost. .................................................. 166
4.37 Database lookups. ............................................. 167
4.38 Database lookups. ............................................. 168
4.39 Database updates. ............................................. 169
4.40 Comparison among three algorithms: HOPPER, On-line:SM, and MR.
The number of database access and the network cost of On-line:SM and MR are normalized to those of HOPPER. The following performance measures are considered: database lookups (DBL), lookup network cost (LNC), database updates (DBU), update network cost (UNC), and total network cost (TNC). ............................................. 170
4.41 Average number of replicas per user. ...................... 171
4.42 Comparison among different sets of traffic parameters. Users in $TP_2$ move 10 times faster than users in $TP_1$. The local callee percentage in $TP_3$ is $\frac{1}{7}$ of that in $TP_1$. The number of database access and the network cost of both On-line:SM and MR are normalized to those of HOPPER. The following performance measures are considered: database lookups (DBL), lookup network cost (LNC), database updates (DBU), update network cost (UNC), and total network cost (TNC). ........ 172

5.1 Location lookup in Meta-HLR MMT. ....................... 181
5.2 Two hierarchical architectures. .............................. 182
5.3 Shared hierarchical architecture. ............................ 183
5.4 Simulation topologies and database hierarchies $\mathcal{S}_0$. 184
5.5 Simulation topologies and database hierarchies $\mathcal{S}_1$. 185
5.6 Simulation topologies and database hierarchies $\mathcal{S}_2$. 186
5.7 Local, intra-network, and inter-network lookup response percentages. 189
5.8 Database lookups and updates, $p = 1.0$. .................. 190
5.9 Database lookups and updates, $p = 0.5$. .................. 191
5.10 Database lookups and updates, $p = 0.0$. ................. 192
Chapter 1

Introduction

Looking at trends in today’s personal communications [19], it is evident that a single network can not satisfy the diversity of communication needs, because people will continue to use multiple devices to communicate with one another. Multiple devices require multiple addresses, making it inconvenient to communicate with another person. To achieve the goal of personal communications, we need a universal personal identification (UPI) that uniquely identifies an individual and is independent of the access network (AN) and the communication device.

We take the stance that such UPIs will become an integral part of future personal communication systems so as to better serve the ever increasing communication needs across heterogeneous networks and through a variety of diverse communication devices. Under such context, the relevant problem to be solved is how to locate a user given his UPI, i.e., the problem of efficient tracking and retrieving of a mobile user’s location information using his UPI, commonly known as the mobility management problem. The goal of our research is to design efficient mobility management techniques (MMTs) to support integrated personal communication services from heterogeneous networks. We identify the following challenges and requirements:

- **Location/Network Independent UPI**: UPIs were introduced to hide the idiosyncrasies of different network addresses and to allow communication at a personal level. A UPI must therefore be independent of both geographical location and AN.
- **Interoperability and Privacy:** A MMT must facilitate interoperability among heterogeneous networks without compromising the autonomy of each AN. In particular, the privacy of a user’s location information should be protected.

- **Scalability:** A MMT must scale to a large number of wide-area roaming users and heterogeneous networks.

- **Efficiency:** Location lookup and update signaling overheads must be minimized.

In general, the network keeps track of a user’s location through an up-to-date user profile stored in various databases. The two basic procedures in mobility management are location update and location lookup. When a user moves across the boundaries of the RAs, the network updates his location information in the pertinent databases. When a caller places a call using the callee’s identification, the network queries the relevant database(s) to obtain the current location and other service information of the callee.

Since at call initiation, the callee’s identification is the only piece of information available to establish the communication link, the network needs to infer from this identification the addresses of relevant location databases from which to retrieve the callee’s location information. In particular, the IS-41 and GSM standards use a geographical phone number as the callee’s identification. This geographical phone number directly points to a home location register (HLR) that serves as a location database in those standards. Geographical numbering, however, ties a user’s identification to a particular network and geographical location, therefore it can not support UPI.

Hierarchical mobility management techniques have been proposed to support UPI. A hierarchical structure scales better because location information is distributed, and there is no bottleneck in the system. Hierarchical structures can also exploit the locality in user calling and mobility patterns. In addition to using a hierarchical structure, the performance of mobility management can be further enhanced by using replicas of user profiles which may be kept at various locations. Replication techniques make profile information more readily available, thus reducing lookup cost and latency,
but to keep these replicas consistent and fresh, they must be updated whenever the
user profile is updated. The principle of replication is to replicate if the benefit of
replication is greater than its overhead.

A replication algorithm can be either off-line or on-line. An off-line algorithm
assumes complete knowledge of user traffic statistics. An on-line algorithm, on the other
hand, does not make any assumption about user traffic patterns, it decides whether to
distribute new replicas or delete existing replicas after serving each request, all based
on the input sequence seen so far. In this dissertation, we propose optimal off-line and
on-line replication algorithms that minimize the network messaging cost based on the
network structure, communication link cost, and user calling and mobility statistics.

Scalable and efficient mobility management architectures are also explored to sup-
port UPI in a heterogeneous network environment. Our first proposal builds upon
the existing location databases and modifies HLR/VLR to accommodate location in-
dependent UPI. To provide better scaling and to exploit locality in user calling and
mobility patterns, our second proposal adopts a hierarchical architecture.

1.1 Dissertation Outline

Chapter 2 covers some groundwork, including an introduction to the wireless network
architecture and the concept of user profile. We present a historical perspective on the
development of MMTs as well as detailed descriptions of two representative MMTs,
HLR/VLR and Hierarchical MMT. Motivated by locality in user calling and mobility
patterns, we introduce profile replication to improve the performance of MMTs. There
are two major questions in profile replication:

1. How should a replica be accessed?

2. Where should a replica be placed?

The first question is answered in Chapter 2, where we introduced replica pointers
to avoid searching the hierarchy for the closest replica, thus limiting the maximum
number of database lookups for each lookup request to two. Chapters 3 and 4 are
devoted to answering the second question. In particular, Chapter 3 focuses on off-line profile replication, where we develop optimal off-line replication algorithms that minimize network cost based on the network structure, communication link costs, and user calling and mobility statistics. Chapter 4 focuses on on-line profile replication, where we develop optimal on-line replication algorithms that dynamically adjust replica placement according to user calling and mobility patterns.

Chapter 5 addresses the mobility management problem in a heterogeneous network environment.

1.2 Contributions

We made a number of contributions in the area of mobility management in PCS networks.

We have formulated the off-line optimization problems for both unicast and multicast replica updates with the objective of minimizing the total network cost. Each of the optimization problems has been reduced to a known general problem. In particular, the unicast replication problem has been reduced to the $p$-median problem in discrete location theory (DLT), for which an efficient dynamic programming solution exists in a tree network. In contrast to a previously unsuccessful attempt to solve this optimal replication problem [39], we draw a clear distinction between database access cost and network cost. Using our replica access scheme, the service of a lookup request requires at most two database lookups with a predetermined search sequence. With the database access cost hence bounded, we can now focus on the network cost and reduce the optimal replication problem to the $p$-median problem. The multicast replication formulation serves as a comparison case and is applicable when the signaling network supports multicast. We have not only generalized a previously proposed algorithm to include non-uniform link costs, but also provided an intuitive explanation for the algorithm. Motivated by the special features of our problem, we have been able to reduce the database update cost by eliminating certain replicas while maintaining the minimum network cost.

We have built a unified framework for solving the on-line file allocation (FA)
1.2. CONTRIBUTIONS

problem on an edge. In addition to characterizing the past input sequence via an offset vector, our framework constructs the overall solution structure through an offset transition diagram. Our framework has been put to test in the following three tasks. First, we have shown the correspondence between two previously proposed algorithms by fitting them into our framework. Second, we have proposed a new 3-competitive algorithm. Third, we have relaxed the assumption in the original edge model, allowing the cost of deleting a replica to be non-zero. Via a distributed edge model, we have included the cost of request propagation for the FA edge problem. Previously, the applicability of an edge algorithm to a tree FA problem has been proved only for individual cases. The design of an edge algorithm is therefore somewhat arbitrary, and one can only hope that the solutions for the individual edges do not conflict with one another. Our study has revealed sufficient conditions that connect the design of an edge algorithm to that of a tree algorithm, such that the design of an edge algorithm is no longer arbitrary, but with the tree structure in mind. Applying the distributed edge model to a tree, we relate request propagation on an edge to that on a tree. We have addressed this previously missing link so that the cost of request propagation can be added to a tree algorithm as well as an edge algorithm while maintaining the same competitive ratio. A tree algorithm implicitly assumes that all edges in the tree know the input requests, thus a request must propagate to all edges. To reduce the overhead of request propagation, we have proposed and analyzed two methods, smart and delayed propagation. The idea of smart propagation is to continue propagation only if it is necessary, and the idea of delayed propagation is to propagate after accumulating a number of requests.

On a separate front, we have proposed two MMTs to support integrated personal communication services from heterogeneous networks using UPI. We have modified HLR/VLR with Meta-HLR databases to map a UPI to multiple HLR addresses. To solve the scalability problem in this centralized approach, we have proposed a shared hierarchical MMT that not only scales better but also takes advantage of locality in user calling and mobility patterns.

The performance of our proposed MMTs has been studied via large scale computer simulations. Both our off-line and on-line replication algorithms are optimal and
perform better than previously proposed threshold-based algorithms.
Chapter 2

Mobility Management Techniques

This chapter begins with an introduction to the wireless network architecture and the concept of user profile in Section 2.1. In Section 2.2, we trace the development of various mobility management techniques (MMTs). Two techniques, HLR/VLR and Hierarchical MMT, are reviewed in Sections 2.2.2 and 2.2.3, respectively. The performance of MMTs can be improved by profile replication as described in Section 2.3. Section 2.4 surveys related work, followed by an introduction to our simulation environment in Section 2.5.

2.1 Basics

2.1.1 Wireless Network Architecture

Figure 2.1 illustrates a generic wireless network architecture. The coverage area of an AN is divided into registration areas (RAs), and each RA is associated with some location database (DB). A registration area is further divided into cells, each served by a single base station. Mobile terminals communicate with base stations through wireless links. The base stations within the same RA are wired to a mobile switching center (MSC) that is a part of the wireline network. Signalings between MSCs and DBs are carried out through the common channel signaling (CCS) network, using the signaling system no. 7 (SS7) protocol [55].
Figure 2.1: Wireless network architecture.
2.1.2 User Profile

A user profile contains not only a user’s current location information, but also service information, such as billing and authentication. Specifically,

- **Location information** consists of the address of the current serving MSC and a temporary routable number. The temporary number is called temporary local directory number (TLDN) in IS-41 and mobile station roaming number (MSRN) in GSM.

- **Status information** describes the status of a mobile terminal, e.g., a mobile terminal can be in either of the three states: busy, idle, or power-off.

- **Service information** includes everything else needed to establish a connection, such as billing, authentication, voice mail and call forwarding options, quality of service requirement, etc. The service information is used not only to authenticate a user, but also for the network to personalize the communication service according to the user’s preference.

2.2 Mobility Management Techniques

In this section, we offer a historical perspective on how different MMTs came into being with an emphasis on Hierarchical MMT, followed by a detailed description of two representative MMTs: HLR/VLR and Hierarchical MMT.

2.2.1 A Perspective

The two related and often confusing problems in mobility management are location registration/paging and database update/lookup. The problem of location registration/paging deals with small scale mobility management, i.e., when to update location information and how to page a user [12]. Its challenge lies in location uncertainty, i.e., there is a gap between the last seen location of a user and the user’s current location. Both location registration and paging procedures consume wireless resources. The problem of database update/lookup, on the other hand, deals with large scale mobility
management, i.e., where to store and retrieve a user’s profile. The lookup procedure is prescribed by where user profiles are stored in the network. The search process either finds the user profile or returns not found. In both cases, the search process terminates, and there is no uncertainty, although the user location information stored in the databases may be obsolete. Both database update and lookup procedures utilize wireline resources.

Although operating on different scales, these two problems bear some parallels. For example, both problems depend on user calling and mobility patterns (on different scales of course), which leads to similar mathematical optimization problems in some cases [3, 8]. The most important parallel between these two problems is the collaboration between the two components in each problem, namely, location registration and paging in the small scale mobility management problem, and database update and lookup in the large scale mobility management problem. In the former case, the registration policy should aim to aid the paging procedure. In the latter case, where we store the user profile dictates where we retrieve it. In both cases, the two components need to be considered in conjunction and optimized as a whole. This dissertation focuses on the database update/lookup aspect of mobility management.

Where should we store a user’s location information? Location lookup and update affect database designs in opposite ways: to save location lookup cost, we would like to store a user’s location information in every node of the network. However, whenever a user moves, all nodes in the network must be notified, resulting in expensive location update procedures. On the other hand, the location update cost is zero if we do not store a user’s location information anywhere except for the user’s current location. To reach the user, we would have to search the entire network, resulting in expensive location lookup procedures.

A third approach is to keep users’ location information in a centralized database, or HLR. HLRs are associated with specific home location areas (HLAs). The UPI of user $i$ maps to the address of a particular HLR either directly or indirectly. When user $i$ moves, his HLR is updated accordingly. Similarly, when user $i$ is called, his HLR is queried to retrieve $i$’s profile. The HLR scheme is inefficient under the following scenario: when user $i$ travels out of his HLA and receives some local calls in that
area, mobility management always needs to retrieve user $i$’s profile from his HLR remotely, even though user $i$ is currently roaming in that area. This problem is partially remedied by visitor location registers (VLRs). If user $i$’s profile is stored not only in his HLR, but also in a VLR covering the area that he is currently visiting, then the network first queries the caller’s local VLR for user $i$’s profile, a query to the remote HLR is only necessary if user $i$ is not found in the local VLR.

What other options do we have? and can we do better? The mobility management problem is not unique and similar problems arise in other areas, such as name server design, where the network needs to find a migrating server given the name of the service [52]. Mullender [65] created a paradigm called “Distributed Matching-making” for these problems. For any node $v$ in the network, define two node sets $P(v)$ and $Q(v)$, such that $P(v) \cap Q(v') \neq \emptyset$ for each ordered pair $(v, v')$. With this matching constraint, various design goals can be defined. For example, one can try to minimize the average of $|P(v)| + |Q(v)|$ over all pairs $(v, v')$. Applying this paradigm to the mobility management problem, suppose user $i$ moves to node $v$. It needs to update its location information for all nodes in $P(v)$. When a call originates from node $v'$ to user $i$, the network queries all the nodes in $Q(v')$. Since $P(v) \cap Q(v') \neq \emptyset$, the network is guaranteed to retrieve the latest location information of user $i$. Note that although some location information from $Q(v')$ may be out-of-date, one can always distinguish the latest location information via a simple time stamp. If we limit $|P(v) \cap Q(v')| = 1$, the match-making strategy can be represented by a rendezvous matrix $R$, where $P(i) \cap Q(j) = \{r_{ij}\},$

$$R = \begin{bmatrix}
  r_{11} & r_{12} & \cdots & r_{ij} & \cdots \\
  r_{21} & r_{22} & & \cdots & \\
  \vdots & & & \ddots & \\
  r_{i1} & \cdots & \cdots & r_{ij} & \cdots \\
  \vdots & & & \cdots & 
\end{bmatrix}, \quad \bigcup_{j=1}^{n} r_{ij} \subseteq P(i) \quad \text{and} \quad \bigcup_{i}^{n} r_{ij} \subseteq Q(j).$$
Example 2.1. $R_1$ is a broadcasting structure.

\[
R_1 = \begin{bmatrix}
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 \\
3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 \\
4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 \\
5 & 5 & 5 & 5 & 5 & 5 & 5 & 5 \\
6 & 6 & 6 & 6 & 6 & 6 & 6 & 6 \\
7 & 7 & 7 & 7 & 7 & 7 & 7 & 7 \\
8 & 8 & 8 & 8 & 8 & 8 & 8 & 8 \\
9 & 9 & 9 & 9 & 9 & 9 & 9 & 9 \\
\end{bmatrix}
\]

Example 2.2. $R_2$ is a centralized structure.

\[
R_2 = \begin{bmatrix}
5 & 5 & 5 & 5 & 5 & 5 & 5 & 5 \\
5 & 5 & 5 & 5 & 5 & 5 & 5 & 5 \\
5 & 5 & 5 & 5 & 5 & 5 & 5 & 5 \\
5 & 5 & 5 & 5 & 5 & 5 & 5 & 5 \\
5 & 5 & 5 & 5 & 5 & 5 & 5 & 5 \\
5 & 5 & 5 & 5 & 5 & 5 & 5 & 5 \\
5 & 5 & 5 & 5 & 5 & 5 & 5 & 5 \\
5 & 5 & 5 & 5 & 5 & 5 & 5 & 5 \\
5 & 5 & 5 & 5 & 5 & 5 & 5 & 5 \\
\end{bmatrix}
\]
Example 2.3. $R_3$ is a hierarchical structure, where two nodes rendezvous at their least common ancestor (LCA).

$$R_3 = \begin{bmatrix}
1 & 7 & 7 & 9 & 9 & 9 & 7 & 9 & 9 \\
7 & 2 & 7 & 9 & 9 & 9 & 7 & 9 & 9 \\
7 & 7 & 3 & 9 & 9 & 9 & 7 & 9 & 9 \\
9 & 9 & 9 & 4 & 8 & 8 & 9 & 8 & 9 \\
9 & 9 & 9 & 8 & 5 & 8 & 9 & 8 & 9 \\
9 & 9 & 9 & 8 & 8 & 6 & 9 & 8 & 9 \\
7 & 7 & 7 & 9 & 9 & 7 & 9 & 9 \\
9 & 9 & 9 & 8 & 8 & 8 & 9 & 8 & 9 \\
9 & 9 & 9 & 9 & 9 & 9 & 9 & 9 & 9 
\end{bmatrix}$$

There are two problems in the design of $R_3$. First, the definitions of $P$ and $Q$ do not include distant limit. In the previous example, the distance between nodes 3 and 4 may not be larger than that between nodes 2 and 3, yet the former meets at node 9 and the latter meets at node 7, and node 7 is one level lower than node 9 in the hierarchy. Second, being the highest level in the hierarchy, node 9 belongs to all $P$ and $Q$ sets. However interactions between nodes 2 and 3 do not need to involve node 9 as node 7 is their LCA. In other words, $R_3$ can not take advantage of locality.

The distance factor is considered in [5], where an $m$-regional matching defines the sets $P(v)$ and $Q(v)$ such that $P(v) \cap Q(v') \neq \emptyset$ if nodes $v$ and $v'$ are within distance $m$ of each other. Based on $m$-regional matching, Awerbuch [6] proposed to build a hierarchy of such matchings, with the $i$th level matchings, $P_i(v)$ and $Q_i(v)$, covering the vicinity of distance $2^i$ from node $v$, as shown in Figure 2.2. By progressively searching a larger and larger area around the caller, callees physically close to the caller can be found in lower levels of the hierarchy, thus taking advantage of locality and reducing signaling cost. The problem with the update procedure remains: while the lookup procedure can terminate the search as soon as it finds the user profile, the update procedure needs to update all levels of the hierarchy. To avoid updating higher levels of the hierarchy, the authors proposed to leave a forwarding pointer at the previous location and only update the lowest $\log d$ levels if the previous and
current locations are of distance \( d \) away.

\[
\begin{array}{c|c}
\text{level } i & P, Q \\
\hline
. & . \\
\hline
\text{level 1} & P, Q \\
\hline
\text{level 0} & P, Q \\
\end{array}
\]

Figure 2.2: Hierarchy based on regional matching.

Why can’t the update procedure take advantage of locality? It is because a user profile is stored in all levels of regional matchings. If we store a pointer to a next lower level matching instead, then the update procedure needs to propagate only as far as the LCA between the previous and current locations. The lookup procedure changes accordingly: if the match returns a pointer, we then follow the pointer to the lowest level where the user profile is stored. Therefore, a hierarchy of regional matchings with pointers at all levels except the leaf level can take full advantage of locality. A simplification of such hierarchy is a tree structure \([85, 59, 74]\), which maintains the distance constraint only within subtrees, while nodes from adjacent subtrees need to meet at the LCA between the two subtrees.

The next question concerns where to place databases in a hierarchy. Badrinath \([7]\) proposed to partition the coverage area according to a user’s mobility patterns and to update his location information whenever he crosses the boundaries of these partitions. The intuition is to group together areas where the user moves frequently to reduce update cost. In addition to mobility patterns, calling patterns also contribute to the effectiveness of a partition scheme. Furthermore, a per-user partition scheme needs to consider global database capacity constraints. All these factors included, Anantharam \([3]\) formulated the optimal placement of databases in a hierarchy given
link capacities, database capacities, and user calling and mobility patterns. The solutions utilize dynamic programming techniques. The objective functions can be the total number of database access, the total link traffic, call setup delays due to queuing at the databases, etc.

In summary, Mullender [65] provided the theoretic foundations for the correctness of a hierarchical database design. Awerbuch [6] added distance constraints to the matchings. A hierarchy of regional matchings makes finding nearby users cheaper than faraway users. Yet this approach lacks an elegant solution to exploit locality in mobility patterns. Replacing the actual user profile with location pointers preserves locality in mobility patterns, but location lookup might require traversing down the hierarchy following location pointers. A simple approximation of such a hierarchy is a tree structure. Per-user partition according to user mobility patterns exploits locality in mobility patterns. A complete formulation and quantitative solution were given by Anantharam [3] on the optimal placement of databases in a hierarchy.

2.2.2 HLR/VLR

Database Structure

Geographical routing is prevalent in wireline telephony, where a user’s directory number is geographical. When a call is routed through the public switched telephone network (PSTN), switches determine the appropriate routes according to the telephone number of the callee. In particular, a telephone number in the United States has ten digits: the first three-digit is the area code, followed by a three-digit central office code and a four-digit line number. These phone numbers are therefore routable.

With mobility, however, a user’s ID does not correspond to a fixed geographical location, hence is not directly routable. The task of mobility management is therefore to translate such ID into some routable number that identifies the MSC currently serving the user. Unlike the wireline case, a routable number is temporary and valid only for the purpose of call setup. A temporary routable number not only conserves the resources of routable numbers, but also keeps a user’s privacy.

Suppose RA X is associated with a location database DB(X), and DB(X) stores
profile information for all the users currently roaming in RA X. Let \( i \) be a user currently roaming in RA X. Although DB(\( X \)) can respond to any location query regarding user \( i \) within RA X, user \( i \)'s whereabouts is unknown outside of RA X. To solve this problem, we need a location database that always knows user \( i \)'s whereabouts. Such location database is called the HLR, covering user \( i \)'s HLA. The other location databases are the VLRs covering user \( i \)'s VLAs. Both HLR and VLRs are defined with respect to user \( i \), with HLR(\( i \)) denoting user \( i \)'s HLR and VLR(\( i \)) denoting user \( i \)'s current VLR. As user \( i \) moves across RAs, his user profile is passed from one VLR to another, while HLR(\( i \)) always maintains a pointer to VLR(\( i \)). In other words, a VLR acts like a local cache for the HLR, and a location lookup request either finds user \( i \)'s profile directly in VLR(\( i \)) or more generally, queries HLR(\( i \)) and follows the pointer to VLR(\( i \)).

Since at call initiation, the callee's identification is the only piece of information available to establish the communication link, the network needs to infer from this identification the address of the callee's HLR. In particular, the IS-41 [29] and GSM [64] standards use a geographical phone number as the callee's identification. This geographical phone number directly points to a user's HLR, i.e., the address of HLR(\( i \)) is encoded in user \( i \)'s ID.

**Location Update**

The location update procedure involves authenticating a user, moving a user's profile from the previous VLR to the current VLR, and changing a user's HLR to point to the current VLR. The signaling for authentication and database updates can be combined. When to update is determined by registration/paging protocols. In particular, each RA broadcasts its location ID over a common control channel. By monitoring such beacon signals, a mobile terminal can detect any RA crossings. When user \( i \) moves from RA \( X \) to RA \( Y \), his mobile terminal initiates a location update procedure, as shown in Figure 2.3.

**Algorithm 2.1.** HLR/VLR location update

1. User \( i \)'s mobile terminal initiates a location update request to MSC(\( Y \)).
2.2. MOBILITY MANAGEMENT TECHNIQUES

1. MSC(Y) authenticates user i by consulting VLR(Y) and HLR(i).

2. MSC(Y) registers user i in VLR(Y) and HLR(i). VLR(Y) obtains user i’s profile from HLR(i) and HLR(i) updates user i’s location pointer to point to VLR(Y).

3. HLR(i) cancels the previous registration at VLR(X).

![Figure 2.3: Location update in HLR/VLR.](image)

Location Lookup

The basic location lookup involves querying the callee’s HLR. When user j attempts a call to user i, his mobile terminal initiates a location lookup procedure, as shown in Figure 2.4.

**Algorithm 2.2. HLR/VLR location lookup**

0. Location lookup request for user i originates from MSC(j).

1. Query VLR(j), if found, then done.

2. Query HLR(i) which contains the address of VLR(i).

3. Query VLR(i).

4. MSC(i) assigns a temporary number to user i. Return this temporary number along with user i’s profile to VLR(j).
The HLR address is needed in both update and lookup procedures. The update procedure requires the mover’s HLR and the lookup procedure requires the callee’s HLR. The former is considered as internal network activities, and the HLR address can be encoded in a private user ID, e.g., the international mobile subscriber identity (IMSI) in GSM. These private IDs are different from dialable phone numbers, because they are not dialable and serve internal identification purpose only. Since these private user IDs are readily available for location updates, our attention has been focused on the public user ID, which encodes the address of a user’s HLR for location lookup purpose.

Other Applications

HLR/VLR-like mobility management also finds its application in mobile IP, wireless ATM and satellite systems. Although the terminologies differ, the concepts of home and visitor areas prevail.

Limitations of HLR/VLR

There are three major limitations of HLR/VLR. First, because location lookups query the callee’s HLR by default, call setup between users who are in adjacent registration areas may need to query a remote HLR, adding lookup latency and network cost. For example, let RAs X, Y, and Z be three RAs where RAs X and Y are far apart and RAs Y and Z are physically adjacent. Suppose a callee, whose HLA is RA X, moves temporarily from RA X to RA Y. While in RA Y, all calls to this callee from RA Z
2.2. MOBILITY MANAGEMENT TECHNIQUES

need to query the remote database covering RA X for the callee’s profile, even though the RAs Y and Z are next to each other. This problem is figuratively referred to as the tromboning problem. Second, HLR/VLR does not scale well with a large number of users because it is centralized and the HLRs can easily become bottle-necks in the network with increasing number of users. Third, HLR/VLR uses a geographical phone number as the callee’s ID. This geographical phone number directly points to the callee’s HLR. Geographical numbering, however, ties a user’s ID to a particular network and geographical location, therefore HLR/VLR can not support UPI.

Improving HLR/VLR

The performance of HLR/VLR can be improved by reducing either the location update or lookup costs. If a user does not receive many calls, we can delay updating his HLR by either leaving a forwarding pointer at his previous VLR [42] or anchoring the change between his HLR and VLRs [36], thus reducing the location update cost. The performance of forwarding pointers was thoroughly studied in [27]. On the other hand, if a callee does not move frequently, we can avoid querying his HLR by either caching his profile at his callers’ current VLRs [43, 34] or placing location pointers between his HLR and VLRs [35], thus reducing the location lookup cost. While a cached user profile may be out-of-date since it is not updated when the user’s location changes, a profile replication technique [76] ensures that all profile replicas are updated whenever the user’s location changes. The trade-off is between the benefit and overhead of replication. Other research efforts involve designing efficient call setup protocols in conjunction with the MMTs [40] [20]. To support UPI, we propose a Meta-HLR scheme that builds upon the existing location databases and modifies HLR/VLR to accommodate location independent UPI [79].

2.2.3 Hierarchical MMT

Database Structure

Hierarchical MMT arranges the location databases into a tree-like structure [51]. From the root down, the databases in each level of the hierarchy form finer and finer
partitions of the total coverage area. Each leaf database serves an individual RA and stores location information of users currently roaming in the RA. A non-leaf database stores a pointer (UPI + database ID) for every user who is roaming in the coverage area of one of its descendant databases, as shown in Figure 2.5.

**Location Update**

In Figure 2.6, when a user $i$ moves from RA $X$ to RA $Y$, the network not only moves user $i$’s profile from DB($X$) to DB($Y$), but also updates user $i$’s location pointers by creating current location pointers from DB($Y$) to the LCA between DB($X$) and DB($Y$), LCA(DB($X$), DB($Y$)), and deleting previous location pointers from LCA(DB($X$), DB($Y$)) to DB($X$). In the following algorithm description, $\text{ptr}(\text{DB},i)$ is user $i$’s location pointer at DB.

**Algorithm 2.3. Hierarchical location update**

User $i$’s mobile terminal initiates a location update request to MSC($Y$)

$DB \leftarrow DB(Y)$

$DB_n \leftarrow \text{parent}(DB)$

**while** $\text{ptr}(DB_n,i) = 0$ **do**

$\text{ptr}(DB_n,i) \leftarrow DB$ \{Create location pointer at DB$_n$ for user $i$\}

$DB \leftarrow DB_n$

$DB_n \leftarrow \text{parent}(DB)$

**end while**\{DB$_n$ is LCA(DB($X$), DB($Y$))\}

$DB_o \leftarrow \text{ptr}(DB_n,i)$

$\text{ptr}(DB_n, i) \leftarrow DB$

**while** $\text{ptr}(DB_o,i) \neq 0$ **do**

$DB \leftarrow \text{ptr}(DB_o,i)$

$\text{ptr}(DB_o,i) \leftarrow 0$ \{Remove location pointer for user $i$\}

$DB_o \leftarrow DB$

**end while**\{DB$_o$ is DB($X$)\}

Transfer user $i$’s profile from DB($X$) to DB($Y$)
2.2. MOBILITY MANAGEMENT TECHNIQUES

Figure 2.5: Database hierarchy.
Location Lookup

In Figure 2.7, DB(j) is the leaf database where caller j currently resides, and DB(i) is the leaf database where callee i currently resides. When j places a call to i, the system first queries DB(j) for i’s profile information. If not found, DB(j) propagates the lookup request up the location hierarchy to LCA(DB(j), DB(i)), the first database that contains i’s location pointer. The lookup request then propagates down the hierarchy following callee i’s location pointers to DB(i) where callee i’s location information is stored.

Algorithm 2.4. Hierarchical location lookup

Location lookup request for user i originates from MSC(j)
Query DB(j), if found, then done
DB ← DB(j)
repeat
  DB ← parent(DB)
  Query DB for user i
until found {DB is LCA(DB(j), DB(i))}
while ptr(DB,i) ≠ 0 do
  DB ← ptr(DB,i) {follow location pointer for user i}
end while {DB = DB(i)}
2.2. MOBILITY MANAGEMENT TECHNIQUES

![Diagram of location lookup in Hierarchical MMT]

Figure 2.7: Location lookup in Hierarchical MMT. Caller \( j \) in DB(\( j \))’s coverage area attempts a call to callee \( i \) in DB(\( i \))’s coverage area. The dotted lines delineate the propagation paths of a lookup request.

Return user \( i \)’s profile to DB(\( j \))

Advantages of Hierarchical MMT

Obviously a hierarchical structure scales better because location information is distributed, and there is no bottleneck in the network. Hierarchical structures can also exploit locality in user calling and mobility patterns. Because users tend to call and move in geographically adjacent areas, signaling mostly involves lower-level databases, while the less frequent long distance calls and movements require signaling involving higher-level databases. More importantly, a hierarchical structure embeds a user’s location information in the hierarchy, and a user’s ID is used to search the hierarchy. A user’s ID is no longer geographical, therefore a hierarchical structure is suitable to support UPI.

Improving Hierarchical MMT

Similar to the performance enhancement techniques for HLR/VLR, caching [41, 82], forwarding pointer [48, 69], and profile replication [45, 51] are applicable to Hierarchical MMT. One drawback of a hierarchical structure is that the load and storage requirements increase as we move up the hierarchy. To alleviate this problems,
Mao [61] and Dolev [22] proposed alternative data and tree structures, respectively. End-to-end service delay for a hierarchical structure was evaluated using queuing models in [26, 71].

2.3 Profile Replication

What is profile replication? It is putting a copy of the user profile where it is needed the most. The difference between caching and replication is that the cached user profile information does not undergo changes if the user’s location changes, while the user profile replica is updated whenever the user’s location changes. The benefit of profile replication is the reduced lookup cost and latency. The overhead of profile replication is the cost of updating these replicas when a user’s location changes.

2.3.1 Motivations

Previous studies have found locality in user calling patterns. For example, one study [76] showed that 70% of the calls made by callers in a week are to their top five callees. For user $i$, if we replicate his top five callees’ profiles at his location, then the lookups to these callees can be resolved locally, reducing lookup cost and latency.

Specifically, in hierarchical mobility management, when a caller places a call to a callee, the lookup request has to traverse the database hierarchy in order to retrieve the callee’s profile from where the callee is currently roaming. If we put a copy of the callee’s profile where the lookup request is originated, this lookup request can then retrieve the callee’s profile from this replica, thus reducing lookup cost and latency. Intuitively, the more lookup requests the more we gain from replication. However if the callee moves around, this replica must be updated thus incurring network cost. On one hand, we would like to put the replica as close as possible to where the lookup requests originate to reduce lookup cost. On the other hand, putting the replica as close as possible to where the update requests originate reduces replica update cost. When lookup and update requests are from multiple sources, replicas should be placed as close as possible to calling and mobility activity centers. However, mobility
2.3. PROFILE REPLICATION

activity centers do not necessarily coincide with calling activity centers, and a trade-off between the benefit and overhead of replication is inevitable. Since it only makes sense to keep a replica if the benefit of replication is greater than its overhead, our task is to determine the benefit and overhead of replication, both of which depend on the network structure, communication link costs, and user calling and mobility patterns.

