ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ

ΤΜΗΜΑ ΜΗΧΑΝΙΚΩΝ
ΗΛΕΚΤΡΟΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ ΚΑΙ ΔΙΚΤΥΩΝ

OmniStore: Μηχανισμοί για την αυτοματοποίηση της διαχείρισης δεδομένων σε ένα προσωπικό σύστημα με πολλές φορητές συσκευές

ΔΙΔΑΚΤΟΡΙΚΗ ΔΙΑΤΡΙΒΗ

του

ΑΛΕΞΑΝΔΡΟΥ Σ. ΚΑΡΥΠΙΔΗ

Απόφοιτου Επιστήμης Υπολογιστών Πανεπιστημίου Κρήτης

Συμβουλευτική Επιτροπή : Σ. Λάλης
Η. Χούστης
Λ. Τασιούλας

Εγκρίθηκε από την επταμελή εξεταστική επιτροπή την 23η Φεβρουαρίου 2007

Βόλος, Φεβρουάριος 2007

.....

ΑΛΕΞΑΝΔΡΟΣ Σ. ΚΑΡΥΠΙΔΗΣ

Μηχανισμοί για την αυτοματοποίηση της διαχείρισης δεδομένων σε ένα προσωπικό σύστημα με πολλές φορητές συσκευές

Εισαγωγή

Όσο ο όγκος των δεδομένων που παράγουμε, συλλέγουμε και χρησιμοποιούμε αυξάνεται συνεχώς, τόσο πιο δυσχερής γίνεται η συμβατική διαχείριση των αρχείων, η οποία συνήθως απαιτεί σημαντική εμπλοκή του χρήστη. Ο μεγάλος αριθμός των φορητών και ενσωματωμένων συσκευών που έχει μαζί του ο καθένας μας απλά επιδεινώνει τη κατάσταση. Η επιστημονική κοινότητα έχει μακράν συμφωνήσει ότι τα υπολογιστικά συστήματα που πρόκειται να διαχυθούν στο ανθρώπινο περιβάλλον θα πρέπει να σχεδιαστούν ώστε να *υποστηρίζουν* τον άνθρωπο στις δραστηριότητες του, όχι να αποσπούν τη προσοχή του από αυτές. Αυτή η εργασία συνεισφέρει σε αυτό το στόχο μειώνοντας τη πολυπλοκότητα της διαχείρισης αρχείων σε ένα προσωπικό υπολογιστικό περιβάλλον που αποτελείται από πολλές φορητές συσκευές. Για τον σκοπό αυτό σχεδιάσαμε και υλοποιήσαμε το “OmniStore”, ένα σύστημα που απαλλάσσει το χρήστη από χρονοβόρες εργασίες διαχείρισης αρχείων. Αυτό επιτυγχάνεται μέσω επικοινωνίας των συσκευών τόσο με υπηρεσίες υποδομής, όσο και μεταξύ τους (αδόμητη επικοινωνία), με στόχο την αυτοματοποίηση των διεργασιών που αφορούν την διαχείριση αρχείων.

Κάθε χρήστης διαθέτει προσωπικό χώρο αποθήκευσης στην υποδομή, με τον οποίο συσχετίζει τις συσκευές που του ανήκουν. Τα αρχεία που δημιουργούνται σε αυτές συγκεντρώνονται αυτομάτως στην αποθήκευση υποδομής. Ο χρήστης έχει πρόσβαση σε αυτήν από οπουδήποτε και μπορεί μέσω κατάλληλης διεπαφής να προγραμματίζει τη μεταφορά αρχείων σε συσκευές της αρεσκείας του. Η αποστολή αρχείων από τις συσκευές προς την αποθήκευση υποδομής και αντιστρόφως πραγματοποιείται όταν αυτές επικοινωνούν, κάτι το οποίο συμβαίνει περιοδικά.

Επιπλέον, οι συσκευές επικοινωνούν και μεταξύ τους, με σκοπό να επιτευχθεί η συλλογική λειτουργία τους. Συγκεκριμένα, ανταλλάσσουν πληροφορίες σχετικά με τις επικρατούσες συνθήκες (τοποθεσία, θερμοκρασία, φωτισμός, κλπ.) οι οποίες εξάγονται από τους διάφορους αισθητήρες που βρίσκονται ενσωματωμένοι στις συσκευές, και οι οποίες στη συνέχεια προσαρτώνται στα αρχεία που δημιουργούν, ώστε να είναι εύκολη η αναζήτησή με βάση αυτές. Επίσης, δημιουργούν αντίγραφα σημαντικών αρχείων για να αυξήσουν τη διαθεσιμότητα αυτών, ή μετακινούν αρχεία για να ανακατανείμουν τον ελεύθερο χώρο ώστε να αποσυμφορηθούν οι έντονα χρησιμοποιούμενες συσκευές. Τέλος, επιτρέπουν την απομακρυσμένη πρόσβαση σε αρχεία μιας συσκευής από εφαρμογές σε άλλη, παρέχοντας ακόμη και ανοχή σε βλάβες όταν το αρχείο είναι διαθέσιμο σε περισσότερες από μία συσκευές. Ο συνδυασμός αυτών των χαρακτηριστικών απαλλάσσει το χρήστη από πολλές εργασίες που πρέπει να πραγματοποιεί σε ότι αφορά την διαχείριση του αποθηκευτικού χώρου.

Λογισμικό υποστήριξης αδόμητης επικοινωνίας

Η λειτουργία του συστήματος προϋποθέτει την δυνατότητα ευκαιριακής επικοινωνίας μεταξύ συσκευών, κάτι που συνεπάγεται σημαντικό προγραμματιστικό κόστος λόγω της αυτογενούς υψηλής δυναμικότητας που εμπεριέχουν τα αδόμητα δίκτυα. Αναπτύξαμε λοιπόν λογισμικό υποστήριξης που παρέχει υπηρεσίες για τη διευκόλυνση της ανάπτυξης εφαρμογών σε τέτοιο περιβάλλον, το οποίο παρέχει:

ανακάλυψη υπηρεσιών που υπάρχουν στο αδόμητο δίκτυο στο οποίο συμμετέχει μια συσκευή, ευριστική μέθοδο επιλογής υπηρεσιών με βάση την αναμενόμενη διαθεσιμότητα της συσκευής που τις παρέχει, επικοινωνία μεταξύ εφαρμογών σε διαφορετικές συσκευές και διαμοιρασμό πληροφοριών για τις επικρατούσες συνθήκες στο χώρο όπου συνευρίσκονται πολλές συσκευές.

Για την υλοποίηση όλων των προαναφερθέντων λειτουργιών, το λογισμικό υποστήριξης εκπέμπει περιοδικά ένα "*μήνυμα παρουσίας*" στο αδόμητο δίκτυο, το οποίο χρησιμοποιείται με πολλαπλούς τρόπους ώστε να παρέχονται όλες οι παραπάνω υπηρεσίες. Η ορθότητα των αλγορίθμων που χρησιμοποιούνται δεν επηρεάζεται από τον ρυθμό εκπομπής αυτών των "μηνυμάτων παρουσίας". Μάλιστα ο ρυθμός αυτός μεταβάλλεται ανάλογα με το φόρτο που δημιουργούν οι εφαρμογές που χρησιμοποιούν τις υπηρεσίες ώστε να εξυπηρετούνται οι ανάγκες τους. Συγκεκριμένα, υπάρχουν τρεις ρυθμοί εκπομπής:

- **Αργός**: Αυτός ο ρυθμός χρησιμοποιείται όταν δεν υπάρχει καμιά εργασία προς εξυπηρέτηση από τις εφαρμογές που χρησιμοποιούν το σύστημα.
- **Κανονικός**: Αυτός ο ρυθμός χρησιμοποιείται όταν υπάρχουν εργασίες προς εξυπηρέτηση από τις εφαρμογές που χρησιμοποιούν το σύστημα, οι οποίες όμως *δεν* είναι καινούριες και έχει υπάρξει ήδη επεξεργασία για αυτές.
- **Γρήγορος**: Αυτός ο ρυθμός χρησιμοποιείται όταν υπάρχουν εργασίες προς εξυπηρέτηση από τις εφαρμογές που χρησιμοποιούν το σύστημα, οι οποίες είναι καινούριες και δεν έχουν επεξεργαστεί καθόλου.

Αρχικά το σύστημα χρησιμοποιεί τον αργό ρυθμό, μεταβαίνοντας στον γρήγορο όποτε κάποια εφαρμογή δημιουργήσει μια νέα εργασία. Όταν όλες οι υπάρχουσες εργασίες έχουν επεξεργαστεί, χρησιμοποιείται ο κανονικός ρυθμός, επιστρέφοντας στον γρήγορο κάθε φορά που δημιουργείται νέα εργασία. Αν κάποια στιγμή αφαιρεθούν όλες οι εργασίες, το σύστημα επιστρέφει στον αργό ρυθμό.

Ευριστική επιλογή υπηρεσιών στο αδόμητο δίκτυο με βάση την αναμενόμενη διαθεσιμότητα

Η ευκαιριακή επικοινωνία εφαρμογών με υπηρεσίες σε ένα αδόμητο δίκτυο είναι συνήθης πρακτική. Σημαντική παράμετρος για την επιλογή της συσκευής με την οποία μια εφαρμογή θα επικοινωνήσει (όταν υπάρχουν περισσότερες επιλογές) ή ακόμη και για το αν θα επιχειρηθεί καν επικοινωνία, αποτελεί η αναμενόμενη διαθεσιμότητα της άλλης συσκευής. Ένας μηχανισμός που παρέχει πληροφορίες για το ιστορικό συνεύρεσης συσκευών μπορεί να βοηθήσει στη λήψη της σχετικής απόφασης.

Το λογισμικό υποστήριξης χρησιμοποιεί τα περιοδικά μηνύματα παρουσίας για να καταγράφει το πλήθος και τη μέση διάρκεια των συνευρέσεων. Για κάθε συσκευή που συναντάται, παρακολουθείται η διάρκεια της συνεύρεσης (που προσδιορίζεται από τη διάρκεια λήψης μηνυμάτων παρουσίας) και χρησιμοποιείται για να υπολογιστεί ο τρέχων μέσος όρος διάρκειας των συνευρέσεων συνολικά, καθώς και το πλήθος τους. Η χρονική περίοδος (παράθυρο) για την οποία καταγράφονται στοιχεία είναι πεπερασμένη και σε τακτά χρονικά διαστήματα γίνεται ομαλοποίηση των μετρήσεων υποθέτοντας μια κανονική κατανομή των συνευρέσεων στο χρόνο. Με βάση τις πληροφορίες αυτές μια εφαρμογή μπορεί να εκτιμά την αναμενόμενη μελλοντική διάρκεια συνεύρεσης με μια άλλη συσκευή, ως το γινόμενο του πλήθους των συνευρέσεων και της μέσης διάρκειας αυτών στο τρέχον παράθυρο.

Ανακάλυψη υπηρεσιών

Το λογισμικό υποστήριξης παρέχει τη δυνατότητα ανακάλυψης της ύπαρξης κάποιας υπηρεσίας στο αδόμητο δίκτυο, καθώς και τη διεύθυνση στην οποία μπορεί μια εφαρμογή να επικοινωνήσει με αυτήν. Οι υπηρεσίες θεωρείται ότι χρησιμοποιούν προσυμφωνημένα αλφαριθμητικά ονόματα με βάση τα οποία γίνεται η αναζήτηση.

Η υλοποίηση αυτής της λειτουργικότητας γίνεται με την επισύναψη πληροφοριών αναζήτησης / διαφήμισης υπηρεσιών στα μηνύματα παρουσίας. Συγκεκριμένα, ανάλογα με τα αιτήματα που δέχεται, το λογισμικό υποστήριξης μπορεί είτε να εκπέμπει μηνύματα παρουσίας με συννημένο ερώτημα αναζήτησης κάποιας υπηρεσίας (αίτημα εφαρμογής), είτε να εκπέμπει μηνύματα παρουσίας με συννημένη διαφήμιση της διεύθυνση μιας υπηρεσίας (αίτημα της υπηρεσίας). Εναλλακτικά, οι εφαρμογές μπορούν να περιοριστούν στο να ζητήσουν από το λογισμικό αναζήτησης να καταγράφει τη διεύθυνση υπηρεσιών κάποιου τύπου όταν λαμβάνεται σχετική διαφήμιση, χωρίς όμως να αποστέλλονται αναζητήσεις. Παρομοίως οι υπηρεσίες μπορούν να ζητήσουν από το λογισμικό υποστήριξης να εκπέμπει διαφημίσεις για αυτές μόνο εφόσον λάβει σχετικό μήνυμα αναζήτησης. Με βάση αυτά υπάρχουν οι εξής δύο μέθοδοι για την ανακάλυψη υπηρεσιών:

Μέθοδος	***Αίτημα Αναζήτησης***	***Αίτημα διαφήμισης***
Ενεργή αναζήτηση	Περιοδική εκπομπή	Εκπομπή μόνο κατά τη λήψη σχετικής αναζήτησης ή περιοδική εκπομπή
Παθητική αναζήτηση	Απλή καταγραφή χωρίς εκπομπή αναζήτησης	Περιοδική εκπομπή

Επικοινωνία

Το λογισμικό υποστήριξης επιτρέπει την δημιουργία συνδέσεων δύο σημείων μέσω των οποίων μπορούν να μεταδίδονται αμφίδρομα ροές δεδομένων, για τις οποίες παρέχεται εγγύηση παράδοσης με ορθή σειρά. Για την δημιουργία μιας σύνδεσης πρέπει να χρησιμοποιηθεί ο μηχανισμός ανακάλυψης ώστε να εντοπιστεί η διεύθυνση με την οποία είναι επιθυμητή σύνδεση.

Επιπλέον, παρέχεται υποστήριξη για την επικοινωνία με υπηρεσίες στο διαδίκτυο. Για τον σκοπό αυτό αναπτύχθηκε μια υπηρεσία *πύλης διαδικτύου*, η οποία μπορεί να διαβιβάζει τις ροές που μεταδίδονται μέσω μιας σύνδεσης σε υπηρεσίες στο διαδίκτυο. Η υπηρεσία πύλης εγκαθιστά μια σύνδεση διαδικτύου με την υπηρεσία που επιθυμεί η εφαρμογή πελάτης και στη συνέχεια αποστέλλει την εισερχόμενη από το πελάτη ροή στην υπηρεσία στο διαδίκτυο, ενώ παρομοίως αποστέλλει στον πελάτη τη ροή δεδομένων που λαμβάνει από την υπηρεσία στο διαδίκτυο. Με τον τρόπο αυτό, μπορούν συσκευές που ανακαλύπτουν την υπηρεσία πύλης διαδικτύου πλησίον τους να συνδέονται με αυτήν και να επικοινωνούν μέσω αυτής με οποιαδήποτε υπηρεσία στο διαδίκτυο.