2.3.2 System Model

In our model, communication links connect different levels of the databases. Let $T = (V, E)$ be a database hierarchy, where $v \in V$ corresponds to a location database and $e = (u, v) \in E$ is the communication bidirectional link between two connected location databases $u$ and $v$. The cost of such a link is $c(u, v)$. Because of the tree structure, there exists a unique path between any two databases $x$ and $y$ with cost $c(x, y)$ equal to the sum of the link costs along the path from $x$ to $y$. A parent-child relation is uniquely defined once we assign a root for the tree. This root can be arbitrary because its sole purpose is to define the parent-child relations. Let $\rho$ be an arbitrary root of $T$. Given an edge $(u, v) \in E$, if $u$ is on the unique path between $\rho$ and $v$, then $u$ is the parent of $v$ and $v$ is a child of $u$. Let $T_v$ be a subtree rooted at $v$, i.e., $T_v$ consists of $v$ and all of its offsprings. Given edge $(a, b)$ in a tree, removing edge $(a, b)$, we get two subtrees $T_a(a, b)$ and $T_b(a, b)$, rooted at nodes $a$ and $b$, respectively. The replica database set $R(i) \subseteq V$ is the set of databases containing replicas for user $i$, as shown in Figure 2.8. Replication for user $i$ is determined by both lookup requests and update requests. Lookup requests are generated by calls to user $i$ and update requests are generated by movements of user $i$. In describing our algorithms, we will focus on the optimal replication for user $i$. In a given period of time, $\forall v \in V$,

- $\mathcal{L}(v)$: the number of lookup requests from database $v$
- $\mathcal{U}(v)$: the number of update requests from database $v$
- $\mathcal{A}(v)$: the number of requests from database $v$, $\mathcal{A}(v) = \mathcal{L}(v) + \mathcal{U}(v)$
- $\mathcal{L}(T_v)$: the number of lookup requests from subtree $T_v$, $\mathcal{L}(T_v) = \sum_{v \in T_v} \mathcal{L}(v)$
\( \mathcal{U}(T_v) \): the number of update requests from subtree \( T_v \), \( \mathcal{U}(T_v) = \sum_{v \in T_v} \mathcal{U}(v) \)

\( \mathcal{A}(T_v) \): the number of requests from subtree \( T_v \),
\( \mathcal{A}(T_v) = \sum_{v \in T_v} \mathcal{A}(v) = \sum_{v \in T_v} \mathcal{L}(v) + \sum_{v \in T_v} \mathcal{U}(v) \)

![System model diagram](image)

Figure 2.8: System model.

We modify the lookup and update protocols to include replication. Lookup and update with replication follow the general rule of ”read-one-and-write-all”.

**Location Lookup with Replicas**

Given the set of replica databases \( R \), there exists a unique database \( \hat{v} \in R \) closest to a database \( v \), \( \forall v \in V \). Following the convention in discrete location theory (DLT) we say that \( v \) is served or covered by \( \hat{v} \). If \( v \notin R \), the address of \( \hat{v} \) is stored in \( v \) as a replica pointer. For a lookup request originating from \( v \), if \( v \) has a replica, or equivalently if \( v \in R \), then we can obtain the user profile information locally at \( v \). Otherwise, we follow the replica pointer to query the closest replica database \( \hat{v} \) with a cost \( c(v, \hat{v}) = \min_{r \in R} (c(v, r)) \), as shown in Figure 2.9. Using replica pointers to access non-local replicas, the number of database lookups for each lookup request is at most two, one for the replica pointer if needed and the other for the replica itself. This replica access model not only defines a predetermined search sequence but also keeps a bound on the the cost of database lookup.
2.3. PROFILE REPLICATION

Figure 2.9: Lookup the closest replica. \( \hat{v} \) is the closest replica to \( v \), such that \( c(v, \hat{v}) = \min_{r \in R} (c(v, r)) \).

Location Update with Replicas

To keep the replicas consistent with the primary profile copy, each location update request triggers a replica update for all databases \( r \in R \). Replica updates can be either unicast or multicast. Let \( v \) be the database where the update request is originated. In unicast, since each database \( r \in R \) is updated separately with cost \( c(v, r) \), the total cost is the sum of the costs of individual replica updates, \( \sum_{r \in R} c(v, r) \), as shown in Figure 2.10(a). In multicast, the minimum steiner tree (MST) [38], covering the replica database set \( R \) and database \( v \), is traversed to reach all database \( r \in R \). The total cost of multicast replica update is the cost of the MST(\( R \cup \{v\} \)), as shown in Figure 2.10(b). Intuitively, multicast not only updates the replica at database \( r \) but also all replicas at databases along the path from \( v \) to \( r \).

2.3.3 Issues

Replica Consistency

Instead of employing expensive locking protocols [67] to maintain replica consistency, we create a temporary forwarding pointer at a user’s previous location to redirect calls to his current location, in case the location information obtained from a profile replica is not yet updated. This is similar to the treatments in [76, 45, 51].
Database Capacity

Profile replicas require additional storage at databases. A simple way to limit the total storage is to limit the maximum number of replicas allowed for each user. However, these limits do not prevent some databases from using up more storage than others. To consider individual database’s capacity, a typical approach is to formulate the replication problem into a network flow problem. Shivakumar et al. [76] took such an approach to solve the replication problem for HLR/VLR. We shall show in Section 3.1.1 why such formulation can not extend to Hierarchical MMT. In this dissertation, our formulations of the replication problem limit the maximum number of replicas for each user whenever possible. Storage requirements for our proposed algorithms are demonstrated via computer simulations.

What to Replicate?

Lam [49] argued that only the address of the current serving database (a VLR in HLR/VLR and a leaf DB Hierarchical MMT) should be replicated because part of the routing information, the temporary routing number, is volatile and valid only for the purpose of call setup, and there would be considerable overhead if we were to update this information each time this routing information changes. The drawback of this approach is that an additional location lookup at the current serving database
is necessary in order to retrieve the routing information. As the caller and the callee might be far away, replication in this case reduces only the number of database lookups, but not the network cost.

We view this issue differently. Both the address of the serving MSC and the temporary routing number lead to the serving MSC, with the temporary number further identifying a user within the coverage area of the serving MSC. We observe that the address of the serving MSC is sufficient to route the call to the serving MSC, and the complete temporary routable number is not needed until the serving MSC has received the call and is ready to establish the connection to the callee. Therefore we can safely replicate the location information in the form of the address of the serving MSC, avoiding the volatile temporary number altogether in the replica. This approach eliminates the additional location lookup at the current serving database and reduces both the number of database lookups and network cost. Similar ideas were investigated in protocols such as the reverse virtual call setup [40] and the reverse connection setup [20].

In addition to location information, various service information (call forwarding options, quality of service requirement, etc.) is also suitable for replication.

Off-line or On-line?

A replication algorithm can be either off-line or on-line. An off-line algorithm makes replication decision based on user traffic statistics. An on-line algorithm, on the other hand, does not make any assumption of user traffic statistics. Instead, it decides whether to add or delete a replica after serving each request, all based on the input requests seen so far.

What to Optimize?

The total cost of a mobility management system consists of both database access cost and network cost. Since our replica access scheme serves a lookup request in at most two database lookups with a predetermined search sequence, the database access cost is hence bounded. From now on, we will be focusing on minimizing the
2.4 Related Work

The concept of UPI is an extension to the three-level aspects of mobility, namely terminal, personal, and service mobility [37]. The concept of person-to-person communications is not entirely new [33, 90]. More recent research efforts have mostly focused on the network protocol aspect of the problem [4, 60, 75, 84], including personalizing communication service according to the user’s preference, translating between different communication formats, and negotiating terminal capabilities, etc. The mobility tracking aspect of the problem, however, is largely unattended and mostly resorts to HLR/VLR-like centralized mobility management. This dissertation fills in this gap.

General survey papers on mobility management can be found in [2, 70]. A mobility management architecture can be either centralized, e.g., HLR/VLR, or distributed. The distributed or hierarchical database structure avoids the bottleneck problem found in a centralized system and therefore scales better. It also takes full advantage of locality in user calling and mobility patterns. Previously proposed mobility management architectures were based on the assumption of a single network environment, and the designs were not necessarily targeted at supporting UPI in a heterogeneous network environment. In Chapter 5, we propose two efficient hierarchical database architectures for this purpose.

Various optimization techniques, such as caching, forwarding pointers, and profile replication, have been proposed to improve the performance of MMTs. Although their complexity and effectiveness may vary, these optimization techniques are generally applicable to both centralized and distributed architectures. In particular, the profile replication problem belongs to the family of file allocation (FA) problems. Dowdy et al. [23] provided a comprehensive survey on FA models which employed mixed integer programming formulations and relied mostly on heuristics for efficient general solutions. Previously proposed profile replication algorithms are threshold-based [45, 51]. Although simple, these algorithms rely on gross simplification to estimate the threshold values and can not provide any performance guarantees partly because of
their dependence on traffic models. In contrast, our profile replication algorithms determine the optimal replica placement according to user traffic statistics, network structure, and communication link costs. They are optimal in minimizing network communication costs, independent of any traffic models.

Realistic large scale computer simulations for mobility management were first developed in the Pleiades project [45, 51]. Using these realistic traffic models previously developed, we conduct extensive computer simulation to demonstrate the performance of our proposed MMTs.

2.5 Simulation Environment

2.5.1 Topology

Our simulation topology models the ten largest cities in U.S.. Each city is divided into a number of RAs, as shown in Figure. 2.11. Users can move within and in between these cities. Next to the RA grids are the database hierarchies for the AN. The leaf databases correspond to RAs. Unless otherwise specified, the communication link cost is unity throughout the hierarchy. The intermediate level databases in the hierarchy serve as signaling routing centers for non-hierarchical MMTs. All our simulation results can easily scale to larger user population sizes. The results shown here are performed on a population of 100,000 users. The choice of the population size reflects a trade-off between running time and accuracy.

2.5.2 Traffic model

The simulation traffic model is based on the realistic teletraffic and mobility models developed in [50] that incorporates real call traffic data, airplane passenger traffic data, and personal transportation surveys. This model accounts for various calling and mobility patterns typical of human behavior, including patterns that change over the course of a day. Our simulation results show such variations with time accordingly. We show the call and movement traffic volume over a 24-hour period in Figure 2.12. The call volume is much larger than the movement volume and the movement curve is
Figure 2.11: Simulation topologies and database hierarchies.
2.5. SIMULATION ENVIRONMENT

not as smooth because of the lower volume. There are also small variations in traffic volume from day to day as shown in Figure 2.13. Our simulations consider time zone difference between the east and west coast, and the time reference is the pacific standard time (PST), which explains why the morning traffic peaks around 7am. Unless otherwise specified, this set of traffic parameters, $TP_1$, is used throughout this dissertation. To explore the performance advantages of our replication algorithms under different traffic conditions, we consider variations on some traffic parameters in the traffic model: $TP_2$ and $TP_3$, where users in $TP_2$ move 10 times faster than users in $TP_1$ and the local callee percentage in $TP_3$ is $\frac{1}{4}$ of that in $TP_1$.

2.5.3 Performance Metrics

We measured the performance of a replication algorithm via the following three sets of metrics:

1. As the target of our optimal replication algorithms, the total network cost is the foremost performance measure. It breaks down into lookup network cost, update network cost, and replication related costs which are also of interest. Local lookups/updates involve zero network cost, while only remote lookups/updates incur network cost. Both primary and replica updates contribute to the update network cost. The update network cost for the primary copies is the same for all algorithms.

2. Although not the direct optimization objectives, various database access counts are closely related to the network cost metric. They provide a consistency measure of an algorithm. In general, a lookup request may generate multiple database lookups. A local lookup request, however, requires only one local database lookup. Both primary and replica copy updates contribute to the number of database updates. Since the number of primary copy updates is the same for all algorithms, the more replicas an algorithm distributes, the more database updates it generates. In this category, we examined local lookup percentages, number of database lookups, and number of database updates.
Figure 2.12: Traffic volume.
2.5. Simulation Environment

(a) Call

(b) Movement

Figure 2.13: Traffic volume variations.
3. Neither of the above metrics deliver explicit indications of the effectiveness of individual replicas. The replica distribution and access statistics are introduced to serve this purpose.

2.5.4 Off-line and On-line Simulations

An off-line simulation is to verify the performance of an off-line algorithm that requires complete knowledge of user traffic statistics. The setup for off-line simulation thus involve two phases: statistics collection and simulation run. As shown in Figure 2.14, the first phase collects relevant lookup and update request statistics. At the end of the first phase, an off-line replication algorithm establishes the replica locations for each user according to the statistics collected so far. With replicas in place, we then run the entire simulation afresh in the second phase. Special care is taken to ensure that the conditions of the two phases in off-line simulations are exactly the same. We shall see how the performance of an off-line algorithm degrades without accurate traffic statistics in Chapter 4. This is where an on-line algorithm kicks in, because it can dynamically adjust replica placement according to user calling and mobility patterns.

The on-line simulation is also conducted in two phases: the first phase collects statistics and distributes replicas according an off-line replication algorithm. In the second phase, an on-line algorithm dynamically adjusts replica placement according to user calling and mobility patterns. The setup for on-line simulation matches how we would like to apply our on-line algorithm in practice: an off-line algorithm determines the initial replica placement, which is modified by an on-line algorithm subsequently as requests arrive. Chapter 4 shows that the difference between simulations with and without such initial placements is fairly small. In other words, the statistics collection phase is dispensable for on-line algorithms. Special care is taken to ensure that the conditions of off-line and on-line simulations are exactly the same.

We filter out a long starting transient period and present the performance for a representative 24-hour period after the transient period.
2.6 Summary

In this chapter, we have provided some background including the wireless network architecture and the concept of user profile. Two representative MMTs, HLR/VLR and Hierarchical MMT, have been described in detail. We have motivated using profile replication to improve the performance of MMTs. In addition to our system model, we have discussed various issues related to profile replication. The material on literature surveys aims to sketch a big picture, and details related to specific topics are included in later chapters. Our simulation environment consists of simulation topology, traffic models, performance metrics, etc.
Chapter 3

Off-line Profile Replication

In addition to using a hierarchical structure, the performance of MMTs can be further enhanced by using replicas of user profiles which may be kept at various locations. This replication technique makes profile information more readily available, thus reducing lookup cost and latency. But to keep these replicas consistent and fresh, they must be updated whenever the user’s location changes. The principle of replication is to replicate if the benefit of replication is greater than its overhead.

In this chapter, we develop optimal off-line replication algorithms that minimize network cost based on the network structure, communication link costs, and user calling and mobility statistics. This chapter begins with a detailed survey of some previously proposed replication algorithms for mobility management in Section 3.1.1. It proceeds by describing some general location problems on tree networks in Section 3.1.2, from which our off-line replication algorithms shall be derived. The optimization objectives of the off-line replication problem are given in Section 3.2. We develop optimal replication algorithms for both unicast and multicast replica updates in Sections 3.3 and 3.4, respectively. In Section 3.5, we compare these two algorithms via two examples. And in Section 3.6, we compare the performance of these algorithms with that of a previously proposed threshold-based algorithm via large scale computer simulations.
3.1 Related Work

3.1.1 Replication Algorithms for Mobility Management

There have been various off-line replication proposals for mobility management, including a minimum-cost maximum-flow algorithm [76] and a threshold-based algorithm [45].

Minimum-cost Maximum-flow Algorithm

Shivakumar et al. [76] considered replications for HLR/VLR. The replication problem is formulated as a network flow problem, where the network nodes consist of user profile $P_i$’s and database $Z_j$’s, a flow from a user profile $P_i$ to a database $Z_j$ corresponds to replicating user profile $P_i$ at database $Z_j$, and the cost associated with such a flow is defined as the difference between the benefit and overhead of replication. The capacity of a database is limited, so is the maximum number of replicas allowed for each user profile. A minimum-cost maximum-flow of this flow network thus maximizes the total system savings within the capacity constraints. Without the database capacity limits, this algorithm degenerates: for each user profile, it replicates at databases with the highest savings, distributing as many replicas as allowed.

The complexity of the network flow problem aside, this formulation avoids measuring the actual benefit and overhead of replications and assumes that they are readily available. Such assumption is valid for HLR/VLR, where a profile replica is treated the same as an entry in a VLR: a lookup request searches for local profile replicas only, if not found, it falls back to search the callee’s HLR, as described in Section 2.2.2. Hence the benefit of replication is uniquely defined. Compared with the flat network structure in HLR/VLR, Hierarchical MMT allows more flexibility in the locations of replicas, i.e., in addition to a local replica, a lookup request may be served by a remote replica. The remote replicas, however, make it much harder to measure the benefit and overhead of replication. In fact, such measurement is generally unavailable until we have decided where to place the replicas. This is because although in general, the benefit of replicating at a particular database is determined
by its usage, i.e., the total number of lookup requests from all the databases that are covered by this replica, whether a database is covered by a specific replica database depends on the placement of other replicas in its neighborhood, since a request retrieves a profile replica from its closest replica database. As a result, the benefit of an individual replica is correlated to the locations of other replicas, and this network flow formulation does not apply to Hierarchical MMT.

Even though the hierarchical structure thwarts the effort to generalize the network flow formulation for HLR/VLR, it actually works to our advantage in our formulations precisely because of its tree-like structure. In our approach, we do not consider database capacity, instead, we assume it is always available. But we will show via simulations the storage requirements to implement our replication algorithms. We also incorporate the constraint on the total number of replicas allowed for each user profile whenever possible. With unicast replica update, this constraint leads to a slightly different optimization problem formulation, yet a much more involved algorithm. With multicast replica update, this constraint leads to a much harder problem and our formulation can only solve the optimization problem without it.

Threshold-based Algorithm

Jamink et al. [45] proposed a threshold-base replication algorithm for Hierarchical MMT. Suppose we want to decide whether to place a replica at a database for user $i$. The threshold-based algorithm first estimates the benefit and overhead of replication according to user traffic statistics. Specifically, the benefit is approximated by the number of calls from this database to user $i$, and the overhead is approximated by the number of movements made by user $i$. Since the true benefit and overhead also depend on the network structure and communication link costs, thresholds ($R_{\text{max}}$ and $R_{\text{min}}$) are introduced to allow some error margin. The algorithm decides whether to replicate by comparing against these thresholds the ratio of the number of calls to the number of movements.

Although simple, the threshold-based algorithm suffers from a few problems. First, its thresholds are difficult to determine. They often have to be picked by hand according to a particular set of user traffic statistics or simplified traffic models. Second,
its performance is difficult to analyze because of its dependency on thresholds and traffic statistics. Overall, the threshold-based algorithms treat calls and movements as isolated events and do not consider the network structure and communication link costs.

3.1.2 General Location Problems for Tree Networks

The $p$-median Problem

The profile replication problem belongs to the family of file allocation (FA) problems [23]. Discrete location theory (DLT) [63] originates from the study of the facility location problems in operations research. The isomorphism between FA and DLT has long been observed [73]. However, it is the recent algorithm developments [18, 77] in DLT that enabled us to solve the FA problem on tree networks efficiently without the crutch of heuristics.

In particular, the $p$-median problem is a general formulation for the optimal placement of facilities given usage statistics and some cost functions. Suppose there are a number of clients $I = \{1, 2, \cdots, i, \cdots, n\}$ who require a certain service. The demand of each client is denoted as $w_i, i \in I$. This service can be provided by some facilities $J = \{1, 2, \cdots, j, \cdots, m\}$. For client $i$, the cost is $c_{ij}$ to get a unit of service from facility $j$. Let the set of facility locations be $K \subseteq J$. Assume that all facilities are identical and have unlimited capacity. Client $i$ chooses facility $\hat{j}$ that provides the service with the minimum cost, i.e., $\hat{j} = \arg \min_{j \in K} c_{ij}$. Furthermore, let the maximum number of facilities allowed be $p$ and the cost of setting up facility $j$ be $f_j$. The $p$-median problem is a general formulation for the optimal placement of facilities so that the total cost, service cost plus facility setup cost, is minimized, i.e.,

\[
\min \{ \sum_{i \in I} w_i \{ \min_{j \in K} c_{ij} \} + \sum_{j \in K} f_j \}, \quad \text{provided that } |K| \leq p.
\]

There exist efficient algorithms for the $p$-median problem on tree networks using dynamic programming techniques [18, 77].

Our unicast replication problem bears close analogy to the $p$-median problem on tree networks, which facilitates the reduction from the former to the latter. In contrast to a previously unsuccessful attempt to solve this optimal replication problem [39],
we draw a clear distinction between database access cost and network cost. Using our replica access scheme, the service of a lookup request requires at most two database lookups according to a predetermined search sequence. With the database access cost hence bounded, we can now focus on network cost and reduce the optimal replication problem to the \( p \)-median problem.

**Replication for Multicast Replica Update**

If the network supports multicast, then replicas can be updated in a multicast fashion. Both Wolfson [88] and Maggs [58] gave an \( O(n) \) algorithm for this general optimization problem. We not only generalize their results to include non-uniform link costs, but also provide an intuitive explanation for the algorithm. Motivated by the special features of our problem, we are able to reduce the database update cost by eliminating certain replicas while maintaining the minimum network cost.

### 3.2 Optimization Objectives

Our optimal replication algorithms minimize the network cost based on the network structure, communication link costs, and user calling and mobility statistics. The optimal solutions establish upper bounds on the performance improvement achievable from replication. Intuitively, a replica in a higher level of the database hierarchy can serve lookup requests from a larger area than a replica in a lower level of the hierarchy. Conversely, even though the number of lookup requests from a single node may not be large enough to qualify for replication, the sum of the lookup requests from a few nodes may, therefore these nodes may share a replica. The replicas should be placed close to the calling activity center so as to reduce the replica lookup cost. Similarly, if a user moves often within a certain area, the replicas should be placed close to the mobility activity center so as to reduce the replica update cost. All these intuitions are captured in our solutions to the replication problem.

Depending on whether replicas are updated in unicast or multicast, we have the following optimization objectives:
3.3. Replication for Unicast Replica Update (UR)

Replication for unicast replica update (UR)

\[
\text{Network cost} = \sum_{v \in V} \mathcal{L}(v) \min_{r \in R} c(v, r) + \sum_{v \in V} \mathcal{U}(v) \sum_{r \in R} c(v, r) \tag{3.1}
\]

Replication for multicast replica update (MR)

\[
\text{Network cost} = \sum_{v \in V} \mathcal{L}(v) \min_{r \in R} c(v, r) + \sum_{v \in V} \mathcal{U}(v) \text{MST}(R \cup v) \tag{3.2}
\]

In both objective functions, the first term represents the replica lookup network cost and the second term represents the replica update network cost. The total network cost depends on the network structure, communication link costs, and user calling and mobility statistics. Although the general optimization problem is hard, we are fortunate to deal with a hierarchical, or tree-like, network. We reduce formulation (3.1) to the \(p\)-median problem in DLT, for which efficient dynamic programming solutions exist on tree networks. We include formulation (3.2) as a comparison case that is applicable when the signaling network supports multicast.

In the system model described in Section 2.3.2, we assume that the lookup and update network costs are the same between any two databases. Such an assumption can be relaxed by a weighting function in addition to \(\mathcal{L}(\cdot)\) and \(\mathcal{U}(\cdot)\). Neither of our replication algorithms are affected by this modification.

3.3 Replication for Unicast Replica Update (UR)

Since a lookup request at database \(v\) is served by its closest replica database \(\hat{v}\), if we imagine replicas as facilities, then the first term in Equation (3.1) matches the \(p\)-median problem description. On the other hand, the second term in Equation (3.1) does not directly fit into the \(p\)-median problem because an update request needs to be propagated to all replica databases. But if we exchange the order of summations in Equation (3.1),

\[
\text{Network cost} = \sum_{v \in V} \mathcal{L}(v) \min_{r \in R} c(v, r) + \sum_{r \in R} \sum_{v \in V} \mathcal{U}(v)c(v, r) \tag{3.3}
\]
the inner sum, \( \sum_{v \in V} \mathcal{U}(v) c(v, r) \), is a fixed constant for every replica database in the network, and corresponds to the facility setup cost in the \( p \)-median problem. Equation (3.3) now conforms to the description of a \( p \)-median problem in DLT. We have thus reduced the unicast replication problem to the general \( p \)-median problem, where replicas correspond to facilities, the replica update cost maps to the facility setup cost, and the lookup request maps to the service demand.

If we limit the maximum number of replicas allowed for each user so that \( |R| \leq p < |V| \), this optimization problem can be solved in \( O(pm^2) \) time [77] or \( O(p^2m^2) \) time [46]. If \( p = |V| \), commonly referred to as the uncapacitated facility location (UFL) problem, it can be solved in \( O(n^2) \) time [18]. Both solutions utilize dynamic programming techniques.

### 3.3.1 1-median Problem

When we allow only one replica, or \( p = 1 \), Equation (3.3) simplifies,

\[
\text{Network cost} = \sum_{v \in V} (\mathcal{L}(v) + \mathcal{U}(v)) c(v, r)
\]  

(3.4)

The minimum cost replica location \( m \) is called the 1-median. Goldman [32] proved the following lemma,

**Lemma 3.1. (Goldman 1971)**

Given \( T = (V, E) \) and an edge \( (v_1, v_2) \in E \), removing \( (v_1, v_2) \) results in two subtrees \( T_{v_1}(v_1, v_2) \) and \( T_{v_2}(v_1, v_2) \) rooted at nodes \( v_1 \) and \( v_2 \), respectively, as shown in Figure 3.1. Let \( m \) be a 1-median of \( T \), the following holds,

- If \( m \in T_{v_1}(v_1, v_2) \), then \( \mathcal{A}(T_{v_1}(v_1, v_2)) \geq \mathcal{A}(T_{v_2}(v_1, v_2)) \)

- If \( \mathcal{A}(T_{v_1}(v_1, v_2)) \geq \mathcal{A}(T_{v_2}(v_1, v_2)) \), then finding a 1-median of \( T \) is equivalent to finding a 1-median for \( T_{v_1}(v_1, v_2) \) when \( \mathcal{A}(v_1) \) is replaced by \( \mathcal{A}(v_1) + \mathcal{A}(T_{v_2}(v_1, v_2)) \).

From Lemma 3.1, we know that a tree network may have one or two 1-medi ans. The location of a 1-median is determined by the number of requests, or “weights”
of nodes, and the communication link costs do not make a difference. Based on Lemma 3.1, the majority algorithm [32] calculates the location of a 1-median in $O(n)$ time.

**Algorithm 3.1. Majority algorithm**

Given $T = (V, E)$, to find a 1-median $m$ of $T$,

1. If $T$ consists of a single node $v$, then $m = v$.

2. Search for node $v_i$ that connects to $T$ via a single edge $(v_i, v_j)$.
   
   - If $A(v_i) \geq A(T)/2$, then $m = v_i$.
   
   - Otherwise, modify $T$ by deleting node $v_i$ and edge $(v_i, v_j)$, and setting $A(v_j) = A(v_j) + A(v_i)$. Return to step 1.

Recall from Section 3.1.1 that the threshold-based algorithm chooses a replica location $v$ if $L(T_v) > R_{\text{max}} \cdot U(T)$, where $R_{\text{max}}$ is some threshold value. When $R_{\text{max}} = 1$ and all lookup requests originate from subtree $T_v$, the threshold-based algorithm finds a 1-median to replicate. The reverse is not true, however, since a 1-median can well be a node with $L(T_v) < U(T)$.

### 3.3.2 Uncapacitated Facility Location Problem

As mentioned in Section 3.1.1, the benefit of an individual replica is correlated to the locations of other replicas, which renders it impossible to measure the benefit.
of individual replicas before distributing replicas. Alternatively, we can view the problem from the perspective of the nodes covered by these replicas: from all possible replica locations that cover a node, we find the one with the minimum cost.

The tree structure of the network leads us to consider constructing recursion according to the parent-child relation among nodes, i.e., we divide a problem for a node into subproblems for its children. Once such recursion is established, the calculations can be carried out in a leaf-to-root fashion, i.e., the solutions to the subproblems for the children become building blocks for the parent problem. The challenge is to relate a problem for a parent node to the subproblems for its children. Suppose node $y$ is one of node $x$’s children, and the closest replicas for $y$ and $x$ are $\hat{y}$ and $\hat{x}$, respectively. The following lemma relates $\hat{y}$ to $\hat{x}$.

**Lemma 3.2.** Given a rooted tree $T = (V, E)$, let node $x$ be node $y$’s parent and the closest replicas to $x$ and $y$ be $\hat{x}$ and $\hat{y}$, respectively.

A. If $\hat{x} \notin T_y$, then either $\hat{y} \in T_y$ or $\hat{y} = \hat{x}$.

B. If $\hat{x} \in T_y$, then $\hat{y} = \hat{x}$.

**Proof.** Consider all possible locations of $\hat{x}$ and $\hat{y}$ with respect to subtree $T_y$:

A. $\hat{x} \notin T_y$, as shown in Figure 3.2(a).

1. $\hat{y} \in T_y$, possible.

2. $\hat{y} \notin T_y \implies \hat{y} = \hat{x}$, since all paths connecting any node in subtree $T_y$ to a node in $T - T_y$ must go through node $x$, node $\hat{x}$ must also be the closest replica location for node $y$, i.e., $\hat{y} = \hat{x}$.

B. $\hat{x} \in T_y$, as shown in Figure 3.2(b). Since all paths connecting node $x$ to any node in subtree $T_y$ must go through node $y$, node $\hat{x}$ must also be the closest replica location for node $y$, i.e., $\hat{y} = \hat{x}$.

The minimum cost of replication for subtree $T_x$, when $x$ is served by $\hat{x}$, consists of two parts:
3.3. Replication for unicast replica update (UR)

\[ \hat{x} \implies \hat{y} = \hat{x} \text{ or } \hat{y} \in T_y \]

(a) \( \hat{x} \notin T_y \)

(b) \( \hat{x} \in T_y \)

Figure 3.2: Lemma 3.2.

- the cost of serving \( x \) at \( \hat{x} \).

- the sum of the minimum costs of replication for subtrees \( T_y \)'s, where \( y \in \text{children}(x) \), with \( \hat{y} \) related to \( \hat{x} \) according to Lemma 3.2.

We have now divided the problem defined on \( T_x \) into subproblems defined on subtrees \( T_y \)'s. The minimum cost of replication for subtree \( T_x \) is the minimum among all \( \hat{x} \in T_x \). We need the following notations to describe the UFL algorithm.

- Let \( \alpha(u, T_v) \) be the minimum network cost of replication for subtree \( T_v \) when \( v \) is served by \( u \).

- Let \( \alpha(T_v) = \min_{u \in T_v} \alpha(u, T_v) \) be the minimum network cost of replication for subtree \( T_v \), where every node in subtree \( T_v \) is served by some replica within subtree \( T_v \), according to Lemma 3.2.
• Let $\beta_{uv}$ be the cost of serving $v$ at $u$,

$$
\beta_{uv} = \begin{cases} 
\sum_{v' \in V} U(v')c(v', v) & v = u \\
\mathcal{L}(v)c(v, u) & v \neq u
\end{cases}
$$

(3.5)

Intuitively, if a lookup request at $v$ is served locally, then we need to maintain a replica at node $v$, incurring a cost equal to the replica update cost $\sum_{v' \in V} U(v')c(v', v)$. Otherwise, the lookup request at $v$ is served at node $u$ with a cost proportional to the cost of the unique path between $v$ and $u$, $c(v, u)$.

• Let $\gamma(v)$ be the optimal replica database to serve database $v$.

**Algorithm 3.2. UFL: Minimum cost algorithm $O(n^2)$**

*From the leaves to the root $\rho,$*

1. If $v$ is a leaf database, $|T_v| = 1$, then $\alpha(T_v) = \alpha(v, T_v) = \beta_{uv}$, and $\alpha(u, T_v) = \beta_{uv}$ if $u \neq v$.

2. If $v$ is a non-leaf database, $|T_v| \neq 1$, then

   (a) If $u \notin T_v$ or $u = v$, then $w \in \text{children}(v)$ is either served by $u$ or some node in $T_w$ according to Lemma 3.2.A, as shown in Figure 3.3(a).

   $$
   \alpha(u, T_v) = \beta_{uv} + \sum_{w \in \text{children}(v)} \min\{\alpha(u, T_w), \alpha(T_w)\}
   $$

   (b) If $u \in T_v$ and $u \neq v$, node $w' \in \text{children}(v)$ on the path joining $v$ and $u$, or $u \in T_{w'}$, must also be served by $u$ according to Lemma 3.2.B, as shown in Figure 3.3(b).

   $$
   \alpha(u, T_v) = \beta_{uv} + \alpha(u, T_{w'}) + \sum_{w \in \text{children}(v) - \{w'\}} \min\{\alpha(u, T_w), \alpha(T_w)\}
   $$

3. $\alpha(T) = \alpha(T_\rho)$ is the minimum replication cost for $T$.

The complexity of this algorithm is $O(n^2)$, where $n = |V|$, because each node can be served by any node in the tree.
### 3.3. Replication for Unicast Replica Update (UR)

![Diagram of UFL: Minimum cost algorithm.](image)

**Figure 3.3: UFL: Minimum cost algorithm.**

**Algorithm 3.3.** UFL: Minimum cost replica location algorithm $O(n)$

> From the root $\rho$ to the leaves,

1. $\gamma(\rho) = \arg\left(\min_{u \in T} \alpha(u, T)\right)$ is the optimal replica database to serve $\rho$.

2. If $w \neq \rho$,

   (a) If $w$ is on the path between $\text{parent}(w)$ and $\gamma(\text{parent}(w))$, then
   
   $\gamma(w) = \gamma(\text{parent}(w))$.

   (b) Otherwise,
   
   $\gamma(w) = \arg\left(\min\{\alpha(\gamma(\text{parent}(w)), T_w), \alpha(T_w)\}\right)$.

Unless otherwise specified, we refer to Algorithms 3.2 and 3.3 as the UR algorithm. The following two examples show the UFL algorithm in action.