Διαμοιρασμός πληροφοριών επικρατουσών συνθηκών

Το λογισμικό υποστήριξης έχει τη δυνατότητα να διαμοιράζει τις πληροφορίες που διαθέτει μια συσκευή για τις επικρατούσες συνθήκες, στο σύνολο των συσκευών που συνευρίσκονται πλησίον της. Έτσι, μπορεί κάθε συσκευή να εκμεταλλεύεται τις

δυνατότητες των υπολοίπων να χαρακτηρίζουν το περιβάλλον γύρω τους μέσω πληροφοριών που λαμβάνουν από ενσωματωμένους αισθητήρες ή εξάγουν από την αλληλεπίδραση με το χρήστη.

Ο μηχανισμός διαμοιρασμού στηρίζεται σε έναν πίνακα του λογισμικού υποστήριξης στον οποίο οι εφαρμογές καταγράφουν τις πληροφορίες για επικρατούσες συνθήκες που διαθέτουν. Το λογισμικό υποστήριξης επισυνάπτει τις πληροφορίες αυτές στα μηνύματα παρουσίας που εκπέμπει η συσκευή ώστε να μεταδοθούν στις συσκευές που βρίσκονται σε εμβέλεια. Παράλληλα, το λογισμικό υποστήριξης εμπλουτίζει τον πίνακα αυτό με πληροφορίες που λαμβάνει από άλλες κοντινές συσκευές, καθιστώντας αυτές διαθέσιμες και στις τοπικές εφαρμογές.

Αυτοματοποίηση διαχείρισης δεδομένων

Η αυτοματοποίηση της διαχείρισης δεδομένων επιτυγχάνεται με τη δράση του συστήματος σε δύο άξονες: τη συνεργασία των συσκευών με κατάλληλες υπηρεσίες υποδομής και τη συνεργασία των συσκευών μεταξύ τους.

Υπηρεσίες υποδομής

Κάθε χρήστης διαθέτει ένα προσωπικό χώρο αποθήκευσης στην υποδομή, με τον οποίο συσχετίζει τις συσκευές που του ανήκουν. Κατά την συσχέτιση οι συσκευές αποκτούν καθολικά μοναδικά αναγνωριστικά με τα οποία μπορεί κανείς να αναφέρεται μονοσήμαντα σε αυτές. Επιπλέον, αποκτούν πιστοποιητικά με τα οποία μπορούν να αποδεικνύουν η μία στην άλλη ότι ανήκουν στον ίδιο χρήστη όταν συναντώνται σε ένα αδόμητο δίκτυο, χωρίς να απαιτείται επικοινωνία με την υπηρεσία υποδομής. Τέλος, τα αναγνωριστικά αυτά χρησιμοποιούνται από τις συσκευές ώστε να μπορούν αυτόνομα να δημιουργούν καθολικά μοναδικά αναγνωριστικά για τα αρχεία που δημιουργεί ο χρήστης σε αυτές. Τα αρχεία του συστήματος μπορούν να έχουν απεριόριστο αριθμό συννημένων πληροφοριών, οι οποίες χρησιμοποιούνται για την ταξινόμηση τους. Μέσω των συννημένων αυτών πληροφοριών το σύστημα – μεαξύ άλλων – διατηρεί και ιστορικό για τις διάφορες εκδόσεις αρχείων τα οποία υπέστησαν μεταβολές από το χρήστη.

Οι εγγεγραμμένες συσκευές επικοινωνούν περιοδικά με την υπηρεσία αποθήκευσης υποδομής ώστε να μεταφέρουν αρχεία από και προς αυτή, σύμφωνα με τις επιθυμίες του χρήστη και τις ανάγκες των εφαρμογών. Όσον αφορά την μεταφορά αρχείων από τις συσκευές προς την υποδομή, το σύστημα υλοποιεί "βαθιά *πλήρη αρχειοθέτηση*" συλλέγοντας *όλα* τα αρχεία που δημιουργεί ο χρήστης στην υπηρεσία υποδομής. Για το λόγο αυτό, όποτε οι συσκευές επικοινωνούν με την υπηρεσία, μεταφέρουν σε αυτή *όλα* τα νέα αρχεία που έχουν δημιουργηθεί σε αυτές. Για τη μεταφορά αρχείων από την υπηρεσία υποδομής σε κάποια συσκευή, θα πρέπει κάποια εφαρμογή να έχει υποβάλλει σχετικό αίτημα. Η υπηρεσία υποδομής ενημερώνει τη συσκευή που πρέπει να παραλάβει το αρχείο όταν αυτή επικοινωνήσει και στη συνέχεια πραγματοποιεί τη μεταφορά. Οι αιτήσεις αυτές μπορούν να αφορούν συγκεκριμένο χρονικό διάστημα για το οποίο το αρχείο θα πρέπει να βρίσκεται στη συσκευή, καθώς και να ορίζουν ότι η συσκευή θα πρέπει να ενημερώνεται και να ανακτά νεότερες εκδόσεις του αρχείου ώστε να διαθέτει πάντα την τελευταία έκδοση αυτού.

Ομαδική λειτουργία συσκευών

Η συσκευές επικοινωνούν μεταξύ τους για την δημιουργία αντιγράφων σημαντικών αρχείων (ώστε να αυξηθεί η διαθεσιμότητά τους) ή για τη μετακίνηση

αρχείων από μια συσκευή σε άλλη (ώστε να αποσυμφορηθεί κάποια συσκευή στην οποία εξαντλείται ο διαθέσιμος αποθηκευτικός χώρος). Η αντιγραφή αρχείων πραγματοποιείται κατόπιν υπόδειξης κάποιας εφαρμογής ότι το αρχείο είναι σημαντικό (τέτοια υπόδειξη δημιουργείται αυτόματα από το σύστημα για αρχεία τα οποία ελήφθησαν από την υπηρεσία υποδομής και για τα οποία είναι προγραμματισμένη συγκεκριμένη χρονική περίοδος παρουσίας). Οι μεταφορές για την εξοικονόμηση χώρου καθοδηγούνται από δύο παραμέτρους της συσκευής: τον ελάχιστο διαθέσιμο χώρο και τον επιθυμητό διαθέσιμο χώρο. Όποτε ο διαθέσιμος χώρος είναι λιγότερος από τον ελάχιστο επιτρεπτό, εκκινεί η διαδικασία αποσυμφόρησης της συσκευής, η οποία περιλαμβάνει και μεταφορά αρχείων σε άλλες συσκευές πλησίον της, και η οποία σταματά μόλις ο διαθέσιμος χώρος ανέλθει στο επιθυμητό επίπεδο.

Για την υποστήριξη της ταξινόμησης και αναζήτησης αρχείων οι συσκευές χρησιμοποιούν τις διαμοιραζόμενες πληροφορίες επικρατουσών συνθηκών για την προσθήκη σχετικών συννημένων πληροφοριών στα αρχεία που δημιουργούνται. Με τον τρόπο αυτό, ο χρήστης μπορεί να αναζητά αρχεία με βάση τις συνθήκες που επικρατούσαν όταν αυτά δημιουργήθηκαν.

Τέλος, οι συσκευές επιτρέπουν την απομακρυσμένη πρόσβαση σε αρχεία σε μια συσκευή από εφαρμογές σε κάποια άλλη. Σε περίπτωση που το αρχείο είναι διαθέσιμο σε περισσότερες από μια συσκευές, είναι δυνατή η μετάβαση από τη μια στην άλλη σε περίπτωση που υπάρξει αστοχία της συσκευής από την οποία προσπελάσεται το αρχείο.

OmniStore: Mechanisms for automating data management in a personal system comprising several portable devices

Alexandros Karypidis

23 February, 2007

Contents

List of Figures

List of Tables

Listings

Acknowledgments

I would like to thank the people who kept me company in the long hours spent working: Thanasis Fevgas, George Giannikis, Manos Koutsoubelias, Marios Pitikakis and George Vasilakis. Special thanks go to my advisor, Spyros Lalis, whose help in the process of writing this dissertation was invaluable.

I am grateful to my parents and sister, who have forever supported me in all my quests. Last but not least, I will not forget how supportive Jim Syrivelis and Catherine Kazantzi were during this stressful period.

P.S. My friends at the department insist that I explicitly state: I thank the J.

Abstract

As the volume of data people generate, collect and use continuously grows, so do the burdens associated with conventional file management, which typically requires considerable user involvement. The large number of portable and embedded devices we carry with us, merely aggravate the situation. The scientific community has long stipulated that the computing systems that will pervade the human habitat will be designed to assist people in, rather than distract them from, their activities. This work contributes towards achieving this objective by reducing the complexity of file management in a personal computing environment comprising several portable devices with ad-hoc networking capability.

To this end, we have designed and implemented OmniStore, a system that combines portable devices and infrastructure-based services to relieve the user from explicit and time consuming file management tasks. In Omnistore, all versions of all files created on any device are incrementally forwarded to a repository. Conversely, a file (and subsequent versions thereof) may be copied from the repository to any device, in an asynchronous and flexible way. Furthermore, portable devices collaborate with each other in the background to replicate files for increased availability, to migrate files for storage reclamation, and to provide transparent and fault-tolerant remote file access within the personal area network. New files are also annotated with context information generated from nearby devices, enabling their flexible organization and lookup. Notably, the mechanisms and protocols of OmniStore have been designed taking into account the intermittent nature of mobile and ad-hoc communication.

As a basis of our system, we have developed a runtime environment which provides core facilities, such as service discovery, device co-location statistics, remote communication and context aggregation, paying special attention to the characteristics of personal area networks. The file management functionality is implemented on top of this runtime and is made available to the application programmer via a library that augments the conventional file system API with the necessary primitives.

Chapter 1

Introduction

Since its inception and through the present day, we have systematically broadened our use of computing technology by applying it to an increasing number of domains. In addition to its steadily growing scope, it deployment has also expanded in depth: we are gradually adapting our lifestyle to suit computer intricacies. This realization became cause for legitimate concern more than ten years ago, inspiring the *Ubiquitous Computing* vision [Wei91]. The scientific community then stipulated that, even though computing elements will unavoidably proliferate in human habitats, they would be designed to serve people by unobtrusively assisting their activities, otherwise constraining themselves to the background of human attention.

Indeed, by continuously transcending new boundaries, computing is gradually establishing its presence in our daily routine. Meanwhile, the ubiquitous computing concept has affected all areas of computer science, such as: hardware design, operating systems, networking, middleware, user interfaces, etc. As numerous objects are augmented with computing and (wireless) networking functionality,

new applications become possible, while at the same time old applications have to be rethought. Personal data management, one of the typical uses of computing technology, is among the areas which need to be revisited.

1.1 The information explosion

With computing elements permeating all aspects of our lives, the volume of data that users must typically manage is escalating. People now employ the use of digital information in a growing number of areas. For example, compared to one or two decades ago, a lot of paperwork has been replaced by electronic documents, both in business and in government. Books, music, photographs and videos are now digitally stored.

In addition to the expanded use of digital storage formats and mediums, the rise in the amount of data managed by a person may also be attributed to the fact that data generation and collection now starts to occur at small ages. Consider for example that, submitting hand-written essays is now outdated even in primary education, as children quickly catch on to word processing software and prefer to type their essays on computers. In fact, nobody has faced the personal data management problem to its full extent so far, as people accompanied by a lifetime of all-digital data do not exist yet: even early-adopters of personal computers are now in their fourties. Moreover, they have created and collected relatively little data, as computing was not pervasively deployed until recently.

Another contributing factor to the bulk of data managed per person is the ease with which we generate it nowadays, given the multitude of digital appliances that are available for such purposes. PDAs, mobile phones, digital cameras, voice-

memo recorders and other such devices are casually carried and used to create data, even when one is on the move. As a result, data generating activities in human lives have increased substantially.

In a recent study [LV03] it was determined that around 800MB of information were produced per person in the year 2003. The study further estimates [Var05] that well over 90% of information currently produced is created in a digital format, anticipating that this percentage will keep increasing in the future. Citing the findings of this report, several storage-organization and information-retrieval challenges were included in a list of meritable long-term research goals [Gra03] for the future.

1.2 The personal data management problem

The growing number of devices via which data is collected with or distributed to for use, in combination with increased data volumes, is causing the amount of effort required for personal data management to reach alarming levels. This problem is quickly becoming a considerable burden, indicating that storage management systems must be adapted to support this new state of affairs.

Specifically, the activities which personal data management entails may be classified into two categories:

- **Organization tasks**: the annotation of files with names and other meta-information, the grouping of files in logical structures (e.g. directory hierarchies) for convenient review and access, etc.
- **Placement tasks**: the transfer of files from one location to another, the

replication of files on multiple storage mediums for availability and fault-tolerance, the removal of files from a storage medium in order to make space, etc.

We next review how each category of storage-related activities is affected by the increasing volumes of data which one must manage.

1.2.1 Organization

File organization is one of the most common – but also fairly awkward – tasks which people must deal with when using personal computing systems. Typically, it is addressed by assigning human-readable names to files and arranging them in hierarchical directory structures that are formed according to the envisaged information access pattern. While this approach works well for most conventional data processing scenarios, it becomes less appropriate when files are being generated via personal mobile devices such as cameras, memo recorders, phones and music players. In this case, the number of files generated can be very large and finding good names and directory structures for storing them becomes increasingly hard; even more so if this should be done on the move. Aggravating the situation is the fact that persistent memory prices are dropping to levels which render efficient use of storage space obsolete; the storage capacity available makes it possible to simply keep *all* data we create. This trend intensifies the severity of the file organization problem, keeping the concept of a personal "memex" [Bus96] elusive.

An accumulating body of work that addresses this issue [GJSO91, GM99, SKW+02, MTX03] is converging to the solution of using unrestricted metadata annotation as a better alternative to traditional hierarchical file systems. In such

so-called *semantic* file systems, applications and users may attach any number and type of annotations to files. File lookup operations are then expressed as queries with respect to these annotations, whereas browsing is supported through flexible virtual directory hierarchies that are dynamically generated using data clustering techniques. Due to the inherent flexibility of metadata-based browsing, time is admittedly better invested in annotating files as opposed to trying to define fixed directory structures, which most likely will be changed (more than once) anyway.

There is no free lunch though. While the user is relieved from having to define and manage file names and directory structures, semantic file systems require files to be decorated with annotations. Still, the process of annotating files in sufficient detail as to enable efficient and flexible lookup can be equally cumbersome and time consuming. For this reason, the research community is now turning its attention towards finding ways to facilitate – or even automate – this process [SG03]. A significant challenge is therefore to allow for seamless organization of files through computer-generated meta-data annotations, not just on the desktop, but also on mobile devices such as cameras, phones, PDAs, etc.

1.2.2 Placement

Given efficient means for organizing and accessing files, the next major hurdle involves transferring files from one storage medium to another. Such transfers may be desired in several cases. For example: to create copies for increased availability (replication), to backup files (archival), to increase access locality (caching), or to move data from a heavily-used storage medium to another in order to make space (load distribution).

Again, users are responsible for performing such awkward tasks. Through file managers they explicitly issue commands to create copies of files on portable media and subsequently update their original location by copying them back, delete files when a storage medium is full, etc. To make things worse, management tasks must usually be performed in synchronous fashion and in real-time. For example, the user is interrupted when storage is depleted and is required to create free space at that instant in order to proceed. Similarly, copying a file to a portable storage medium is a task which requires the user to combine: having the medium handy, issuing the transfer command and waiting until the copy operation is complete.

1.3 Distraction-free personal data management

The extensive adoption of computing in various devices has dispersed the points of data production and consumption across numerous storage elements that may reside in different locations, or be casually carried or worn by the user. As a consequence, data management tasks are becoming increasingly distributed in space and time. Consequently, we must perform such operations on several storage mediums, while on the move, struggling with the limited user interaction facilities that are in place at the time. This is hardly convenient, or unobtrusive. The essence of ubiquitous computing however, lies in allowing users to attend to whatever tasks must be accomplished, without diverting their attention to the technicalities involved in operating computers. In other words, people should be assisted in achieving their goals, rather than be distracted by computing systems.

Our thesis is that *personal data management can be automated so as to minimize its reliance on user input.*

Chapter 2

Data management in ubiquitous computing environments

Technological advances have allowed an increasing number of objects to be augmented with computing resources. A wealth of devices is now available for assisting people in performing various tasks, such as: communicating with others using mobile phones, navigating assisted by positioning systems, entertaining themselves with music players or gaming consoles, taking photographs with digital cameras, etc. We point out that, in most cases, computing-assisted activities are accompanied by the production and / or consumption of data; Table 2.1 gives examples of the data-related actions made possible with typical devices of our time.

This work attempts to tackle the issue of managing personal data[1] in ubiquitous computing environments, where people generate or consume data in a multitude of ways. In this chapter, we review how storage elements are utilized, con-

[1] System files and binaries our outside the scope of this work.

Device	*Data production*	*Data consumption*
Mobile Phone	Record voice-memos Record phone-calls Take photographs Create video clips	Playback recordings Review photographs and clips
Digital camera	Take photographs Create video clips	Review photographs and clips
Music player	Record radio shows Record voice-memos	Playback music Playback recordings
Game console	-	Play games
e-book reader	-	Read books
Navigator	Record route	Plot course

Table 2.1: Data-related activies of typical portable devices.

sidering their characteristics, discussing which of these are desirable or not and pinpointing how user experience can be improved. We thus outline the requirements of a storage management system in line with the ubiquitous computing vision.

2.1 Storage element characteristics and challenges

2.1.1 A plethora of portable devices with storage

An important recent development is that the number of devices with storage has increased significantly. Consequently, data organization and placement becomes more complicated, requiring increased effort from the user both mentally and in volume of work. Most of these newly introduced storage elements reside in portable, special-purpose gadgets: digital cameras, music players, gaming consoles, voice-memo recorders and mobile phones are just some examples. One should also consider the novel devices being introduced by researchers, such as

the Personal Server [WPD+02], or even every-day objects augmented with computing capability (wallets [LKSS03], watches [NKR+02], etc.). This is a major deviation from prior computing environments where people handled data using "real" computers, such as personal workstations and laptops.

Another aspect which needs to be taken into account is that, given the multitude and diversity of devices with storage, several of them being wearable or portable, it is quite likely for one to carry more than one such device. Furthermore, it is common to change the set of devices accompanying the mobile user, as it is easy to pick up or drop off such a device (which may even happen inadvertently). Effort must be put into dealing with such user activity by arranging device contents to efficiently use storage space.

2.1.2 Ad-hoc networking potential

The increasing adoption of ad-hoc wireless networking technologies in portable devices is changing the landscape of data management. Until recently, portable storage elements typically had to be placed into a suitable host device, or connected with a computer, in order for their contents to be accessible. Nowadays, by exploiting ad-hoc networking capabilities, the data on any device can be made available to all applications in nearby devices. Each storage element can, in principle, act as a network-accessible storage server.

There is a more important aspect to this networking ability though: a portable storage element which can initiate communication with other devices is no longer constrained to a passive role, but is free to observe its environment autonomously and interact with other elements and services in its vicinity. We can thus de-

part from the traditional "dumb" storage medium model and move on to the *active store* model, in which devices can discover and communicate among themselves or with servers residing on the Internet. By embedding appropriate logic in portable devices, we can turn them into smart, collaborating and self-organizing storage elements. The potential of such an approach has not yet been investigated.

2.1.3 User interaction restrictions

As the number of computing elements in typical human environments grows, the various processes requiring user input to proceed must compete for the privilege of user attention. However, interaction with computing devices is more often a source of distraction for the user and is in direct opposition with the ubiquitous computing vision: human attention should be treated as a scarce resource [CGS+02].

We can bring storage management close to our goal by incorporating logic that initiates storage maintenance tasks without user intervention. In addition, chances of obtaining implicit input from the user (e.g. through contextual means [Dey01]), must be exploited, allowing for interaction through "natural" user activity rather than explicit interaction with a computing system. Furthermore, whenever user involvement can not be avoided, measures should be taken to reduce the impact of requiring human attention. Specifically, the situation can be greatly improved by making it possible for interaction to occur when it is convenient for *the user*. In other words, computing elements should avoid seeking out the user to *request* input, but rather be designed so as to be ready to *receive* input whenever interaction is initiated by the user.

We emphasize that, in the case of portable computing elements, the user is further handicapped by the limited interaction facilities of such devices. This is due to their small form-factor and the fact that they are used while the user is on the move. Therefore, the importance of making user input requirements minimal and convenient can not be exaggerated. Equally useful is the ability to provide input from alternative, more convenient devices.

2.2 Usage review of storage on portable devices

With these characteristics of storage elements in ubiquitous computing environments in mind, we next review the typical roles of portable devices with storage: as producers, consumers and couriers of files. Each role is discussed in the following, giving examples for every case. It should be noted that these are not mutually exclusive, but rather outline the primary usage patterns exhibited. As such, they can provide insight and help derive the requirements for a storage system that is adapted to the needs of users.

2.2.1 Producers

Producers create files. Devices assuming this role are capable of producing certain types of data (pertaining to their application domain) and are especially designed for achieving that purpose. The general usage pattern in this case is a flow of this data from the portable device to the infrastructure, where it is archived for later review and / or processing.

A common example are digital cameras, with which one takes photographs that are later moved to personal computers in order to archive them and release

storage space on the device. Similarly, digital recorders – often used by interviewers, investigators and other people which must meticulously collect information while on the move – produce sound files which serve as audible notes that may be played back later. Mobile phones with call-recording capability can also serve this purpose by recording conversations in which the other party is telling us something we would like to remember.

2.2.2 Consumers

Consumers are devices on which files are accessed. In this case, the device is designed to provide some kind of service based on the data it holds. The usage pattern is to select the desired data and place it on the special-purpose device.

The most typical representative of this class is a music player. Applications for managing music collections allow people to purchase music from Internet stores and create playlists that are transferred (along with the respective music files) on the device. In this manner, the desired music selection is made available for listening on the move. Other examples of the data consumer type are portable gaming devices and e-book readers. These employ a similar model in which data (games and books respectively) are downloaded from digital stores and placed on a device to be used on the move.

2.2.3 Couriers

Couriers transfer files from one location to another. In contrast to producers and consumers, this type of usage is totally unrelated to the processing capabilities of a portable device. Any device with storage can be exploited in this manner. In this

case, one puts selected files on the device to be used as a reliable provider of the data, which is then carried along to make sure the data will be nearby at the time when it is needed. This may occur whenever a user feels that a file is so important as to warrant having it physically available on a carried device, during important time / location frames.

For example, prior to going to a conference, a user may place her presentation on her watch, to avoid any access problems. Even if the file were accessible through the Internet, a portable can be useful in case of rare (but always possible) network and server failures, or when network security restrictions at the user's (or the visiting) site prohibit access to the remote server where the file is stored.

2.3 Storage in ubiquitous computing environments

Based on the characteristics of typical storage elements in the ubiquitous computing era, outlined in Section 2.1, along with the typical usage patterns exhibited, discussed in Section 2.2, we designed OmniStore, a storage management system for personal computing.

OmniStore facilitates personal data management by addressing the technicalities forced upon users with regard to data organization and placement activities. It automates most of these tasks, both in the case of infrastructure and portable device storage. Specifically, it employs opportunistic interaction among portable devices to achieve collaborative storage management among them, also coordinating their contents in reference to an infrastructure storage service that belongs to the user. To provide this functionality, we have developed both a set of infrastructure services required to support devices in their storage-related operations,

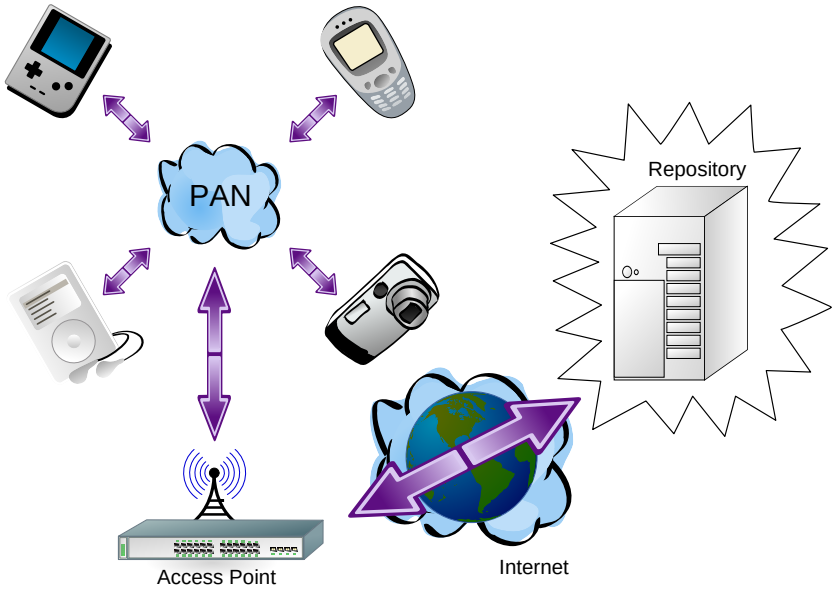


Figure 2.1: OmniStore's design

as well as the necessary runtime mechanisms required to support efficient ad-hoc interaction between wearable and portable devices, when disconnected from the infrastructure.

Our system forwards all files generated on portable devices to a storage service referred to as the *repository*. The newly created files are automatically decorated with semantic annotations derived from the context-sensing capabilities of devices within range, which are used for organizational purposes. The inverse data flow is also supported, by allowing the user and applications to select files in the repository and request that they be cached on portable devices of their choice.

Given that portable devices may also interact with each other, we introduce collaborative behavior among them. In addition to traditional access to files on any device, from applications anywhere in the personal area network (PAN), we implement further functionality such as "off-loading"' of files from one device to another to distribute storage load, or replication of files on several devices to

increase availability and reliability. With this design the various portable caches are merged into a *single* collective portable cache that accompanies the user.

This combination of features, as is discussed in detail in Chapter 5.1, addresses the requirements analyzed in this Section, allowing for convenient personal data management with significantly reduced human effort. We believe that our approach contributes towards the realization of the ubiquitous computing vision.

Chapter 3

Core runtime mechanisms

Portable and wearable devices with significant processing and networking capability are becoming increasingly common. Their growing use is altering the conventional personal computing paradigm. What we usually refer to as the "personal computer" becomes a collection of separate and possibly autonomous elements that co-operate with each other without relying on external infrastructure or a prearranged setup. As we move beyond the physical – and mental – boundaries of the desktop, one of the key challenges becomes to combine the devices and artifacts that are available, whether carried by people or situated in a given environment, to accommodate personal computing.

Functions and applications are distributed on different platforms, that can be widely heterogeneous in terms of computing resources and user interaction capability. Also, the system configuration can change several times during application execution, due to devices being switched on and off, or moved into and out of range. In order to cope with this dynamic nature, suitable supporting mechanisms are required.

The results of our initial work in this area can be found in [LKS05, LKSS03], which discusses a runtime system designed to support application execution on multi-device platforms that are formed in ad-hoc fashion. Since OmniStore targets such environments, we went on to evolve those mechanisms, adding important facilities that better support opportunistic collaboration among devices. In this Chapter, we present the functionality offered by our refined version the runtime: discovery of services in the PAN, intelligent selection among several possible matching services, communication among nodes and acquisition of context information.