**Example 3.1.** For the networks in Figure 3.4, $v_4$ is the root and all communication link costs are unity. $\delta$’s represent update requests and $\lambda$’s represent lookup requests. Given the network structure and the number of lookup and update requests, we can...
calculate the optimal replication cost by calculating the α's from leaf to root.

\[
\begin{align*}
\alpha(T_{v_0}) &= \beta_{v_0v_0} \\
\alpha(T_{v_1}) &= \beta_{v_1v_1} \\
\alpha(v_0, T_{v_2}) &= \beta_{v_0v_2} + \alpha(v_0, T_{v_0}) + \min\{\alpha(v_0, T_{v_0}), \alpha(T_{v_1})\} \\
\alpha(v_1, T_{v_2}) &= \beta_{v_1v_2} + \min\{\alpha(v_1, T_{v_0}), \alpha(T_{v_0})\} + \alpha(v_1, T_{v_1}) \\
\alpha(v_2, T_{v_2}) &= \beta_{v_2v_2} + \min\{\alpha(v_2, T_{v_0}), \alpha(T_{v_0})\} + \min\{\alpha(v_2, T_{v_1}), \alpha(T_{v_1})\} \\
\alpha(v_3, T_{v_2}) &= \beta_{v_3v_2} + \min\{\alpha(v_3, T_{v_0}), \alpha(T_{v_0})\} + \min\{\alpha(v_3, T_{v_1}), \alpha(T_{v_1})\} \\
\alpha(v_4, T_{v_2}) &= \beta_{v_4v_2} + \min\{\alpha(v_4, T_{v_0}), \alpha(T_{v_0})\} + \min\{\alpha(v_4, T_{v_1}), \alpha(T_{v_1})\} \\
\alpha(T_{v_2}) &= \min\{\alpha(v_0, T_{v_2}), \alpha(v_1, T_{v_2}), \alpha(v_2, T_{v_2})\} \\
\alpha(T_{v_3}) &= \beta_{v_3v_3} \\
\alpha(v_0, T_{v_4}) &= \beta_{v_0v_4} + \alpha(v_0, T_{v_0}) + \min\{\alpha(v_0, T_{v_0}), \alpha(T_{v_3})\} \\
\alpha(v_1, T_{v_4}) &= \beta_{v_1v_4} + \alpha(v_1, T_{v_1}) + \min\{\alpha(v_1, T_{v_1}), \alpha(T_{v_3})\} \\
\alpha(v_2, T_{v_4}) &= \beta_{v_2v_4} + \alpha(v_2, T_{v_2}) + \min\{\alpha(v_2, T_{v_2}), \alpha(T_{v_3})\} \\
\alpha(v_3, T_{v_4}) &= \beta_{v_3v_4} + \min\{\alpha(v_3, T_{v_2}), \alpha(T_{v_2})\} + \alpha(v_3, T_{v_3}) \\
\alpha(v_4, T_{v_4}) &= \beta_{v_4v_4} + \min\{\alpha(v_4, T_{v_2}), \alpha(T_{v_2})\} + \min\{\alpha(v_4, T_{v_3}), \alpha(T_{v_3})\} \\
\alpha(T_{v_4}) &= \min\{\alpha(v_0, T_{v_4}), \alpha(v_1, T_{v_4}), \alpha(v_2, T_{v_4}), \alpha(v_3, T_{v_4}), \alpha(v_4, T_{v_4})\}
\end{align*}
\]

I. For the network in Figure 3.4(a), according to Equation (3.5), the β matrix is defined as follows,

\[
\beta = \begin{bmatrix}
4 & 0 & 0 & 6 & 0 \\
10 & 2 & 0 & 6 & 0 \\
5 & 0 & 3 & 4 & 0 \\
15 & 0 & 0 & 9 & 0 \\
10 & 0 & 0 & 2 & 6
\end{bmatrix}
\]
3.3. REPPLICATION FOR UNICAST REPLICA UPDATE (UR)

The α’s can be calculated,

\[ \alpha(v_0, T_{v_2}) = 4 \quad \alpha(v_0, T_{v_4}) = 10 \]
\[ \alpha(v_1, T_{v_2}) = 6 \quad \alpha(v_1, T_{v_4}) = 12 \]
\[ \alpha(v_2, T_{v_2}) = 7 \quad \alpha(v_2, T_{v_4}) = 11 \]
\[ \alpha(v_3, T_{v_2}) = 4 \quad \alpha(v_3, T_{v_4}) = 13 \]
\[ \alpha(v_4, T_{v_2}) = 4 \quad \alpha(v_4, T_{v_4}) = 12 \]
\[ \alpha(T_{v_0}) = 4 \quad \alpha(T_{v_1}) = 2 \quad \alpha(T_{v_2}) = 4 \quad \alpha(T_{v_3}) = 9 \quad \alpha(T_{v_4}) = 10 \]

The UFL algorithm determines that the optimal replication cost is 10, corresponding to replicating at node \( v_0 \), which is also the 1-median.

II. For the network in Figure 3.4(b), according to Equation (3.5), the β matrix is defined as follows,

\[ \beta = \begin{bmatrix}
4 & 0 & 0 & \mathbf{12} & 0 \\
10 & 2 & 0 & \mathbf{12} & 0 \\
5 & 0 & 3 & 8 & 0 \\
15 & 0 & 0 & \mathbf{9} & 0 \\
10 & 0 & 0 & 4 & 6
\end{bmatrix} \]

The difference between Examples 3.1-I and 3.1-II is highlighted in the β matrices. The α’s can be calculated,

\[ \alpha(v_0, T_{v_2}) = 4 \quad \alpha(v_0, T_{v_4}) = 13 \]
\[ \alpha(v_1, T_{v_2}) = 6 \quad \alpha(v_1, T_{v_4}) = 15 \]
\[ \alpha(v_2, T_{v_2}) = 7 \quad \alpha(v_2, T_{v_4}) = 15 \]
\[ \alpha(v_3, T_{v_2}) = 4 \quad \alpha(v_3, T_{v_4}) = 13 \]
\[ \alpha(v_4, T_{v_2}) = 4 \quad \alpha(v_4, T_{v_4}) = 14 \]
\[ \alpha(T_{v_0}) = 4 \quad \alpha(T_{v_1}) = 2 \quad \alpha(T_{v_2}) = 4 \quad \alpha(T_{v_3}) = 9 \quad \alpha(T_{v_4}) = 13 \]
The UFL algorithm determines that the optimal replication cost is 13, corresponding to replicating at nodes \( v_0 \) and \( v_3 \).

(a) Example I, replicate at \( v_0 \).  
(b) Example II, replicate at \( v_0 \) and \( v_3 \).

Figure 3.4: Example 3.1.

3.3.3 \( p \)-median Problem

If we limit the number of replicas to be \( p \), such that \( p < |V| \), the minimum cost replication algorithm is much more involved. While the general dynamic programming techniques still apply, the constraint on the total number of replicas must be translated to each subproblem accordingly, in addition to the network cost. For detailed analysis, please refer to [77].

3.3.4 Discussion

General vs. Tree Networks

One possible way to find an optimal replica placement is to enumerate all possible combinations of replica locations and find the minimum cost among them. In doing so, however, we inevitably fall into the curse of a combinatorial problem, i.e., there are \( \binom{n}{p} = \frac{n!}{p!(n-p)!} \) possibilities. If the network size \( n = 20 \) and \( p = 5 \), this number amounts to 15,504. If \( n = 50 \), and \( p = 10 \), this number grows to about \( 10^{10} \)! How big is this number? The age of the Earth is generally believed to be 4.55\( \times 10^9 \) years.
Although hard for general networks, the optimal replication problem on tree networks has efficient algorithms because of following theory [47].

**Theorem 3.1. (Kolen 1983)**

\[ \forall r \in R, V(r) = \{ v \in V : \text{node } v \text{ is served by replica } r \} \text{ induces a subtree.} \]

This property of tree networks significantly reduces the number of possible combinations of replica locations. The dynamic programming techniques then allow us to enumerate these possibilities systematically, i.e., from leaf to root.

**Other Problem Formulations**

The \( p \)-median problem locates \( p \) medians in a network such that the total distance of every node to its closest median is minimized. In our mobility management problem, the \( p \)-median formulation corresponds to minimizing the total network cost. Another related problem in DLT is the \( p \)-center problem [28, 62], which locates \( p \) centers in a network such that the maximum distance of every node to its closest center is minimized. In our mobility management problem, the \( p \)-center formulation corresponds to minimizing the maximum network cost. This is useful in limiting the maximum latency in the call setup process, where the cost of signaling is related to the number of hops, and more hops means longer delay. A hybrid of the \( p \)-median and \( p \)-center problems is defined as the \( p \)-median problem that minimizes a convex combination of the objective functions of the median and center problems. Tamir et al. [78] proposed a polynomial time algorithm for the \( p \)-median problem on tree networks. Depending on our optimization objectives, all these formulations are applicable to the unicast replication problem.

**Reduction Obstacles**

We face a few obstacles in reducing the unicast replication problem to the \( p \)-median problem. The first difficulty results from the asymmetry between lookup and update requests, i.e., a lookup request at database \( v \) is served by its closest replica database \( \hat{v} \), whereas an update request needs to be propagated to all replica databases. The following two special cases remove this asymmetry:
• If there is no update request, i.e., replicas are read-only, then most of the $p$-median problem formulations readily fit the unicast replication problem.

• If the number of replicas is limited to 1, then it is a 1-median problem.

The second difficulty has to do with replica access. All DLT models assume that each client knows the location of its closest facility once the facilities are in place. We introduce replica pointers to meet this requirement. In general, for a lookup request from node $v$, it is difficult to measure the network cost to search for its closest replica $\hat{v}$ because the search path is undefined. A replica pointer not only defines a search sequence, but also limits the number of database lookups for each lookup request to two, one for the replica pointer if needed and the other for the replica itself.

Primary Copy

In our problem formulation, we set aside the primary copy of the user profile; the cost of updating the primary copy does not appear in the objective functions because it is independent of replica placement, and we assume that all lookup requests are served by replicas. But a primary copy lookup may be cheaper than a replica lookup, depending on their relative positions. To handle this possibility, we can compare the cost of the primary and replica lookups, and replicate only if the replica lookup costs less than the primary lookup.

Replica Distribution Cost

Also not included in our problem formulation is the cost of distributing a replica. Similar to unicast replica update, the cost of distributing a replica is purely a function of the replica location. Moreover, this cost is a constant for each node in the tree. Therefore, the replica distribution cost can be modeled as part of the facility setup cost, along with the replica update costs.

Network Size

When describing our algorithms, we have used the network $T = (V, E)$ with $|V| = n$ extensively. But in actual implementation, what network size should be used in these
3.3. REPLICATION FOR UNICAST REPLICA UPDATE (UR)

calculations? These algorithms would not be as tractable if we were to take the whole network into account. In fact, we need to consider only the part of a hierarchical network that covers all the databases with lookup or update requests, i.e., a subtree induced by the databases from which lookup and update requests originate.

Suppose no requests originate from a leaf node \( v \), i.e., \( \mathcal{L}(v) = \mathcal{U}(v) = 0 \), then \( v \) can be served by any replica \( r \in R \) with zero cost. If we were to add a replica at node \( v \), it would be more expensive to update a replica at node \( v \) than a replica at its parent node. Since there is no advantage to replicate at node \( v \), we don’t need to consider node \( v \) in the calculation. Suppose only update requests originate from a leaf node \( v \), i.e., \( \mathcal{L}(v) = 0 \), but \( \mathcal{U}(v) \neq 0 \). Similar to the case when \( \mathcal{L}(v) = \mathcal{U}(v) = 0 \), \( v \) can be served by any replica \( r \in R \) with zero cost. However, it might be cheaper to update a replica at node \( v \) than a replica at its parent node, since the update requests \( \mathcal{U}(v) \) from node \( v \) incur a zero update cost at node \( v \), but a non-zero update cost at its parent. Hence node \( v \) should be included in the calculation, as illustrated by the following example.

**Example 3.2.** In Figure 3.5, the optimal replica location is \( v_1 \) from which only update requests originate.

![Diagram](image)

**Figure 3.5:** Example 3.2, replicate at node \( v_1 \).

We can continue to trim off leaf nodes with no requests until there are no more such nodes left. Whatever is left of the network should be included in the calculation.
Requests from leaf nodes only

The UR algorithms described here are general enough to handle lookup and update requests from any database in a tree network. In our mobility management problem, however, the lookup and update requests originate entirely from the leaf databases. One may wonder if this special feature leads to further improvement for the UR algorithms. The answer is no, because the complexity of the problem is determined by the number of possible replica locations where a node can be served, and limiting where the requests originate does not restrict placement of replicas. Besides, the UR algorithm allocates a replica only if it serves some requests. Therefore, other than trimming out the nodes with no requests as described earlier, we do not gain from this special feature.

3.4 Replication for Multicast Replica Update (MR)

In the objective function (3.2), the cost to update a replica database set $R$ from database $v$ is the cost of the minimum Steiner tree covering $R \cup \{v\}$. The following two lemmas show the properties of a multicast replica update strategy.

**Lemma 3.3.** With multicast replica update, there exists an optimal replica placement that induces a connected subtree.

*Proof.* Let $R'$ be an optimal replica placement. Construct replica set $R$ such that $R$ is the minimum connected set covering $R'$, i.e., $R' \subseteq R$. Since $R' \subseteq R$, both replica sets incur the same replica lookup cost. Similarly, for an update request from node $v$, the minimum Steiner tree covering $R' \cup \{v\}$ is the same as that covering $R \cup \{v\}$. Therefore, both replica sets also incur the same replica update cost. Therefore, $R$ is an optimal replica placement that induces a connected subtree. \qed

**Lemma 3.4.** Let $m$ be a 1-median, with multicast replica update, there exists an optimal replica placement $R$, such that $R$ is connected and $m \in R$.

*Proof.* From Lemma 3.3, we know that with multicast replica update, there exists an optimal connected replica placement $R'$. Suppose $m \notin R'$. We show how to construct
3.4. REPLICA FOR MULTICAST REPLICA UPDATE (MR)

$R$ from $R'$, such that $R$ is connected, $m \in R$, and $\text{Cost}(R) \leq \text{Cost}(R')$. Let $\hat{m} \in R'$ be the closest replica to $m$ and the path between $m$ and $\hat{m}$ be $P_{m \rightarrow \hat{m}}$. Let $m'$ be the first node along $P_{m \rightarrow \hat{m}}$ adjacent to $m$. Upon removing edge $(m, m')$, $T$ is divided into two subtrees $T_m(m, m')$ and $T_{m'}(m, m')$, as shown in Figure 3.6.

- If $\mathcal{L}(T_m(m, m')) > \mathcal{U}(T_{m'}(m, m'))$, we can add replicas along $P_{m \rightarrow \hat{m}}$ without increasing the total cost. In doing so, the replica update cost is increased by at most $\mathcal{U}(T_{m'}(m, m')) \cdot c(P_{m \rightarrow \hat{m}})$ for all the update requests originated from subtree $T_{m'}(m, m')$, since all or part of path $P_{m \rightarrow \hat{m}}$ must be traversed to update the replica at $m$. None of the update requests from subtree $T_m(m, m')$ generates additional replica update cost. On the other hand, the new replica at node $m$ reduces the lookup requests from subtree $T_m(m, m')$ by $\mathcal{L}(T_m(m, m')) \cdot c(P_{m \rightarrow \hat{m}})$. The net increase $\leq (\mathcal{U}(T_{m'}(m, m')) - \mathcal{L}(T_m(m, m'))) \cdot c(P_{m \rightarrow \hat{m}}) < 0$, since $\mathcal{L}(T_m(m, m')) > \mathcal{U}(T_{m'}(m, m'))$.

- If $\mathcal{L}(T_m(m, m')) \leq \mathcal{U}(T_{m'}(m, m'))$, since $m$ is a 1-median, $\mathcal{A}(T_m(m, m')) \geq \mathcal{A}(T_{m'}(m, m'))$ by Lemma 3.1. We have

\[
\mathcal{U}(T_m(m, m')) = \mathcal{A}(T_m(m, m')) - \mathcal{L}(T_m(m, m')) \\
\geq \mathcal{A}(T_{m'}(m, m')) - \mathcal{L}(T_m(m, m')) \\
\geq \mathcal{A}(T_{m'}(m, m')) - \mathcal{U}(T_{m'}(m, m')) \\
= \mathcal{L}(T_{m'}(m, m')).
\]

Let $r_1$ be a leaf node in the subtree induced by $R'$, and $(r_1, r_2)$ be an edge in $R'$. Upon removing edge $(r_1, r_2)$, $T$ is divided into two subtrees $T_{r_1}(r_1, r_2)$ and $T_{r_2}(r_1, r_2)$. $\mathcal{U}(T_{r_2}(r_1, r_2)) \geq \mathcal{U}(T_m(m, m')) \geq \mathcal{L}(T_{m'}(m, m')) \geq \mathcal{L}(T_{r_1}(r_1, r_2))$. If we remove replica $r_1$, the net increase $(\mathcal{L}(T_{r_1}(r_1, r_2)) - \mathcal{U}(T_{r_2}(r_1, r_2))) \cdot c(r_1, r_2) \leq 0$. Similarly, we can continue to remove all the replicas except $\hat{m}$ without increasing the total cost. With a single replica $\hat{m}$, from the definition of 1-median, $\text{Cost}(\{m\}) \leq \text{Cost}(\{\hat{m}\})$.

In both cases, we have constructed an optimal replica placement $R$, such that $R$ is connected and $m \in R$.

\[\square\]
Next we show how a 1-median serves as a seed from which new replicas “sprout”.

### 3.4.1 General MR Algorithm

**Theorem 3.2.** Given $T = (V, E)$, let $m$ be a 1-median of $T$, and $T$ is rooted at $m$. With multicast replica update, there exists an optimal replica placement $R$, such that \(orall v \in R, \text{ either } v = m \text{ or }\)

\[
\mathcal{A}(T_v) = \mathcal{L}(T_v) + \mathcal{U}(T_v) > \mathcal{U}(T).
\] (3.6)

**Proof.** From Lemma 3.4, we know there exists an optimal replica placement $R'$, such that $R'$ is connected and $m \in R'$. We show how to construct $R$ from $R'$, such that $R$ contains only 1-median and nodes that satisfy Equation 3.6, and $\text{Cost}(R) \leq \text{Cost}(R')$. To construct $R$, for $v \in T$, we perform the following operations,

- If $v \notin R'$ and $\mathcal{A}(T_v) > \mathcal{U}(T)$, then add a replica at node $v$.
- If $v \in R' - \{m\}$ and $\mathcal{A}(T_v) \leq \mathcal{U}(T)$, then delete a replica at node $v$.

Next we show that neither operation increases the total cost.
If \( v \notin R' \) and \( A(T_v) > U(T) \), consider the path between \( v \) and its closest replica \( \hat{v} \), \( P_{v \rightarrow \hat{v}} \), let \( u \) be the first node along \( P_{v \rightarrow \hat{v}} \) adjacent to \( \hat{v} \), as shown in Figure 3.7(a). We have \( T_v \subseteq T_u \), since all paths from subtree \( T_v \) to root \( m \) go through \( u \), hence \( A(T_u) \geq A(T_v) > U(T) \). If we add a replica at node \( u \), the replica update cost increases by \( (U(T) - U(T_u)) \cdot c(\hat{v}, u) \) and the lookup cost decreases by \( L(T_u) \cdot c(\hat{v}, u) \). The net increase \( (U(T) - U(T_u) - L(T_u)) \cdot c(\hat{v}, u) = (U(T) - A(T_u)) \cdot c(\hat{v}, u) \leq 0 \), since \( A(T_u) \geq U(T) \). Similarly, we can add replicas for all nodes along this path \( P_{v \rightarrow \hat{v}} \) without increasing the total cost.

On the other hand, if \( v \in R' - \{m\} \) and \( A(T_v) \leq U(T) \), consider subtree \( T_v \). Suppose \( r_1 \) is a leaf node in the subtree induced by \( R' \), \( r_1 \in T_v \), and edge \( (r_1, r_2) \) is an edge in \( R' \), as shown in Figure 3.7(b). We have \( T_{r_1} \subseteq T_v \), since all paths from subtree \( T_{r_1} \) to root \( m \) go through \( v \), hence \( A(T_{r_1}) \leq A(T_v) \leq U(T) \). Removing the replica at node \( r_1 \), the lookup cost increases by \( L(T_{r_1}) \cdot c(r_1, r_2) \), and the replica update cost decreases by \( (U(T) - U(T_{r_1})) \cdot c(r_1, r_2) \). The net increase \( (L(T_{r_1}) - U(T) + U(T_{r_1})) \cdot c(r_1, r_2) = (A(T_{r_1}) - U(T)) \cdot c(r_1, r_2) \leq 0 \), since \( A(T_{r_1}) \leq U(T) \). Similarly, we can remove all replicas in subtree \( T_v \) without increasing the total cost.

\[
\begin{align*}
\text{(a) } v \notin R' & \text{ and } A(T_v) > U(T) \\
\text{(b) } v \in R' - \{m\} & \text{ and } A(T_v) \leq U(T)
\end{align*}
\]

Figure 3.7: Theorem 3.2.

In Theorem 3.2, a 1-median serves as a seed from which new replicas "sprout", 
resulting in a connected replica set. We have the following $O(n)$ replication algorithm for multicast replica update.

**Algorithm 3.4. MR: Minimum cost replica location algorithm $O(n)$**

Given $T = (V, E)$, let $m$ be a 1-median of $T$, and $T$ is rooted at $m$. \( \forall v \in V, \) if \( \mathcal{L}(T_v) + \mathcal{U}(T_v) > \mathcal{U}(T) \), then replicate at $v$.

This algorithm does not explicitly include link costs in its calculation. We shall provide an intuitive argument as to why this is the case. As shown in Figure 3.8, given $u \in R$ and $(u, v) \in E$, suppose we want to decide whether to replicate at $v$. All update requests from subtree $T_v$ go through $v$ regardless of whether $v$ holds a replica or not. Therefore they do not generate additional overhead. Likewise, all lookup requests from subtree $T_v$ go through $v$, and the benefit is \( \mathcal{L}(T_v) \cdot c(u, v) \) if $v$ holds a replica. The additional replica update overhead results from update requests originating from outside of $T_v$, denoted by \( \mathcal{U}(T) - \mathcal{U}(T_v) \). Consequently, replicating at $v$ brings an overhead of \( (\mathcal{U}(T) - \mathcal{U}(T_v)) \cdot c(u, v) \). We would replicate if the benefit of replication is greater than its overhead, i.e., \( \mathcal{L}(T_v) \cdot c(u, v) > (\mathcal{U}(T) - \mathcal{U}(T_v)) \cdot c(u, v) \). We note that the communication link cost $c(u, v)$ drops out from both sides and this is the same replication condition given in Algorithm 3.4.

![Diagram](image_url)

Figure 3.8: Multicast replication algorithm: $u \in R$, replicate at node $v$ if \( \mathcal{L}(T_v) > \mathcal{U}(T) - \mathcal{U}(T_v) \).
Example 3.3. Revisit example 3.1 given in Figure 3.4. Node $v_0$ is a 1-median and $U(T) = 3$ in both examples. We redraw the tree with $v_0$ as the root in Figure 3.9.

In Example I,

\[
\begin{align*}
\mathcal{L}(T_{v_2}) + U(T_{v_2}) &= 4 > U(T) & \text{Replicate at node } v_2 \\
\mathcal{L}(T_{v_4}) + U(T_{v_4}) &= 2 < U(T) & \text{Do not replicate at node } v_4 \\
\mathcal{L}(T_{v_1}) + U(T_{v_1}) &= 2 < U(T) & \text{Do not replicate at node } v_1 \\
\mathcal{L}(T_{v_3}) + U(T_{v_3}) &= 2 < U(T) & \text{Do not replicate at node } v_3 \\
\end{align*}
\]

In Example II,

\[
\begin{align*}
\mathcal{L}(T_{v_2}) + U(T_{v_2}) &= 6 > U(T) & \text{Replicate at node } v_2 \\
\mathcal{L}(T_{v_4}) + U(T_{v_4}) &= 4 > U(T) & \text{Replicate at node } v_4 \\
\mathcal{L}(T_{v_1}) + U(T_{v_1}) &= 2 < U(T) & \text{Do not replicate at node } v_1 \\
\mathcal{L}(T_{v_3}) + U(T_{v_3}) &= 4 > U(T) & \text{Replicate at node } v_3 \\
\end{align*}
\]

Note that in Example I, we do not need to consider $v_3$ once we have determined that $v_4$ does not have a replica because of the connectivity requirement.

(a) Example I, replicate at $v_0$ and $v_2$. (b) Example II, replicate at $v_0$, $v_2$, $v_3$, and $v_4$.

Figure 3.9: Example 3.3.
Unlike the UR algorithm, which distributes only replicas that serve some lookup requests, the MR algorithm may distribute replicas solely to satisfy the connectivity requirement. It is these interior replicas that give us an opportunity for improvement. Our approach is to eliminate some replicas so as to reduce the number of database updates while maintaining the minimum overall network cost.

3.4.2 Modified MR Algorithm

The total network cost is the sum of lookup network cost and update network cost. The former is the cost of the unique path connecting a requesting database \( v \) and its closest replica database \( \hat{v} \), and the latter is the cost of the MST(\( R \cup \{ v \} \)). To maintain the minimum network cost, we need to attend to both replica lookup and update costs. The replica lookup cost remains unchanged if we keep all the replicas that serve some lookup requests. Let the set of replicas serving some lookup requests be \( R_{\Lambda} \). The replica update cost, on the other hand, depends on the replica set \( R \). Since \( R \) induces a subtree, let the terminal nodes\(^1\) in \( R \) be \( R_t \). The MST(\( R \)) is the union of paths between all pairs of nodes in \( R_t \). The replica update cost remains unchanged if we keep all the nodes in \( R_t \).

**Lemma 3.5.** Let \( R \) be an optimal replica placement calculated by Algorithm 3.4. For \( u \in R_t \), if \( u \) does not serve any lookup requests, then \( u \) must be a the root of \( T \) and \(|R| = 1\).

*Proof.* Because of the tree structure, all nodes served by the same replica induce a subtree. Except for the root of \( T \), the 1-median \( m \), the number lookup requests served by \( u \in R_t \) is \( \mathcal{L}(T_u) \). Suppose \( u \) is not the root of \( T \), then \( \mathcal{L}(T_u) = 0 \rightleftharpoons \mathcal{U}(T_u) > \mathcal{U}(T) \). But we know \( \mathcal{U}(T) \geq \mathcal{U}(T_u) \), since \( T_u \subseteq T \). A contradiction! Therefore \( u \) must be the root of \( T \). Suppose \(|R| > 1\), and \( u \) is connected to the rest of \( R \) via node \( v \in R \). Removing edge \((u, v)\) result in two subtrees \( T_u(u, v) \) and \( T_v(u, v) \). Node \( u \) not serving any lookup requests implies that \( \mathcal{L}(T_u(u, v)) = 0 \), and \( \mathcal{U}(T_u(u, v)) = \mathcal{A}(T_u(u, v)) \geq \mathcal{A}(T_v(u, v)) \geq \mathcal{A}(T_v(u, v)) - \mathcal{U}(T_v(u, v)) = \mathcal{L}(T_v(u, v)). \) Furthermore,

\(^1\)Terminal nodes are those with only one neighbor
\[ T_v = T_v(u,v) \text{ since } u \text{ is the root of } T. \text{ But } v \in R \text{ must satisfy } \mathcal{L}(T_v) + \mathcal{U}(T_v) > \mathcal{U}(T) \implies \mathcal{U}(T_v(u,v)) < \mathcal{L}(T_v(u,v)). \text{ A contradiction! Therefore } |R| = 1. \]

From Lemma 3.5, if node \( u \in R_t \), and \( |R| > 1 \), then \( u \) must serve some lookup requests. To maintain the minimum network cost, we must keep all the replicas in \( R \cup R_t \).

**Request From Leaf Nodes Only**

The above discussion is general enough to include lookup and update requests from any database in a tree network. In our mobility management problem, however, the lookup and update requests originate entirely from the leaf databases, which means \( R \subseteq R_t \). Therefore, in addition to the 1-median, we keep a replica at a database only if it serves some lookup requests. For example, we can remove the replicas from the two intermediate level databases, \( v_2 \) and \( v_4 \), in Example 3.3-II. The resulting replica set, however, may no longer be connected.

**Algorithm 3.5. Modified MR**

*Given \( T = (V, E) \), let \( m \) be the 1-median of \( T \), and \( T \) is rooted at \( m \).*

\[ P1. \ \forall v \in V, \text{ if } \mathcal{L}(T_v) + \mathcal{U}(T_v) > \mathcal{U}(T), \text{ then add a replica at } v. \]

\[ P2. \ \forall v \in R - \{m\}, \text{ if } v \text{ does not serve any lookup request, i.e., } \mathcal{L}(T_v) = 0, \text{ then remove the replica at } v. \]

**3.4.3 Discussion**

**Why Connected \( R \)?**

In the multicast replication problem, replicas induce a connected subtree. The connectivity of \( R \) simplifies the replication problem considerably. First, the connectivity of \( R \) isolates the impact of adding or deleting a replica to a single edge. Since both the benefit of replication, from replica lookup, and the overhead of replication, from replica update, share the same link cost, they cancel out and the link costs do not
matter. Second, when deciding whether to replicate at node \( v \), the original problem is equivalent to the replication problem with subtree \( T_v \) collapsed into one single supernode \( v' \) with the number of lookup requests being \( L(T_v) \) and the number of update requests \( U(T_v) \). Third, once a replica \( u \) enters \( R \), a later replica \( v \), such that \( T_v \subseteq T_u \), can not push out \( u \), because if \( A(T_v) > U(T) \), then \( A(T_u) > A(T_v) > U(T) \) must hold. If the number of replicas is limited to \( p \), the optimal replica set \( R \) may not be connected. Without connectivity, however, the replication problem is much more difficult. For example, we can not simply reverse the order by which replicas are added into the replica set \( R \), hoping to maintain the optimality for a given size of \( R \). The following is a counter example:

**Example 3.4.** For the network on the left, suppose all link costs are unity. \( \delta \)'s represent update requests and \( \lambda \)'s represent lookup requests. The cost of all possible replications are listed on the right.

![Diagram](image)

Since node \( v_1 \) is the 1-median, according to Algorithm 3.4, the optimal solution is to replicate at nodes \( v_1, v_2, \) and \( v_3 \), which is also the order that replicas are added. However, if we want to limit the number of replicas to 2, removing the last replica \( v_3 \) yields a cost of 4, while the optimal solution is to replicate at nodes \( v_1 \) and \( v_3 \) with a cost of 3.

As discussed in Section 3.4.2, only replicas that serve some lookup requests are necessary, while others can be removed, which suggests that an optimal replica set \( R \), such that \( R \) is connected and \(|R| \leq p \), may not exists, as in Example 3.4.
3.5. **COMPARISON BETWEEN UR AND MR**

**Replica Distribution Cost**

As in off-line unicast replication, we do not consider the cost of distributing replicas in off-line multicast replication. Such cost is, however, important in on-line replication algorithms in Chapter 4. Like multicast replica update, replicas can be distributed in a multicast fashion, and the distribution cost is determined by the MST covering all replicas. This is different from unicast replication, where each replica is distributed separately.

**Tree Orientation**

Unlike in the UR algorithm, the orientation of the tree is not arbitrary in Algorithm 3.4. The replication algorithm chooses a 1-median as the root of the tree. The order hence established in the tree might be different from the database hierarchy since the 1-median might be different from the distributed root in our model.

**Network Size**

Similar to the UR algorithm, the MR algorithm only needs to consider the network determined by the leaf databases with lookup or update requests. Suppose no requests originate from a leaf node \( v \), then \( v \) can not be a 1-median, and \( A(v) > U(T) \) does not hold. Therefore \( v \) is not a candidate for replication. Suppose only update requests originate from a leaf node \( v \), i.e., \( L(v) = 0 \), but \( U(v) ≠ 0 \). Since node \( v \) could be a 1-median which is the starting point for the MR algorithm, we need to include it in the calculation. We can continue to trim off leaf nodes with no requests until there are no more such nodes left. Whatever is left of the network should be included in the calculation.

**3.5 Comparison Between UR and MR**

In this section, we show two examples to demonstrate the difference in replica locations chosen by the optimal replication algorithms discussed in Sections 3.3 and 3.4.
The threshold-based algorithm HIPER [45] is also included as a reference. All communication link costs are unity in both examples. In Figures 3.10 and 3.11, δ’s represent update requests, and λ’s represent lookup requests. In both examples, the filled circles are the replica locations determined by the optimal replication algorithms.

Example 3.5. As shown in Figure 3.10

Example 3.6. As shown in Figure 3.11

Table 3.1 summarizes the various network costs associated with these two examples. Comparing the two replication algorithms, we observe that MR does not require communication link costs in calculating replica placement. The multicast replica update strategy effectively reduces replication overhead, but MR does not guarantee a smaller lookup network cost than UR. Their performances vary with traffic statistics (lookup and update requests), as shown in Examples 3.5 and 3.6. When |R| = 1, i.e., a single replica, there is no difference between unicast and multicast update strategies.

![Diagram](attachment://image.png)

Figure 3.10: Example 3.5. MR has smaller lookup network cost than UR. Section 3.4.2 shows that we can remove replicas at v1 and v2 without changing the overall network cost.
Figure 3.11: Example 3.6. UR has smaller lookup network cost than MR.

<table>
<thead>
<tr>
<th></th>
<th>Example I</th>
<th></th>
<th>Example II</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>UR</td>
<td>MR</td>
<td>HIPER</td>
<td>UR</td>
</tr>
<tr>
<td>Lookup network cost</td>
<td>24</td>
<td>20</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>Update network cost</td>
<td>34</td>
<td>25</td>
<td>60</td>
<td>8</td>
</tr>
<tr>
<td>Total network cost</td>
<td>58</td>
<td>45</td>
<td>64</td>
<td>14</td>
</tr>
</tbody>
</table>

Table 3.1: Network cost in Examples I and II.
3.6 Simulations

We have simulated the following replication algorithms in the environment described in Section 2.5,

1. **HIPER** threshold-base algorithm [45] with the following parameters:
   - Maximum number of replicas $N = 5$.
   - Maximum level of replication $L = 2$, corresponding to replication at all levels of the database hierarchy except the root.
   - $R_{max} = R^{opt}_{max} = 1$, $R_{min} = R^{opt}_{min}$ (topology dependent).

2. **UR** optimal replication algorithm for unicast replica update

3. **MR** optimal replication algorithm for multicast replica update

We shall compare the performance of these algorithms in terms of network cost, database access, and replica distribution and access.

3.6.1 Network Cost

As shown in Figure 3.12, both UR and MR perform very well and lead to over 54% and 76% improvement over HIPER in lookup network cost, respectively. Figure 3.13 depicts the update network cost. The improvements of the optimal algorithms over the threshold-based algorithm range from over 13% to 23%. Both primary copy and replica updates contribute to the update network cost. The update network cost for primary copy is the same for all three algorithms. Removing this common portion of the cost, the performance gain in replica update network cost alone is even higher, ranging from over 23% to 41%. The total network cost is the sum of the lookup network cost and the update network cost, as shown in Figure 3.14. For all three algorithms, because of the high local lookup percentage due to replication and offline simulation, the update network cost dominates in the total network cost, and the curves follow the shape of the movement traffic volume. The overall performance gain of UR and MR is over 19% and 31%, respectively.
Figure 3.12: Lookup network cost.
Figure 3.13: Update network cost.
Figure 3.14: Total network cost.
3.6.2 Database Access

The next group of figures compares the performance of database access for each of these algorithms. The goal is to examine the competitiveness of the optimal replication algorithms against the threshold-based algorithm in the database access arena, given their superb network cost performance. Figure 3.15 shows the local lookup percentage, i.e., the percentage of the lookup requests that can be resolved locally. MR performs the best among the three. Knowing the traffic condition completely from off-line simulation, all three algorithms had very high local lookup percentage (> 95%). The number of database lookups and updates are shown in Figures 3.16 and 3.17, respectively. Given the high local lookup percentage, we expect a very small difference in the number of database lookups. Both primary copy and replica copy updates contribute to the number of database updates. Since the number of primary
copy updates is the same for all three algorithms, the more replicas an algorithm distributes, the more database updates it generates. UR has the least amount of replicas, hence the smallest number of database updates. Indeed, both UR and MR perform competitively in the amount of database access.