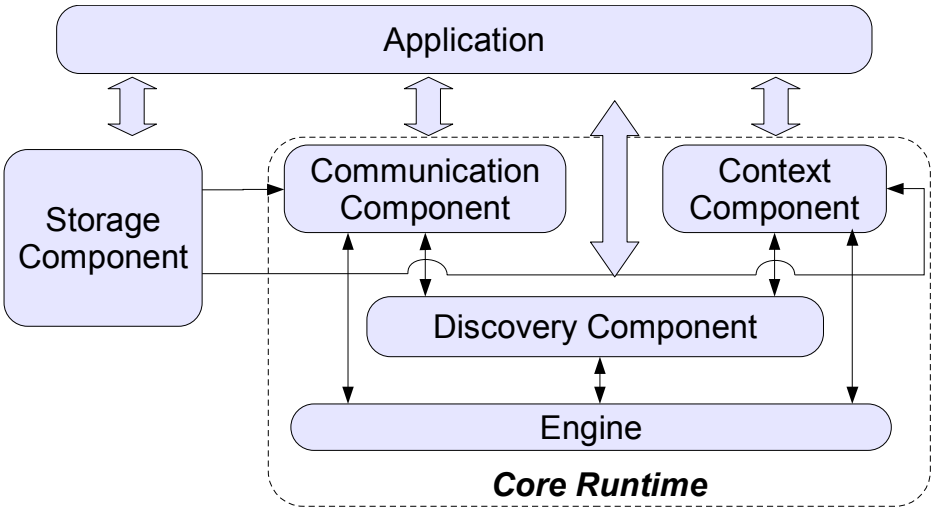


Figure 3.1: Core runtime components

Figure 3.1, shows the design of our underlying runtime. Our implementation is written in the Java language. However, we restricted ourselves to the use of only certain basic classes from the `java.lang` package of the J2SE (Java 2 Standard Edition) libraries. Therefore, the runtime can execute on virtually any Java virtual machine, even on restricted J2ME (Java 2 Micro Edition) implementations. The binary size of the runtime is about 350KB.

It should be noted that all the algorithms and protocols discussed in the following are designed assuming the existence of a broadcast primitive for sending a data packet to all nodes in the network. At the lowest layer of the communication component lies an abstract class providing this broadcast primitive. To create a network driver for a specific type of networking technology, one must implement this function in a subclass of the abstract network driver, along with a means to deliver received network packets to the network driver for processing. Also, a class representing the device address must be implemented. This design makes it easy to support most types of wireless networks (e.g. ZigBee and WLAN). Furthermore, it is easy to create drivers for wired networks (e.g. using multicast UDP/IP or Ethernet broadcasts) for testing purposes.

3.1 Beaconing: the runtime's heartbeat

Broadcasts are an essential tool for implementing functionality targeting ad-hoc networking environments. For example, source routing algorithms [JM96, HJ04] use broadcasts for route discovery, whereas their distance-vector counterparts [PB94, CBR04] rely on them even more heavily, as they perform them periodically in order to keep routing tables up-to-date. Most service discovery mechanisms also typically base their operation on broadcasts to varying degrees [Cha06, Yau03]. Evidently, cross-layered approaches which exploit broadcasts for multiple purposes can be very beneficial [VRdL05].

Our system is no exception. Specifically, it uses broadcasts for: (a) maintaining device co-location information which is used to guide service selection, (b) discovering services present in the network, and (c) disseminating context in-

formation. In the interest of efficiency, we have designed all of these functions around a single periodic packet broadcast (beaconing). The algorithms are designed to work orthogonally, with their correctness being immune to the beaconing frequency. In fact, this can be adjusted at runtime, to facilitate the operational requirements of services and applications executing on top of our runtime. The beaconing rate varies in order to accommodate higher-layer tasks, using one of three different intervals among consecutive beacons: *idle*, *normal* and *fast*:

Idle This is the lowest rate at which a device will emit consecutive beacons when no tasks exist.

Normal This is the interval among consecutive beacons when pending tasks exist, none of which was added within the last processing iteration over such tasks.

Fast This is the interval among consecutive beacons when a new task has just been added. The new task is normally processed immediately after its addition. The fast beaconing cycle persists until the new task is processed for a second time, at which point the system falls back to *normal* beaconing rate.

Figure 3.2 depicts the transition diagram among these states. The reader will be referred to it in the following sections, where each of these mechanisms is discussed in detail.

3.2 Maintaining co-location history for service selection

Ad-hoc computing systems rely on spontaneous interaction among computing elements which collaborate in order to achieve some goal. The first requirement

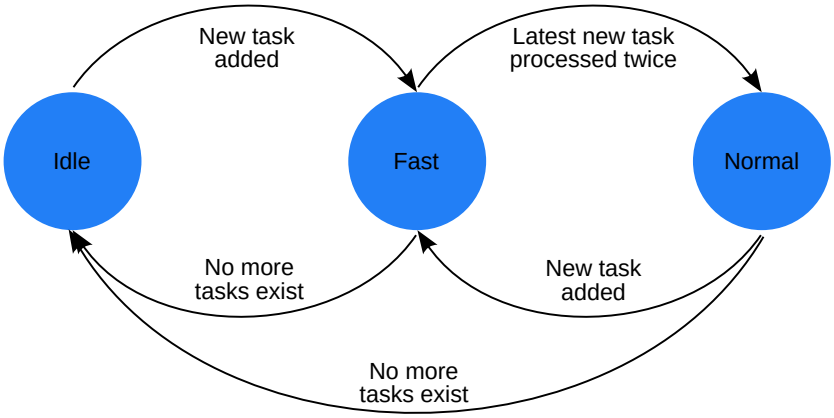


Figure 3.2: State transition diagram for the cycling rates used by the runtime.

for this to occur is to detect the presence of nearby services, a process which is referred to as *discovery*. However, simple service discovery is not a sufficient solution to the *association problem* [Tim02] of locating suitable partners for interaction. When multiple options exist, the selection between competing service offerings is non-trivial and requires more elaborate matchmaking between consumers and providers, otherwise systems may exhibit undesirable behavior. Mechanisms that go beyond static property matching are needed to automate such decisions.

In the context of storage management, a most significant factor in service selection is that of expected availability. When an application stores data on a device that is present in the ad-hoc network, it is – at best – a risky venture: the target device may at any time exit the network and render the data unavailable. It is therefore desirable to take into account expected availability when performing file transfers for load distribution or fault-tolerance purposes. In fact, as discussed in [KL05], a means of estimating future availability of services can be beneficial when selecting an interaction candidate for many other types of services (in addition to storage).

3.2.1 Collecting co-location statistics

Our approach towards "educated" service selection, is based on capturing the *co-location relationship* between a local device (and potential application host) and other remote devices (offering services to applications). To keep both current and historic information about such encounters, we adopt a simple information model comprising the following information for each remote device: (a) duration of the current encounter, (b) number and mean duration of previous encounters, and (c) time of the last encounter.

Table 3.1 shows co-location entries for a user's mobile phone. Based on these entries it can be inferred that the phone is constantly near the wristwatch, having only two stable encounters of very high duration. The user seems to have entered her office half an hour ago, where she has been roughly 22 times this week, averaging three hours for every stay. Moreover, she was previously in a car (probably driving to work) which was used about 34 times, with an average driving time of about half an hour. Earlier, the user was at home, as can be inferred from the bedroom and living room entries. At present, another (unknown) phone has been encountered, indicating that a person is perhaps visiting the user at the office.

Device	*Current duration*	*Previous encounters*	*Mean duration*	*Time of last encounter*
Wristwatch	50 hours	2	82 hours	53 hours
Office room	30 mins	22	3 hours	16 hours
Car	N/A	34	25 mins	40 mins
Living room	N/A	28	4 hours	55 mins
Bedroom	N/A	15	6 hours	45 mins
Other Phone	1 min	N/A	N/A	N/A

Table 3.1: Sample co-location information maintained by a mobile phone.

A more elaborate information model could be employed. For example it would

be possible to record a (bounded) number of separate entries for each encounter thus allowing us to deduce the distribution of co-location occurrence over time. One could then perform more informed analysis of co-location history and prediction of future behavior. Nevertheless, considerable information can already be deduced even from simple co-location data which can be maintained using a very small memory footprint. More importantly, it becomes possible to capture the basic relationship of *familiarity* among devices, which in turn can be easily exploited to guide service selection: Devices with a high *previous encounters* count can be expected (statistically) to meet frequently. Devices with a high *mean duration* can be expected (again, statistically) to be available in the PAN for significant time spans. The extent of expected availability can be deduced by the product of these two metrics.

We now describe how this information is efficiently maintained by our runtime system.

Each device emits beacons periodically, to make its presence known to nearby devices. Based on the receipt of such beacons, which we refer to as "sensing events", the discovery component keeps a set of *device history* records, shown in Listing 3.1. A record holds the time of the first sensing event (`tsCurEncStartEvnt`) and the time of the last sensing event (`tsCurEncLastEvnt`) contributing to the current encounter. It also holds the number (`prvEncCnt`) and mean duration (`meanPrvEncDur`) of previous encounters as well as the time (`tsLastPrvEnc`) when the last encounter occurred. The start time of the so-called observation window is stored (for purposes which will be discussed later on) in `tsObsrvWinStart`. Finally, the `maxBeaconPeriod` field holds the maximum interval among subsequent beacon emissions from that device. Table

```
public class DeviceHistory {
    AbstractDeviceAddress addr;
    long maxBeaconPeriod, tsObsrvWinStart;
    long tsCurEncStartEvnt, tsCurEncLastEvnt;
    long tsLastPrvEnc, meanPrvEncDur;
    int prvEncCnt;
    // ...
    long getCurEncDuration() {
        return
            tsCurEncLastEvnt - tsCurEncStartEvnt;
    }

    int getPrevEncCount () {
        return prvEncCnt;
    }

    int getMeanPrevEncDuration () {
        return meanPrvEncDur;
    }
    // ...
}

public interface IDiscovery {
    // ...
    DeviceHistory getDeviceHistory(
        AbstractDeviceAddress a);
    // ...
}
```

Listing 3.1: The `DeviceHistory` class

3.2 summarizes how these fields map to the elements of the co-location information model shown in Table 3.1.

When a device is sensed for the first time, a new history record is created for it and properly initialized. The `tsCurEncStartEvnt` and `tsCurEnc-LastEvnt` fields in the (new) history record are set equal to the time of the sensing event that triggered the encounter. The running duration of the encounter is maintained as subsequent sensing events cause updates to the `tsCurEnc-`

History Information	*Implementation*
Device	`d.getAddress()`
Current duration	`tsCurEncLastEvnt - tsCurEncStartEvnt`
Previous encounters	`prvEncCnt`
Mean duration	`meanPrvEncDur`
Last encounter	`tsLastPrvEnc`
Time to next beacon	`maxBeaconPeriod`

Table 3.2: Mapping of co-location data model to `DeviceHistory` fields.

`LastEvnt` field. The device's maximum beaconing period is retrieved from the beacon packet and stored in `maxBeaconPeriod`.

The encounter with the device is considered *active* until the expiration of the `maxBeaconPeriod` timeout: if the elapsed time since the last sensing event for that device (`tsCurEncLastEvnt`) exceeds the stated maximum beaconing period, the encounter is declared inactive. Whenever a runtime emits is own beacon, it checks all active history records to see whether any have expired. In this case, the record's fields are updated as follows: the `prvEncCnt` is incremented by one, the `tsLastPrvEnc` is updated to hold the value of `tsCurEncLastEvnt`, the `meanPrvEncDur` is recalculated to incorporate the duration of the encounter, and the `tsCurEncStartEvnt` and `tsCurEncLastEvnt` are reset to zero, indicating that the device is no longer co-located with the local host. Figure 3.3 illustrates a sample series of sensing events and the corresponding device history record updates performed by device A with respect to the presence of device B.

3.2.2 Pruning co-location history

The large number of devices that can be encountered in a ubiquitous computing setting, along with the typical resource constraints of embedded devices, makes

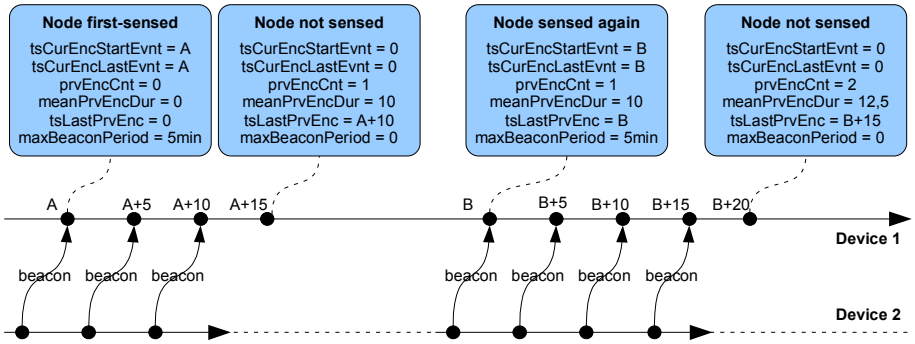


Figure 3.3: Time diagram of sensing events for co-location history recording.

it unrealistic to let co-location information grow ad infinitum. This necessitates a policy for replacing or collecting device history records.

For this purpose, co-location history is maintained with respect to an *observation period* P (e.g. one week), which is a system configuration parameter that makes it possible to remove old co-location records. Since we do not keep a separate timestamp for each previous encounter, garbage collection is done by approximation, as follows. When a new history record is created, its `tsObsrvWinStart` field is initialized with the time of the first corresponding sensing event. This field is used to periodically check whether the record contains data beyond the observation period P, i.e. `currentTime - tsObsrvWinStart > P`. In this case the co-location history is *pruned* by increasing the value of the `tsObsrvWinStart` field by an advancement time slot `T`. Assuming a uniform distribution of encounters in time, the number of previous encounters can be adjusted by decreasing the `prvEncCnt` field in proportion to `T`. The `meanPrvEncDur` field remains unchanged. As a result of this update process a history record may end up containing merely "outdated" information and can be removed. We consider this to be the case when: `prvEncCnt < T/P`, which allows a record based

on a single encounter to persist for at least two consecutive periods. Other metrics could easily be employed for discarding records.

Even with this garbage-collecting procedure in place, it is still possible that a new history record may need to be created, while the system's device history table is full. In this case, a *victim* record is chosen (randomly) for removal, out of all records concerning devices that (a) are currently not co-located, (b) have the smallest total duration of previous encounters (approximated as the product of the number of previous encounters and mean duration) and (c) have the oldest latest encounter. If no such record exists, i.e. all records are about devices that are currently co-located, then the new encounter is ignored.

This approach results in the system being "blind" to new devices if the size of the set is smaller than the number of currently co-located devices, in the sense that no co-location statistics are collected for them. Nevertheless, this limitation is of minor practical importance, as the memory requirements of our approach are very modest and allow the monitoring of thousands of devices using small amounts of memory[1]. Finally, it is important to stress the fact that co-location information is an auxiliary hint, and that it is possible to discover and access a device and service independently of whether a corresponding device history record exists.

As a last point we stress that, as was discussed in Section 3.1, the algorithm presented here is not dependent on the beaconing rate of devices, which may be adjusted at runtime without affecting the validity of results. Specifically, increased beaconing rates merely translate into greater sampling rates, thus providing greater accuracy for the statistics collected. The idle beaconing rate of a device

[1]Assuming 16-byte device addresses, one thousand device records can be maintained in roughly 64KB of memory

defines the smallest granularity of statistics for it.