Figure 3.16: Database lookups.

Figure 3.18 summarizes the performance of all three algorithms by their relative ratios for database access and network cost over a 24-hour period. It is interesting to observe that even though the number of replica updates (RDBU) for MR is very close to that for HIPER, the replica update network cost (RUNC) for MR amounts to only about 60% of the cost for HIPER. This reduction comes from better replica locations being chosen by MR and the multicast replica update strategy. Multicast plays an important role in the performance difference between UR and MR by saving on replica update cost. Overall, MR performs the best in all metrics except in the area of database updates.
Figure 3.17: Database updates.
Figure 3.18: Comparison among the three algorithms: HIPER, UR, and MR. The number of database access and the network cost of UR and MR are normalized to those of HIPER. The following performance measures are considered: database lookups (DBL), lookup network cost (LNC), database updates (DBU), update network cost (UNC), replica database updates (RDBU), replica update network cost (RUNC), and total network cost (TNC).
3.6.3 Replica Distribution and Access

Figure 3.19 shows the replica distribution histograms. Each user enters one of the eleven bins depending on the number of replicas allocated for that user. The large increase in the five replica bin for HIPER is due to the limit (five) on the maximum number of replicas allowed for each user. Even though neither UR nor MR imposes such limit, the total number of replicas distributed by all three algorithms are similar. Besides, both UR and MR tend to give more users a smaller number of replicas. With multicast, MR can afford more replicas without additional replica update network overhead. Figure 3.20 provides the replica lookup access statistics. All three algorithms perform similarly except that only HIPER has replicas that are never used.

![Replica distribution histogram.](image)
Figure 3.20: Replica lookup access statistics, normalized by the total number of replicas.
3.6.4 Discussion

All the simulation results shown above are from a specific set of traffic parameters. To explore the performance advantages of our optimal replication algorithms under different traffic conditions, we examine the effect of variations in the traffic model. For example, increasing user movement speed (or equivalently shrinking RA size) increases the number of RA crossing, hence the number of location update requests for a given period of time. Replication is therefore more expensive. The threshold-based algorithm tends to withhold replicas from a user in this case. Similarly, decreasing call frequency would also frustrate the replication attempt based on thresholds. This limitation is due to the fact that threshold values are traffic dependent and difficult to estimate. Our optimal replication algorithms, on the other hand, work very well with these traffic conditions and show more performance gain over the threshold-based algorithm. Another way to worsen the traffic condition is to reduce the local callee percentage that characterizes locality of calls. We observe that as the local callee percentage decreases, the optimal replication algorithms perform even better than the threshold-based algorithm. Figure 3.21 summarizes the performance of our off-line algorithms with respect to the threshold-based algorithm under these traffic conditions. $TP_1$, $TP_2$, and $TP_3$ represent three different sets of traffic parameters, where $TP_1$ is the original set of traffic parameters used throughout this dissertation, users in $TP_2$ move 10 times faster than users in $TP_1$, and the local callee percentage in $TP_3$ is $\frac{1}{4}$ of that in $TP_1$. The network costs of both UR and MR are normalized to that of HIPER. Both UR and MR perform consistently better than HIPER. These experiments demonstrate that the optimal replication algorithms are not traffic dependent and that they perform competitively under mild traffic condition with respect to the threshold-based algorithm. The performance gap between the optimal and the threshold-based algorithm is even greater under severe traffic conditions.

One additional advantage of the optimal replication algorithm UR is that the communication link cost is included in the calculation, which is not possible in the threshold-based algorithm. In Figure 3.22, the difference between $L_1$ and $L_{10}$ is that the communication link cost is unity in $L_1$ throughout the hierarchy, and uniformly distributed between 1 and 10 in $L_{10}$. The number of database access and network
3.6. SIMULATIONS

Figure 3.21: Comparison among different sets of traffic parameters. Users in $TP_2$ move 10 times faster than users in $TP_1$. The local callee percentage in $TP_3$ is $\frac{1}{4}$ of that in $TP_1$. The network cost of both UR and MR is normalized to that of HIPER. The following performance measures are considered: lookup network cost (LNC), update network cost (UNC), and total network cost (TNC).
costs of both UR and MR are normalized to that of HIPER. With multicast, since replica placement is independent of communication link costs, the number of database access is the same for $L_1$ and $L_{10}$.

![Comparison among different sets of network link costs.](image)

Figure 3.22: Comparison among different sets of network link costs. The network cost of both UR and MR is normalized to that of HIPER. The difference between $L_1$ and $L_{10}$ is that the communication link cost is unity in $L_1$ throughout the hierarchy, and uniformly distributed between 1 and 10 in $L_{10}$. The following performance measures are considered: database lookups (DBL), lookup network cost (LNC), database updates (DBU), update network cost (UNC), and total network cost (TNC).

### 3.7 Summary

In this chapter, we have proposed two optimal off-line replication algorithms that minimize network cost based on the network structure, communication link costs, and user calling and mobility statistics. We have developed our algorithms for both unicast and multicast replica updates. Extensive computer simulations have confirmed that these optimal algorithms perform better than the previously proposed threshold-based
3.7. SUMMARY

algorithm.
Chapter 4

On-line Profile Replication

We begin by making an important distinction between an off-line and an on-line algorithm. An off-line algorithm assumes complete knowledge of user calling and mobility statistics, which may not be available in practice. An on-line algorithm, on the other hand, does not make any assumption about user traffic patterns. Instead, it decides whether to distribute new replicas or delete existing replicas after serving each request, all based on the input sequence seen so far.

In this chapter, we develop optimal on-line replication algorithms that dynamically adjust replica placement according to user calling and mobility patterns. This chapter begins with a detailed survey of some previously proposed replication algorithms and an introduction to the concept of competitive analysis in Section 4.1. It proceeds by solving the replication problem on a single edge in Section 4.2. The edge solutions then become building blocks for solving the replication problem on a tree in Section 4.3. In Section 4.4, we discuss issues related to implementing our on-line algorithms. And in Section 4.5, we compare the performance of these algorithms with that of a previously proposed threshold-based algorithm via large scale computer simulations.
4.1 Related Work

4.1.1 A Replication Algorithm for Mobility Management

As in the case of off-line profile replication, previous researchers have proposed a threshold-based algorithm [51] that compares a user's call to mobility ratio against a pair of thresholds, \( R_{repl} \) and \( R_{det} \). The idea is to replicate when the ratio rises above \( R_{repl} \) and to delete a replica when the ratio drops below \( R_{det} \). The inherent problems with a threshold-based algorithm remain: both threshold determinations and performance analysis are difficult.

4.1.2 General File Allocation Problems for Tree Networks

More generally, the on-line profile replication problem belongs to the family of on-line file allocation (FA) problems that determine the placement of copies of an object, \( \mathcal{X} \), according to an input request sequence \( \sigma \). The number of copies of \( \mathcal{X} \) is at least one, i.e., the replica set \( R \neq \emptyset \). A request can be either a lookup or an update\(^1\). While a lookup request can be served by any copy of \( \mathcal{X} \), an update request changes all copies of \( \mathcal{X} \).

Two special cases of the FA problem impose further restrictions: the migration problem allows only a single copy of \( \mathcal{X} \), and the replication problem deals with a lookup-only request sequence.\(^2\) An on-line algorithm \( A \) is \( c \)-competitive if there exists some constant \( B \) such that the cost of algorithm \( A \) is within a factor \( c \) of the cost of the optimal off-line algorithm for any request sequence \( \sigma \), i.e., \( C_A(\sigma) \leq c \cdot C_{OPT}(\sigma) + B \), where \( C_A(\sigma) \) and \( C_{OPT}(\sigma) \) are the costs incurred by algorithm \( A \) and the optimal off-line algorithm, respectively. While an on-line algorithm can be either deterministic or randomized, we consider only deterministic algorithms in this dissertation. Motivated by the hierarchical database structure in mobility management, we shall focus on

---

\(^1\)The more popular terms are read and write requests in the literature. Lookup is equivalent to read, and update is equivalent to write.

\(^2\)Note this terminology is different from what we use throughout this dissertation. Our replication problem is equivalent to the FA problem without any constraints on the number of replicas allowed and the input sequence.
tree networks instead of general networks. Black et al. [13] proposed a 3-competitive algorithm for the migration problem and a 2-competitive algorithm for the replication problem on tree networks. They also proved that no deterministic algorithm can be better than 3-competitive for the migration problem.

On-line algorithms on a tree network utilize two important techniques: work function and factoring. Via dynamic programming formulations, the work function [15] reconstructs step-by-step the actions taken by an algorithm given an input sequence. The factoring technique [13] transforms a FA problem on a tree into FA problems on individual edges of the tree, such that the total cost of the tree FA problem is the sum of individual costs of the edge FA problems. The method of factoring originates from amortized analysis where the total cost is distributed among individual components. For example, the lookup cost from an originating node $v$ to its closest replica $\hat{v}$ is amortized to each edge along the path from $v$ to $\hat{v}$.

Lund et al. [57] and Heide et al. [1] independently proposed two optimal 3-competitive algorithms for the FA problem in tree networks. Our work not only generalizes their algorithms, but also effectively reduces the overhead associated with them, hence making a family of on-line FA algorithms feasible in practice.

What’s New?

Lund’s algorithm $A_L$ used an offset vector to capture the past input. Our work generalizes this idea and builds a unified framework for solving the FA problem on an edge. In addition to characterizing the past input sequence via an offset vector, our framework constructs the overall solution structure through an offset transition diagram. Specifically, from the counting actions observed in an offset transition diagram, we may draw an analogy between the FA problem on an edge and the classical ski problem in on-line analysis. This analogy gives an intuitive explanation for these algorithms. Our framework is put to the test in the following three tasks. First, we show the correspondence between two previously proposed algorithms by fitting them into our framework, as discussed in Section 4.2.3. Second, we propose a new 3-competitive algorithm $A_T$ in Section 4.2.3. Third, we relax the assumption in the original edge model, allowing the cost of deleting a replica to be non-zero. While
requiring derivations of the offset function from scratch, this modified model bears 
close resemblance to the original one and lends itself to a 3-competitive algorithm 
$A_d$, as described in Section 4.2.3. Proofs of theorems 4.1 and 4.5 are similar to those 
in [57]. They are included not only for completeness but also as reference cases from 
which other variations are derived.

While Lund’s model completely ignored the cost of request propagation, Heide’s 
model include the cost of request propagation via a distributed edge model. The idea 
is to propagate a request from one end to the other only if necessary. In addition to 
the distributed edge model, Heide’s algorithm $A_H$ used a counter pair to capture the 
past input, and they proved that with the cost of request propagation added, their 
algorithm $A_H$ is still 3-competitive. Using offset functions, the proofs of Theorems 4.3 
and 4.9 are alternatives to those in [1]. Theorem 4.9 further states that all other 
algorithms ($A_H$, $A_T$, and $A_d$) also remain 3-competitive with the cost of request 
propagation added.

As proved by previous researchers, neither algorithm $A_L$ nor $A_H$ causes any conflict 
when applied to edges in a tree. Therefore tree algorithms based on algorithms $A_L$ 
and $A_H$ are also 3-competitive. However, the applicability of an edge algorithm to 
a tree FA problem has been proved only for individual cases. The design of an 
edge algorithm is thus somewhat arbitrary, and one can only hope that the solutions 
of individual edges do not conflict with one another. Our study reveals sufficient 
conditions that connect the design of an edge algorithm to that of a tree algorithm, 
such that the design of an edge algorithm is no longer arbitrary, but with the tree 
structure in mind. The foundation of our derivations is Lemma 4.1 proved in [57].

In applying the distributed edge model to a tree, we face the question of how to 
relate the request propagation on an edge to that on a tree. This question is left 
unanswered in [1]. Our study not only identifies this previously missing link, but also 
addresses it in Theorem 4.10. Without this theorem, the cost of request propagation 
can not be added to a tree algorithm while maintaining the same competitive ratio. 
A second cost reduction method, delayed propagation, is considered in the second 
part of Section 4.3.5.

Various implementation issues for these algorithms are discussed in Section 4.4,
where we devise a hybrid approach to further limit the computation and network overhead. The performance of these algorithms are studied via large scale computer simulations.

4.2 Edge Algorithms

4.2.1 Problem Description

An edge problem considers replications on a single edge. We define an edge \((a, b)\) as two nodes, \(a\) and \(b\), connected by a communication link. From Section 4.1, the replica set \(R \neq \emptyset\) implies that either node \(a\) or \(b\) of edge \((a, b)\) must have a replica. Therefore an edge can be in any of three replica states, denoted by \(s^3\): node \(a\) has a replica, \(s = a\); node \(b\) has a replica, \(s = b\); or both \(a\) and \(b\) have replicas, \(s = ab\). Let \(S = \{a, b, ab\}\) be the set of possible replica states. A request can be either a lookup or an update request, originating from either node \(a\) or \(b\). Let \(a^\lambda (b^\lambda)\) denote a lookup request from node \(a (b)\) and \(a^\delta (b^\delta)\) an update request from node \(a (b)\).

Let \(serv(s, \sigma_i)\) be the cost of serving request \(\sigma_i\) in state \(s\). It is defined as follows: for a lookup request \(\sigma_i\), if there is a replica locally, then \(serv(s, \sigma_i) = 0\), otherwise \(serv(s, \sigma_i) = 1\). The cost of serving an update request is 1 unless there is only a local replica, which incurs zero cost. Let \(tran(s', s)\) be the cost of changing from state \(s'\) to state \(s\). The cost of adding a new replica is \(D\) and the cost of deleting a replica is 0. We shall relax this assumption and solve the edge problem when deletion cost is non-zero in Section 4.2.3. Table 4.1 lists both functions \(serv(s, \sigma_i)\) and \(tran(s', s)\).

\[
\begin{array}{c|cccc}
\text{serv}(s, \sigma_i) & a^\lambda & a^\delta & b^\lambda & b^\delta \\
\hline
s & a & 0 & 0 & 1 & 1 \\
\hline
b & 1 & 1 & 0 & 0 \\
ab & 0 & 1 & 0 & 1 \\
\hline
\end{array}
\]

\[
\begin{array}{c|ccc}
\text{tran}(s', s) & a & b & ab \\
\hline
s & a & 0 & D & D \\
\hline
s' & b & D & 0 & D \\
\hline
ab & 0 & 0 & 0 \\
\hline
\end{array}
\]

Table 4.1: Cost functions \(serv(s, \sigma_i)\) and \(tran(s', s)\).

Upon a request, an edge serves the request with some service cost, then it decides

\(^3\)A replica state is equivalent to an edge state. We use these two terms interchangeably.
whether to add or delete a replica. Since an on-line algorithm does not see the complete input sequence \(\sigma_1, \sigma_2, \cdots, \sigma_i, \cdots\), its decisions are based on the requests seen so far, i.e., \(\sigma_1, \sigma_2, \cdots, \sigma_i\), a.k.a. the past input sequence. Given the past input sequence, an edge algorithm resolves where to put replicas, or equivalently, the replica state of the edge.

### 4.2.2 Framework

#### Work Functions

We start with the optimal off-line algorithm, which takes a complete input sequence and produces the optimal replica state. One possible edge algorithm is to run the optimal off-line algorithm with the past input sequence. In particular, we define the optimal cost \(\text{opt}_i\) as the minimum cost of serving requests 1 through \(i\). This edge algorithm calculates the optimal cost \(\text{opt}_i\) and selects the corresponding final state as the replica state. Is this algorithm competitive? The answer is no, because the optimal for the past does not necessarily correspond to the optimal for the future. This suggests that we look at other cost measures beyond the optimal past.

Specifically, we are interested in the minimum cost of ending in all possible states. The minimum cost of serving requests 1 through \(i\) ending in state \(s\) is defined as the work function \(W_i(s)\). The following dynamic programing formulation calculates \(W_i(s)\) from \(W_{i-1}(s)\): 

\[
W_i(s) = \min_{s' \in S} \{W_{i-1}(s') + \text{serv}(s', \sigma_i) + \text{tran}(s', s)\}
\]

It is easy to see that the optimal cost \(\text{opt}_i = \min_{s \in S} W_i(s)\). Without loss of generality, suppose after processing requests 1 to \(i - 1\), \(\text{opt}_{i-1} = W_{i-1}(a), W_{i-1}(b) = W_{i-1}(a) + k\), and \(W_{i-1}(ab) = W_{i-1}(a) + l\), with \(0 \leq k \leq l \leq D\). The work function \(W_i\)'s after processing \(\sigma_i\) are shown in Table 4.2.

**Example 4.1.** Suppose initially only node \(a\) has a replica, \(W_0(a) = 0, W_0(b) = 2, \) and \(W_0(ab) = 2\). Given input sequence \(\sigma = \{b^\lambda, b^\delta, a^\delta, b^\delta, b^\lambda, a^\delta\}\), Figures 4.1 and 4.2 show sample paths of replica states and work function values, respectively.

Since work functions are minimum costs with constraints on the ending states, they can not deviate too much from one another. For our purpose, it is sufficient to
<table>
<thead>
<tr>
<th>$k$</th>
<th>$\sigma_i$</th>
<th>$W_i(a)$</th>
<th>$W_i(b)$</th>
<th>$W_i(ab)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\geq 1$</td>
<td>$a^\lambda$</td>
<td>$W_{i-1}(a)$</td>
<td>$W_{i-1}(a) + \min(k + 1, l)$</td>
<td>$W_{i-1}(a) + l$</td>
</tr>
<tr>
<td></td>
<td>$a^\delta$</td>
<td>$W_{i-1}(a)$</td>
<td>$W_{i-1}(a) + \min(k + 1, D)$</td>
<td>$W_{i-1}(a) + \min(l + 1, D)$</td>
</tr>
<tr>
<td></td>
<td>$b^\lambda$</td>
<td>$W_{i-1}(a) + 1$</td>
<td>$W_{i-1}(a) + k$</td>
<td>$W_{i-1}(a) + l$</td>
</tr>
<tr>
<td></td>
<td>$b^\delta$</td>
<td>$W_{i-1}(a) + 1$</td>
<td>$W_{i-1}(a) + k$</td>
<td>$W_{i-1}(a) + l + 1$</td>
</tr>
<tr>
<td>$= 0$</td>
<td>$a^\lambda$</td>
<td>$W_{i-1}(a)$</td>
<td>$W_{i-1}(a) + \min(1, l)$</td>
<td>$W_{i-1}(a) + l$</td>
</tr>
<tr>
<td></td>
<td>$a^\delta$</td>
<td>$W_{i-1}(a)$</td>
<td>$W_{i-1}(a) + 1$</td>
<td>$W_{i-1}(a) + \min(l + 1, D)$</td>
</tr>
<tr>
<td></td>
<td>$b^\lambda$</td>
<td>$W_{i-1}(a) + \min(1, l)$</td>
<td>$W_{i-1}(a)$</td>
<td>$W_{i-1}(a) + l$</td>
</tr>
<tr>
<td></td>
<td>$b^\delta$</td>
<td>$W_{i-1}(a) + 1$</td>
<td>$W_{i-1}(a)$</td>
<td>$W_{i-1}(a) + \min(l + 1, D)$</td>
</tr>
</tbody>
</table>

Table 4.2: Work function $W_i$, where $W_{i-1}(b) = W_{i-1}(a) + k$ and $W_{i-1}(ab) = W_{i-1}(a) + l$. The opt $i$’s are highlighted.

![Example 4.1: a sample path of replica states](image)

Figure 4.1: Example 4.1: a sample path of replica states. The square box contains the work function value upon request $\sigma_i$. Replica states for $W_i(a)$ are \{a, a, a, a, a, a, a\}, replica states for $W_i(b)$ are \{b, b, b, b, b, b, b\}, and replica states for $W_i(ab)$ are \{b, b, b, b, ab, ab, ab\}. 

88 \hspace{1cm} \textit{CHAPTER 4. ON-LINE PROFILE REPLICA}
Figure 4.2: Example 4.1: a sample path of work function values.
consider their distances from the optimal. The logic behind this is that we don’t need to keep track of the accumulated cost, only how far each replica state is away from the optimal. We define such distance from the optimal as the offset function.

Offset Functions

The offset function \( w_i(s) = W_i(s) - opt_i \) measures the cost difference between an arbitrary state \( s \) and state \( s^* \) chosen by the optimal off-line algorithm after the first \( i \) requests. Since the cost of deleting a replica is zero, \( w_i(ab) \geq w_i(a) \) and \( w_i(ab) \geq w_i(b) \), which implies that either \( a \) or \( b \) is an optimal state, i.e., at least one of \( w_i(a) \) and \( w_i(b) \) must be zero. If \( w_i(s) = 0 \), where \( s = a \) or \( b \), then \( s \) is called a zero offset state (ZOS). The offset values of all three states \( (w_i(a), w_i(b), w_i(ab)) \) form an offset vector. Without loss of generality, we consider offset vector \((0k1)\), with \( w_i(a) = 0 \), \( w_i(b) = k \), \( w_i(ab) = l \), and \( 0 \leq k \leq l \leq D \). The initial value of an offset vector is determined by the initial state of an edge, i.e., \((0DD)\) if only node \( a \) has a replica or \((000)\) if both ends have replicas.

**Example 4.2.** An offset vector \((012)\) means that for the input sequence seen so far, the optimal cost of ending in state \( a \) is the same as \( C_{OPT} \), the cost of the optimal off-line algorithm. The optimal cost of ending in state \( b \) is \( C_{OPT} + 1 \) and the optimal cost of ending in state \( ab \) is \( C_{OPT} + 2 \).

An offset vector captures all the information we need from the past. Upon the current request, we can calculate the next offset vector from the previous one, as shown in Table 4.3 that is derived from Table 4.2.

**Example 4.3.** Suppose \( w_{i-1} = (012) \) and \( D = 2 \). If a request \( \sigma_i = a^k \), then \( w_i = (022) \). Similarly, \( w_i = (022), (001), \) and \( (002) \) upon requests \( a^5 \), \( b^l \), and \( b^k \), respectively.

Another way to visualize the changes in offset vectors according to input requests is via the offset transition diagram as shown in Figure 4.3. We show the offset transition diagram for \( D = 1, 2, \) and \( 3 \). For other values of \( D \), the same principle applies. We make a number of observations regarding the transition diagram. First,
Figure 4.3: Offset transition diagrams.
the offset transition diagram is symmetric between \( a \) and \( b \) because the distinction between them is only a matter of naming. Second, the diamond shape reflects the counting actions in the calculation of offset functions, i.e., if we count the number of lookup requests from node \( b \), \( |b^\lambda| \), and the number of update requests from node \( a \), \( |a^\delta| \), their difference, \( |b^\lambda| - |a^\delta| \), marks the position along the SW-NE diagonals. By symmetry, the difference between the number of lookup requests from node \( a \) and the number of update requests from node \( b \), \( |a^\lambda| - |b^\delta| \), marks the position along the SE-NW diagonals. Essentially, we are counting \(|b^\lambda| - |a^\delta|\) for any prefix of the complete input sequence. Such counting, however, is bounded by the replication cost \( D \), and a count above \( D \) results in self-loops. In other words, the value of \( D \) determines the range of counting, which also explains why the transition diagram for \( D = n - 1 \) is contained within the transition diagram for \( D = n \). This counting behavior is not entirely unexpected, because the calculation of work functions implicitly weighs the benefit of replica state transition against the accumulated service cost. It is through such counting that we are able to characterize an input sequence by an offset vector. The value of an offset vector does not depend on the edge algorithm, but is completely determined by the input sequence. For example, given \( D = 2 \) and an initial offset vector \((0 DD)\), a past input sequence \{\(b^\lambda, b^\delta, a^\delta, b^\delta, b^\lambda, a^\delta\}\) results in an offset vector \((1 02)\), independent of any edge algorithm.
### 4.2. EDGE ALGORITHMS

**Framework**

Instead of looking only at the optimal past, we keep track of all possible replica states by their distances from the optimal via the offset vectors. The offset calculation thus reduces the infinite number of input sequences to a finite number of offset vectors. What remains is to assign some replica state to each offset vector. With a finite number of offset vectors, this assignment is essentially a table lookup. Figure 4.4 illustrates our framework for solving the edge problem, where we first compute the offset vector from the input sequence, then use a table to find an appropriate replica state for this offset vector. The backbone for our framework is the offset transition diagram: the offset vector for any input sequence can be calculated from the offset transition diagram, and as we shall see in Section 4.2.3, the counting actions in the offset transition diagram intuitively explain how replica states are assigned to offset vectors. It is essential to distinguish an offset transition from a replica state transition. The former is a property of the input sequence whereas the latter is the result of replica state assignments.

![Framework for the edge problem](image)

*Figure 4.4: Framework for the edge problem: $f$ calculates the offset vector, $g$ assigns an appropriate replica state according to the offset vector.*
4.2.3 Replica State Assignments

While the offset function compounds the past information into a single offset vector, the assignment of a replica state for each offset vector must be done step-by-step, i.e., upon processing each request. To bridge between these two realms, we introduce the concept of amortized analysis that distributes the total cost over the entire input sequence. This approach simplifies the matter considerably because we only need to deal with the current input request, not the entire past input sequence, essentially converting the competitive requirement for an entire input sequence to that for an individual request. However, the current input request is not isolated, and for its replica state assignment, we must consider the past, because what we can afford in the current step is affected by the total cost paid so far. Moreover, how the cost is distributed directly influences how replica states are assigned. This section begins with the derivation of the minimum requirements on replica state assignments to achieve 3-competitiveness, and ends with discussions on three different algorithms.

Amortized Analysis

In amortized analysis, the total cost is averaged over the entire input sequence: even though the cost of serving an individual request may be high, the average cost can still be low. One popular approach to amortized analysis is the potential method that represents the already paid costs as potentials applicable to serving future requests. By definition, any potential change upon a series of transitions is solely determined by the beginning and ending potentials, independent of any intermediate values.

From Section 4.2.4, we know that an edge algorithm can be at best 3-competitive. Our goal, then, is to design a replica state assignment so that the cost of the edge algorithm is within a factor of 3 of the optimal algorithm, i.e., \( C_e(\sigma) \leq 3 \cdot C_{OPT}(\sigma) + B \) for any sequence \( \sigma \), where \( B \) is some constant. Consider the cost of serving a single request \( \sigma_i \).

- Let \( \Delta opt_i = opt_i - opt_{i-1} \) be the cost increment for the optimal off-line algorithm upon request \( \sigma_i \). The total cost \( C_{OPT}(\sigma) = \sum_{i=1}^{\text{\# of requests}} \Delta opt_i \).

- Let \( \Delta C_{e,i} = serv(s', \sigma_i) + trans(s', s) \) be the cost increment for an edge algorithm
4.2. EDGE ALGORITHMS

serving request $\sigma_i$ in state $s'$ and transitioning to state $s$. The total cost $C_e(\sigma) = \sum_{i=1}^{|v|} \Delta C_{e,i}$.

If we can show that

$$\Delta C_{e,i} + \Delta \Phi_i \leq 3 \cdot \Delta opt_i$$

(4.1)

for all request $\sigma_i$, where $\Phi$ is a potential function and $|\Delta \Phi_i|$ is bounded by some constant $B$, summing the inequalities for each request in the sequence $\sigma$, we obtain $C_e(\sigma) \leq 3 \cdot \text{OPT}(\sigma) + B$. Using a potential function, we can thus prove an edge algorithm’s competitiveness in serving any input sequence through its competitiveness in serving individual requests. Furthermore, because an offset vector essentially captures all the relevant information from the past, we need to consider only the possible combinations of current input requests, offsets, and replica states. Similarly, potentials should also depend on offsets and replica states.

Minimum Requirements

As the left hand side of Inequality 4.1 is determined by $\Delta opt_i$, we tabulate its values upon a request $\sigma_i$ in Table 4.4, and in Figure 4.5, we show the offset transitions with non-zero $\Delta opt_i$. Note that only transitions with non-zero $\Delta opt_i$ are shown. In

<table>
<thead>
<tr>
<th>$k$</th>
<th>$\sigma_i$</th>
<th>$\Delta opt_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\geq 1$</td>
<td>$a^\lambda$</td>
<td>0</td>
</tr>
<tr>
<td>$\geq 1$</td>
<td>$a^\delta$</td>
<td>0</td>
</tr>
<tr>
<td>$\geq 1$</td>
<td>$b^\lambda$</td>
<td>1</td>
</tr>
<tr>
<td>$\geq 1$</td>
<td>$b^\delta$</td>
<td>1</td>
</tr>
<tr>
<td>$= 0$</td>
<td>$a^\lambda$</td>
<td>0</td>
</tr>
<tr>
<td>$= 0$</td>
<td>$a^\delta$</td>
<td>0</td>
</tr>
<tr>
<td>$= 0$</td>
<td>$b^\lambda$</td>
<td>0</td>
</tr>
<tr>
<td>$= 0$</td>
<td>$b^\delta$</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 4.4: $\Delta opt_i$ values upon $\sigma_i$, where $w_{i-1}(a) = 0$, $w_{i-1}(b) = k$, $w_{i-1}(ab) = l$.

In particular, self-loops do not appear in Figure 4.5, therefore $\Delta opt_i = 0$ for these transitions. With no offset change and assuming no replica state transitions associated with them, self-loops carry no potential change, i.e., $\Delta \Phi = 0$. Therefore, Inequality (4.1)
Figure 4.5: $\Delta opt_i = 1$ for the transitions shown, and $\Delta opt_i = 0$ for the transitions not shown. $D = 2$.

becomes

$$\Delta C_{e,i} \leq 0. \quad (4.2)$$

This is possible only if all self-loops are served with zero cost, and we term this condition as the minimum requirements. The corresponding replica state assignments are shown in Figure 4.6, and summarized as follows,

1. $w_i = (0DD), \ s = a$.

2. $w_i = (000), \ s = ab$.

3. $w_i = (0ll), \text{ where } 0 < l < D, \ s = a \text{ or } s = ab$.

State transitions include replicating at node $a$ if the offset vector is $(0ll)$ and deleting the replica at node $b$ if the offset vector is $(0DD)$.

In order to provide an intuitive explanation to the above replica state assignments, we take a small digression to introduce a classical problem in on-line algorithm design: the skiing problem. Suppose a skier must decide every day he goes skiing whether to rent or buy skis, until he decides to buy them. The skier does not know how many days he will go on skiing. Call this number of days $G$. The cost to rent skis for a day is
Figure 4.6: Minimum requirements. Next to each offset are the permitted replica states.

1 unit, while the cost to buy skis is \( H \) units. The optimal off-line algorithm is to buy the skis the first day if \( G > H \) and to rent every day otherwise. The optimal on-line algorithm is to rent skis for \( H - 1 \) days and then buy on the \( H \)th day, or equivalently, \( G > H - 1 \). Its performance is within a factor of \( (2 - \frac{1}{\pi}) \) of the optimum. Intuitively, the skier will delay buying until the accumulated cost of rental reaches the cost of purchase.

In the edge problem, the cost of replication \( D \) resembles the cost of buying the skis. The difference between the number of lookup requests from node \( a \), and the number of update requests from node \( b \), \(|a^\lambda| - |b^\ell|\), parallels the number of rental days, since a lookup request from node \( a \) and an update request from node \( b \) have opposite effects on the replica decision for node \( a \). Intuitively, the more lookup requests from node \( a \), the more we gain by replicating at node \( a \), whereas the more update requests from node \( b \), the more we pay in updating a replica at node \( a \). Like the skiing problem, we delay replicating at node \( a \) until the difference \(|a^\lambda| - |b^\ell|\) reaches the replication cost \( D \).
3-competitive Algorithms

Without defining a particular potential function, we derived the minimum requirements from zero potential change. Next we shall delve into defining a potential function. What factors does a potential function depend upon? First, potentials should relate to the past input sequence. Since an offset vector completely characterizes an input sequence, it is natural to define potentials as a function of the offsets. Second, although the cost of distributing a new replica is $D$, such “prepaid” replication cost may benefit future lookup requests. Potentials then should also incorporate the replica state of the edge.

Let $\Phi(a)$, $\Phi(b)$, and $\Phi(ab)$ be the potentials for states $a$, $b$, and $ab$, respectively. Since both service cost and replica state transition cost contribute to $\Delta C_{e,i}$, to satisfy Inequality (4.1), we may define potentials such that,

- The cost of replica state transitions is completely absorbed by changes in the potential, i.e., $\Delta C_{e,i} + \Delta \Phi_i \leq 0$ holds.

- Inequality (4.1) is true for all combinations of input requests, offsets, and allowable replica states when there are no changes in replica state. In particular, since we have assumed so far that the cost of replica deletion is zero, the potential change for deleting a replica must be $\leq 0$, or $\Phi(a) - \Phi(ab) \leq 0$. To balance the cost of distributing a replica, the potential change for adding a replica should be at least $-D$, or $\Phi(ab) - \Phi(b) \leq -D$.

Inequality (4.1) can be rewritten as,

$$\Delta \Phi_i \leq 3 \cdot \Delta opt_i - \Delta C_{e,i}, \quad (4.3)$$

which a potential function must satisfy. Intuitively, the potential change $\Delta \Phi_i$ increases when $\Delta opt_i = 1$, because the service cost is $\leq 1$, i.e., we have some “savings” that can be used later.

Figure 4.7(a) shows how the optimal cost $\Delta opt_i$ changes in the offset transition diagram. While $\Delta opt_i$ does not depend on the replica state because of optimality, $\Delta C_{e,i}$ does because of service cost $serv(\cdot)$. We shall investigate potential changes
in the offset transition diagram for state \( a, b, \) and \( ab \) separately. By symmetry, we consider only half of the diagram. In state \( a, \) as shown in Figure 4.7(c), offset vector \((000)\) expects \( \Delta \Phi_i \leq -1 \) for \( b^\lambda, \) which is impossible because \( \Delta \Phi_i = 0 \) for self-loops where there are no replica state transitions. This constraint eliminates state \( a \) as a candidate for offset \((000)\). Similarly, in Figure 4.7(d), offsets of the form \((0ll)\) eliminate state \( b \) because of self-loops generated by \( a^\lambda \)'s. In Figure 4.7(b), offset \((0DD)\) eliminates state \( ab \) because of the self-loops generated by \( a^b \)'s. These constraints are the same as the minimum requirements discussed previously, and they necessitate certain replica state transitions, the cost of which must be balanced by potential changes. Replica state assignments, then, must go hand in hand with the definition of a potential function. Specifically, offset \((000)\) calls for a transition to state \( ab \) from state \( a \) or \( b, \) with \( \Phi(ab) - \Phi(a) \leq -D \) and \( \Phi(ab) - \Phi(b) \leq -D. \) Similarly, offsets of the form \((0ll)\) call for a transition to state \( ab \) from state \( b, \) with \( \Phi(ab) - \Phi(b) \leq -D, \) and offset \((0DD)\) calls for a transition to state \( a \) from state \( ab, \) with \( \Phi(a) - \Phi(ab) \leq 0. \) With certain replica states excluded by these constraints, the remaining potential changes must satisfy \( 0 \leq \Delta \Phi_i(a) \leq 2 \) and \( -1 \leq \Delta \Phi_i(b) \leq 3, \) as shown in Figures 4.7(c) and (d). The symmetry between \( a \) and \( b \) requires that \( \Phi(a) = \Phi(b) \) for offsets of the form \((00l), \) and the potential function must be non-negative. We summarize these conditions as follows,

1. Minimum requirements (for state transition):
   
   \( \Phi(ab) - \Phi(a) \leq -D \) and \( \Phi(ab) - \Phi(b) \leq -D \) for offset \((000).\)
   
   \( \Phi(ab) - \Phi(b) \leq -D, \) for offsets of the form \((0ll), \) where \( 0 < l < D. \)
   
   \( \Phi(a) - \Phi(ab) \leq 0 \) for offset \((0DD).\)

2. Constraints on the potential change:
   
   \( 0 \leq \Delta \Phi_i(a) \leq 2. \)
   
   \( 1 \leq \Delta \Phi_i(b) \leq 3. \)
   
   \( 1 \leq \Delta \Phi_i(ab) \leq 2 \) along SE-NW diagonals and \( 1 \leq \Delta \Phi_i(ab) \leq 3 \) along SW-NE diagonals.
3. Symmetric requirement: $\Phi(a) = \Phi(b)$ for offsets of the form $(00l)$.

4. Non-negative requirement: $\Phi(s) \geq 0$, where $s \in \{a, b, ab\}$.

The minimum requirements are pertinent only to offset vectors along the uppermost diagonals in the offset transition diagram, and there are no restrictions on the other offset vectors, i.e., all replica states are allowable. Thus it is conceivable that various definitions for a potential may satisfy Inequality (4.1). We shall show three possible definitions along with their replica state assignments. If we choose $\Phi(a) = \Phi(ab) = 0$ for $(0DD)$, the smallest potential values for $(000)$ are $\Phi(ab) = D$, $\Phi(a) = \Phi(b) = 2D$. We have the following potential function and algorithm $A_L$.

$$
\Phi(s, k) = \begin{cases} 
2D - 2k & s = a \\
2D - k & s = b \\
D - k & s = ab 
\end{cases} \quad (4.4)
$$

This definition satisfies $\Phi \geq 0$ because $k \leq l \leq D$. When $k = 0$, we have $\Phi(a, 0) = \Phi(b, 0) = 2D$. This potential function and algorithm $A_L$ were first proposed by Lund et al. [57], and the replica state assignments fulfill the minimum requirements. Our analysis here leads to the same potential function. The proof of Theorem 4.1 is included not only for completeness but also as a reference from which other variations are derived, as shown in the proofs of Theorems 4.2, 4.3, and 4.4.

**Algorithm 4.1. $A_L$ (Lund 1999) [57]**

- **Replica state assignments:**
  1. $w_i = (0DD)$, $s = a$.
  2. $w_i = (000)$, $s = ab$.
  3. $w_i = (0ll)$, where $0 < l < D$, $s = a$ or $s = ab$.

- **State transitions:** as shown in Table 4.5.

**Theorem 4.1.** Algorithm $A_L$ is 3-competitive.
(a) $\Delta opt_i$.

(b) $\Delta \Phi_i(ab)$.

(c) $\Delta \Phi_i(a)$.

(d) $\Delta \Phi_i(b)$.

Figure 4.7: Offset transition diagrams with input requests replaced by $\Delta opt_i$ and $\Delta \Phi_i$ constraints. $D = 3$. 
Table 4.5: State transitions for algorithm $A_L$.

<table>
<thead>
<tr>
<th>$w_i$</th>
<th>$s_{i-1} \leadsto s_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0DD)</td>
<td>$b \leadsto a$</td>
</tr>
<tr>
<td></td>
<td>$ab \leadsto a$</td>
</tr>
<tr>
<td>(000)</td>
<td>$a \leadsto ab$</td>
</tr>
<tr>
<td></td>
<td>$b \leadsto ab$</td>
</tr>
<tr>
<td>(0ll), $0 &lt; l &lt; D$</td>
<td>$b \leadsto ab$</td>
</tr>
</tbody>
</table>

Proof. Using the potential function defined in (4.4), we show that Inequality (4.1) holds for all possible replica state transitions prescribed by algorithm $A_L$ and all combinations of input requests, offsets, and replica states, as shown in Tables 4.6 and 4.7.

<table>
<thead>
<tr>
<th>$w_i$</th>
<th>$s_{i-1} \leadsto s_i$</th>
<th>$\Delta C_{e,i}$</th>
<th>$\Delta \Phi_i$</th>
<th>$\Delta C_{e,i} + \Delta \Phi_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0DD)</td>
<td>$b \leadsto a$</td>
<td>D</td>
<td>$-D$</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>$ab \leadsto a$</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>(000)</td>
<td>$a \leadsto ab$</td>
<td>D</td>
<td>$-D$</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>$b \leadsto ab$</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>(0ll), $0 &lt; l &lt; D$</td>
<td>$b \leadsto ab$</td>
<td>D</td>
<td>$-D$</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 4.6: Inequality (4.1) holds for all possible replica state transitions prescribed by algorithm $A_L$.

For potential definition (4.4), the potential difference between state $b$ and state $ab$ is always $D$, large enough to afford distributing a replica at node $a$. This fact is used by the following algorithm $A_T$.

**Algorithm 4.2. $A_T$**

- **Replica state assignments:**
  1. $w_i = (0DD)$, $s = a$.
  2. $w_i = (000)$, $s = ab$.
  3. $w_i = (0kl)$, where $0 < k < D$, $s = a$ or $s = ab$.

- **State transitions:** as shown in Table 4.8.
### 4.2. EDGE ALGORITHMS

<table>
<thead>
<tr>
<th>( w_{i-1} )</th>
<th>( \sigma_i )</th>
<th>( s )</th>
<th>( \Delta C_{e,i} )</th>
<th>( \Delta \Phi_i )</th>
<th>( \Delta \text{opt}_i )</th>
<th>( w_{i-1} )</th>
<th>( \sigma )</th>
<th>( s )</th>
<th>( \Delta C_{e,i} )</th>
<th>( \Delta \Phi_i )</th>
<th>( \Delta \text{opt}_i )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a^\lambda )</td>
<td>( a )</td>
<td>0</td>
<td>0, -2</td>
<td>0</td>
<td>( a )</td>
<td>0</td>
<td>0, -2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( b )</td>
<td>1</td>
<td>-1</td>
<td></td>
<td></td>
<td>( b )</td>
<td>1</td>
<td>-1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( ab )</td>
<td>0</td>
<td>0, -1</td>
<td></td>
<td></td>
<td>( ab )</td>
<td>0</td>
<td>0, -1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( a^\delta )</td>
<td>( a )</td>
<td>0</td>
<td>0, -2</td>
<td>0</td>
<td>( a )</td>
<td>0</td>
<td>-2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( b )</td>
<td>1</td>
<td>-1</td>
<td></td>
<td></td>
<td>( b )</td>
<td>1</td>
<td>-1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( ab )</td>
<td>1</td>
<td>-1</td>
<td></td>
<td></td>
<td>( ab )</td>
<td>1</td>
<td>-1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( b^\lambda )</td>
<td>( a )</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>( a )</td>
<td>1</td>
<td>-1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( b )</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td>( b )</td>
<td>0</td>
<td>0, -2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( ab )</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td>( ab )</td>
<td>0</td>
<td>0, -1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( b^\delta )</td>
<td>( a )</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>( a )</td>
<td>1</td>
<td>-1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( b )</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td>( b )</td>
<td>0</td>
<td>-2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( ab )</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td>( ab )</td>
<td>1</td>
<td>-1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.7: Inequality (4.1) holds for all combinations of input requests, offsets, and replica states in algorithm \( A_L \).

### Table 4.8: State transitions for algorithm \( A_T \).

<table>
<thead>
<tr>
<th>( w_i )</th>
<th>( s_{i-1} \rightarrow s_i )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( (0DD) )</td>
<td>( b \rightarrow a )</td>
</tr>
<tr>
<td></td>
<td>( ab \rightarrow a )</td>
</tr>
<tr>
<td>( (000) )</td>
<td>( a \rightarrow ab )</td>
</tr>
<tr>
<td></td>
<td>( b \rightarrow ab )</td>
</tr>
<tr>
<td>( (0kl), 0 &lt; k &lt; D )</td>
<td>( b \rightarrow ab )</td>
</tr>
</tbody>
</table>
Unlike algorithm $A_L$, where all replica state transitions take place along the uppermost diagonals, algorithm $A_T$ distributes replicas much “earlier”, in fact as soon as we pass offsets of the form $(00l)$.

**Theorem 4.2.** Algorithm $A_T$ is 3-competitive.

**Proof.** Using the same potential function definition in (4.4), we show that Inequality (4.1) holds for all possible replica state transitions prescribed by algorithm $A_T$ and all combinations of input requests, offsets, and replica states, as shown in Tables 4.9 and 4.10. □

<table>
<thead>
<tr>
<th>$w_i$</th>
<th>$s_{i-1} \sim s_i$</th>
<th>$\Delta C_{e,i}$</th>
<th>$\Delta \Phi_i$</th>
<th>$\Delta C_{e,i} + \Delta \Phi_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(0DD)$</td>
<td>$b \sim a$</td>
<td>$D$</td>
<td>$-D$</td>
<td>$0$</td>
</tr>
<tr>
<td></td>
<td>$ab \sim a$</td>
<td>$0$</td>
<td>$0$</td>
<td>$0$</td>
</tr>
<tr>
<td>$(000)$</td>
<td>$a \sim ab$</td>
<td>$D$</td>
<td>$-D$</td>
<td>$0$</td>
</tr>
<tr>
<td></td>
<td>$b \sim ab$</td>
<td>$D$</td>
<td>$-D$</td>
<td>$0$</td>
</tr>
</tbody>
</table>

Table 4.9: Inequality (4.1) holds for all possible replica state transitions prescribed by algorithm $A_T$. This table is identical to Table 4.6 except for the replica state transition for $w_i = (0kl)$, where $0 < k < D$.

A third variation was proposed by Heide et al. [1], and they proved that algorithm $A_H$ is 3-competitive using a data structure called counter pairs. The following is our alternative proof of its competitiveness via offset functions.

**Algorithm 4.3.** $A_H$ (Heide 1999) [1]

- **Replica state assignments:**
  1. $w_i = (0DD)$, $s = a$.
  2. $w_i = (000)$, $s = ab$.
  3. $w_i = (0ll)$, where $0 < l < D$, $s = a$ or $s = ab$.
  4. $w_i = (0kD)$, where $k < D$, $s = a$ or $s = b$.

- **State transitions:** as shown in Table 4.11.
### 4.2. Edge Algorithms

<table>
<thead>
<tr>
<th>$w_{i-1}$</th>
<th>$\sigma_i$</th>
<th>$s$</th>
<th>$\Delta C_{e,i}$</th>
<th>$\Delta \Phi_i$</th>
<th>$\Delta \text{opt}_i$</th>
<th>$w_{i-1}$</th>
<th>$\sigma$</th>
<th>$s$</th>
<th>$\Delta C_{e,i}$</th>
<th>$\Delta \Phi_i$</th>
<th>$\Delta \text{opt}_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$a^\lambda$</td>
<td>$a$</td>
<td>0</td>
<td>0, $-2$</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>$a^\lambda$</td>
<td>$a$</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$b$</td>
<td>Forbidden</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$b$</td>
<td>1</td>
<td>$-1$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$ab$</td>
<td>0</td>
<td>0, $-1$</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>$ab$</td>
<td>0</td>
<td>0, $-1$</td>
</tr>
<tr>
<td></td>
<td>$a^\delta$</td>
<td>$a$</td>
<td>0</td>
<td>0, $-2$</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>$a$</td>
<td>0</td>
<td>$-2$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$b$</td>
<td>Forbidden</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$b$</td>
<td>1</td>
<td>$-1$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$ab$</td>
<td>1</td>
<td>$-1$</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>$ab$</td>
<td>1</td>
<td>$-1$</td>
</tr>
<tr>
<td></td>
<td>$b^\lambda$</td>
<td>$a$</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>$a$</td>
<td>1</td>
<td>$-1$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$b$</td>
<td>Forbidden</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$b$</td>
<td>0</td>
<td>0, $-2$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$ab$</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>$ab$</td>
<td>0</td>
<td>0, $-1$</td>
</tr>
<tr>
<td></td>
<td>$b^\delta$</td>
<td>$a$</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>$a$</td>
<td>1</td>
<td>$-1$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$b$</td>
<td>Forbidden</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$b$</td>
<td>0</td>
<td>$-2$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$ab$</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>$ab$</td>
<td>1</td>
<td>$-1$</td>
</tr>
</tbody>
</table>

Table 4.10: Inequality (4.1) holds for all combinations of input requests, offsets, and replica states in algorithm $A_T$. This table is identical to Table 4.7 except for the forbidden state $b$ when $w_i = (0kl)$, with $k > 0$.

<table>
<thead>
<tr>
<th>$w_i$</th>
<th>$s_{i-1} \leadsto s_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0DD)</td>
<td>$b \leadsto a$</td>
</tr>
<tr>
<td></td>
<td>$ab \leadsto a$</td>
</tr>
<tr>
<td>(000)</td>
<td>$a \leadsto ab$</td>
</tr>
<tr>
<td></td>
<td>$b \leadsto ab$</td>
</tr>
<tr>
<td>(0ll), $0 &lt; l &lt; D$</td>
<td>$b \leadsto ab$</td>
</tr>
<tr>
<td>(0kD), $k &gt; 0$</td>
<td>$ab \leadsto a$</td>
</tr>
</tbody>
</table>

Table 4.11: State transitions for algorithm $A_H$. 
CHAPTER 4. ON-LINE PROFILE REPLICATION

Compared with algorithm $A_L$, algorithm $A_H$ prescribes additional state transitions along the lowest diagonals in the offset transition diagram, which the potential function defined in (4.4) can not sustain, since $\Phi(a) - \Phi(ab) = D - k > 0$, if $k < D$. How should we modify the potential function to accommodate algorithm $A_H$? Note that additional state transitions occur when $l = D$, yet the potential function defined in (4.4) does not depend on $l$, i.e., offsets with the same $k$ value share the same potential. To differentiate these offsets, potential $\Phi(ab)$ must include $l$. Because $l - k$ is constant along the SW-NE diagonals, if we increase potential $\Phi(ab)$ by $l - k$, $\Delta\Phi_i(ab)$ along the SW-NE diagonals remains unchanged. Furthermore, since $\Delta(l - k) = 1$ along the SE-NW diagonals, increasing $\Phi(ab)$ by $l - k$ results in $\Delta\Phi_i = 2$ along these diagonals, and $0 \leq \Delta\Phi_i(ab) \leq 2$ is still satisfied. Our potential function is defined as follows,

$$\Phi(s, k, l) = \begin{cases} 
2D - 2k & s = a \\
2D - k & s = b \\
D - k + (l - k) & s = ab 
\end{cases} \quad (4.5)$$

Again $\Phi \geq 0$, because $k \leq l \leq D$. When $k = 0$, we have $\Phi(a, 0, l) = \Phi(b, 0, l) = 2D$. When $l = D$, we have $\Phi(ab, k, D) = \Phi(a, k, D) = 2D - 2k$, thus $\Delta\Phi_i = 0$ for a transition from state $ab$ to state $a$. Figure 4.8 compares the potentials defined in (4.4) and (4.5).

**Theorem 4.3.** Algorithm $A_H$ is 3-competitive.

*Proof. Using the potential function defined in (4.5), we show that Inequality (4.1) holds for all possible replica state transitions prescribed by algorithm $A_H$ and all combinations of input requests, offsets, and replica states, as shown in Tables 4.12 and 4.13.*

**Example 4.4.** Suppose initially the replica is at node $a$, $w_0 = (022)$, $D = 2$, and the input sequence $\sigma = \{b^\lambda, b^\delta, a^\delta, b^\delta, b^\lambda, a^\delta\}$. Table 4.14 lists the replica states taken by the four algorithms: Optimal off-line, $A_L$, $A_H$, and $A_T$. Algorithms $A_L$ and $A_H$ behave identically for this example, whereas $A_T$ acts differently at the 5-th step. All three algorithms are 3-competitive, i.e., their cost is within a factor of 3 of the optimal off-line algorithm.
4.2. EDGE ALGORITHMS

(a) \(\Phi(ab)\) defined in (4.4).

(b) \(\Phi(ab)\) defined in (4.5).

Figure 4.8: Comparison between the potentials defined in (4.4) and (4.5). The numbers on top of the offsets are the potentials. \(D = 3\).

<table>
<thead>
<tr>
<th>(w_i)</th>
<th>(s_{i-1} \sim s_i)</th>
<th>(\Delta C_{e,i})</th>
<th>(\Delta \Phi_i)</th>
<th>(\Delta C_{e,i} + \Delta \Phi_i)</th>
</tr>
</thead>
<tbody>
<tr>
<td>((0DD))</td>
<td>(b \sim a)</td>
<td>(D)</td>
<td>(-D)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(ab \sim a)</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>((000))</td>
<td>(a \sim ab)</td>
<td>(D)</td>
<td>(-D)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(b \sim ab)</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>((0ll), 0 &lt; l &lt; D)</td>
<td>(b \sim ab)</td>
<td>(D)</td>
<td>(-D)</td>
<td></td>
</tr>
<tr>
<td>((0kD), k &gt; 0)</td>
<td>(ab \sim a)</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.12: Inequality (4.1) holds for all possible replica state transitions prescribed by algorithm \(A_H\). The additional state transitions are made possible by the potential function defined in (4.5) such that \(\Phi(a) - \Phi(ab) = 0\) for \(l = D\).
<table>
<thead>
<tr>
<th>$w_{i-1}$</th>
<th>$\sigma_i$</th>
<th>$s$</th>
<th>$\Delta C_{e,i}$</th>
<th>$\Delta \Phi_i$</th>
<th>$\Delta opt_i$</th>
<th>$w_{i-1}$</th>
<th>$\sigma_i$</th>
<th>$s$</th>
<th>$\Delta C_{e,i}$</th>
<th>$\Delta \Phi_i$</th>
<th>$\Delta opt_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a^\lambda$</td>
<td>$a$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>$a^\lambda$</td>
<td>$a$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$b$</td>
<td>1</td>
<td>$-1$</td>
<td>0</td>
<td> </td>
<td> </td>
<td>$b$</td>
<td>1</td>
<td>$-1$</td>
<td>0</td>
<td> </td>
<td> </td>
</tr>
<tr>
<td>$ab$</td>
<td>0</td>
<td>0</td>
<td>$-2$</td>
<td>0</td>
<td> </td>
<td>$ab$</td>
<td>0</td>
<td>0</td>
<td>$-2$</td>
<td>0</td>
<td> </td>
</tr>
<tr>
<td>$a^\delta$</td>
<td>$a$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>$a^\delta$</td>
<td>$a$</td>
<td>0</td>
<td>$-2$</td>
<td>0</td>
<td> </td>
</tr>
<tr>
<td>$b$</td>
<td>1</td>
<td>$-1$</td>
<td>0</td>
<td> </td>
<td> </td>
<td>$b$</td>
<td>1</td>
<td>$-1$</td>
<td>0</td>
<td> </td>
<td> </td>
</tr>
<tr>
<td>$ab$</td>
<td>1</td>
<td>$-1$</td>
<td>0</td>
<td> </td>
<td> </td>
<td>$ab$</td>
<td>1</td>
<td>$-1$</td>
<td>0</td>
<td> </td>
<td> </td>
</tr>
</tbody>
</table>

Table 4.13: Inequality (4.1) hold for all combinations of input requests, offsets, and replica states in algorithm $A_H$. This table is similar to Table 4.7 except for the underlined entries.

<table>
<thead>
<tr>
<th>$i$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A_{L}(A_H)$</td>
<td>(022)</td>
<td>(011)</td>
<td>(001)</td>
<td>(012)</td>
<td>(002)</td>
<td>(102)</td>
<td>(202)</td>
<td>(102)</td>
</tr>
<tr>
<td>$A_T$</td>
<td>$a$</td>
<td>$a$</td>
<td>$a$</td>
<td>$a$</td>
<td>$a$</td>
<td>$a$</td>
<td>$b$</td>
<td>$b$</td>
</tr>
</tbody>
</table>

Table 4.14: Example 4.4.
4.2. EDGE ALGORITHMS

Correspondence Between $A_L$ and $A_H$

Using the framework developed in Section 4.2.2 and potential functions defined in Section 4.2.3, we have derived both algorithms $A_L$ and $A_H$. Yet they were not first proposed within the context of our framework. Before fitting them into our framework, we note that the correspondence between them is not obvious because they use different cost measures and prescribe different sets of replication rules. However, if we fit both of them into our general framework, we can quickly see their correspondence. While Lund’s algorithm $A_L$ calculates offset vectors, Heide’s algorithm $A_H$ calculates counter pairs. We incorporate both of these algorithms $A_L$ and $A_H$ into our framework by creating their corresponding transition diagrams, as shown in Figure 4.9. From these transition diagrams, we can easily see the correspondence between an offset vector and a counter pair. In fact, offset vectors and counter pairs are equivalent to one another, $(0kl) \Leftrightarrow (D - (l - k), D - l)$. Both algorithms behave identically in reducing an input sequence. Using the transition diagram, we can also match the replica states assigned by these two algorithms. We see how both algorithms make similar choices of replica states. Heide’s algorithm $A_H$ has additional deletion rules. They are the counterparts of replication rules. i.e., if we count up to $D$ to replicate, then we should delete the replica after counting down $D$.

By putting these two previously proposed algorithms in our framework, we have shown the correspondence between them.

Deletion Cost $\neq 0$

So far, we have assumed that the cost of deleting a replica is zero. This assumption is relaxed in this section. Let the cost of deleting a replica be $d$, and $0 < d \leq D$. Table 4.15 defines $\mathit{trans}(\cdot)$ with deletion cost $d$.

Without loss of generality, we consider offset vector $(k_a k_b l)$, with $w_i(a) = k_a$, $w_i(b) = k_b$, $w_i(ab) = l$, and $0 \leq k_a, k_b \leq l + d$, $0 \leq l \leq D$. The initial value of the offset vector is determined by the initial state of an edge, i.e., $(0, D + d, D)$ if only node $a$ has a replica or $(dd0)$ if both ends have replicas. Upon the current request, we calculate the next offset vector from the previous one, as shown in Table 4.16.
(a) Lund’s algorithm $A_L$.

(b) Heide’s algorithm $A_H$.

Figure 4.9: Correspondence between algorithms $A_L$ and $A_H$. An offset $(0kl)$ is equivalent to a counter pair $(D - (l - k), D - l)$. Next to each offset are the permitted replica states. The differences in replica state assignments are underlined. Input requests are not shown in the transition diagrams for clarity.
<table>
<thead>
<tr>
<th>$w_{i-1}$</th>
<th>$\sigma_i$</th>
<th>$w_i(a)$</th>
<th>$w_i(b)$</th>
<th>$w_i(ab)$</th>
<th>$\Delta \text{opt}_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$a^\lambda$</td>
<td>0</td>
<td>$\min(k_b + 1, l + d)$</td>
<td>$l$</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>$a^\delta$</td>
<td>0</td>
<td>$\min(k_b + 1, D + d)$</td>
<td>$\min(l + 1, D)$</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>$b^\lambda$</td>
<td>0</td>
<td>$k_b - 1$</td>
<td>$l - 1$</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>$b^\delta$</td>
<td>0</td>
<td>$k_b - 1$</td>
<td>$l$</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>$a^\lambda$</td>
<td>0</td>
<td>1</td>
<td>$l$</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>$a^\delta$</td>
<td>0</td>
<td>1</td>
<td>$\min(l + 1, D)$</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>$b^\lambda$</td>
<td>1</td>
<td>0</td>
<td>$l$</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>$b^\delta$</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>$\min(l + 1, D)$</td>
</tr>
<tr>
<td></td>
<td>$a^\lambda$</td>
<td>$k_a$</td>
<td>$\min(k_b + 1, d)$</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>$a^\delta$</td>
<td>$k_a - 1$</td>
<td>$k_b$</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>$b^\lambda$</td>
<td>$\min(k_a + 1, d)$</td>
<td>$k_b$</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>$b^\delta$</td>
<td>$k_a$</td>
<td>$k_b - 1$</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>$a^\lambda$</td>
<td>0</td>
<td>$\min(k_b + 1, d)$</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>$a^\delta$</td>
<td>0</td>
<td>$k_b + 1$</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>$b^\lambda$</td>
<td>1</td>
<td>$k_b$</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>$b^\delta$</td>
<td>0</td>
<td>$k_b - 1$</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 4.16: Offset change upon $\sigma_i$, where $0 < d < D$, $w_{i-1}(a) = k_a$, $w_{i-1}(b) = k_b$, $w_{i-1}(ab) = l$. 

\[
\begin{array}{c|c|c|c|c|c}
\text{tran}(s', s) & s \\
\hline
a & 0 & D & D \\
\hline
b & D & 0 & D \\
\hline
ab & d & d & 0 \\
\end{array}
\]

Table 4.15: $\text{Trans}(\cdot)$ function
We show the offset transition diagram for $D = 2, d = 1$ in Figure 4.10. If we represent an edge problem by a replication and deletion cost pair, $(D, d)$, there can be a one-to-one correspondence between the offset transition diagrams of $(D, d)$ and $(D + d, 0)$. From the transition diagram, we can see that both $D$ and $d$ contribute to the counting range. In addition to accumulating cost with respect to the replication cost $D$, referred to as $D$-counting, the top portion of the transition diagram, as outlined by the dashed lines, is devoted to counting with respect to $d$, referred to as $d$-counting. Figure 4.11 shows $d$-counting for $d = 0, 1,$ and $2$.

![Figure 4.10: Offset transition diagram, $D = 2, d = 1$. The dashed lines outlines the $d$-counting region.](image)

Similar to the $d = 0$ case, we shall design a potential function and a 3-competitive algorithm together for the $d \neq 0$ case. Because of the non-zero deletion cost, the optimal cost increment $\Delta opt_i$ varies differently from the $d = 0$ case. In fact, along the two diagonals, $\Delta opt_i$’s are opposite in the $d$-counting region to those in the $D$-counting region, as shown in Figure 4.12(a). Consequently, the rates of potential change differ as well, as shown in Figures 4.12(b), (c), and (d). The minimum requirements,
4.2. EDGE ALGORITHMS

(a) $d = 0$.

(b) $d = 1$.

(c) $d = 2$.

Figure 4.11: $d$-counting regions in the offset transition diagrams, $d = 0, 1,$ and 2.
however, remain the same, and so do the symmetric and non-negative requirements. We define the following potential function:

\[
\Phi(s, k_a, k_b) = \begin{cases} 
2 \cdot (D + d) - 2 \cdot k_b - k_a & s = a \\
2 \cdot (D + d) - k_b - 2k_a & s = b \\
(D + 2d) - k_b - 2k_a & s = ab
\end{cases}
\] (4.6)

In this case \( \Phi \geq 0 \), because \( d \leq D \). When \( k_a = k_b = k \), we have \( \Phi(a, k, k) = \Phi(b, k, k) = 2D + 2d - 3k \). When \( d = 0 \) and \( k_a = 0 \), (4.6) is the same as (4.4). Based on the one-to-one correspondence between an edge problem with parameters \((D, d)\) and \((D + d, 0)\), our algorithm \(A_d\) imitates algorithm \(A_L\) whenever possible.

Algorithm 4.4. \(A_d\)

- **Replica state assignments:**
  1. \(w_i = (0, D + d, D), s = a\).
  2. \(w_i = (dd0), s = ab\).
  3. \(w_i = (0, l + d, l), \text{where } 0 < l < D, s = a \text{ or } s = ab\).

- **State transitions:** as shown in Table 4.17.

<table>
<thead>
<tr>
<th>(w_{i-1})</th>
<th>(s_{i-1} \sim s_i)</th>
</tr>
</thead>
<tbody>
<tr>
<td>((0, D + d, D))</td>
<td>(b \sim a)</td>
</tr>
<tr>
<td>((dd0))</td>
<td>(a \sim ab)</td>
</tr>
<tr>
<td>((0, l + d, l), 0 &lt; l &lt; D)</td>
<td>(b \sim ab)</td>
</tr>
</tbody>
</table>

Table 4.17: State transitions for algorithm \(A_d\) when deletion cost \(d \neq 0\).

**Theorem 4.4.** Algorithm \(A_d\) is 3-competitive.

**Proof.** Using the potential function defined in (4.6), we show that Inequality (4.1) holds for all possible replica state transitions prescribed by algorithm \(A_d\) and all combinations of input requests, offsets, and replica states, as shown in Tables 4.18 and 4.19. \(\square\)
Figure 4.12: Offset transition diagrams with input requests replaced by $\Delta opt_i$ and $\Delta \Phi_i$ constraints, $D = 3$, $d = 2$
<table>
<thead>
<tr>
<th>$w_i$</th>
<th>$s_{i-1} \sim s_i$</th>
<th>$\Delta C_{e,i}$</th>
<th>$\Delta \Phi_i$</th>
<th>$\Delta C_{e,i} + \Delta \Phi_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(0, D + d, D)$</td>
<td>$b \sim a$</td>
<td>$D + d$</td>
<td>$-D - d$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$ab \sim a$</td>
<td>$d$</td>
<td>$-d$</td>
<td></td>
</tr>
<tr>
<td>$(dd0)$</td>
<td>$a \sim ab$</td>
<td>$D$</td>
<td>$-D$</td>
<td>$0$</td>
</tr>
<tr>
<td>$(0, l + d, l)$</td>
<td>$b \sim ab$</td>
<td>$D$</td>
<td>$-D$</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.18: Inequality (4.1) holds for all possible replica state transitions prescribed by algorithm $A_d$.