3.3 Service discovery

The discovery mechanism supports two modes of operation: active and passive. *Active* discovery allows an application in need of a service to probe the PAN in order to locate it. On the other hand, *passive* discovery relies on overhearing information regarding services in the PAN. The names used for the two modes reflect the service client's perspective in the discovery process[2]. Active discovery is useful when an application needs a service in order to perform some task the soonest possible and therefore does not want to miss out on opportunities to do so. Passive discovery is preferable when an application is not eager to perform some task immediately, but may opportunistically do so when circumstances permit.

This functionality is accessed via the programmatic interface shown in Listing 3.2, in which service types are represented by strings whose values are presumably agreed upon by application developers. Communication occurs via `Endpoint` objects (see Section 3.4) which represent unique addresses on hosts and allow for multiplexed communication among them.

Discovery is implemented by exploiting the beaconing cycle discussed in Section 3.2. Specifically, we use the emitted beacons to piggy-back lookup and advertisement information. The mechanism used to decorate beacons is based on two lists maintained by the discovery component: one with lookup tasks and the other with advertisement tasks.

Lookup tasks are created by applications when they require services of a cer-

[2]Evidently, the exact opposite holds from the service provider's perspective.

```
public interface IDiscovery {
    // ...
    // register a local service provider
    public void registerService(String serviceType,
        Endpoint ep);

    // support passive discovery of a local service
    public void setActiveAdvertise(Endpoint ep,
        boolean persistentAds);

    // register interest in specific service types
    public void registerInterest(String serviceType,
        IServiceFoundListener listener);

    // enable active discovery for some service type
    public void setActiveDiscovery(String serviceType,
        boolean activeDiscovery);

    // list known service providers in the PAN
    public List getProviders(String serviceType);
    // ...
}
```

Listing 3.2: The discovery component API

tain type to be *actively* discovered; these tasks cause the runtime to emit *lookup requests*, informing nearby devices of its need to locate service offerings of that type. The runtime notifies local applications whenever it discovers a suitable provider.

Advertisement tasks on the other hand are used to advertise the existence of a service to other entities in the PAN. They cause the runtime to emit *advertisements* that contain contact information for the local service provider, so that nearby entities may detect it. These tasks are created by the runtime when lookup requests are received from the PAN, causing an advertisement to be sent as a reply for the actively searching entity. In addition, a service provider may also create an advertisement task, to *periodically* advertise the service, regardless of whether lookup

requests are received for it. Doing so allows the service to be passively discovered by nearby clients.

Advertisement and lookup tasks are processed by alternating among them with a ratio equal to that of the respective list lengths. For example, if twice as many lookup as advertisement tasks exist, then two beacons will be sent out with lookup information attached, for every beacon that is emitted with advertisement information. Respectively, whenever a beacon is received (and after co-location statistics are updated as discussed in Section 3.2), it is passed on to the discovery component for processing. The discovery component then extracts and processes any piggy-backed discovery information which may exist in the beacon. In the following sections, we discuss each of the two modes of discovery (active and passive) in detail.

We point out that, as was discussed in Section 3.1, the creation of advertisement and lookup tasks affects the beaconing rate according to the state transition diagram of Figure 3.2. In short, when a task is added, the runtime beacons at fast rate. It then falls back to normal rate unless another (new) task is added in the meantime. When all tasks have been serviced, beacons are emitted at idle rate.

3.3.1 Active discovery

Active discovery is triggered by applications that wish to locate a service in order to perform some task. The process is initiated by the interested application through a call to the `registerInterest()` method, with the desired service type as a parameter, along with a listener method to be called when a service offering is found. A subsequent call to the `setActiveDiscovery()` method is

required to create a lookup task for this service type, which periodically broadcasts requests for it.

The discovery component distributes lookup requests evenly among the service types registered in its lookup task list. It selects the first node in the list to piggy-back a request to the outgoing beacon and then adds the node back to the end of the list. New registrations are added to the beginning of the list so that the first beacon for them is emitted with priority to older running lookups.

Whenever a beacon is received, the list of locally available services, populated by applications via `registerService()` calls, is checked to see if matching services exist. If a match is found, a one-off advertisement task is added to the head of the advertisement tasks list.

Figure 3.4 depicts the active discovery process. The application on the left creates an endpoint (step 1) and calls `registerInterest()` and `setActiveDiscovery()` (step 2). A lookup task is thus created by the discovery component for that service type, which also records the submitted listener method to be used when matches are found. The application on the right creates an endpoint (step 3) for providing this service. It then calls `registerService()` in order to notify the discovery component of its existence (step 4). At some point, the application on the left emits beacons with a lookup request for the desired service type (step 5). The runtime of the application on the right, upon receiving such beacons, enqueues advertisements for the endpoint providing the service (we assume that both applications are using the same service type resulting in a match) and therefore emits advertisements beacons (step 6) with contact information for the endpoint. Finally, the application's listener is called (step 7) to deliver the address of the service to the application.

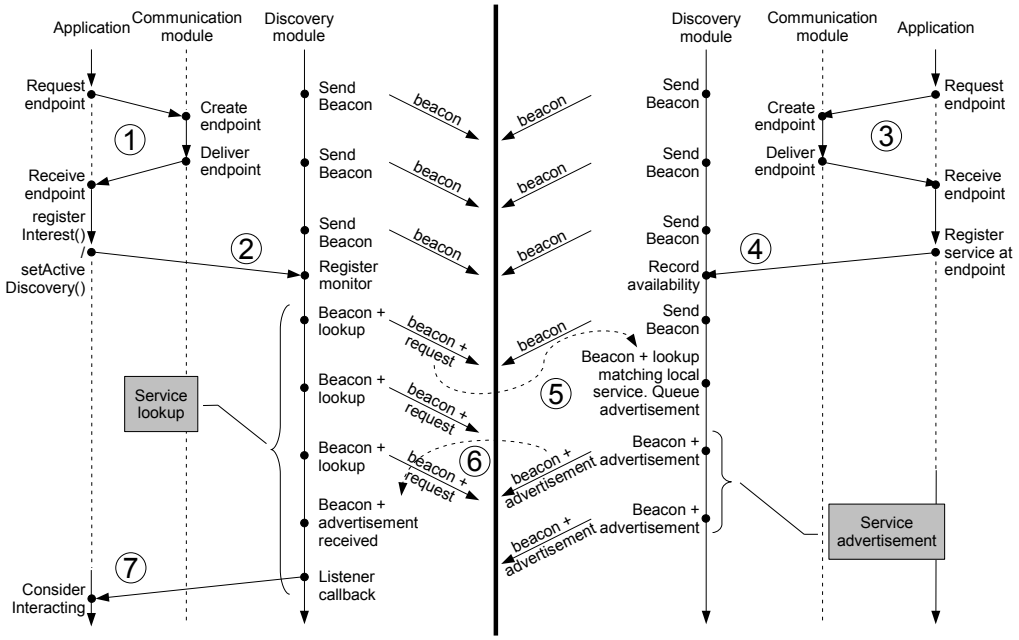


Figure 3.4: Active service discovery using lookup requests.

3.3.2 Passive discovery

The runtime maintains a list of service types of interest to local applications. When an advertisement beacon is received, the service type is looked up to determine whether local interest exists. In this case, the contact address for the service is recorded in this list. The contact information (if any exists) is returned to the application when it calls the `getProviders()` method. In addition, the listeners provided by `registerInterest()` method calls are used to deliver notifications to the applications interested in that service type, every time a new service offering is found.

Clients that are interested in exploiting passive discovery call the `registerInterest()` method to declare their interest for a service type, but do not call the `setActiveDiscovery()` method. This means that even though the ap-

plication will be notified when a service offering is detected, the runtime will not emit any lookup requests for the service. Instead, discovery relies on overhearing advertisements sent out by service providers. While this may occur coincidentally if another nearby device happens to emit a lookup request for that service type, the mechanism is intended for use by service providers that periodically advertise their presence, without being provoked to do so by lookup requests. To this end, the runtime can be instructed to emit advertisement beacons for services, independently of whether a lookup request was received for them. In other words, the advertisements are broadcast in a "to whom it may concern" fashion.

To cause this type of advertising the service provider must call the `registerService()` method, followed by a call to `setActiveAdvertize()`. This creates a periodic advertisement task in the discovery component that repeatedly sends out advertisements for the service offering, regardless of whether a lookup request was received for the service. Figure 3.5 depicts the passive discovery process.

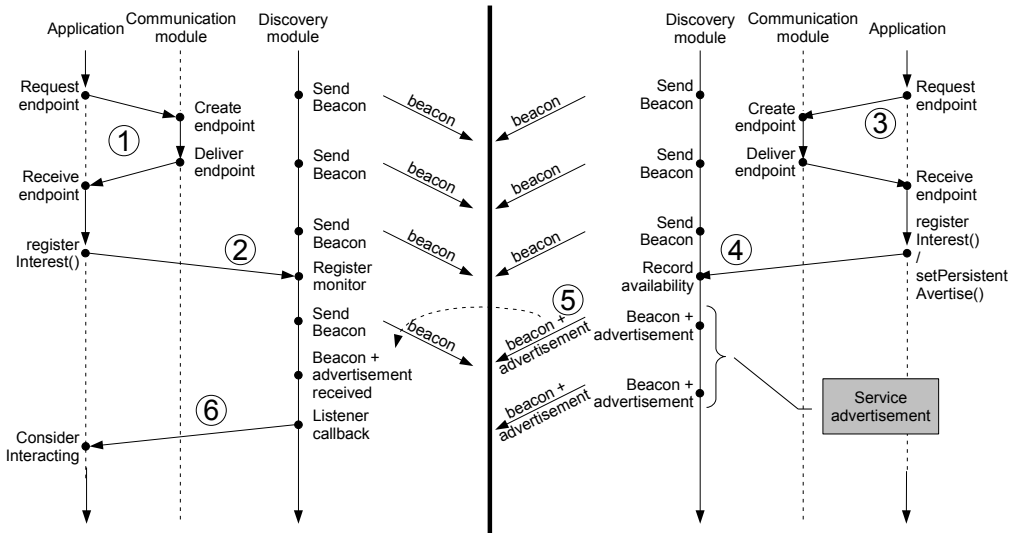


Figure 3.5: Passive service discovery using persistent advertisements.

3.4 Communication

The communication component provides support for applications wishing to exchange messages with peers on other devices in the PAN. The basic element of this mechanism is the *endpoint*, a uniquely addressable element which can be used as a contact point when exporting a service to the PAN. Its address consists of the device's network adapter address (which is network technology dependent) and a 16-bit integer that is guaranteed to be locally unique for each endpoint in a runtime. The endpoint's address can be obtained using the discovery mechanisms discussed in Section 3.3.

3.4.1 Unreliable communication

Applications can exchange datagrams by creating an `Endpoint` object, which allows its holder to send packets of data to other endpoints in the PAN. This is done via the `sendPacket()` method, which emits a packet targeted to the endpoint whose address is given as a parameter. The source endpoint's address is automatically placed in the packet's header. The runtime processes packets received from the PAN and delivers them accordingly to local endpoints for processing. Specifically, each endpoint can define a callback method (`processPacket()`) via `setPacketListener()`. The runtime calls this method for all packets targeting the endpoint, also giving the source address of the incoming packet to the listener, so that its origin can be reviewed (and optionally used to send replies). The most relevant parts of the communication API can be seen in Listing 3.3.

```
public interface IPacketListener {
    public void processPacket (Address sourceAddr,
        byte [] payload);
}

public class Endpoint {
    // ...
    public int getLocalId();

    public int sendPacket(Address targetAddr,
        byte [] payload);
    public void setPacketListener(
        IPacketListener callbackMethod);
    // ...
}

public class ReliableBidiEndpoint extends Endpoint {
    // ...
    public void connect(Address targetAddr);
    public ReliableBidiEndpoint accept();
    public void close();

    public int writeBytes(byte[] outDataBuf);
    public int readBytes(byte[] inDataBuf,
        int ofs, int maxLen);

    public void writeBoolean(boolean b);
    public void writeInt(int i);
    public void writeLong(long l);
    public void writeFloat(float f);
    public void writeDouble(double d);
    public void writeUTFString(String s);

    public boolean readBoolean();
    public int readInt();
    public long readLong();
    public float readFloat();
    public double readDouble();
    public String readUTFString(s);
    // ...
}
```

Listing 3.3: The communication component API

3.4.2 Reliable communication

`ReliableBidiEndpoint` is a descendant of the `Endpoint` class, which builds upon its functionality providing reliable, bi-directional, streaming communication among two connected endpoints. It is basically an implementation of the Transmission Control Protocol (TCP) specification [ISI81], with minor modifications. The most important deviation from the TCP standard is that the underlying network layer is not assumed to be the Internet Protocol (IP). Instead, the addresses used in packets are endpoint addresses (rather than the typical IP address / TCP port combination). Another difference is that the window size is fixed. Using the `ReliableBidiEndpoint` class (Listing 3.3) applications can communicate using byte streams.

The `ReliableBidiEndpoint` class defines internal outgoing and incoming data buffers which are used to generate outgoing packets and store the payload of incoming packets respectively. Applications must use the `writeBytes()` method to fill the outgoing buffer and the `readBytes()` method to extract data from the incoming buffer. The base `sendPacket()` and `setPacketListener()` methods are overridden to prevent applications from accessing them directly. The `ReliableBidiEndpoint` uses the outgoing buffer's contents to send packets to the connected peer, appending the required header information to achieve reliable and ordered transmission. It also processes incoming packets by installing its own `processPacket()` handler, which acknowledges the receipt of data and enqueues it for delivery in the incoming buffer. Retransmission, order preservation and receipt acknowledgment are all handled internally as defined in the TCP specification [ISI81].

Finally, we provide simple data marshaling routines for all primitive data types (integers, long integers, floats, doubles, booleans, characters and strings). We consciously refrain from supporting Java object serialization, thus allowing the implementation of the core runtime mechanisms in other programming languages[3].

3.4.3 Accessing infrastructure services

Network gateways are devices which have both a wireless ad-hoc adapter, as well as an adapter which connects them to the Internet. Their role is to bridge the two networks, allowing nearby devices in the PAN to connect to services on the Internet. The functionality is provided by the *Internet access daemon*, which creates tunnels for such connections.