<table>
<thead>
<tr>
<th>$w_{i-1}$</th>
<th>$\sigma_i$</th>
<th>$s$</th>
<th>$\Delta C_{e,i}$</th>
<th>$\Delta \Phi_i$</th>
<th>$\Delta \text{opt}_i$</th>
<th>$w'_{i-1}$</th>
<th>$\sigma$</th>
<th>$s$</th>
<th>$\Delta C_{e,i}$</th>
<th>$\Delta \Phi_i$</th>
<th>$\Delta \text{opt}_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(0k_bl)$</td>
<td>$a^\lambda$</td>
<td>$a$</td>
<td>$0$</td>
<td>$0, -2$</td>
<td>$0$</td>
<td>$a^\lambda$</td>
<td>$a$</td>
<td>$0$</td>
<td>$0, -2$</td>
<td>$0$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$b$</td>
<td>$1$</td>
<td>$-1$</td>
<td></td>
<td></td>
<td>$b$</td>
<td>$1$</td>
<td>$-1$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$ab$</td>
<td>$0$</td>
<td>$0, -1$</td>
<td>$0$</td>
<td></td>
<td>$ab$</td>
<td>$0$</td>
<td>$-1$</td>
<td>$0$</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$a^\delta$</td>
<td>$a$</td>
<td>$0$</td>
<td>$0, -2$</td>
<td>$0$</td>
<td>$a^\delta$</td>
<td>$a$</td>
<td>$0$</td>
<td>$0, -2$</td>
<td>$0$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$b$</td>
<td>$1$</td>
<td>$-1$</td>
<td></td>
<td></td>
<td>$b$</td>
<td>$1$</td>
<td>$-1$</td>
<td>$0$</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$ab$</td>
<td>$1$</td>
<td>$-1$</td>
<td>$0$</td>
<td></td>
<td>$ab$</td>
<td>$1$</td>
<td>$-1$</td>
<td>$0$</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$b^\lambda$</td>
<td>$a$</td>
<td>$1$</td>
<td>$2$</td>
<td>$1$</td>
<td></td>
<td>$a$</td>
<td>$1$</td>
<td>$-1$</td>
<td>$0$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$b$</td>
<td>$0$</td>
<td>$1$</td>
<td>$1$</td>
<td></td>
<td>$b$</td>
<td>$0$</td>
<td>$-2$</td>
<td>$0$</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$ab$</td>
<td>$0$</td>
<td>$1$</td>
<td>$1$</td>
<td></td>
<td>$ab$</td>
<td>$0$</td>
<td>$-1$</td>
<td>$0$</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$b^\delta$</td>
<td>$a$</td>
<td>$1$</td>
<td>$2$</td>
<td>$1$</td>
<td></td>
<td>$a$</td>
<td>$1$</td>
<td>$-1$</td>
<td>$0$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$b$</td>
<td>$0$</td>
<td>$1$</td>
<td>$1$</td>
<td></td>
<td>$b$</td>
<td>$0$</td>
<td>$-2$</td>
<td>$0$</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$ab$</td>
<td>$1$</td>
<td>$1$</td>
<td>$1$</td>
<td></td>
<td>$ab$</td>
<td>$1$</td>
<td>$-1$</td>
<td>$0$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$(k_a k_b 0)$</td>
<td>$a^\lambda$</td>
<td>$a$</td>
<td>$0$</td>
<td>$0, -2$</td>
<td>$0$</td>
<td></td>
<td>$a^\lambda$</td>
<td>$a$</td>
<td>$0$</td>
<td>$0, -2$</td>
<td>$0$</td>
</tr>
<tr>
<td></td>
<td>$b$</td>
<td>$1$</td>
<td>$-1$</td>
<td></td>
<td></td>
<td>$b$</td>
<td>$1$</td>
<td>$-1$</td>
<td>$0$</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$ab$</td>
<td>$0$</td>
<td>$0, -1$</td>
<td>$0$</td>
<td></td>
<td>$ab$</td>
<td>$0$</td>
<td>$0, -1$</td>
<td>$0$</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$a^\delta$</td>
<td>$a$</td>
<td>$0$</td>
<td>$0, -2$</td>
<td>$0$</td>
<td></td>
<td>$a^\delta$</td>
<td>$a$</td>
<td>$0$</td>
<td>$0, -2$</td>
<td>$0$</td>
</tr>
<tr>
<td></td>
<td>$b$</td>
<td>$1$</td>
<td>$2$</td>
<td>$0$</td>
<td></td>
<td></td>
<td>$b$</td>
<td>$1$</td>
<td>$-1$</td>
<td>$0$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$ab$</td>
<td>$1$</td>
<td>$2$</td>
<td>$0$</td>
<td></td>
<td>$ab$</td>
<td>$1$</td>
<td>$-1$</td>
<td>$0$</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$b^\lambda$</td>
<td>$a$</td>
<td>$1$</td>
<td>$-1$</td>
<td>$0$</td>
<td></td>
<td>$a$</td>
<td>$1$</td>
<td>$-1$</td>
<td>$0$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$b$</td>
<td>$0$</td>
<td>$-2$</td>
<td>$0$</td>
<td></td>
<td>$b$</td>
<td>$0$</td>
<td>$-2$</td>
<td>$0$</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$ab$</td>
<td>$0$</td>
<td>$0, -2$</td>
<td>$0$</td>
<td></td>
<td>$ab$</td>
<td>$0$</td>
<td>$-2$</td>
<td>$0$</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$b^\delta$</td>
<td>$a$</td>
<td>$1$</td>
<td>$2$</td>
<td>$0$</td>
<td></td>
<td>$a$</td>
<td>$1$</td>
<td>$2$</td>
<td>$0$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$b$</td>
<td>$0$</td>
<td>$1$</td>
<td>$1$</td>
<td></td>
<td>$b$</td>
<td>$0$</td>
<td>$1$</td>
<td>$1$</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$ab$</td>
<td>$1$</td>
<td>$1$</td>
<td>$1$</td>
<td></td>
<td>$ab$</td>
<td>$1$</td>
<td>$1$</td>
<td>$1$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.19: Inequality (4.1) holds for all combinations of input requests, offsets, and replica states in algorithm $A_d$. An offset can be in any of the four possible forms: (1) $(0k_bl)$, where $1 \leq k_b \leq l + d$ and $1 \leq l \leq D$. (2) $(00l)$, where $0 \leq l \leq D$. (3) $(k_a k_b 0)$, where $1 \leq k_a \leq k_b \leq d$. (4) $(0k_b 0)$, where $1 \leq k_b \leq d$. 
4.2.4 Discussion

Lower Bound

Black et al. [13] proved that the migration problem, where only a single replica is allowed, has a lower bound of 3. Our edge problem can be reduced to the migration problem, as shown in [57], hence 3-competitive is the best possible. Alternatively, if we allow only one replica, the offset transition diagram degenerates into a single line, as shown in Figure 4.13, because state \(ab\) is forbidden, and an offset transition is only determined by the origin of a request, regardless of its type. In other words, lookup or update requests from the same node cause the same offset transitions. A potential function similar to (4.4) can prove its 3-competitiveness. (4.4) is applicable to the single replica scenario because it does not depend on \(l\), the offset value of state \(ab\).

\[
\begin{array}{ccccccc}
0 & 3 & b & 0 & 2 & b & 0 \\
& a & & a & & a & \\
\end{array}
\]

Figure 4.13: The migration problem, \(D = 3\)

Steady State Probability Distributions

Suppose all requests are equally likely, i.e., \(p\{a^\lambda\} = p\{a^\delta\} = p\{b^\lambda\} = p\{b^\delta\} = 0.25\). We can calculate the steady state probability distribution for all possible offset vectors. Figure 4.14 shows the steady state probabilities for \(D = 2\).

Relation to the Optimal Off-line Algorithm

From Section 4.2.2, we know that an algorithm that replicates according to the optimal past is not necessarily competitive. The following example illustrates why such an algorithm fails.

Example 4.5. Suppose \(D = 2\), \(w_0 = (011)\), and the input sequence \(\sigma = \{(b^\delta b^\lambda a^\lambda a^\lambda)^+\}\). The optimal off-line algorithm maintains a single replica at node \(a\). If we try to follow
Figure 4.14: Offset transition diagram with input requests replaced by transition probabilities. All requests are equally likely, i.e., \( p\{a^\lambda\} = p\{a^\delta\} = p\{b^\lambda\} = p\{b^\delta\} = 0.25 \). Steady state probabilities are calculated for \( D = 2 \).
the optimal past, we would end up tossing the replica back and forth between nodes a and b. Specifically, suppose initially the replica is at node a, after processing $b^\phi b^\phi$, since the ZOS has moved from node a to node b, or equivalently, since node b corresponds to the optimal past, the replica moves from node a to node b accordingly. Similarly, after processing the next two input $a^\lambda a^\lambda$, the replica moves back to node a, as shown in Figure 4.15. In contrast, an optimal off-line algorithm avoids thrashing a replica because it knows the future. Without such knowledge, the optimal for the past does not necessarily correspond to the optimal for the future.

Figure 4.15: Example 4.5. Transitions relevant to the input sequence $\sigma = \{(b^\phi b^\phi a^\lambda a^\lambda)^+\}$ are highlighted.

Relation to the Static Off-line Algorithms

Recall two properties of the off-line replication algorithms described in Chapter 3. First, they are static, i.e., once the replicas are distributed, no changes are made to them. Second, the optimal replica placement is based on the number of lookup and update requests in an input sequence. Having the same number of lookup and update requests, all permutations of an input sequence share the same optimal replica
placement. In contrast, an off-line algorithm in the context of competitive analysis is dynamic, allowing changes in replica placement during execution. The difference between static and dynamic algorithms is illustrated by the following example.

**Example 4.6.** Suppose the cost of replication $D$ is 1 and initially there is a replica at node $a$. Given an input sequence $\sigma = \{b^{\gamma} a^{\delta}\}$, the static off-line algorithm replicates at node $b$ and incurs a total cost of 2. The dynamic off-line algorithm also replicates at node $b$, but such replica is dropped before input request $a^{\delta}$, incurring a total cost of 1. The dynamic off-line algorithm therefore performs better than the static off-line algorithm.

**Step-by-step vs. One-shot Processing**

In all previously discussed edge algorithms, an edge must process each input request, in a step-by-step manner. To alleviate some of the computational burden, one may think of processing a chunk of input requests together, in a one-shot manner. Is this improvement valid? Let’s look at two examples:

**Example 4.7.** Let the replication cost $D$ be 2, the initial offset $u_0 = (202)$ with a replica at node $b$, and the input sequence $\sigma = \{a^{\delta}, a^{\lambda}, b^{\lambda}, b^{\delta}\}$, as shown in Figure 4.16. The cost of an optimal off-line algorithm is 3. In step-by-step processing, the final edge offset is $(101)$ and the replica state is $ab$, with a total cost of 6. If we skip the intermediate replica state transition, i.e., one-shot processing, based on the final offset $(101)$, an edge algorithm then decides to remain in state $b$, incurring a total cost of 3. In both cases, the costs are within a factor of 3 of the optimal, although their final replica states are different.

**Example 4.8.** Same as Example 4.7 except that the input sequence $\sigma = \{a^{\delta}, a^{\lambda}, b^{\lambda} (a^{\lambda})^n, b^{\delta}\}$, where $n > 8$. The cost of an optimal off-line algorithm remains unchanged, so does the total cost of the step-by-step processing. The cost of one-shot processing, however, increases above the bound, $3 \cdot C_{OPT}$, i.e., the one-shot processing no longer preserves the 3-competitiveness of an edge algorithm.

From these examples, it is evident that processing input requests in chunks does not guarantee the same edge algorithm performance. This result is not unexpected
because edge algorithms were proven to be 3-competitive by satisfying Inequality (4.1) for each request. In particular, one-shot processing fails in Example 4.8 because it misses a replication opportunity. Although multiple replica states may be possible for an offset vector, the route — a chain of previous replica states — via which an edge arrives at the current offset determines the current replica state. Without replica state changes, step-by-step processing is equivalent to one-shot processing.

Figure 4.16: Examples 4.7 and 4.8, step-by-step vs one-shot processing.

4.3 Tree Algorithms

4.3.1 Problem Description

A tree problem deals with replications on a tree network, i.e., given the past input sequence, a tree algorithm resolves where to put replicas, or equivalently the replica set of the tree. Knowing how to replicate for a single edge, we can solve the tree problem by transforming it into individual edge problems. The technique for such transformation is called factoring. Factoring guarantees that the cost of the tree
problem is the sum of the costs for the individual edge problems. In order for an edge algorithm to function, both input requests and replica states must be in terms of the two end nodes of an edge. Hence we need to factor a request from any node of the tree to a request from either end node of an edge. Similarly, the replica set for the tree needs to be factored to replica states for all edges.

Figure 4.17 summarizes how a tree algorithm works: we first factor both the input sequence and the replica set \( R \) to each edge in the tree. Each edge then runs some edge algorithm and produces some replica state. Coalescing these replica states, we get a replica set \( R \) for the tree.

\[
\sigma(T) \xrightarrow{\text{Factoring}} \sigma(e_1), R(e_1) \xrightarrow{\text{Edge Algo}} s(e_1) \\
\sigma(e_2), R(e_2) \xrightarrow{\text{Edge Algo}} s(e_2) \\
\vdots \\
\sigma(e_{n-1}), R(e_{n-1}) \xrightarrow{\text{Edge Algo}} s(e_{n-1})
\]

Figure 4.17: Tree algorithm.

### 4.3.2 Factoring

Given edge \((a, b)\) in a tree, removing edge \((a, b)\), we get two subtrees \( T_a(a, b) \) and \( T_b(a, b) \), rooted at nodes \( a \) and \( b \), respectively, as shown in Figure 4.18 (a). A request originating from subtree \( T_a(a, b) \) \((T_b(a, b))\) is factored to a request from node \( a \) \((b)\) in the induced edge problem. If the replica set \( R \) falls entirely in subtree \( T_a(a, b) \) \((T_b(a, b))\), then the replica state of edge \((a, b)\) is \( a \) \((b)\) in the induced edge problem. If \( R \) spans both subtrees \( T_a(a, b) \) and \( T_b(a, b) \), then the replica state of edge \((a, b)\) is \( ab \) in the induced edge problem.

Instead of sending a request to each edge separately, we choose to propagate the request along the tree, as shown in Figure 4.18 (b). Because of the tree structure,
4.3. **TREE ALGORITHMS**

the path between any node in subtree $T_a(a, b)$ and node $b$ must go through node $a$. In other words, if a request reaches node $a$ before node $b$ for edge $(a, b)$, it must have come from subtree $T_a(a, b)$, therefore should be factored to a request from node $a$ in the induced problem for edge $(a, b)$. Hence edge $(a, b)$ can correctly factor a request by simply observing which end node the request propagation reaches first.

![Diagram](image)

Figure 4.18: Factoring and request propagation. In (a), a request from subtree $T_a(a, b)$ ($T_b(a, b)$) is factored to a request from node $a$ ($b$) in the induced edge problem. If the replica set $R$ falls entirely in subtree $T_a(a, b)$ ($T_b(a, b)$), then the replica state of edge $(a, b)$ is $a$ ($b$) in the induced edge problem. If $R$ spans both subtrees $T_a(a, b)$ and $T_b(a, b)$, then the replica state of edge $(a, b)$ is $ab$ in the induced edge problem. In (b), a request from node $v$ reaches node $a$ before node $b$, therefore it is factored to a request from node $a$ for edge $(a, b)$.

### 4.3.3 Offset Distribution

The factoring technique produces a strong correlation among the offsets from the induced edge problems, allowing consistent coalescing of replication decisions, as shown in Section 4.3.4. In this section, we focus on the properties of offset distribution along adjacent edges. The following lemma was first proved by Lund et al.

**Lemma 4.1.** (Lund [57])

A. There exists a root $\rho_z$ such that $R = \{\rho_z\}$ is a ZOS in all induced edge problems. $\rho_z$ is called the zero offset root (ZOR).
B. Let \((a, b)\) and \((b, c)\) be adjacent edges with a ZOR \(\rho_z \in T_b\), where \(T_b\) is a subtree formed by removing both \((a, b)\) and \((b, c)\) from \(T\). Let the offsets for edges \((a, b)\) and \((b, c)\) be \((k_{ab}, l_{ab})\) and \((0, k_{bc}, l_{bc})\), respectively. Then \(l_{ab} \geq (l_{bc} - k_{bc})\) holds.

C. Let \((a, b)\) and \((b, c)\) be adjacent edges with a ZOR \(\rho_z \in T_a(a, b)\), where \(T_a(a, b)\) is a subtree formed by removing edge \((a, b)\) from \(T\). Let the offsets for edges \((a, b)\) and \((b, c)\) be \((0, k_{ab}, l_{ab})\) and \((0, k_{bc}, l_{bc})\), respectively. Then

1. \(l_{ab} \leq l_{bc}\)
2. \(k_{ab} \leq k_{bc}\)
3. \(l_{ab} - k_{ab} \geq l_{bc} - k_{bc}\)
4. if \(k_{bc} = 0\), then \(k_{ab} = 0\) and \(l_{ab} = l_{bc}\) hold.

From Lemma 4.1, a ZOR \(\rho_z\) has the following properties,

- \(\rho_z\) defines an order “\(\leftarrow\)” between the two nodes of an edge: \(a \leftarrow b\) iff \(a\) is nearer to \(\rho_z\) than \(b\). We call \(a\) the near end (NE) and \(b\) the far end (FE). Given any node \(v\), if \(v \neq \rho_z\), then among all the edges connected to \(v\), there exists a unique edge \((u, v)\) such that \(u \leftarrow v\). This order serves as a next hop routing table from any node \(v\) to \(\rho_z\). We shall use this self-addressing property of \(\rho_z\) to reduce the overhead for request propagation in Section 4.3.5.

- For edge \((a, b)\), if \(a \leftarrow b\), then \(a\) is a ZOS. Conversely, if \(a\) is a ZOS, the order between \(a\) and \(b\) is not uniquely defined because both nodes can be ZOS’s. The location of \(\rho_z\) is therefore not unique. But once we have chosen a valid \(\rho_z\), the order “\(\leftarrow\)” is uniquely defined between the two nodes of any given edge in a tree.

- A request can change the ZOS(s) of an edge, and the location of ZOR must alter accordingly. More precisely, the current ZOR \(\rho_z\) moves toward the requesting node along path \(P_{\rho_z \rightarrow v}\) between the previous ZOR \(\rho_z\) and the requesting node \(v\). Details can be found in Algorithm 4.5 in Section 4.3.5. For an edge that is not on the path \(P_{\rho_z \rightarrow v}\), it perceives the current request as from its NE, and the
previous ZOS remains a ZOS, as seen from Table 4.3 or Figure 4.3. In other words, such edge is oblivious to any changes in the ZOR.

- \( \rho_z \) may or may not be in the replica set \( R \) depending on the edge algorithm and the value of \( D \).

Lemma 4.1 dictates that edges with offsets of the form \((00l)\) induce a connected subtree \( T_0 \) containing \( \rho_z \). Moving away from \( \rho_z \), \( k \) and \( l \) are non-decreasing and \((l-k)\) is non-increasing; moving towards \( \rho_z \), \( k \) and \( l \) are non-increasing and \((l-k)\) is non-decreasing. In the offset transition diagram, as shown in Figure 4.19, directions \( d_1 \) and \( d_2 \) move away from \((00l)\)'s, while the opposite directions \( \tilde{d}_1 \) and \( \tilde{d}_2 \) move towards \((00l)\)'s.

Ultimately, Lemma 4.1 limits the valid offset values along adjacent edges, i.e., given the offsets of one edge, the offsets of its adjacent edges are restricted. The valid offsets are regularly located in the offset transition diagram. Consider adjacent edges \((a,b)\) and \((b,c)\), suppose \( a \leftarrow b \), or equivalently, \( \rho_z \in T_a(a,b) \). The valid offsets for edge \((b,c)\) are those reachable from the offset of edge \((a,b)\) along directions \( d_1 \) and \( d_2 \). For example, for \( D = 3 \), if edge \((a,b)\) has an offset \((012)\), all edges connected to \( b \) must have one of the following offsets: \((012)\), \((022)\), \((023)\), or \((033)\).

The offset distribution is also related to \( \Delta opt_i \) and the potential function: along directions \( d_1 \) and \( d_2 \), \( \Delta opt_i = 0 \) and the potential decreases; along directions \( \tilde{d}_1 \) and \( \tilde{d}_2 \), \( \Delta opt_i = 1 \) and the potential increases. Incidentally, directions \( d_1 \) and \( d_2 \) correspond to offset changes upon a lookup or update request from a ZOS.

### 4.3.4 Sufficient Conditions

In Chapter 3, we have shown that with multicast replica update, there exists an optimal replica set that induces a connected subtree. If we begin with a connected replica set, can an edge algorithm maintain such connectivity? Toward the end of this section, we shall prove that all previously discussed edge algorithms \((A_L, A_T, A_H, \text{ and } A_d)\) maintain a connected replica set. For now, let's look at how the replica states of adjacent edges relate to one another.
Figure 4.19: Offset distribution along directions \( d_1 \) and \( d_2 \), where \( \uparrow \) means non-decreasing and \( \downarrow \) means non-increasing. Given \( D = 3 \) and \( w(a, b) = (012) \), the valid offsets for its adjacent edges are \((012), (022), (023), \) and \((033)\).

In an induced edge problem, if the replica state of the edge is \( a \) (or \( b \)), node \( a \) (or \( b \)) may not have a physical replica, because factoring only requires that the replica set \( R \) falls entirely in subtree \( T_a(a, b) \) (or \( T_b(a, b) \)). On the other hand, if the edge replica state is \( ab \), both nodes \( a \) and \( b \) must have physical replicas as a result of the connectivity in the replica set \( R \).

How does this affect replica states along adjacent edges? Consider adjacent edges \((a, b)\) and \((b, c)\). Suppose only node \( c \) has a replica initially. Upon factoring, edge \((b, c)\) is in state \( c \), and edge \((a, b)\) is in state \( b \) because the replica falls in subtree \( T_b(a, b) \), even though node \( b \) does not have a physical replica. After processing some input request, suppose edge \((a, b)\) decides to change from state \( b \) to state \( ab \), so it simply adds a replica at node \( a \), believing that it already has a replica somewhere in subtree \( T_b(a, b) \). Now if this request does not cause any replica state change for edge \((b, c)\), we would face a conflict: edge \((b, c)\) thinks there is no replica in subtree \( T_b(b, c) \) that includes edge \((a, b)\), while edge \((a, b)\) has just added a replica at node \( a \). In other
4.3. TREE ALGORITHMS

words, although locally, edge \((b, c)\) is in state \(c\), globally both nodes \(a\) and \(c\) have replicas, which factors to state \(bc\) for edge \((b, c)\). The local and global views for edge \((b, c)\) do not match, resulting in a disconnected replica set \(R\). A similar situation may arise upon replica deletions along adjacent edges \((a, b)\) and \((b, c)\). Suppose initially all three nodes have replicas, edge \((a, b)\) is in state \(ab\) and edge \((b, c)\) is in state \(bc\). After processing some input requests, suppose edge \((a, b)\) drops its replica at node \(b\). Now if this request does not cause any replica state change for edge \((b, c)\), we would again face a conflict: edge \((a, b)\) thinks there is no replica in subtree \(T_b(a, b)\) that includes edge \((b, c)\), and edge \((b, c)\) still has a replica at node \(c\). Again, the local and global views for edge \((a, b)\) do not match, and the replica set \(R\) is no longer connected.

What causes these conflicts? In solving the tree problem, we first factor both the replica set \(R_{i-1}\) and the input request \(\sigma_i\) to each edge \((a, b)\) in the tree. The tree algorithm operates in a distributed manner. Locally, each edge \((a, b)\) processes the factored input request \(\sigma_i(a, b)\) and determines the appropriate edge replica state \(s_i(a, b)\). Globally, the replica states of all edges constitute the replica set \(R_i\) of the tree. Behind these seemingly independent operations, the edges in the tree are physically connected, in other words, the subtrees defined by adjacent edges overlap. Therefore, adding a replica at node \(a\) tacitly results in two-replica states for all edges along the path between node \(a\) and the previous replica set \(R_{i-1}\). Similarly, deleting a replica at node \(b\) also implies all replicas in the subtree \(T_b(a, b)\) should be dropped. If \(\forall\) edge \((a, b) \in E\), the replica state maintained locally by edge \((a, b)\) conforms with the replica state from factoring \(R_i\) to edge \((a, b)\), then the local edge view would be consistent with the global tree view.

However, not all edge algorithms satisfy this condition readily. We can easily construct one that does not. In Section 4.2.3, apart from the minimum requirements on replica state assignments, both states \(a\) and \(ab\) are possible for offset vectors of the form \((0kl)\), where \(0 < k \leq l < D\). Furthermore, the potential function defined in (4.4) always maintains a distance \(D\) between state \(b\) and state \(ab\), which permits state transitions from state \(b\) to state \(ab\) at any time. We shall construct algorithm \(A_q\) by making the following modifications to algorithm \(A_{I'}\): instead of replicating at node \(a\) for offsets of the form \((0kl)\) with \(k > 0\), we replicate at node \(a\) only if
$w = (013)$. If $D < 4$, then algorithm $A_Q$ is the same as algorithm $A_T$. Algorithm $A_Q$ is 3-competitive because it satisfies the minimum requirements on replica state assignments and complies with the same potential function defined in (4.4). The sole difference between these two algorithms is that offset $(013)$ is the only replication candidate in algorithm $A_Q$, as compared with all offsets of the form $(0kl)$ with $k > 0$ in algorithm $A_T$.

How does a tree algorithm based on algorithm $A_Q$ behave? Consider two adjacent edges $(a, b)$ and $(b, c)$ in a tree with the cost of replication $D \geq 4$. Suppose $w_{i-1}(a, b) = w_{i-1}(b, c) = (024)$, and the replica set $R_{i-1}$ falls entirely within the subtree $T_c(b, c)$, i.e., edge $(a, b)$ is in state $b$ and edge $(b, c)$ is in state $c$. Such initial condition is valid because there is a path from $(D0D)$ to $(024)$ without passing through $(013)$ in the offset transition diagram. An example request sequence is $\{(a^\lambda)^l(a^\delta)^n(a^\lambda)^m\}$, where $l$, $n$, and $m$ are related to the replication cost $D$ and the initial offset vector. Upon processing a lookup request $\sigma_i$ from node $b$, which factors to $b^\Lambda$ for both edges $(a, b)$ and $(b, c)$, $w_i(a, b) = (013)$ and $w_i(b, c) = (034)$, as shown in Figure 4.20. According to algorithm $A_Q$, edge $(a, b)$ needs to replicate at node $a$. However edge $(b, c)$ does not agree with this replication, i.e., locally it does not replicate at node $b$, resulting in a disconnected replica set $R_i$. Therefore algorithm $A_Q$ does not lead to a valid tree algorithm.

We learn from this example that individual edges are not in conflict as long as they maintain a connected replica set $R$ and the local edge view is consistent with the global tree view. These constraints limit the possible edge replica states generated by an edge algorithm. If an edge algorithm satisfies these constraints, the tree algorithm based on the edge algorithm has the same competitive ratio as the edge algorithm [57]. Applying offset distribution along adjacent edges in Section 4.3.3, we list these constraints as sufficient conditions for the design of edge algorithms.

- $s_{i-1}(a, b) = b$, adding a replica at node $a$. To maintain the connectivity in the current replica set $R_i$, all edges along the path $P_{b \leftrightarrow R_{i-1}}$ between node $b$ and the previous replica set $R_{i-1}$, must also replicate. Depending on the relative positions among the ZOR $\rho_z$, the previous replica set $R_{i-1}$, and edge $(a, b)$, the following offsets are valid for an edge on path $P_{b \leftrightarrow R_{i-1}}$. 

Figure 4.20: Edge algorithm $A_q$ replicates for offsets in gray. Since it results in a disconnected replica set $R$ for $D \geq 4$, it does not lead to a valid tree algorithm.
— If \( \rho_z \in T_a(a, b) \), as shown in Figure 4.21 (a), all offsets along directions \( d_1 \) and \( d_2 \) in the left half of the transition diagram, as shown in Figure 4.21 (c).

— If \( \rho_z \in T_b(a, b) \),
  * If \( \rho_z \notin P_{b \leftarrow R_{i-1}} \), as shown in Figure 4.22 (a), all offsets along \( \vec{d}_1 \) and \( \vec{d}_2 \) directions in the right half of the transition diagram, as shown in Figure 4.22 (b).
  * If \( \rho_z \in P_{b \leftarrow R_{i-1}} \), as shown in Figure 4.23 (a), all offsets along
    · \( \vec{d}_1 \) and \( \vec{d}_2 \) directions in the right half of the transition diagram and
    · \( d_1 \) and \( d_2 \) directions in the left half of the transition diagram, as shown in Figure 4.23 (b).

An edge algorithm must replicate for these offsets.

bullet \( s_{i-1}(a, b) = ab \), deleting a replica at node \( b \). All replicas connecting to node \( b \) must also be deleted lest the connectivity in \( R_i \) be destroyed. Because the cost of deleting a replica is zero, \( \rho_z \) must be in \( T_a(a, b) \), as shown in Figure 4.21 (b). An edge algorithm should delete for all offsets along directions \( d_1 \) and \( d_2 \) in the left half of the transition diagram, as shown in Figure 4.21 (c).

By symmetry, we can argue the cases for adding a replica at node \( b \) and deleting a replica at node \( a \).

Example 4.9. In Figure 4.24(a), the state transitions for offsets (022), (023), and (033) should include those for offset (012), i.e., \( A_{012} \subseteq A_{022}, A_{023}, A_{033} \), since they are reachable from offset (012) along directions \( d_1 \) and \( d_2 \).

Example 4.10. In Figure 4.24(b), the state transitions for offset (022) should include those for all offsets that can reach (022) along directions \( d_1 \) and \( d_2 \), i.e., \( A_{022} \supseteq \cup\{A_{011}, A_{012}, A_{000}, A_{001}, A_{002}\} \).

All previously discussed algorithms, \( A_L \), \( A_H \), \( A_T \), and \( A_d \), satisfy these conditions. Therefore the combination of the edge solutions provides a 3-competitive solution in the original tree network.
4.3. **TREE ALGORITHMS**

(a) Adding a replica at node $a$.

(b) Deleting the replica at node $b$.

(c) Offset transition diagram, $D = 3$.

Figure 4.21: Sufficient conditions if $\rho_z \in T_a(a, b)$. 
Figure 4.22: Sufficient conditions if $\rho_z \in T_b(a, b)$ and $\rho_z \notin P_{b \leftrightarrow R_{i-1}}$. 
4.3. TREE ALGORITHMS

(a) Adding a replica at node \( a \).

(b) Offset transition diagram, \( D = 3 \).

Figure 4.23: Sufficient conditions if \( \rho_z \in T_b(a, b) \) and \( \rho_z \in P_{b \rightarrow R_{i-1}} \).
(a) $A_{012} \subseteq A_{022}, A_{023}, A_{033}$

(b) $A_{022} \supseteq \bigcup\{A_{011}, A_{012}, A_{000}, A_{001}, A_{002}\}$

Figure 4.24: Examples 4.9 and 4.10. $D = 3$. 
4.3. **TREE ALGORITHMS**

**Theorem 4.5.** The tree algorithm based on algorithm $A_L$ is 3-competitive.

**Proof.** We prove by induction on the number of requests. Initially, $|R| = 1$, i.e., there is a single replica. Let $s'$ be the global replica state factored from the replica set $R$ on an edge, and $s$ be the local replica state of the edge upon processing a factored input request. After processing requests $\sigma_1, \sigma_2, \ldots, \sigma_{i-1}$, suppose $R$ is connected and $s' = s$ holds for all edges. We show that all possible replica additions and deletions maintain the connectivity in $R$ and ensure $s' = s$ for all edges upon a request $\sigma_i$. Subscripts $i-1$ and $i$ differentiate between before and after processing request $\sigma_i$.

- **Replica additions**
  Suppose after processing request $\sigma_i$, algorithm $A_L$ replicates for edge $(a, b)$, and $w(a, b) = (0ll)$, where $0 \leq l < D$.

  - If $w(a, b) = (0ll)$ where $0 < l < D$ as shown in Figure 4.25 (a), $\rho_z \in T_a(a, b)$ and algorithm $A_L$ replicates at node $a$. From the inductive hypothesis, $s_{i-1} = b$ and $R_{i-1} \subseteq T_b(a, b)$. Upon processing request $\sigma_i$, $s_i = ab$. Let $P_{b \leftrightarrow R_{i-1}}$ be the unique path between node $b$ and $R_{i-1}$. From Lemma 4.1.C.3, edge $(u, v)$ along $P_{b \leftrightarrow R_{i-1}}$, must have an offset of the form $(0ll'')$, which in turn adds a replica to its NE node $u$. These newly added replicas are connected to one another and to the previous replica set $R_{i-1}$. Hence the current replica set $R_i$ remains connected, and the local replica states are consistent with the globally factored replica states for edge $(a, b)$ and all edges along the path between edge $(a, b)$ and $R_{i-1}$.

  - If $w(a, b) = (000)$, $s_{i-1} = a$ or $s_{i-1} = b$.

    1. If $s_{i-1} = b$, this case is the same as having $w(a, b) = (0ll)$ where $0 < l < D$.

    2. If $s_{i-1} = a$, algorithm $A_L$ replicates at node $b$.

      (a) If $\rho_z \in T_b(a, b)$, as shown in Figure 4.25 (b), this case is the same as having $w(a, b) = (0ll)$, where $0 < l < D$, with $a$ and $b$ swapped.

      (b) If $\rho_z \in T_a(a, b)$, as shown in Figure 4.25 (c). Let $P_{a \leftrightarrow R_{i-1}}$ be the unique path between node $a$ and $R_{i-1}$ and edge $(u, v) \in P_{a \leftrightarrow R_{i-1}}$. 

i. If edge \((u, v)\) is on the same side of \(\rho_z\) as edge \((a, b)\), from Lemma 4.1.C.4, \(w(u, v) = (000)\), which requires replication at its FE \(u\).

ii. If edge \((u, v)\) is on the opposite side of \(\rho_z\) as edge \((a, b)\), from Lemma 4.1.B, \(w(u, v) = (0l''l')\), which requires replication at its NE \(v\).

In both cases, the newly added replicas are connected to one another and to the previous replica set \(R_{i-1}\). Hence the current replica set \(R_i\) remains connected, and the local replica states are consistent with the globally factored replica states for edge \((a, b)\) and all edges along the path between edge \((a, b)\) and \(R_{i-1}\).

- **Replica deletions**

Suppose after processing request \(\sigma_i\), algorithm \(A_L\) deletes a replica at node \(b\) for edge \((a, b)\), and \(w(a, b) = (0DD)\). We know \(\rho_z \in T_a(a, b)\) and \(s_{i-1} = ab\). All the nodes in \(R_{i-1}\) that are connected to node \(b\) induce a connected subtree \(R_b\). From Lemma 4.1.C.2, edge \((u, v)\) in \(R_b\) must also have offset \((0DD)\), which in turn removes the replica at its FE node \(v\) for all edges in \(R_b\). Hence the current replica set \(R_i\) remains connected and the local replica states are consistent with the globally factored replica states for edge \((a, b)\) and all edges in \(R_b\).

Alternatively, we can prove that algorithm \(A_L\) satisfies the sufficient conditions. To add a replica at node \(a\), edge \((a, b)\) must have an offset of the form \((0ll)\), where \(0 \leq l < D\). Starting from such an offset, all offsets reachable along directions \(d_1\) and \(d_2\) in the offset transition digram are also of the form \((0ll)\). The replica deletion case can be argued similarly.

\(\square\)

**Theorem 4.6.** The tree algorithm based on algorithm \(A_T\) is 3-competitive.

**Proof.** We prove by induction on the number of requests. In addition to the replica state transitions prescribed by algorithm \(A_L\), algorithm \(A_T\) replicates at node \(a\) when \(w(a, b) = (0kl)\) with \(k > 0\). Suppose after processing request \(\sigma_i\), algorithm \(A_T\) replicates at node \(a\) for edge \((a, b)\) when \(w(a, b) = (0kl)\) with \(k > 0\). We know that
4.3. TREE ALGORITHMS

(a) $(0lt)$, where $0 < l < D$, $s_{i-1}(a, b) = b$, $s_i(a, b) = ab$.

(b) $(000)$, $\rho_z \in T_i(a, b)$, $s_{i-1}(a, b) = a$, $s_i(a, b) = ab$.

(c) $(000)$, $\rho_z \in T_i(a, b)$, $s_{i-1}(a, b) = a$, $s_i(a, b) = ab$.

Figure 4.25: Theorem 4.5.
\( \rho_z \in T_a(a, b) \). From inductive hypothesis, \( s_{i-1} = b \) and \( R_{i-1} \subseteq T_b(a, b) \). Let \( P_{b \leftrightarrow R_{i-1}} \) be the unique path between node \( b \) and \( R_{i-1} \). From Lemma 4.1.C.2, edge \((u, v)\) along \( P_{b \leftrightarrow R_{i-1}} \) must have an offset of the form \((0k'k')\) with \( k' > 0 \), which in turn adds a replica to its NE \( u \). These newly added replicas are connected to one another and to the previous replica set \( R_{i-1} \). Hence the current replica set \( R_i \) remains connected, and the local replica states are consistent with the globally factored replica states for edge \((a, b)\) and all edges along the path between edge \((a, b)\) and \( R_{i-1} \).