The Internet access daemon uses the discovery mechanism (Section 3.3) to emit advertisements for its tunnelling service to the PAN. Clients that wish to contact Internet services must use the discovery mechanism to search for gateways, and then connect to them using reliable communication endpoints (Section 3.4.2). In order to setup the tunnel, the *Internet access daemon* expects the client to send a target host (either its DNS name or its IP address) and port number. Once these are received, it establishes a normal TCP/IP connection to that host and sends the connection result to the client in the PAN. If the connection has succeeded, the Internet access daemon starts tunneling all data received from the PAN through the TCP/IP connection and back. From that point onward, the portable device must interact with the service using whatever protocol is employed by it (e.g. HTTP/SOAP for web services).

[3]We also remind that our Java implementation even restricts itself to using the facilities of the `java.lang` package.

Network gateways allow any protocol implemented on top of TCP/IP to be accessible via tunnels. Of course, in this case, the appropriate interaction logic used by that protocol must be installed on the portable devices which intend to use the service. In any case, we would like to support the case where a service must be accessible to all classes of clients:

- **Nearby portable devices**: The client application is hosted on a portable device which is physically located near the server.
- **Remote portable devices**: The client application is hosted on a portable device which is using a network gateway to contact the server.
- **Internet hosts**: The client application is on another host in the Internet and is communicating with the server via normal TCP/IP.

Any services hosted in the user's house fall into this category. To facilitate such cases we have created two classes especially suited for this purpose: `TunneledEndpoint` and `TCPIPEndpoint`.

The `TCPIPEndpoint` class extends `ReliableBidiEndpoint`. It replaces the network I/O primitives of its `Endpoint` ancestor to read incoming / write outgoing packets from / to a normal TCP/IP socket, instead of using the runtime's network adapter driver. Specifically, it overrides the `sendPacket()` method to write the packets to a normal TCP/IP connection, while at the same time creating incoming packets by reading data from the connection and calling `processPacket()` to process them. Suitable constructors are defined for this class, which take the host and port to connect to (client operation) or the port to listen on for connections (server operation) as parameters. Services running on

Internet hosts that wish to be accessible through network gateways must listen for connections using `TCPIPEndpoint` objects.

The `TunneledEndpoint` is similar, but replaces the network I/O primitives of the base `Endpoint` class to use a `ReliableBidiEndpoint` connection instead of calling into the runtime's network driver. In this case, the `sendPacket()` method is overridden to write packets to a `ReliableBidiEndpoint` connection's stream, while at the same time creating incoming packets by reading data from that connection and calling `processPacket()` to process them. A new constructor is defined that takes the `ReliableBidiEndpoint` object to be used for this purpose as a parameter (it is assumed that it is a connection to a network gateway in the PAN), along with the host and port to which a tunnel is to be requested (it is assumed that a `TCPIPEndpoint` is listening on that address). The constructor negotiates the tunnel creation with the gateway and proceeds to establish the tunneled connection.

Figure 3.6 depicts an Internet server which is interacting with three clients: one is on another Internet host whereas, one is a portable device which is using network gateway tunnel, and one is an Internet host which is communicating through a direct TCP/IP connection. Because both `TCPIPEndpoint` and `TunneledEndpoint` inherit from `ReliableBidiEndpoint`, the interaction logic (excluding connection establishment) is the same for all cases on both sides (client and server).

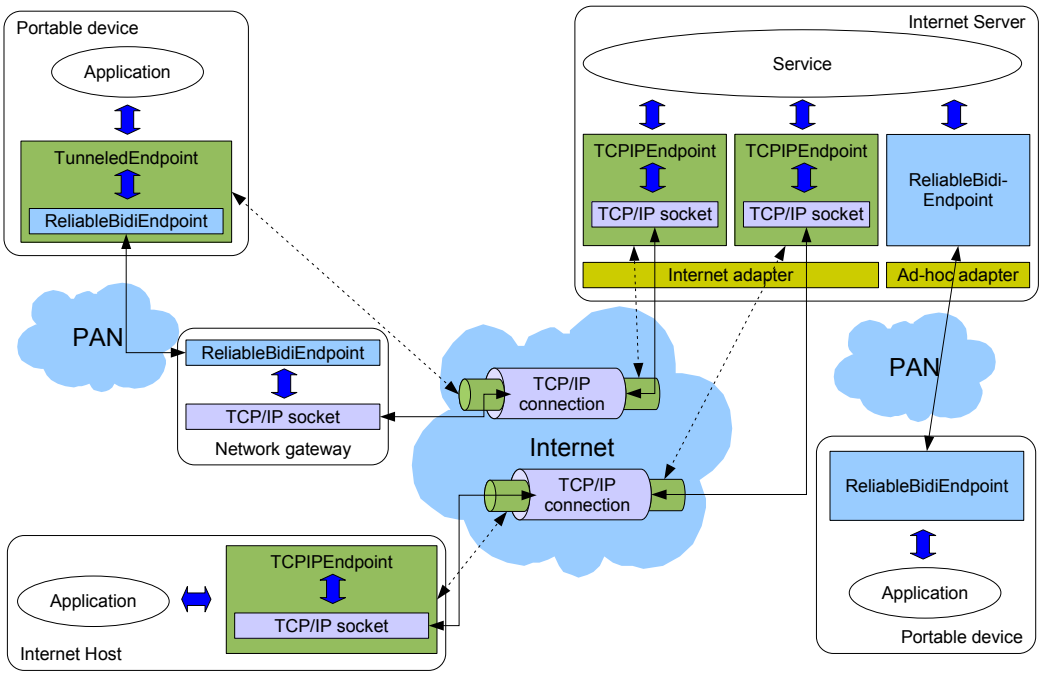


Figure 3.6: A tunneled connection from the PAN to the Internet

3.5 Context aggregation

Wearable and portable devices can provide a wealth of contextual information via a variety of sensing capabilities. For example, a GPS navigator may provide latitude, longitude, speed and direction information, a camera may provide illuminance information, and a person's active badge may provide identification information. Although each such piece of contextual information can be useful by itself, it is likely to become more valuable when combined with others. Ideally a group of devices can complement each other by providing different contextual aspects, thereby enabling the synthesis of the current contextual situation [GSB02]. Along this rationale the role of the context component is to allow any entity to publish and access context information, as well as to let such information propagate in the PAN in order to create a more complete aggregate contextual perception.

3.5.1 Generating and reviewing context information

The context component maintains information in the form of a tuple space where each entry consists of seven fields: a name indicating the type of information (key), a value, its creation timestamp (`CTS`), its lifetime or time-to-live (`TTL`), its validity scope, its emission timestamp (`ETS`) and the originating device address (`ODA`). The `key` and `value` fields correspond to an attribute-value pair that holds the actual context information. The `CTS` and `TTL` fields are used for garbage collection. The `scope`, `ETS` and `ODA` fields are used to control the propagation of context information to other devices.

The API for accessing the tuple space is shown in Listing 3.4. To add a tuple, the application invokes the `post()` method, supplying values for the `key`, `value`, `TTL` and `scope` parameters, which are used to initialize the respective fields of the new entry. The `CTS` and `ODA` fields are set equal to the current time and the local device address respectively. The `ETS` field is set to zero.

When generating contextual information the device's application is expected to post new values for its keys at the rate of change of the respective information. The `post()` method searches and removes duplicate entries (i.e. entries that have the same `key` and `ODA` values) as new tuples are added. Entries originating from different devices (with an `ODA` value that is not equal to the local device address) are not considered as duplicates even if they have the same `key`. This is to allow for different devices to provide different *flavors* or *granularity* of the "same" type of context information, without introducing any conflicts or having to build elaborate metadata mapping mechanisms at a low system level.

Applications may review the currently available context information via the

```
public ContextTuple {
    AbstractDeviceAddress oda;

    String key, value;
    int ttl, scope;
    long cts, ets;

    public String getKey();
    public String getValue();
    public AbstractDeviceAddress getOrigDevAddr();
    // ...
}

public interface IContext {
    public static final int LOCAL_SCOPE = 1,
        DEV_RANGE_SCOPE = 2, TTL_SCOPE = 3;

    public static final String ANY_KEY = null;
    public static final String ANY_DEVICE = null;

    public void post (String key, String value,
        int ttl, int scope);
    public ContextTuple[] read (
        AbstractDeviceAddress devid, String key);
    public void clear(
        AbstractDeviceAddress devid, String key);
}
```

Listing 3.4: The context component API

`read()` method which returns the entries matching the supplied key name and originating device address. The special device address value `null` can be used in combination with a specific key, to obtain all entries with the specified key produced by any device. Similarly, the special key value `null` can be used in combination with a specific device address, to obtain all entries produced by the specified device with any key. Using both special values, the entire contents of the tuple space are retrieved. The caller may inspect the returned entries via the getter methods of the `ContextTuple` class.

Garbage collection is performed as a side-effect of the `post()`, `read()` and `clear()` methods, thus avoiding the need of an extra thread to perform housekeeping tasks. As the tuple space is being searched to find duplicate or matching entries, outdated entries are detected by comparing the current time with their `CTS` plus `TTL` fields. Any such tuples are removed. Entries may also be explicitly removed prior to their `TTL` expiration, via the `clear()` method.

3.5.2 Context information propagation

To exploit the collective context-sensing capabilities of the PAN, the contents of the tuple space are disseminated in an asynchronous fashion to nearby devices. As in the case of service discovery, this occurs in tight coupling with the operation of the beaconing mechanism. More specifically, prior to emitting the beacon, the runtime up-calls a method of the context component that may append context tuples to the beacon. Conversely, every time a beacon is received, the runtime up-calls a method of the context component that extracts context information from the beacon and inserts it to the tuple space. We point out that the beaconing rate

Value	*Meaning*
local	Entry which applies to the originating device only and does not propagate to other devices. Discarded when its TTL expires.
dev-range	Entry which propagates to other devices but is valid only within range of the originating device. It is discarded when out of the originating device's range, or when its TTL expires.
ttl-range	Entry which propagates to other devices and is valid everywhere, until its TTL expires.

Table 3.3: Possible values for the scope of context information

is affected by the existence on newly generated local context information in an identical manner as in the case of newly created discovery tasks (see Section 3.1).

The scope and emission timestamp are used to control tuple propagation. The scope of an entry determines its effective range. Entries with `local` scope are confined to the local context component while entries with `dev-range` or `ttl-range` scope are disseminated to the PAN. The meaning of these values is summarized in Table 3.3 and their usage is explained in more detail in Section 3.5.3. The `ETS` records the time when an entry was last beaconed. In case there are too many entries to fit in the beacon packet, the context component selects the ones with the smallest `ETS` value. This ensures that all entries get a fair chance of being propagated to other devices and that a newly added tuple has higher priority over older ones.

For each entry that is selected for dissemination, only its `key`, `value` and `TTL` fields are piggy-backed on the discovery beacon (Figure 3.7). There is no need to send the `ODA` field since only locally generated entries are allowed to propagate to other devices and the emitting device's address is already part of the beacon header. Also, the sent `TTL` indicates the remaining (as opposed to

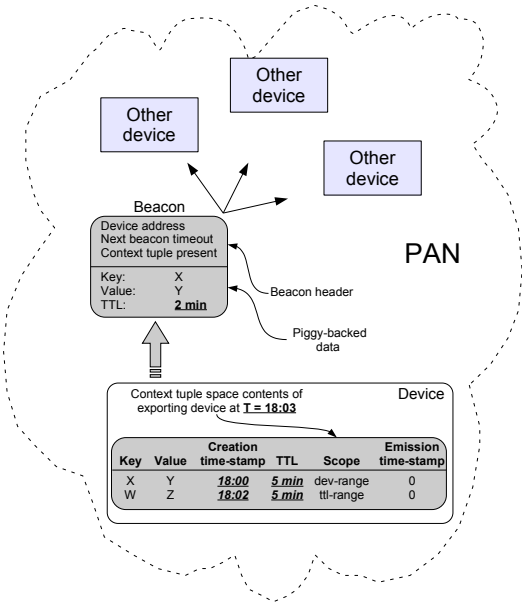


Figure 3.7: Exporting tuples to nearby devices in the PAN

the originally specified) lifetime of the entry. This is calculated as the difference between the original `TTL` value and the time that has elapsed between the creation of the tuple (`CTS`) and its transmission (piggy-backing on the beacon). As a result it is not required to send the `CTS` field of each entry. More importantly, devices are not obliged to maintain synchronized clocks.

Incoming context data is handled in an analogous fashion. For each entry carried by the beacon, a new tuple is created and assigned with the received `key`, `value` and `TTL` fields. The `CTS` field is initialized to the current local time at the moment of the reception (extraction from the beacon) and the `ODA` field is set equal to the address of the device that sent the beacon. Just like when adding a new tuple via the `post()` method, the new entry replaces any duplicate entry with identical `key` and `ODA` values.