Alternatively, we can prove that algorithm \( A_T \) satisfies the sufficient conditions. To add a replica at node \( a \), edge \((a, b)\) must have an offset of the form \((0k\ell)\) with \( k > 0 \). Starting from such an offset, all offsets reachable along directions \( d_1 \) and \( d_2 \) in the offset transition diagram are also of the form \((0k\ell)\) with \( k > 0 \). The replica deletion case can be argued similarly. \( \square \)

**Theorem 4.7.** The tree algorithm based on algorithm \( A_H \) is 3-competitive.

**Proof.** We prove by induction on the number of requests. In addition to the replica state transition prescribed by algorithm \( A_L \), algorithm \( A_H \) deletes the replica at node \( b \) when \( w(a, b) = (0kD) \). Suppose after processing request \( \sigma_i \), algorithm \( A_H \) deletes the replica at node \( b \) for edge \((a, b)\) when \( w(a, b) = (0kD) \). We know that \( \rho_z \in T_a(a, b) \) and \( s_{i-1} = ab \). All the nodes in \( R_{i-1} \) that are connected to node \( b \) induce a connected subtree \( R_b \). From Lemma 4.1.C.1, edge \((u, v)\) in \( R_b \) must have an offset of the form \((0kD)\), which in turn removes the replica at its FE node \( v \) for all edges in \( R_b \). Hence the current replica set \( R_i \) remains connected and the local replica states are consistent with the globally factored replica states for edge \((a, b)\) and all edges in \( R_b \).

Alternatively, we can prove that algorithm \( A_H \) satisfies the sufficient conditions. To delete the replica at node \( b \), edge \((a, b)\) must have an offset of the form \((0kD)\). Starting from such an offset, all offsets reachable along directions \( d_1 \) and \( d_2 \) in the offset transition diagram are also of the form \((0kD)\). The replica addition case can be argued similarly. \( \square \)

**Theorem 4.8.** The tree algorithm based on algorithm \( A_d \) is 3-competitive.

**Proof.** Algorithm \( A_d \) behaves the same as algorithm \( A_L \) because of the one-to-one
correspondence between the offset transition diagrams of \((D, d)\) and \((D + d, 0)\). Algorithm \(A_d\) is 3-competitive as a result of Theorem 4.5.

4.3.5 Propagation Cost Reduction

In Section 4.3.2, factoring is implemented by propagating requests along the tree. Since request propagation is not free, its cost has to be added to both edge and tree algorithms. We propose and analyze two methods, smart and delayed propagation, both aiming at reducing the overhead of request propagation. The idea of smart propagation is to continue propagation only if there is an offset change, and the idea of delayed propagation is to propagate after accumulating a number of requests.

**Smart Propagation**

Up to this point, our edge model has not included the cost of request propagation. To add such cost, we introduce a distributed edge model, where both end nodes calculate the offset vector. The two copies of the offset value must be kept consistent, thus requiring explicit request propagation between the two end nodes. For edge \((a, b)\), after processing a lookup request \(\sigma^\lambda\) from node \(a\), \(\sigma^\lambda\) must propagate from \(a\) to \(b\) if the offset has changed and node \(a\) has a replica. If node \(a\) does not have a replica, no explicit request propagation is needed because \(\sigma^\lambda\) has to be resolved at node \(b\), which in turn can update its copy of the offset vector. Similarly, after processing an update request \(\sigma^\delta\) from node \(a\), \(\sigma^\delta\) must propagate from \(a\) to \(b\) if the offset has changed and node \(b\) does not have a replica, as shown in Figure 4.26. By symmetry, we can define when it is necessary to propagate a request from node \(b\).

Two questions remain: Are edge algorithms still competitive with the cost of request propagation added? Are the tree algorithms based on these edge algorithms also competitive? It turns out that the gap between the cost of an edge algorithm and the 3-competitive bound is large enough to accommodate the cost of request propagation, as stated in the following theorem.

**Theorem 4.9.** Algorithms \(A_L\), \(A_H\), \(A_T\), and \(A_d\) are 3-competitive with the cost of request propagation added.
Figure 4.26: Explicit request propagation in the distributed edge model. The filled circles correspond to replicas and the arrows mark explicit request propagation, assuming the offset of edge \((a, b)\) changes upon the current request.

Proof. Let \(C_p\) be the cost of request propagation. We prove that Inequality (4.1) holds with \(C_p\) added to its left hand side.

- Algorithm \(A_L\), as shown in Table 4.20.
- Algorithm \(A_T\), the table is identical to Table 4.20 except for the forbidden state \(b\) when \(w_i = (0kl)\), with \(k > 0\).
- Algorithm \(A_H\), as shown in Table 4.21, which is similar to Table 4.20 except for the underlined entries.
- Algorithm \(A_d\)
  1. \((0k_bl)\), where \(1 \leq k_b \leq l + d\) and \(1 \leq l \leq D\). The table is the same as Table 4.20.
  2. \((00l)\), where \(0 \leq l \leq D\), as shown in Table 4.22.
  3. \((k_alb0)\), where \(1 \leq k_a \leq k_b \leq d\), as shown in Table 4.23.
  4. \((0kal0)\), where \(1 \leq k_b \leq d\), as shown in Table 4.24.

As all our edge algorithms remain 3-competitive with the cost of request propagation added, can we automatically assume that tree algorithms based on these edge algorithms are also 3-competitive? Consider adjacent edges \((a, b)\) and \((b, c)\), suppose
### 4.3. TREE ALGORITHMS

Table 4.20: Theorem 4.9: Algorithm $A_k$ is 3-competitive with the cost of request propagation added.
<table>
<thead>
<tr>
<th>( w_{i-1} )</th>
<th>( \sigma_i )</th>
<th>( s )</th>
<th>( \Delta C_{e,i} )</th>
<th>( \Delta \Phi_i )</th>
<th>( C_p )</th>
<th>( \Delta \text{opt}_i )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a^\lambda )</td>
<td>( a )</td>
<td>0</td>
<td>( k = l )</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>( k &lt; l )</td>
<td>-2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( b )</td>
<td>1</td>
<td>-1</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( ab )</td>
<td>0</td>
<td>( k = l )</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>( k &lt; l )</td>
<td>-2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>( b^\lambda )</td>
<td>( a )</td>
<td>0</td>
<td>( k = D )</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>( k &lt; D )</td>
<td>-2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( b )</td>
<td>1</td>
<td>-1</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( ab )</td>
<td>1</td>
<td>-1</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>( b^\delta )</td>
<td>( a )</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( b )</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( ab )</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>( a^\delta )</td>
<td>( a )</td>
<td>0</td>
<td>( l = 0 )</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>( l &gt; 0 )</td>
<td>-2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( b )</td>
<td>1</td>
<td>-1</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( ab )</td>
<td>0</td>
<td>( l = 0 )</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>( l &gt; 0 )</td>
<td>-2</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.21: Theorem 4.9: Algorithm \( A_T \) is 3-competitive with the cost of request propagation added.
## 4.3. TREE ALGORITHMS

<table>
<thead>
<tr>
<th>$w_{i-1}$</th>
<th>$\sigma_i$</th>
<th>$s$</th>
<th>$\Delta C_{e,i}$</th>
<th>$\Delta \Phi_i$</th>
<th>$C_p$</th>
<th>$\Delta \text{opt}_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a^\lambda$</td>
<td>$a$</td>
<td>0</td>
<td>$-2$</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>$b$</td>
<td>1</td>
<td>$-1$</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>$ab$</td>
<td>0</td>
<td>$-1$</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>(00l)</td>
<td>$a^\delta$</td>
<td>$a$</td>
<td>0</td>
<td>$-2$</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>$b$</td>
<td>1</td>
<td>$-1$</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>$ab$</td>
<td>1</td>
<td>$-1$</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$b^\lambda$</td>
<td>$a$</td>
<td>1</td>
<td>$-1$</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>$b$</td>
<td>0</td>
<td>$-2$</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>$ab$</td>
<td>0</td>
<td>$-1$</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$b^\delta$</td>
<td>$a$</td>
<td>1</td>
<td>$-1$</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>$b$</td>
<td>0</td>
<td>$-2$</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>$ab$</td>
<td>1</td>
<td>$-1$</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 4.22: Theorem 4.9: Algorithm $A_l$ is 3-competitive with the cost of request propagation added. (00l), where $0 \leq l \leq D$.

<table>
<thead>
<tr>
<th>$w_{i-1}$</th>
<th>$\sigma_i$</th>
<th>$s$</th>
<th>$\Delta C_{e,i}$</th>
<th>$\Delta \Phi_i$</th>
<th>$C_p$</th>
<th>$\Delta \text{opt}_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a^\lambda$</td>
<td>$a$</td>
<td>0</td>
<td>$k_b = d$</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>$b$</td>
<td>1</td>
<td>$k_b &lt; d$</td>
<td>$-2$</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>$ab$</td>
<td>0</td>
<td>$k_b = d$</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$k_b &lt; d$</td>
<td>$-1$</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>(k_a_k_b_0)</td>
<td>$a^\delta$</td>
<td>$a$</td>
<td>0</td>
<td>$1$</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>$b$</td>
<td>1</td>
<td>$2$</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>$ab$</td>
<td>1</td>
<td>$2$</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>$b^\lambda$</td>
<td>$a$</td>
<td>1</td>
<td>$-1$</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>$b$</td>
<td>0</td>
<td>$-2$</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>$ab$</td>
<td>0</td>
<td>$k_a = d$</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$k_a &lt; d$</td>
<td>$-2$</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>$b^\delta$</td>
<td>$a$</td>
<td>1</td>
<td>$2$</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>$b$</td>
<td>0</td>
<td>$1$</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>$ab$</td>
<td>1</td>
<td>$1$</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 4.23: Theorem 4.9: Algorithm $A_l$ is 3-competitive with the cost of request propagation added. (k\_a\_k_b\_0), where $1 \leq k_a \leq k_b \leq d$. 
Table 4.24: Theorem 4.9: Algorithm $A_d$ is 3-competitive with the cost of request propagation added. $(0k_b0)$, where $1 \leq k_b \leq d$.

A request originating from node $a$ does not change the offset of edge $(a, b)$. An edge algorithm does not propagate this request along edge $(a, b)$ according to the distributed edge model. However, factoring requires propagating this request to edge $(b, c)$. We have a conflict! Fortunately, Lemma 4.1 limits the valid offset values along adjacent edges, and we shall prove that if a request from node $a$ does not change the offset of edge $(a, b)$, then it does not change the offset of edge $(b, c)$ either. Request propagation can therefore be *smart*, i.e., propagation should continue until it reaches an edge with no offset change upon this request. All the edges after this edge are not required to process this request since their offsets would not be affected by it.

**Theorem 4.10.** If the offset of edge $(a, b)$ does not change upon a request $\sigma_i$ from node $a$, then the offsets of all edges in subtree $T_b(a, b)$ remain unchanged upon the same request.

*Proof.* Let the offset of edge $(a, b)$ be $(0, k_{ab}, l_{ab})$. On the offset transition diagram, the self-loops correspond to requests that do not change the offsets. These offsets can come in one of the following three forms: $(0DD), (0ll)$ with $0 < l < D$, and $(000)$. 

<table>
<thead>
<tr>
<th>$w_{i-1}$</th>
<th>$\sigma_i$</th>
<th>$s$</th>
<th>$\Delta C_{e,i}$</th>
<th>$\Delta \Phi_i$</th>
<th>$C_p$</th>
<th>$\Delta opt_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a^\lambda$</td>
<td>$a$</td>
<td>0</td>
<td>$k_b = d$</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$b$</td>
<td>1</td>
<td>$k_b &lt; d$</td>
<td>$-2$</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>$ab$</td>
<td>0</td>
<td>$k_b = d$</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$k_b &lt; d$</td>
<td>$-1$</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>$(0k_b0)$</td>
<td>$a^\delta$</td>
<td>$a$</td>
<td>0</td>
<td>$-2$</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$b$</td>
<td>1</td>
<td>$-1$</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$ab$</td>
<td>1</td>
<td>$-1$</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>$b^\lambda$</td>
<td>$a$</td>
<td>1</td>
<td>$-1$</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$b$</td>
<td>0</td>
<td>$-2$</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$ab$</td>
<td>0</td>
<td>$-2$</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>$b^\delta$</td>
<td>$a$</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$b$</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$ab$</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
\( (0, k_{ab}, l_{ab}) = (0DD) \) is invariant in serving requests \( a^\lambda \) and \( a^\delta \).

Let \((u, v)\) be an edge in subtree \( T_b(a, b) \). From Lemma 4.1.A, \( \rho_z \in T_a(a, b) \).
Requests \( a^\lambda \) and \( a^\delta \) factor to NE requests \( u^\lambda \) and \( u^\delta \) for edge \((u, v)\). Let the offset of edge \((u, v)\) be \((0, k_{uv}, l_{uv})\).
From Lemma 4.1.C.2, \( k_{ab} \leq k_{uv} \), since \( k_{ab} = D \), \( k_{uv} \) must also be \( D \). Furthermore, \( k_{uv} \leq l_{uv} \leq D \), both \( k_{uv} \) and \( l_{uv} \) must be \( D \), i.e., the offset of edge \((u, v)\) is \((0DD)\), which is invariant in NE requests \( u^\lambda \) and \( u^\delta \).

\( (0, k_{ab}, l_{ab}) = (0ll) \), where \( 0 < l < D \), is invariant in request \( a^\lambda \).

Let \((u, v)\) be an edge in subtree \( T_b(a, b) \). From Lemma 4.1.A, \( \rho_z \in T_a(a, b) \).
Request \( a^\lambda \) factors to a NE request \( u^\lambda \) for edge \((u, v)\). Let the offset of edge \((u, v)\) be \((0, k_{uv}, l_{uv})\).
From Lemma 4.1.C.3, \( l_{ab} - k_{ab} \geq l_{uv} - k_{uv} \), since \( l_{ab} = k_{ab} \), we have \( l_{uv} = k_{uv} \), i.e., the offset of edge \((u, v)\) is \((0, l_{uv}, l_{uv})\), which is invariant in NE request \( u^\lambda \).

\( (0, k_{ab}, l_{ab}) = (000) \) is invariant in requests \( a^\lambda \) and \( b^\lambda \).

If \( \rho_z \in T_a(a, b) \), the offset of an edge in subtree \( T_b(a, b) \) is invariant in request \( a^\lambda \), as in the case of \((0ll)\), where \( 0 < l < D \).
Let \( P_{a \leftrightarrow \rho_z} \) be the path between node \( a \) and \( \rho_z \).

- For all edges along path \( P_{a \leftrightarrow \rho_z} \), request \( b^\lambda \) factors to a FE request. From Lemma 4.1.C.4, these edges must have offset \((000)\), which is invariant in a FE request.

- For all edges off path \( P_{a \leftrightarrow \rho_z} \), \( b^\lambda \) factors to a NE request. From Lemma 4.1.B and Lemma 4.1.C.3, these edges must have offsets of the form \((0ll)\), which is invariant in a NE request.

Therefore the offsets of all edges in \( T_a(a, b) \) are invariant in request \( b^\lambda \). By symmetry, we can argue the case for \( \rho_z \in T_b(a, b) \).

We have proved that request propagation can be \textit{smart}, i.e., propagation only needs to go as far as there are changes in offsets. \( \square \)
In normal propagation, a request reaches all edges even though it may cause no offset change for some edges, as shown in Figure 4.27(a). In smart propagation, however, request propagation stops when it reaches an edge with no offset change upon this request, as shown in Figure 4.27(b). Smart propagation therefore reduces the overhead of request propagation.

![Diagram of normal and smart propagation]

Figure 4.27: Normal propagation vs. smart propagation. In (b), request propagation does not pass edge \((a, b)\) because its offset does not change upon the current request.

**Delayed Propagation**

Previously, when a request arrives, it propagates to all edges in the tree. We relax this condition and suggest the following idea: instead of propagating as a request arrives, we can propagate after accumulating a number of requests to reduce the cost of propagation. Recall that an update request propagates to all nodes with replicas, and these nodes can update their offsets without additional overhead. On the other hand, a lookup request is served by its closest replica, incurring additional overhead to notify other edges. Because request propagation can piggyback on request service, the propagation of a lookup request is more costly than that of an update request. Instead, we choose to accumulate lookup requests. Here is a simple scheme: we can record the time and origin of lookup requests, store delayed lookup requests in a central location, and periodically propagate the accumulated requests. A few questions remain,
4.3. TREE ALGORITHMS

- **Where to store?** This central location should be close to the center of activity so that the overhead of recording requests is kept at a minimum. A fixed location, however, may not be desirable, due to changing user traffic statistics.

- **How to factor?** Since propagating from a central location is different from propagating from the node where the request is originated, it may seem that we have to keep track of the origin of a request. In other words, every edge needs to know the topology of the network in order to correctly factor a request.

- **When to propagate?**

  To answer the question of when to propagate, we observe a special property of a lookup-only request sequence. Without update requests, the offset vector is determined by the number of lookup requests in the sequence. In Figure 4.28, we show that the counting processes for $a^\lambda$ and $b^\lambda$ are along separate dimensions, hence independent. In other words, the order of lookup requests does not affect the final offset values, only their counts matter. This eliminates the need of recording a time stamp for a lookup request, and also suggests that we propagate the delayed lookup requests upon an update request.

  A central storage avoids system wide synchronization when propagating the delayed lookup requests. However, a fixed location is undesirable in a distributed environment with changing user traffic statistics. For this reason, we propose using ZOR as a suitable alternative, because of its self-addressing property, as discussed in Section 4.3.3.

  At first glance, it may seem that we have to record the precise origin of a lookup request in order to correctly factor it. Such information is actually not necessary. What we need is a common reference so that lookup requests can be correctly recorded with respect to it. Again, a ZOR is a convenient choice for such purpose because it uniquely defines an order between the two end nodes of any edge in a tree. Given a ZOR as the common reference, we then record a lookup request for edge $(a,b)$ as originating from either its NE or FE, instead of its fixed a end or b end. These counters are meaningful as long as we can tell which end is closer to the ZOR, i.e., whether $a \leftarrow b$ or $b \leftarrow a$, which is guaranteed by the definition of the ZOR. Moreover,
Figure 4.28: Lookup-only requests, $D = 2$. The counting processes for $a^\lambda$ and $b^\lambda$ are independent.
for an edge along the path $P_{v \mapsto \rho_z}$ between a requesting node $v$ and the current ZOR $ho_z$, this lookup request is coming from its FE. While for an edge off path $P_{v \mapsto \rho_z}$, this request is coming from its NE. Since it is inevitable for a lookup request to propagate to the ZOR, only edges on path $P_{v \mapsto \rho_z}$ process the request, and the request does not propagate to any edges off path $P_{v \mapsto \rho_z}$. FE requests are processed as they arrive, while NE requests are delayed. We can now correctly factor a request using the following simple counters without origin and topology information.

- The ZOR stores the number of lookup requests $L$ since the last update request.
- Each edge $(a, b)$ stores the number of lookup requests $l(a, b)$ processed so far since the last update request.

The difference between these two counters is the number of NE requests that have been delayed for edge $(a, b)$. The propagation of the delayed lookup requests is triggered by an update request: the update request is first propagated to the ZOR, which in turn propagates the counter $L$ to all edges. Since no FE requests are delayed, edge $(a, b)$ needs to process $(L - l(a, b))$ NE lookup requests and one update request to bring its offset up to date.

In summary, we partition an input request sequence into phases, each ending with an update request. Within each phase, the ZOR maintains the number of lookup requests and the location of the ZOR changes according to input requests. At the end of each phase, the ZOR propagates its counter $L$ to all edges so that their offsets are current.

**Algorithm 4.5. ZOR location**
Let the location of the ZOR be $\rho_z$ and $\rho_z'$ before and after the current lookup request, $v$ be the requesting node, and $P_{v \mapsto \rho_z}$ be the path between $v$ and $\rho_z$.

1. If $v = \rho_z$, then $\rho_z' = \rho_z$, $L = L + 1$.
2. If $v \neq \rho_z$,

(a) From $v$ to $\rho_z$ along path $P_{v \mapsto \rho_z}$, for all edges on $P_{v \mapsto \rho_z}$, update their offsets according to the current lookup request. $L = L + 1$. 

(b) From \( \rho_z \) to \( \rho'_z \) along path \( P_{v \to \rho_z} \), for edge \((a, b) \in P_{v \to \rho_z} \), suppose \( a \leftarrow b \), i.e., node \( a \) is nearer to \( \rho_z \) than node \( b \). After processing all previously delayed lookup requests, i.e., \((L - l(a, b))\) lookup requests from node \( a \), if node \( a \) is no longer a ZOS, then the ZOR moves across this edge. We continue to propagate the counter \( L \) until an edge whose NE remains a ZOS after processing all previously delayed lookup requests, this NE is then the current ZOR \( \rho'_z \).

In Algorithm 4.5, the request propagates from \( v \) to \( \rho_z \) and from \( \rho_z \) to \( \rho'_z \). All edges between \( \rho_z \) and \( \rho'_z \) have processed all lookup requests, thus guaranteeing the correct location of the current ZOR. For an edge between the current ZOR \( \rho'_z \) and \( v \), its NE may not be a ZOS due to delayed lookup requests. Such inconsistency is temporary and would be fixed upon an update request.

Since we delay the propagation of lookup requests to some edges in the tree, the offsets of these edges are not up-to-date. Even with this partial information, it is still possible to distribute replicas achieving performance comparable to that with normal request propagation. We modify the tree algorithm to accommodate the partial propagation of lookup requests. These modifications evidently depend upon the edge algorithm chosen. The algorithm presented here is suitable for all three edge algorithms, \( A_L \), \( A_T \), and \( A_H \), as presented in Section 4.2.3. The following theorem describes how the replica set expands in these algorithms.

**Theorem 4.11.** In a tree network, upon a request \( \sigma_i \) from node \( v \), the new replicas distributed by algorithms \( A_L \), \( A_T \), or \( A_H \) induce a connected subpath along the path between the requesting node \( v \) and its closest replica \( \hat{v} \).

**Proof.** We first prove by contradiction that there can be no more than one expansion point from \( R \) to \( v \). Let \( \hat{v}' \neq \hat{v} \) be a second expansion point from replica set \( R \), then the two paths \( P_{v \to \hat{v}} \) and \( P_{v \to \hat{v}'} \) do not overlap. But nodes \( \hat{v} \) and \( \hat{v}' \) are connected, resulting in a loop. Contradiction!

Next we show that these algorithms add replicas only along a unique path from the replica set \( R \) to \( v \), and nowhere else. We observe that these edge algorithms do not replicate at node \( a \) unless the following three conditions are satisfied:
4.3. TREE ALGORITHMS

- Node $a$ does not already have a replica, $a \notin R$.
- Request $\sigma_i$ is either $a^\lambda$ or $a^\delta$.
- The offset of edge $(a, b)$ satisfies the rules of a given edge algorithm.

Let $\rho_z$ be the ZOR after processing $\sigma_i$ and $P_{v \leftrightarrow \rho_z}$ be the path between $\rho_z$ and $v$. Suppose $\rho_z \notin R$. For edge $(a, b)$, we name its NE $a$ and its FE $b$, i.e., $\rho_z \in T_a(a, b)$. We consider all possible locations among $v$, $\hat{v}$, and $\rho_z$.

- If $\rho_z \in P_{v \leftrightarrow \hat{v}}$, as shown in Figure 4.29(a).
  
  1. For edge $(a, b) \notin P_{v \leftrightarrow \rho_z}$, if $s_{i-1} \neq ab$, $s_{i-1} = a$. With request $\sigma_i$ factoring to a request from node $a$, none of the edge algorithms replicates at node $b$.
  
  2. For edge $(a, b) \in P_{\rho_z \leftrightarrow \hat{v}}$, $s_{i-1} = b$. With request $\sigma_i$ factoring to a request from node $a$, these edge algorithms may replicate at node $a$.
  
  3. For edge $(a, b) \in P_{v \leftrightarrow \rho_z}$, $s_{i-1} = a$. With request $\sigma_i$ factoring to a request from node $b$, these edge algorithms may replicate at node $b$.

- If $\hat{v} \in P_{v \leftrightarrow \rho_z}$, as shown in Figure 4.29(b).
  
  1. For edge $(a, b) \notin P_{v \leftrightarrow \rho_z}$, if $s_{i-1} \neq ab$, $s_{i-1} = a$. With request $\sigma_i$ factoring to a request from node $a$, none of the edge algorithms replicates at node $b$.
  
  2. For edge $(a, b) \in P_{\rho_z \leftrightarrow \hat{v}}$, $s_{i-1} = b$. With request $\sigma_i$ factoring to a request from node $b$, none of the edge algorithms replicates at node $a$.
  
  3. For edge $(a, b) \in P_{v \leftrightarrow \hat{v}}$, $s_{i-1} = a$. With request $\sigma_i$ factoring to a request from node $b$, these edge algorithms may replicate at node $b$.

- If $\rho_z \notin P_{v \leftrightarrow \hat{v}}$ and $\hat{v} \notin P_{v \leftrightarrow \rho_z}$, as shown in Figure 4.29(c). Let $w$ be the first node where $P_{v \leftrightarrow \hat{v}}$ diverges from $P_{v \leftrightarrow \rho_z}$.

  1. For edge $(a, b) \in P_{v \leftrightarrow \rho_z}$
     
     (a) If edge $(a, b) \in P_{w \leftrightarrow \rho_z}$, $s_{i-1} = b$. With request $\sigma_i$ factoring to a request from node $b$, none of the edge algorithms replicates at node $a$. 

(b) If edge $(a, b) \in P_{v \rightarrow w}$, $s_{i-1} = a$. With request $\sigma_i$ factoring to a request from node $b$, these edge algorithms may replicate at node $b$.

2. For edge $(a, b) \notin P_{v \rightarrow \rho_z}$
   
   (a) If edge $(a, b) \notin P_{w \rightarrow v}$, if $s_{i-1} \neq ab$, $s_{i-1} = a$. With request $\sigma_i$ factoring to a request from node $a$, none of the edge algorithms replicates at node $b$.

   (b) If edge $(a, b) \in P_{w \rightarrow v}$, $s_{i-1} = b$. With request $\sigma_i$ factoring to a request from node $a$, these edge algorithms may replicate at node $a$.

A similar argument holds for the case when $\rho_z \in R$. We conclude that the new replicas are added along the path between the requesting node and its closest replica. □

With partial lookup request propagation, although an edge might have missed some previous NE lookup requests, offsets of the form $(0ll)$ are invariant to any NE lookup requests. We can also use the connectivity in the replica set $R$ to infer where to replicate. The following algorithm describes how to replicate with partial propagation of lookup requests.

**Algorithm 4.6. Replication with partial propagation of requests**

Let the location of ZOR be $\rho_z$ and $\rho_z'$ before and after the current lookup request. Let $v$ be the requesting node and $P_{v \rightarrow \rho_z}$ be the path between $v$ and $\rho_z$.

- For edges on path $P_{v \rightarrow \rho_z}$, the current request factors to a FE request.

  - For edge $(a, b) \in P_{v \rightarrow \rho_z'}$, where $a \leftarrow b$ and $s = a$. Edge $(a, b)$ may not have processed all lookup requests since the last update request. Upon the current request, its offset changes from $(0kl)$ to $(0, k-1, l-1)$, where $k \geq 1$. If an edge algorithm replicates at node $b$, its offset must be $(000)$, which is invariant to NE lookup requests. We can therefore proceed with replication without delayed lookup requests.

  - For edge $(a, b) \in P_{\rho_z \rightarrow \rho_z'}$, its offset is up-to-date, and an edge algorithm can replicate accordingly.
4.3. TREE ALGORITHMS

(a) $\rho_z \in P_{v \leftrightarrow \hat{v}}$.

(b) $\hat{v} \in P_{v \leftrightarrow \rho_z}$.

(c) $\rho_z \notin P_{v \leftrightarrow \hat{v}}$, and $\hat{v} \notin P_{v \leftrightarrow \rho_z}$.

Figure 4.29: Theorem 4.11. Possible replications are along path $P_{v \leftrightarrow \hat{v}}$. 
• For edges off path $P_{v \rightarrow \rho_z}$, the current request factors to a NE request.

From Theorem 4.11, the new replicas are added along the path between the requesting node and its closest replica. We search for a possible expansion of the replica set $R$ along the unique path from $\rho_z$ to $R$, $P_{\rho_z \rightarrow R}$. This path can overlap completely or partially with path $P_{v \rightarrow \rho_z}$, depending on the relative locations of $v$, $R$, and $\rho_z$.

\[- \rho_z \notin R \]

* For algorithms $A_I$ and $A_H$, let $(a, b)$ be the first edge along path $P_{\rho_z \rightarrow R}$ with offset $0kl$. From Lemma 4.1.C.3, the remainder of the edges along path $P_{\rho_z \rightarrow R}$ must have offset $(0l'l')$, which allows $R$ to expand to node $a$ while maintaining connectivity in $R$.

* For algorithm $A_T$, let $(a, b)$ be the first edge along path $P_{\rho_z \rightarrow R}$ whose offset is $(0kl)$ with $k > 0$. From Lemma 4.1.C.2, the remainder of the edges along path $P_{\rho_z \rightarrow R}$ must have offset $(0k'l')$ with $k' > 0$, which also allows $R$ to expand to node $a$ while maintaining connectivity in $R$.

\[- \rho_z \in R, \text{i.e., the NE of an edge has a replica. The only replication candidate is } (000). \text{ Let the offset of an edge } (a, b) \text{ be } (000). \]

Since $(000)$ is invariant in both $a^\lambda$ and $b^\lambda$, the offset of edge $(a, b)$ must have been $(000)$ before the current request. A replication triggered by $(000)$ must have already been carried out along path $P_{v \rightarrow \rho_z}$.

Unlike replica additions, replica deletions remove a subtree in $R$. When $\rho_z \in R$, it is difficult to locate the first replica to delete without propagating requests, so we delay replica deletions until the next update request.

We have proposed and analyzed two new methods, smart and delayed propagation, to reduce the overhead of request propagation. The idea of smart propagation is to continue propagation only if there is an offset change, and the idea of delayed propagation is to propagate after accumulating a number of requests. Naturally, the next step is to apply the idea of smart propagation to delayed propagation. In delayed propagation, the FE requests are processed as they arrive, while the NE requests are
4.3. TREE ALGORITHMS

delayed. Among the three invariant states \((0DD), (0ll)\) with \(0 < l < D\), and \((000)\), only \((000)\) is invariant to FE lookup requests. Consequently, propagation of such lookup requests stop at an edge with offset \((000)\) according to smart propagation. Overall, the extent of smart propagation is restricted because the offsets of some edges are not up-to-date due to delayed NE requests.

4.3.6 Discussion

Order of an Input Sequence

In general, the order of requests in a sequence affects the offset value. Having the same lookup and update statistics from nodes \(a\) and \(b\), two sequences may still end up with different offset values, as shown in the following example.

**Example 4.11.** Consider two input sequences \(\sigma_1 = \{a^\lambda, b^\lambda, b^\delta, b^\delta, a^\delta\}\) and \(\sigma_2 = \{b^\lambda, b^\delta, b^\delta, a^\delta, a^\delta\}\), with the first request in \(\sigma_1\) being shifted to the last in \(\sigma_2\). For an initial offset \((000)\), \(w(\sigma_1) = (102)\) and \(w(\sigma_2) = (001)\).

Intuitively, 2-dimensional counting is equivalent to a bounded 2-dimensional random walk, where \(a^\lambda\) corresponds to a step in \(+x\) direction, \(b^\delta\) a step in \(-x\), \(b^\lambda\) a step in \(+y\), and \(a^\delta\) a step in \(-y\). Without the bounds, the number of lookup and update requests determines the \(x\) and \(y\) coordinates, and the order of requests does not matter. However, acting as bounds, the loops in the transition diagram consume input requests without changing offset values. Thus the order of requests in a sequence affects the offset value. In the following cases, the order of requests does not matter because the two directions are independent and there is no back-tracking movements, as shown in Figure 4.30.

- An input sequence consisting of only \(b^\lambda\)'s and \(b^\delta\)'s.
- An input sequence consisting of only \(a^\lambda\)'s and \(b^\lambda\)'s.
- An input sequence consisting of only \(a^\delta\)'s and \(b^\delta\)'s.
- An input sequence consisting of only \(a^\lambda\)'s and \(a^\delta\)'s.
Figure 4.30: Cases when the order of requests in a sequence does not affect the offset value.

4.4 Implementation Issues

Network Size

A tree algorithm operates in a distributed manner: each edge processes a factored input request and determines its appropriate edge replica state, which constitutes the replica set for the tree. However, such an algorithm implicitly assumes that all edges know the input requests, thus every request must propagate to all edges. We have proposed and analyzed two methods, smart and delayed propagation, to reduce the overhead of request propagation. Yet the size of the network also contributes to this propagation overhead. In actual implementation, we apply additional heuristics so that only part of the hierarchical network needs to be included in the calculation.

Let $T_1$ be the subtree induced by the replica set $R$ and all nodes from which lookup and update requests originate. All edges in $T_1$ are active, because they carry network traffic in the form of lookup and update requests. The remainder of the edges in $T_2 = T - T_1$, on the other hand, are not involved in any signaling, hence
inactive. Initially, their offset vectors are of the form \((0DD)\). With all the input requests factored to NE requests, their offset vectors remain unchanged. Therefore a tree algorithm can exclude these inactive edges.

A tree algorithm should include all edges in the replica set \(R\) because \(R\) expands or contracts according to the input requests. Moreover, replica additions and deletions take place at the boundary of \(R\). Therefore, one might speculate whether it is sufficient to consider only the boundary edges in addition to the edges in \(R\). An obvious drawback of this scheme is that we can only add replicas to the immediate neighbors of \(R\), as opposed to distributing replicas all the way to the requesting node. More importantly, upon adding a new replica, a new boundary edge is also added, but this new edge is given no information of previous input requests. If edge \((v_1, v_2)\) were not included in the calculation until node \(v_1\) is included into \(R\), it would have missed out all requests from both nodes \(v_1\) and \(v_2\). Unlike the case of inactive edges, these requests could have changed its offset from \((0DD)\).

Alternatively, we make a boundary node \(u \in R\) the computation point for subtree \(T(u)\) containing all the nodes covered by \(u\). This computation point maintains offsets for all edges within the subtree, therefore avoiding the cost of any request propagation within the subtree, as shown in Figure 4.31. In maintaining offsets for all active edges, this hybrid approach combines the distributed computation, for edges in \(R\), and centralized computation in boundary nodes, for edges outside of \(R\). In contrast to the boundary-edge-only approach, it does not suffer from any loss in performance. During the course of execution, an edge joins \(T(u)\) as soon as it is on the path between \(u\) and some requesting node \(v\). For a request originating from outside of \(T(u)\), it is equivalent to a request from node \(u\) for all edges in \(T(u)\). For a request originating from within \(T(u)\), request propagation can therefore piggyback on request service, since both lookup and update requests have to follow the replica pointer to its closest replica \(u\).

**Replica Pointers**

As in the off-line replication case, a replica search path does not necessarily coincide with the parent-child relation of a database hierarchy. To avoid searching the database
Figure 4.31: The hybrid implementation of an on-line replication algorithm. The replica set $R$ is the set of filled nodes. Each boundary replica node $u$ covers a subtree $T(u)$, and computes offsets for all the edges in $T(u)$, as denoted by dashed lines.

hierarchy for the closest replica, we store a replica pointer, pointing to the closest replica $\hat{v}$, if a node $v$ does not already have a replica locally. Unlike the off-line case, a replica pointer is dispensable if every active edge in the tree maintains its replica state, because one can infer replica locations from the replica states of the edges. In particular, to find the closest replica to a node $v$ without a local replica, we examine all the edges connected to node $v$. All but one of them must have a replica state other than $v$. Suppose edge $(u,v)$ is in state $u$ and $R \subseteq T_u(u,v)$. We repeat the same search process at node $u$ until we reach $R$. This property is valid because $R$ is connected and there exists a unique path from node $v$ to its closest replica $\hat{v}$ in a tree network. The replica state of an edge serves as a next-hop routing table from any node to its closest replica.

However, if we adopt the hybrid implementation discussed earlier, the replica state information is available only for edges in $R$, and nodes without replicas have to rely on replica pointers, which requires updates whenever the replica set $R$ changes. Recall from Chapter 3 that in a tree network, all nodes covered by the same replica induce a subtree. A change in replica pointer is therefore localized and no global recalculation is needed to determine the closest replica. Although the cost of replica pointer updates
is not included in our edge model, it is incorporated in our computer simulations. For all our simulation scenarios, the cost of request propagation dominates the cost of replica pointer update, thus cost reduction of the former is effective in sliming down the total overhead.

**Lookup vs. Update Requests**

In our simulation model, the ratio between messaging for lookup and update requests is 2:1. The network cost for serving a lookup request is two-way, i.e., one for looking up the user profile (the primary copy or a replica), and the other for transferring the user profile itself, whereas the network cost for serving an update request is one way. In the edge model, however, both lookup and update requests are one-way. To bridge the simulation model to the edge model, we simply execute an edge algorithm twice for each lookup request.

### 4.5 Simulations

We have simulated the following algorithms in the environment described in Section 2.5,

1. **HLR/VLR** as described in Section 2.2.2.

2. **Hier** Hierarchical MMT without replication as described in Section 2.2.3.

3. The on-line threshold-based algorithm proposed by Lam et al. [51] with the maximum number of replicas $N = 30$ and $R_{repl} = R_{del} = 1$, as described in Section 4.1.1.

   (a) **HOPPER** initial placement of replicas determined by HIPER [45].

   (b) **HOPPER:NI** no initial placement of replicas.

4. Off-line algorithms, UR and MR, as described in Chapter 3

   (a) **Off-line** based on the traffic statistics of the current day
(b) **Off-line:IT** based on the traffic statistics of the previous day

(c) **Off-line:MR** MR algorithm based on the traffic statistics of the current day

5. On-line algorithms

Since all three edge algorithms \( A_L, A_T, \) and \( A_H \) behave similarly in our simulations, we only show the performance of algorithm \( A_L. \)

(a) Initial placement of replicas determined by MR

   i. **On-line:NR** no overhead reduction

   ii. **On-line:SM** smart propagation

   iii. **On-line:DL** delayed propagation

(b) No initial placement of replicas

   i. **On-line:NI**

Off-line replication does not consider the cost of distributing replicas, but such costs play an important role in on-line replication. Our on-line algorithms model the cost of distributing replicas by the parameter \( D \), and \( D = 2 \) in our simulations. In addition to the cost of distributing replicas, the overhead of our on-line algorithms also include the costs of request propagation and replica pointer updates for both replica additions and deletions. The overhead associated with the threshold-based algorithm includes the costs of distributing replicas, notifying replica deletions, and searching for potential replica exchange targets. While our on-line algorithms apply multicast whenever possible, the threshold-based algorithm uses unicast to update replicas.

### 4.5.1 Network Cost

The total network cost is the sum of lookup network cost, update network cost, and replication related costs that include the costs of replica distribution, request propagation, and replica pointer updates. Figure 4.32 shows the total network cost per second over a 24-hour period. Off-line replication results in the lowest network cost,
followed by on-line replication and Hierarchical MMT without replication. HLR/VLR has the highest network cost because it is a centralized system whereas Hierarchical MMT takes advantage of locality in user calling and mobility patterns. The total network cost of Hierarchical MMT is dramatically reduced by profile replications. Off-line replication performs the best because it takes advantage of complete knowledge of user traffic statistics, which is not available to on-line replication. It is according to the request sequence seen so far that an on-line algorithm decides whether and where to replicate. Due to low local lookup percentage, for HLR/VLR and Hierarchical MMT without replication, the lookup network cost dominates and the curves follow the shape of the call traffic volume. Off-line replications, on the other hand, result in very high local lookup percentage, thus the update network cost dominates in the total network cost, and the curves follow the shape of the movement traffic volume. On-line replication also results in high local lookup percentage, and the lookup network cost is comparable to the update network cost. To understand how heavily the performance of an off-line algorithm is dependent on accurate traffic statistics, suppose we do not have such knowledge and an off-line algorithm runs on the traffic statistics from a previous day, as would be the likely case in practice. Although small in volume, the traffic variations from day to day are significant enough to cause the performance of the off-line algorithms to degrade drastically. This is where an on-line algorithm kicks in, because it can dynamically adjust replica placement according to user calling and mobility patterns. Without reducing the cost of request propagation, however, the total network cost of an on-line algorithm is greater than Hierarchical MMT without replication. Both smart and delayed propagation algorithms are effective in reducing this overhead.

The next three figures compare our on-line algorithms with the threshold-based algorithm. In Figure 4.33, the network cost of both smart and delayed propagation is lower than that of the threshold-based algorithm. Our on-line simulations initially place replicas according to an off-line algorithm based on the previous day’s traffic statistics. We observe little difference in performance without such initialization, as shown in Figure 4.34. The lookup and update network costs are shown in Figures 4.35 and 4.36, respectively. The peaks are results of high traffic volume in the morning.
Figure 4.32: Total network cost.
4.5. SIMULATIONS

Although our on-line algorithm allocates more replicas than the threshold-based algorithm, as shown in Figure 4.41, its replica update network cost is lower due to multicast.

![Graph showing network cost per second over time]

Figure 4.33: Total network cost. The network cost of both smart and delayed propagation is lower than that of the threshold-based algorithm.

4.5.2 Database Access

The next group of figures compares the performance in terms of database access. In Figure 4.37, HLR/VLR results in a smaller number of database lookups than Hierarchical MMT without replication because the latter requires traversal of a database hierarchy to retrieve a user profile, while the former requires a maximum two database lookups for each lookup request. This is also the case for our replication algorithms
Figure 4.34: Total network cost. With or without initial placement of replicas, the network costs of our on-line algorithms are similar.
4.5. SIMULATIONS

Figure 4.35: Lookup network cost.
Figure 4.36: Update network cost.
because of replica pointers. Our replication algorithms generate a smaller number of lookup requests than HLR/VLR because they place replicas close to where they are needed most, so that more lookup requests can be resolved locally. Figures 4.38 and 4.39 compare our on-line replication algorithms with the threshold-based algorithm in terms of database lookups and updates, respectively. Our on-line algorithm has higher local lookup percentage, therefore requires a smaller number of total database lookups than the threshold-based algorithm. With more replicas allocated, our on-line algorithm generates more database updates than the threshold-based algorithm, but the network cost of updating these replicas is not higher due to multicast, as shown in Figure 4.36. Figure 4.40 summaries the performance of three algorithms:

![Database Lookups Graph](image)

Figure 4.37: Database lookups.

HOPPER, On-line:SM, and MR. The number of database access and the network cost
Figure 4.38: Database lookups.
4.5. SIMULATIONS

Figure 4.39: Database updates.
of On-line:SM and MR are normalized to that of HOPPER. Except in the category of database updates (DBU), On-line:SM has a performance gain of 11% in database lookups (DBL), 61% in lookup network cost (LNC), 16% in update network cost (UNC), and 27% in total network cost (TNC).

![Graph](image)

Figure 4.40: Comparison among three algorithms: HOPPER, On-line:SM, and MR. The number of database access and the network cost of On-line:SM and MR are normalized to those of HOPPER. The following performance measures are considered: database lookups (DBL), lookup network cost (LNC), database updates (DBU), update network cost (UNC), and total network cost (TNC).

### 4.5.3 Storage

Figure 4.41 shows the number of replicas allocated by the various algorithms. In HLR/VLR, a user profile stored at a VLR resembles a replica, thus storage in a VLR counts as one. Although all off-line replication algorithms allocate about the same number of replicas, our optimal algorithms perform better than the threshold-based algorithm by choosing more strategic locations for placing replicas, as shown
in Section 3.6. On-line replication algorithms allocate more replicas than off-line algorithms because on-line replication dynamically decides replica placement based on the past requests, whereas off-line replication decides replica placement based on complete knowledge of user calling and mobility statistics. Our on-line replication algorithm allocates the most number of replicas because it maintains a connected replica set, even though some of the replicas in intermediate levels may never be accessed.

![Graph showing average number of replicas per user.](image)

**Figure 4.41**: Average number of replicas per user.

### 4.5.4 Discussion

All the simulation results shown above are from a specific set of traffic parameters. To explore the performance advantages of our on-line replication algorithms...
under different traffic conditions, we examined the effect of variations in the traffic model. Figure 4.42 summarizes the performance ratios of our on-line algorithm to the threshold-based algorithm under different traffic conditions. $TP_1$, $TP_2$, and $TP_3$ represent three different sets of traffic parameters, where $TP_1$ is the original set of traffic parameters used throughout this dissertation, users in $TP_2$ move 10 times faster than users in $TP_1$, and the local callee percentage in $TP_3$ is $\frac{1}{4}$ of that in $TP_1$. The total number of database access and the network cost of both On-line:SM and MR are normalized to that of HOPPER. The off-line algorithm MR is included for comparison. Except in the category of database updates (DBU), On-line:SM performs consistently better than HOPPER. These simulations demonstrate that our on-line replication algorithm is not traffic dependent.

![Figure 4.42: Comparison among different sets of traffic parameters. Users in $TP_2$ move 10 times faster than users in $TP_1$. The local callee percentage in $TP_3$ is $\frac{1}{4}$ of that in $TP_1$. The number of database access and the network cost of both On-line:SM and MR are normalized to those of HOPPER. The following performance measures are considered: database lookups (DBL), lookup network cost (LNC), database updates (DBU), update network cost (UNC), and total network cost (TNC).](image)

**Figure 4.42: Comparison among different sets of traffic parameters.** Users in $TP_2$ move 10 times faster than users in $TP_1$. The local callee percentage in $TP_3$ is $\frac{1}{4}$ of that in $TP_1$. The number of database access and the network cost of both On-line:SM and MR are normalized to those of HOPPER. The following performance measures are considered: database lookups (DBL), lookup network cost (LNC), database updates (DBU), update network cost (UNC), and total network cost (TNC).
4.6 Summary

In this chapter, we have developed a unified framework for the on-line edge replication problem using the offset transition diagram. By putting the previously proposed algorithms, $A_L$ and $A_H$, in our framework, we have shown the correspondence between them. We have also proposed a new 3-competitive edge algorithm $A_T$, and extended the edge model to include the cost of request propagation. We have derived sufficient conditions for consistently coalescing edge solutions into a tree solution. Most importantly, we have made the tree algorithms practical by reducing the overhead imposed by factoring. Our simulation results have demonstrated that our on-line algorithms incur smaller network cost and enable more lookups to be resolved locally than the threshold-based algorithm.

Although both the tree algorithm and the threshold-based algorithm follow the same principle: replicate if the benefit of replication is greater than the cost paid so far, their scope is quite different. The threshold-based algorithm is myopic in nature since it treats lookup and update requests as isolated events. In contrast, the tree algorithm is more holistic in its treatment. The factoring technique connects individual edges, so an edge is no longer isolated, but affected by all input requests. Its decision to add or delete a replica is no longer a local one but a global one with network structure under consideration.
Chapter 5

Mobility Management For Heterogeneous Networks

The previous two chapters addressed the scalability and performance issues of mobility management via hierarchical database structure and profile replication. The subject of this chapter is efficient and scalable mobility management techniques (MMTs) to support universal personal identification (UPI) in a heterogeneous network environment. We focus on database architectures that affect the performance of MMTs. This chapter begins with an introduction to the access network model in Section 5.1, followed by a detailed survey of related research work in Section 5.2. Section 5.3.1 describes our first proposal that builds upon existing location databases and modifies HLR/VLR to accommodate location independent UPI, as introduced in Chapter 1. To provide better scaling and to exploit locality in user calling and mobility patterns, our second proposal adopts a hierarchical architecture in Section 5.3.2. Section 5.4 shows the performance of our proposed MMTs via large scale computer simulations.

5.1 Access Network Model

In a heterogeneous network environment, a user is reachable via multiple access networks (ANs). The services and coverage areas offered by these ANs may differ. In particular, the coverage areas of these ANs may overlap, and the selection of an AN
depends on the availability of an AN, a user’s preference, and roaming agreements among ANs. The service contract relationship between a user and an AN is described by a subscription, which specifies the kind of services a user desires from an AN. In a heterogeneous network environment, a user can access a wide variety of services from different ANs. Logically, a user’s subscription consists of individual fragments of subscriptions to different networks. Since a user may have access to multiple mobile terminals and the locations of these terminals may be different at any given time, a user may have multiple physical presences in the communication network. Similar to a homogeneous network environment, each AN in a heterogeneous network environment encapsulates information about its subscribers in user profiles, as described in Section 2.1.2. To locate a user, the UPI of the user must be mapped to its corresponding last known network address(es), which can be a local telephone switch address, a mobile switching center (MSC) address, or an IP address, etc. Due to multiple ANs and multiple terminals, a user can hold multiple valid network addresses simultaneously.

5.1.1 Multiple Registration Scheme

When a mobile terminal moves to a new registration area (RA) of an AN, it initiates a location registration procedure, which not only registers the user at his current location, but also cancels any registrations at his previous location. In single registration, a user registers with at most one AN at any time and location, whereas in multiple registration, a user can register with more than one available AN at any time and location. Compared with the single registration scheme, the multiple registration scheme provides users with independent services through cooperating or non-cooperating ANs. Users can therefore obtain services from different ANs at the same time. Furthermore, with the multiple registration scheme, users have more flexibility to switch services from one network to another during a connection. This switching is determined by communication conditions such as network load, user mobility patterns, and the user’s preference, etc. Dynamic and seamless switching between ANs is absolutely essential for most multitier applications.
However, the benefits of the multiple registration scheme come with a potential complication in the registration process: when a user moves out of the coverage area of AN A and moves into the coverage area of AN B, the previous registration at AN A can not be invalidated, i.e., the registration at AN A is obsolete. This is referred to as the missing anomaly in [54]. The following two factors cause this complication. First, cancellations of previous registrations are triggered only by the registration procedure. In other words, there is no explicit deregistration. Second, the paradigm of HLR/VLR implies that a user’s registration with a particular AN is completely handled by that AN.

We assume for the rest of this chapter that a mobile terminal can remember a list of its registrations with ANs. By checking the information broadcast on a control channel, a mobile terminal detects environmental changes and selectively cancels the previous registration(s) based on the availability and coverage area of ANs. For simplicity, we further assume that as long as a user can receive services from a network access point, all the necessary explicit cancellations (both intra- and inter-network) can be performed through this network access point independent of any roaming agreements.

5.1.2 Access Control of Location Information

The presence of multiple ANs in a heterogeneous network environment requires access control of location information that is owned or shared by ANs. From a user’s standpoint, access control measures allow him to specify the range over which his location information should be made available to receive calls and the range over which the network should look for his called parties. These range specifications can be either geographic (e.g., San Francisco) or service-oriented (e.g., voice service). We will see in Section 5.3.2 that it is easy to implement the geographic ranges in Hierarchical MMT. From the network management’s standpoint, access control measures define an AN’s private and public realms and permit exchange of location information among ANs without compromising the autonomy of individual ANs.
5.1.3 System Model

We extend the system model of the Pléiades simulator [51] to the heterogeneous network environment by introducing an AN model. The key aspects of our AN model include:

1. Each AN has its own designated coverage area, and it can offer services to its subscribers only within this coverage area. When a user roams outside his own AN’s coverage area, he may be able to receive services from an AN that has a roaming agreement with his own AN. A roaming group is a collection of ANs sharing a roaming agreement which is in effect for any pair of ANs in the group. Compared with a bilateral roaming agreement, the roaming group model simplifies the description of multilateral collaborations because when an AN joins a roaming group, it automatically establishes a roaming agreement with all the existing members in the roaming group.

2. We model the availability of an AN as the service probability at a given time and location. The service probability describes the likelihood of a user receiving services from a particular AN. It reflects the network management (e.g., call admission control) of an AN and limits the choice of ANs for location updates and lookups.

3. A multiple subscription scheme permits a user to subscribe to more than one AN. This allows greater service flexibility at the cost of additional network signaling overhead. Since the exact user subscription pattern in the future is unknown, we experiment with a subscription distribution defined by a geometric series. Given a parameter \( p_s, \{1, p_s, p_s^2, \ldots, p_s^{(n-1)}\} \) determines the probabilities that a user subscribes to the first through the \( n \)th AN.

5.2 Related Work

Proposals for local number portability (LNP) in the wire-line case rely on number translation databases and intelligent network (IN) supports [24]. For example, global
title translation (GTT) is performed at signal transfer points (STPs) in the signaling system no. 7 (SS7) network. Applying the same number translation idea, various research efforts [16, 31, 53] have proposed to extend the GTT functions to achieve number portability in the wireless case.

In place of the SS7 network, Jain et al. [44] investigated how to support non-geographical phone numbers (NGPN) with an ATM backbone. In particular, VLRs perform number translations by hashing a NGPN to the address of a translation server (TS) that contains the NGPN-to-HLR mappings. A Meta-HLR database, as described in Section 5.3.1, is similar to a TS because it stores the UPI-to-HLR mappings. However, our proposals focus on location lookup requests where the caller side has no other information except for the callee’s UPI. For location update requests, we assume that a mobile can provide necessary addressing information, via the international mobile subscriber identity (IMSI), for example. More importantly, with this two-level translation scheme, multiple ANs can operate autonomously in a heterogeneous network environment. We do not specify the implementation for the mapping between UPIs and Meta-HLRs. Several proposals in [44] are applicable for this translation task.

Issues regarding user profile identifications were considered in the context of universal mobile telecommunications system (UMTS) [25]. Lin et al. [54] proposed a multiple registration scheme in conjunction with a multitier HLR scheme in which a tier manager coordinates heterogeneous HLRs. The single registration scheme was studied in [68] for a multitier PCS, where tier integration is based on the merging of HLRs. The authors also proposed new registration algorithms to reduce the network signaling traffic from tier switching.

Wang et al. [86, 87] proposed augmenting HLR/VLR with boundary location registers (BLRs) to speed up registration and paging along the boundary between two ANs. BLRs are essentially special purpose VLRs, covering some predefined boundary areas. These proposals avoid translating a user’s UPI to HLR addresses by assuming a unique HLR for each user across all ANs, similar to the current implementation of roaming agreements among ANs. Although simple, such an assumption seriously hinders the autonomous operations among ANs, therefore this proposal can not support
5.3. MOBILITY MANAGEMENT TECHNIQUES

UPI.

Interworking of the following existing wireless networks have been studied: GSM and IS-41 [30], GSM and DECT [66], GSM and PDC [89], GSM and North American TDMA/AMPS [11]. Our work focuses on the mobility management aspect of the interworking problem and is not limited to any specific system.

5.3 Mobility Management Techniques

To support UPI, we augment HLR/VLR with Meta-HLR databases to map a UPI to multiple HLR addresses. To solve the scalability problem in this centralized approach, we propose a shared hierarchical MMT that not only scales better but also takes advantage of locality in user calling and mobility patterns.

5.3.1 Meta-HLR MMT

Our first proposal builds upon existing location databases and modifies HLR/VLR to accommodate location independent UPI. Since a user is reachable via multiple ANs, we assume that each network stores a user’s location information in one of its HLRs. Note that in a homogeneous network environment, only one HLR stores a user’s location information, whereas in a heterogeneous network environment, a user’s location information is stored in multiple HLRs in the different networks. In order to locate a user via his UPI, we need to query these HLRs from different networks to obtain a user’s location information. To address these HLRs, we introduce a new database, Meta-HLR, which stores the mapping between a UPI and its corresponding HLR(s). In other words, a user’s UPI uniquely identifies a Meta-HLR database. Query of the Meta-HLR maps a UPI to one or more HLR addresses. Subsequent queries of these HLRs retrieve the user’s location information from various ANs.

Although the total number of ANs is fairly large, only a few of them are connected with a particular user through subscriptions, and the call setup procedure requires only the address(es) of the HLR(s) to which a user has subscription(s). This is a key observation that facilitates this MMT. We can support UPI by maintaining the
mapping between a user and his HLR(s) in a separate database — Meta-HLR. The correspondence between UPIs and Meta-HLRs bears close analogy to that between geographical phone numbers and HLRs in HLR/VLR, as described in Section 2.2.2.

The location update protocol is similar to that of HLR/VLR. The location lookup protocol initially queries the Meta-HLR to obtain all the relevant HLR address(es), followed by HLR queries. In Figure 5.1, the local translation refers to translation performed by individual ANs on the UPIs of their subscribers. The global translation maps a UPI to its Meta-HLR address for the subsequent HLR queries. There are three possible levels of lookup in this MMT:

- **level 0** looks up the callee in the local VLR only.
- **level 1** looks up the callee within a single AN's coverage area.
- **level 2** looks up all possible locations of the callee.

Due to its centralized nature, the Meta-HLR MMT suffers from problems similar to HLR/VLR, i.e., it does not scale well and the Meta-HLR database may become a single-point of failure. Despite their many similarities, the contents of Meta-HLRs are much more stable than those of HLRs because they only need to be updated when a user changes his subscription contract with an AN. This happens much less frequently than a user's location updates.

### 5.3.2 Shared Hierarchical MMT

To provide better scaling and to exploit locality in user calling and mobility patterns, our second proposal adopts a hierarchical architecture. Based on the basic hierarchical MMT, as described in Section 2.2.3, we develop a shared hierarchical MMT that logically separates the mobility management structure from individual ANs. It not only reduces signaling overhead for inter-network activities, but also allows fine grain control over the range of location updates and lookups. When applying the general hierarchical structure to a heterogeneous network environment, we have two choices:
Figure 5.1: Location lookup in Meta-HLR MMT.
(a) **Basic hierarchical architecture** where each individual AN has its own location hierarchy and communicates with other ANs only at the root database level, as shown in Figure 5.2(a).

(b) **Shared hierarchical architecture** where all ANs share a single location hierarchy, as shown in Figure 5.2(b).

Figure 5.2: Two hierarchical architectures.

The advantage of a shared location hierarchy becomes clear from the following example. Suppose users $i$ and $j$ are located near one another, but are using different ANs. Under choice (a), all calls between $i$ and $j$ have to go through the root databases of the hierarchies. This is highly inefficient. But under choice (b), because $i$ and $j$ are geographically close to one another, location lookups can always be resolved at a lower-level of the location hierarchy.

The following example further illustrates the organization of a shared hierarchical architecture. Suppose user $i$ subscribes to three ANs and these subscriptions correspond to three different terminals that may appear at different locations at any given time. Figure 5.3 shows a snapshot of user $i$’s location records throughout the hierarchy. At the root level, we observe that user $i$ has three subscriptions and two of them are in one partition of the area while the third belongs to the other partition. Moving down the hierarchy, we further narrow down the possible location regions. Finally, at
the bottom of the hierarchy, three regional databases contain the user profiles, all of which bear user $i$’s identity.

![Diagram of hierarchical architecture](image)

**Figure 5.3: Shared hierarchical architecture.**

Next we address some privacy concerns for this shared hierarchical architecture. In our proposal, only the location pointers — not the user profiles — are shared in higher level databases. The leaf-level databases consist of private regional databases from different ANs, and it is in these private databases where the actual user profiles are stored. Furthermore, since the hierarchical architecture directly corresponds to the geographical area partitions, access control measures can be implemented at each level of the database hierarchy according to a user’s specifications.
5.4 Simulations

Figures 5.4, 5.5, and 5.6 show three simulation topologies representing a range of RA densities. In particular, $S_1$ is the common simulation topology used in previous chapters, $S_0$ is sparser than $S_1$, and $S_2$ is denser than $S_1$. All the ANs in $S_0$ constitute a roaming group, and the same is true for $S_1$ and $S_2$. We consider roaming agreements only in the absence of subscription services. The choice of a roaming AN is arbitrary if more than one roaming AN is available. The service probability is uniformly distributed between 0.9 and 1.0. We consider three different user subscription distributions: $p_s = 1.0$, 0.5, and 0.0, where $p_s = 1.0$ corresponds to a subscription
Figure 5.5: Simulation topologies and database hierarchies $S_1$. 
Figure 5.6: Simulation topologies and database hierarchies $\mathcal{S}_2$. 
scheme including all ANs, and \( p_s = 0.0 \) corresponds to a subscription scheme including only one of the ANs. We implement a multiple registration scheme according to the subscription distribution. The location lookup procedure searches for all possible registered locations of a user. Each of the three different MMTs — Meta-HLR, basic hierarchical and shared hierarchical MMT — runs against three different subscription distributions: \( p_s = 1.0, 0.5, \) and \( 0.0 \).

We classify database lookup responses into three categories: local, intra-network, and inter-network. Figure 5.7 shows the lookup response percentage (LRP) in each category. We observe that the single subscription scheme \( (p_s = 0.0) \) results in the lowest local and intra-network LRPs and the highest inter-network LRP, since it has the highest probability of the caller and the callee using different ANs. As \( p_s \) increases from 0.0 to 1.0, it is more likely that the caller and the callee subscribe to a common AN. Therefore, the inter-network LRP decreases and the local and intra-network LRPs increase. The following two factors contribute to the lookup performance gain of the shared hierarchical MMT:

1. Because the shared hierarchical MMT merges the intermediate level databases from different ANs, a single query to a merged database can return a callee’s location information from all possible ANs in the area covered by this database. In other words, the shared hierarchical MMT queries the databases of a single shared hierarchy, whereas the basic hierarchical MMT needs to query the databases of different AN’s hierarchies. This saving factor dominates when \( p_s = 1.0 \).

2. With the shared hierarchical MMT, the inter-network lookups do not necessarily go through the distributed root as it always does for the basic hierarchical MMT. This saving factor dominates when \( p_s = 0.0 \).

Figures 5.8, 5.9, and 5.10 show the database lookup and update performance of these MMTs for \( p = 1.0, 0.5, \) and \( 0.0 \), respectively. The number of database lookups generated by the shared hierarchical MMT is significantly fewer than that generated by the basic hierarchical MMT: during peak hours, the performance gains in database lookups are 41\%, 45\%, and 37\%, for \( p = 1.0, 0.5, \) and \( 0.0 \), respectively. The shared
hierarchical MMT performs better than the Meta-HLR MMT when $p = 1.0$ which corresponds to a subscription scheme including all ANs. The performance gap between the Meta-HLR MMT and the shared hierarchical MMT ranges from 8% when $p = 0.5$ to 27% when $p = 0.0$ during peak hours. When $p_s = 0.0$ (single subscription), the two hierarchical MMTs generate the same number of database updates.

5.5 Summary

In this chapter, we have described an AN model in a heterogeneous network environment. To support UPI in this environment, we have augmented HLR/VLR with Meta-HLR databases to map a UPI to multiple HLR addresses. To solve the scalability problem in this centralized approach, we have proposed a shared hierarchical MMT that not only scales better but also takes advantage of locality in user calling and mobility patterns. Our simulation results have shown that the shared hierarchical MMT performs significantly better than the basic hierarchical MMT in database lookups, and that the performance gap between the shared hierarchical and Meta-HLR MMTs is relatively small.
Figure 5.7: Local, intra-network, and inter-network lookup response percentages.
Figure 5.8: Database lookups and updates, $p = 1.0$. 
5.5. SUMMARY

(a) Database lookups.

(b) Database updates.

Figure 5.9: Database lookups and updates, $p = 0.5$. 
Figure 5.10: Database lookups and updates, $p = 0.0$. 
Chapter 6

Conclusions

6.1 Summary

Chapters 1 and 2 provided necessary background materials. Chapters 3, 4 and 5 described our new mobility management techniques (MMTs).

In Chapter 1, we motivated the need to support universal personal identification (UPI) in a heterogeneous network environment. The mobility management problem deals with the efficient tracking and retrieving of a mobile user’s location information given his UPI. The challenges and requirements of MMTs include 1) Location/network independent UPI support, 2) Interoperability and privacy, 3) Scalability, and 4) Efficiency.

An overview of a generic wireless network architecture was given in Section 2.1. In this architecture, the three entities related to mobility management are location databases (DBs), mobile switching centers (MSCs), and the common channel signaling (CCS) network carrying signaling traffic between DBs and MSCs. A network keeps track of a user’s location through an up-to-date user profile stored in various databases. A user profile contains not only user’s current location information, but also service information, such as billing and authentication. The coverage area of an access network is divided into registration areas (RAs), and each RA is associated with a location database. The two basic operations in mobility management are location update and location lookup. When a user moves across the boundaries of these
CHAPTER 6. CONCLUSIONS

RAs, the network updates his location information in the pertinent databases. When a caller places a call using the callee’s identification, the network queries the relevant database(s) to obtain the current location and other service information of the callee.

Current standards (IS-41 and GSM) can not support UPI because of geographical numbering. To support UPI, hierarchical MMTs have been proposed. A hierarchical structure scales better because location information is distributed, and there is no bottleneck in the system. Hierarchical structures can also exploit locality in user calling and mobility patterns.

This dissertation deals with the mobility management problem in two aspects:

1. Profile replication to enhance the performance of MMTs. Replication makes profile information more readily available, thus reducing lookup cost and latency. But to keep these replicas consistent and fresh, they must be updated whenever the user profile is updated. The principle of replication is to replicate if the benefit of replication is greater than its overhead.

2. Efficient and scalable MMTs to support integrated personal communication services from heterogeneous networks using UPI.

A replication algorithm can be either off-line or on-line. While an off-line algorithm assumes complete knowledge of user calling and mobility statistics, an on-line algorithm does not make any assumption about user traffic patterns. Instead, it decides whether to distribute new replicas or delete existing replicas after serving each request, all based on the input sequence seen so far. In both cases, the profile replication problem belongs to the family of file allocation (FA) problems, and a hierarchical database structure corresponds to a tree network. Our optimal off-line replication algorithms minimize the network messaging cost based on the network structure, communication link costs, and user calling and mobility statistics. In Chapter 3, we developed optimal off-line replication algorithms for both unicast and multicast replica updates. Our on-line replication algorithms dynamically adjust replica placement according to user calling and mobility patterns. In Chapter 4, we developed a unified framework for the on-line edge replication problem using the
6.2. **FUTURE WORK**

offset transition diagram. The edge solutions then become building blocks for solving the replication problem on a tree via factoring. We derived sufficient conditions for consistently coalescing edge solutions into a tree solution. Most importantly, we made the tree algorithms practical by reducing the overhead imposed by factoring. Although our profile replication algorithms and the threshold-based algorithms follow the same replication principle, their scopes are quite different. The threshold-based algorithms are myopic in nature, because they treat lookup and update requests as isolated events and their replication decisions are local. In contrast, our off-line and on-line algorithms are more holistic in their treatment. For example, in our on-line algorithm, all edges are connected by factoring, so an edge is no longer isolated but affected by all input requests and its decision to add or delete a replica is no longer a local one but a global one with network structure under consideration.

In Chapter 5, we proposed two MMTs to support integrated personal communication services from heterogeneous networks using UPI. We augmented HLR/VLR with Meta-HLR databases to map a UPI to multiple HLR addresses. To solve the scalability problem in this centralized approach, we proposed a shared hierarchical MMT that not only scales better but also takes advantage of locality in user calling and mobility patterns.

The performance of our proposed MMTs was studied via large scale computer simulations. Both our off-line and on-line replication algorithms were shown to be optimal and to perform better than previously proposed threshold-based algorithms.

### 6.2 Future Work

We reduced the unicast off-line replication problem to the $p$-median problem to minimize the total network cost. As discussed in Section 3.3.4, other formulations, such as reductions to the $p$-center problem and to the $p$-median problem, are also open for further study.

We enlisted the help of replica pointers in the hybrid implementation of our on-line replication algorithms, as described in Section 4.4. We would like to include the cost of updating these pointers in our model. With multicast, the optimal replica set
induces a connected subtree. Some replicas are distributed solely to maintain this connectivity. In the case of off-line replication, we were able to remove replicas that do not serve any requests while maintaining the minimum network cost. A similar approach does not apply to on-line replication due to its dynamic nature. In the case of on-line replication, we should be able to further reduce database updates and storage requirement by removing unused replicas. Our on-line replication algorithms were derived from a multicast model, and multicast was applied whenever possible, including replica updates, replica pointer updates, and replica distributions. If the signaling network does not support multicast, unicast must be adopted. The on-line replication problem with unicast is a problem to be solved in the future. Our on-line replication algorithms are deterministic. Randomized algorithms [57] achieve lower competitive ratio but require more computational complexity.
Bibliography


[79] Q. Tian and D. C. Cox. Location management in a heterogeneous network
environment. In Proceedings of WCNC’00, volume 2, pages 753–758, Chicago,
IL, USA, Sep 2000.

[80] Q. Tian and D. C. Cox. On-line profile replication for hierarchical mobility
York, Jun 2002.

[81] Q. Tian and D. C. Cox. Optimal replication algorithms for hierarchical mobility

[82] M. v. Steen, F. J. Hauck, P. Homburg, and A. S. Tanenbaum. Locating objects in

[83] M. Vudali. The location area design problem in cellular and personal communications
system. In Proceedings of ICUPC’95, volume 2, pages 591–595, Cambridge,
MA, USA, Sep 1995.

[84] H. J. Wang, B. Raman, C. Chuah, R. Biswas, R. Gummadi, B. Hohlt, X. Hong,
E. Kiciman, Z. Mao, J. Shih, L. Subramanian, B. Y. Zhao, A. D. Joseph, and
R. H. Katz. ICEBERG: an Internet core network architecture for integrated

communication systems. IEEE Journal on Selected Areas in Communications,

for multtier wireless networks. In Proceedings of MOBICOM’00, pages 99–109,
Boston, MA, USA, Aug 2000.