Figure 3.8 illustrates this dissemination process for a scenario with three de-

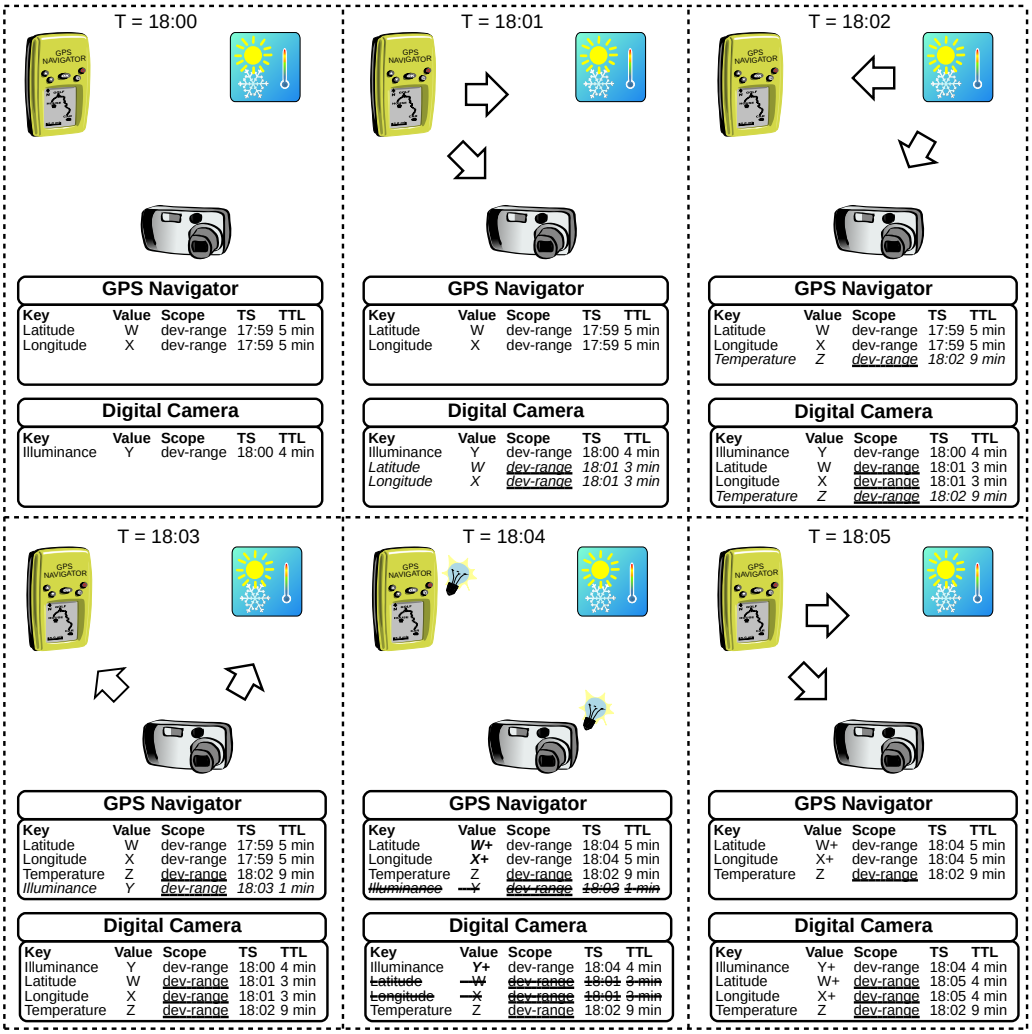


Figure 3.8: Context dissemination process

vices: a GPS navigation unit, a camera and an information beacon present in the surrounding infrastructure, which has a temperature sensor. Subsequent snapshots are shown in one-minute intervals. For simplicity, only the tuple space contents of the GPS and camera devices are presented. Furthermore, we omit the emission time-stamps and originating device address for brevity. Instead, non-local entries in each tuple space have their scope underlined to stand out. The scenario is discussed in the following, starting from the top left snapshot and moving from left to right.

The context component of each device initially contains only information that was generated locally. Therefore, in the example, the GPS navigator has entries for latitude and longitude whereas the camera has a single entry for illuminance. The nearby information beacon whose context space is not shown, has a single locally-generated entry with the current temperature. In the next step, the GPS emits a beacon carrying its own context information and as a result the camera tuple space is populated with two new entries (shown in italic fonts). Next, the temperature sensor transmits a beacon carrying its own context information that ends up in the tuple spaces of the GPS and camera. In a similar fashion, the subsequent transmission of a beacon from the camera leads to the population of the GPS tuple space with a corresponding illuminance entry. In the following, the GPS generates updated latitude and longitude values which replace the old entries; as a side effect, the expired illuminance entry that was imported from the camera is also deleted. At the same time, the camera generates an updated illuminance value; leading to the deletion of the expired latitude and longitude entries imported from the GPS. As a last step, the GPS emits its fresh latitude and longitude values, resulting in the addition of corresponding new entries in the

tuple space of the camera.

It is important to note that beaconing is both unreliable and asynchronous. For this reason there is no guarantee as to whether or when a particular piece of local context information will propagate to other devices. While the tuple spaces of devices do have the *tendency* to converge, this cannot be fully accomplished as long as context information and/or the connectivity between devices change frequently (with respect to the beaconing rate). In the general case, the tuple spaces of devices will be different, each corresponding to a partial and possibly outdated view of the global contextual picture. Despite its *opportunistic* nature, periodic beaconing achieves satisfactory information propagation for most practical purposes. It is also simple to implement and was efficiently integrated as part of our beaconing mechanism.

As a last note, we point out that certain devices need not process contextual information. The information beacon for example, whose sole purpose is to provide information to devices present in the room, need not do any actual processing of the GPS navigator and camera beacons, as it does not run any application that requires contextual information.

3.5.3 Controlling context propagation

A PAN computing system corresponds to an inherently dynamic environment with devices continuously coming in and out of range. From the perspective of a wearable or portable device, the network neighborhood can change either as a result of its own movement (it is carried along by the user) or due to the movement of other devices (it is left behind). This is a tricky issue for any distributed context pro-

vision service precisely because context information is often tightly coupled with the physical location where it was produced. In our case, the problem – which we refer to as *context littering* – is exhibited as follows: a device may use and propagate imported context information that is no longer valid in the current situation. For example, a mobile device may pick up temperature information from a nearby sensor in an air-conditioned room and then keep as well as disseminate this entry after it has been moved to an outdoor location with completely different environmental conditions.

To prevent context littering, imported entries are not eligible for further propagation. We therefore compare a tuple's originating device address against the local device address prior to piggy-backing it on outgoing beacons, allowing only locally-generated tuples to be exported. This prevents context from being "carried over" from one location to another by a moving device, thus protecting devices at the new location.

Even so, the moving device itself is still not protected from formerly acquired context tuples, since they are considered to be valid until their `TTL` expires. One approach to address this would be to immediately remove imported context tuples when their originating device goes out of range, regardless of whether their `TTL` has expired. However, this precaution may be too strict in some cases. It is namely possible for a device to provide reliable context information within some advertised `TTL`, regardless of whether receivers stay within its range. For example, a service in the center of Volos might export a "location" entry with the value of "Volos, Greece" and a `TTL` of 5 minutes. It is indeed unlikely for a device that receives this information to move outside of Volos in such a short time. This would obviously not hold if location information was provided at a granularity of

just a few meters, for instance via a GPS device. Of course this depends on the type of the information produced.

Rather than having devices *interpret* the information received in order to decide how to handle `TTL` values, we let the context source explicitly state the information's "sensitivity" to movement. This is done via the `post()` method, by setting the `scope` field of the new entry accordingly, to either `dev-range` or `ttl-range` (see Table 3.3). While both entry types are properly disseminated via the beaconing mechanism, their type is used at the receiver to decide how to handle the advertised `TTL`. If the information received has a `scope` field equal to `ttl-range`, its `TTL` is adopted as advertised. Else, if the `scope` field is equal to `dev-range`, it is considered valid only as long as both its `TTL` has not expired *and* the originating device remains within range. Therefore entries whose `scope` field is equal to `dev-range` but whose history record in the discovery module is inactive are removed regardless of their `TTL` value.

In the spirit of the previous scenario, Table 3.4 shows a possible state of the digital camera tuple space with five entries. Only the `illuminance` reading is locally generated, whereas all others were imported from the nearby devices. The rest of the information in the example is published by the GPS navigator device and a nearby information beaconing device in the infrastructure. In addition to `latitude` and `longitude` information, the navigator uses an internal database to produce coarse-grained location information which in this case is the `city` tuple. Attention should be paid to the fact that the `latitude` and `longitude` tuples are published with `dev-range` scope, whereas the `city` tuple is published with `ttl-range` scope, given that it is unlikely to change rapidly. The example tuple space also contains a tuple from an infomation beaconing device

which we assume exists at the cafeteria where the user's camera is currently located. This device publishes temperature and location information. Again, the highly "movement-sensitive" tuples `indoor-temp` and `cafeteria` are published with `dev-range` scope, whereas the `outdoor-temp` tuple is published with `ttl-range` scope, as it is not expected to change considerably within its 15-minute TTL.

Key	**Value**	**Creation CTS**	**TTL**	**Scope**	**Emit TS**	**Orig. device address**
illuminance	9857 lx	18:00	4 min	dev-range	18:03	A
latitude	X^d Y^m N	18:01	3 min	dev-range	0	B
longitude	K^d L^m E	18:01	3 min	dev-range	0	B
city	Volos	18:01	15 min	ttl-range	0	B
cafeteria	Grappa	18:01	14 min	dev-range	0	C
indoor-temp	25^oC	18:00	15 min	dev-range	0	C
outdoor-temp	32^oC	18:00	15 min	ttl-range	0	C

Table 3.4: Sample contents for a digital camera's context component

Chapter 4

File management system

OmniStore provides the following functionality to assist users in managing their personal data:

- **Deep archival:** All versions of all files created on all devices owned by the user are automatically archived at an infrastructure storage service referred to as the *repository*.

- **Push-caching:** The repository allows clients to submit requests for sending files to devices owned by the user. A time period during which files should reside on the target devices may be specified. Furthermore, the device can optionally be kept up-to-date with the latest version of the file being push-cached.

- **Collaborative disconnected operation:** Portable devices interact in order to co-operate in managing storage. They may autonomously off-load files from one device to another in order to balance free space, or replicate important files to multiple devices in order to increase availability. Remote

access from applications on one device to files on another device is also supported.

- **Automated context-based annotation:** File indexing and retrieval is facilitated by automatically attaching semantic annotations to files as they are created. These annotations are derived from context information collected from nearby devices.

While each of these functional aspects is useful in itself, the full potential of the system is made clear when they are considered in combination; we refer the reader to Chapter 5.1, which presents compelling use cases made possible by OmniStore.

4.1 Overall architecture

The system's implementation [KL06b] comprises several components. These are:

- The portable device daemon and device library
- The device registry and the device registry library
- The data repository and the data repository library
- Internet access gateways

Figure 4.1 depicts an indicative configuration that includes the repository, the registry, a single access point, a typical Internet-connected personal computer and a PAN with three portable devices. Although the registry and repository services

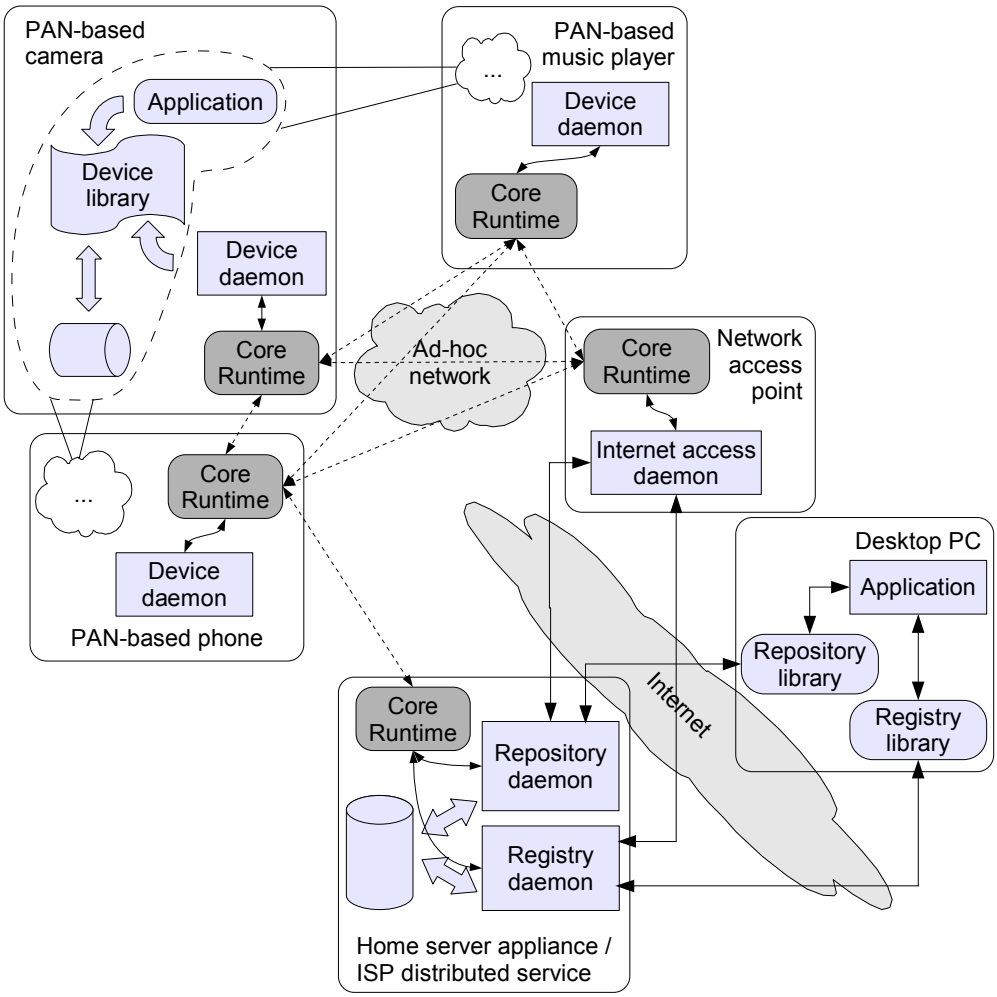


Figure 4.1: OmniStore architecture

need not be co-hosted, they are generally presumed to be installed on a household device referred to as the *home server appliance*.

Portable devices run the *device daemon* which is responsible for managing their storage. It supervises the storage-related activity on the portable, also monitoring the device's environment for the presence of other devices. The daemon communicates with devices detected in the PAN in order to perform collaborative storage management. Moreover, it periodically contacts the data repository

in order to perform various synchronization tasks. Applications use the *device library* to access the device's storage medium. The device library performs behind the scenes interaction with the device daemon, transparently triggering various storage-maintenance actions. The device library can also be used to access the functionality provided by the data repository and the device registry.

The data repository's role is to manage the infrastructure storage that belongs to the user and is implemented by the *repository daemon*. The repository daemon interacts with device daemons to synchronize the contents of each device with the repository. In addition, it allows applications to access the archived files, which may be located using semantic lookup queries. Finally, applications may use its push-caching service to schedule file transfers from the repository to portables. In order to access these services, applications on computers with Internet connections use the *repository client library*, whereas applications on portable devices use the *device library*.

The device registry is an infrastructure service which keeps track of user devices. It records information regarding their capabilities (e.g. what services they provide), as well as device configuration parameters. The *registry daemon* is the component responsible for implementing all these functions, which can be invoked via the *registry client library* (for applications on computers with Internet connections) and the *device library* (for applications on portables devices).

The protocols used in all cases employ the `ReliableBidiEndpoint` data marshaling routines (see Section 3.4). The repository and registry daemons accept connections through both `TCPIPEndpoint` (Section 3.4.3) as well as `ReliableBidiEndpoint` objects. This allows the services to be accessible via normal TCP/IP connections, but also directly through the ad-hoc network if an

application is on a portable device which is adjacent to the servers. The registry library and repository library communicate with their respective daemons through normal TCP/IP connections, using `TCPIPEndpoint` endpoints. Applications on portables access the functionality through the device library, which uses `ReliableBidiEndpoint` connections when the device is in close proximity to the servers. In most cases however, `TunneledEndpoint` objects are used instead, which connect to the `TCPIPEndpoint` of the servers (like infrastructure applications do) with the help of network access gateways (Section 3.4.3).

4.2 Device management

The device registry holds information regarding the computing devices owned by the user. It implements a *device registration* service which allows a device to enter the user's domain and become trusted by the other devices owned by her. To this end, it acts as a certificate authority on the user's behalf, which issues and stores digitally signed certificates during registration. Devices may use these certificates to determine co-ownership and establish trust when they meet in ad-hoc networks.

The registry is identified by its DNS name[1], which we use as the root of a device namespace for each user. For example, its name could be `userHome-Server.someISP.com`. The registry generates identifiers for each device during registration (Section 4.2.1), which are guaranteed to be unique within the context of the device registry. It then serves as a name service for the user's portables, allowing applications to unambiguously refer to specific devices owned by the user.

[1] This is sufficient for our purpose in spite of the numerous problems and shortcomings of DNS; see [BLR+04, WBS04].

In addition, the registry exports services for applications (both on portables as well as in the infrastructure) that wish to record or review information regarding the user's devices and their capabilities. Services and applications hosted on the user's various devices may record their presence there, along with configuration settings that control their operational characteristics. Applications may contact the device registry to inquire such information. For example, an application may request a list of the user's devices which support some type of functionality (e.g. have OmniStore managed storage).

4.2.1 Device registration

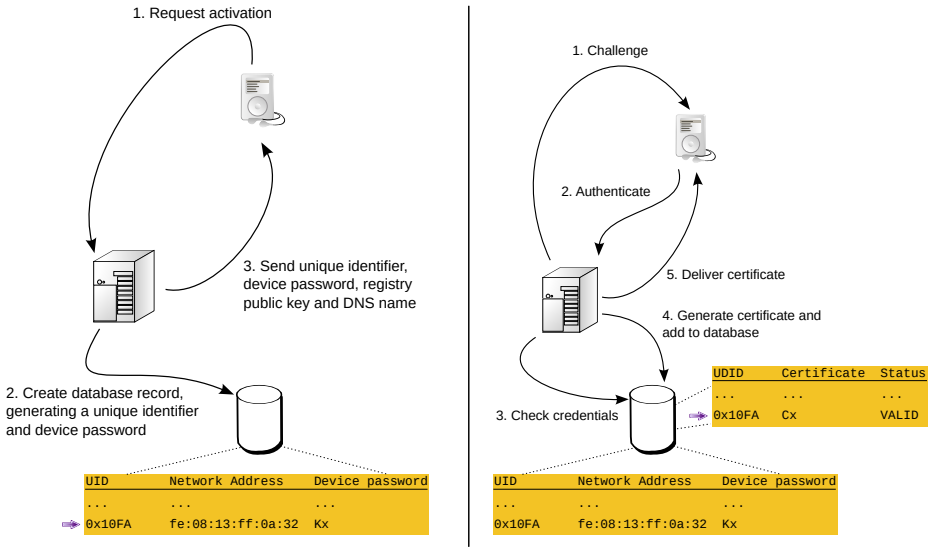
The device registry daemon is hosted on the home server appliance, located in the user's house. Users are expected to power up newly acquired devices in the area of the house where this appliance is located, to trigger the registration process.

As part of its normal operation, the registry daemon advertises the registration service to nearby devices. A device will check during its startup whether it is registered or not. Registered devices proceed to normal operation, loading the device daemon and launching local applications. Non-registered devices enter a special "registration mode", during which they passively monitor the environment (see Section 3.3.2), to discover a registration service.

The first stage of registration is as follows (Figure 4.2): When the new device detects the registry, it sends a registration request. The registry records the device's network adapter address in its database, giving the device a unique (within the registry) 32-bit identifier and a random password. It then responds to the device with: the device's newly-generated "device identifier" and random password,

along with the registry's DNS name and public key.

The issued device identifier is unique within this specific device registry. In combination with the generated password, they constitute a set of credentials which can be used in the future to authenticate the device to the registry. The device records all this information along with the registry's DNS name (which can be used to access the registry remotely through Internet access points) and the registry's public key. After this stage of registration is complete, the device is considered a candidate for entry to the user's device domain, but is not yet part of it.



(a) (b)

Figure 4.2: The two phases of device registration

In order to complete the registration, the user must accept the device's request. This is done via the web-based registry management application (Section 5.1.2), with which the user can view devices pending registration and accept or reject its completion in each case. Once a device's request has been accepted, the second

stage of the registration is triggered (Figure 4.2). The device registry performs an active lookup for the device, in order to contact it and complete the registration. When the device is discovered, the registry opens a connection, challenges the device to authenticate itself using the previously agreed credentials (device identifier and password) and then proceeds to issue a certificate signed by the registry for the device. The certificate comprises of the device identifier and adapter address, signed with the registry's private key; it is recorded in the registry database and delivered to the device[2]. Registered devices may use the registry's public key to verify the authenticity of another device's certificate. Thus, devices can use this to establish that they belong to the same user.

4.2.2 Device configuration

Once device registration is complete, additional functionality of the registry becomes available. Specifically, applications and services hosted on registered devices may record information regarding their capabilities and configuration parameters with the registry. This service is accessible both through the ad-hoc network, as well as through the Internet.

The registry maintains device capabilities and configuration settings in a table with a simple schema comprising of the following columns: device identifier, component identifier, key and value. When adding information regarding a device, the device identifier is set to the identifier issued for the device by the registry. The component identifier is submitted by the application / service creating the entry and is a value defined by its manufacturer, uniquely referencing that application

[2]Until the receipt of this certificate, a device is unable to prove its participation in the user's domain. The password produced for the device during stage one, is merely used to ensure safe delivery of the certificate to that device.

/ service on the device. The key and value hold the configuration setting's type and value respectively; their interpretation is application-specific. We note that even though we use the device registry to primarily record OmniStore configuration settings, the design of the system is generic, allowing it to be used by any application / service which may be present on portables.

Table 4.1 shows a sample of the device registry's contents. The component identifier `runtime` is reserved for use by our portable device runtime (Section 3) to store some device-wide configuration settings. For example, our runtime can be configured (by the device manufacturer) to create entries with the manufacturer's name, the device's model, its serial number, etc. The configuration setting `mnemonic-name` allows the user to specify a human-readable way with which to reference a device.

The special key `service` is reserved for publishing the presence of some type of service on a device (regardless of the component identifier under which it is defined). Its value is the same well-known name under which a service is advertised via the discovery mechanism (see Section 3.3).

The OmniStore device daemon uses the `service` key, along with its component identifier (`OmniStore`) to publish its presence on a device. It then exploits this facility to store configuration information which is used to trigger and control data management activities. Specifically, the device daemon registers three settings: `sync-period`, `min-free-ratio` and `desired-free-ratio`, which are given default values[3]. These parameters are discussed in subsequent sections.

The device registry allows applications to review device capabilities and in-

[3]Each manufacturer is free to specify sensible defaults for the devices they produce.

Device Id	***Component Id***	***Key***	***Value***
4	*runtime*	manufacturer	Smart music corp.
4	*runtime*	model	XJC-400
4	*runtime*	serial	W5-2005-0127-QC14
4	*runtime*	description	Portable music player
4	*runtime*	mnemonic-name	Walkman
4	OmniStore	*service*	`gr.omnistore.pan.server`
4	OmniStore	sync-period	120min
4	OmniStore	desired-free-ratio	0.2
4	OmniStore	min-free-ratio	0.1
5	*runtime*	manufacturer	Smart photos corp.
5	*runtime*	model	MC-1
5	*runtime*	mnemonic-name	My old camera
...	...	...	...

Table 4.1: Sample device configuration data stored in the device registry

spect or modify operational parameters. This enables applications to identify devices providing a specific type of service by searching for all device identifiers having a `service` key with the desired service's name as its value. As a concrete example, the repository management application (see Section 5.1.3) uses this service to lookup devices that support OmniStore functionality, using the results to present the user with a list of the possible target devices for push-caching transfers (Section 4.4.2).

4.3 File naming and access model

4.3.1 Naming scheme

The device registry is the basis upon which OmniStore's file naming scheme is built. This scheme allows us to generate globally unique identifiers for every file

created on any device owned by any user, even when the device is disconnected from the Internet and with no other devices in its vicinity.

In order to generate globally-unique identifiers for every file, devices maintain a 32-bit *local seed* that is incremented with every file creation. Its value is used as the *local identifier* of the created file, which combined with the generating device's identifier becomes unique across all devices in the user's realm. The globally unique file identifier is produced by further adding the device registry's name. An example of file and device identifiers is given in Figure 4.3, which depicts a mobile phone with two files: a photograph (which was presumably generated on the user's camera and subsequently transferred to the phone) and a locally created phone-call recording.

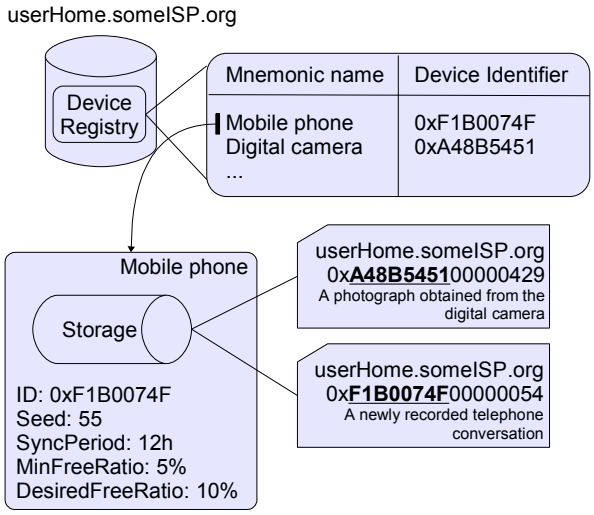


Figure 4.3: Various elements labeled using our naming scheme

The use of file identifiers instead of human-issued filenames is an important feature when considered with OmniStore's target environment in perspective: requesting a file name from a mobile user who, for example, has just recorded a

phone call or taken a photograph on the move, is rather inconvenient, especially through the limited user interfaces offered by such devices. In fact, for this exact reason, most contemporary portable devices will automatically generate cryptic filenames as well, albeit in an isolated device-specific manner, which generally only guarantees that name clashes will not occur for the files generated on that *one* specific device.

We note that files, both on devices as well as at the repository, are stored in a single directory in a *flat* manner. The complete file identifiers are used as file-system names. The globally unique nature of these identifiers makes name clashes impossible.

4.3.2 File organization with semantic annotations

File identifiers are hard to memorize and therefore inconvenient for people. To support flexible organization and browsing, files may be annotated with extra (semantic) information which is used to group or sort files at will. This approach, presented in [GJSO91, GM99, MTX03], is increasingly gaining in popularity. In these so-called semantic file systems, one generally uses annotation-based queries and is presented with dynamic views of storage contents, a model which deviates from the traditional static structure of hierarchical file systems.

OmniStore annotations are defined as in [GJSO91], using key-value pairs whose interpretation is left to applications. Some annotations (see Table 4.2) are reserved for system use and cannot be set by applications (except for `mnemonic-name` and `replicate-count`). Otherwise, the device library (Listing 4.1) allows applications to create, review and edit annotations .

Key	*Values and use*
dirty	Flag indicating whether the file exists in the repository or not.
mnemonic-name	A human-readable name for referring to the file.
derived-from	The value contains a file identifier indicating that the annotated file is a subsequent version of that file.
identical-content	Flag indicating that this file's content is identical to the one it is derived from (see Section 4.5.3).
avail-start	Timestamp indicating the start of the time period for which the file *must* reside on this device (see Section 4.4.2).
avail-end	Timestamp indicating the end of the time period for which the file *must* reside on this device (see Section 4.4.2).
replicate-count	Indicates that this file must be replicated (see Section 4.5.2).

Table 4.2: System-defined annotations.

As an example of the expressiveness possible with this scheme, consider the sample annotations for a file, depicted in Table 4.3. The file is a photograph, presumably taken at a public square in the city of Volos. Using file annotations, it is possible to lookup files with annotation-based queries, such as "show me all files which are photographs, taken in Volos". Of course, by assigning a mnemonic name to the file, a user may also request the file with the traditional method of supplying the filename "VolosNewYearPhoto2006.jpg".

Key	*Value*
mnemonic-name	VolosNewYearPhoto2006.jpg
format	jpg
type	photograph
city	Volos
location	Ag. Nikolaos square
date	2006.01.01
temperature	5C

Table 4.3: Annotated file example (a photograph).

4.3.3 File access model

OmniStore employs the write-once-read-many (WORM) model, as introduced in the Cedar system [SGN85, Hag87]. In other words, a file remains *immutable* once created. Specifically, while it is possible to alter the contents of a file, this is done by creating a new copy of it, with its own globally-unique identifier. To this end, when a file is opened in read-write mode (Listing 4.1), the device library creates a new copy on-the-fly and the returned file handle refers to the copy. The copy also inherits the annotations of the original file.

File annotations are frozen when a file is closed. They may only be edited by applications if the file is opened in read-write mode, thus creating a new revision of the file, derived from the original. Therefore, even the annotations of files with the same identifier are guaranteed to be identical.

To maintain file history, OmniStore creates system annotations. It defines the reserved `derived-from` key, whose value is the file's immediate ancestor (i.e. the identifier of the file which was opened in read-write mode, subsequently creating the new revision).

It is possible that an application may open a file in read-write mode in order to edit its annotations *only*. When a file is opened in read-write mode, the device library keeps track of whether file contents or annotations were actually modified. If the file is closed with no modifications to its data or annotations, the copy is deleted. If only file annotations were modified, a second system annotation using the reserved `identical-content` key is used to hold the identifier of the file whose content is identical to the new file. This annotation is used to optimize both storage usage as well as the automated archival process (Section 4.4.1).

We point out that the `derived-from` and `identical-content` annotations will not necessarily hold the same value. The former will always be set to the file's immediate ancestor. The latter however may point to an ancestor further up the revision hierarchy. Specifically, this occurs when a file is successively edited more than once, making changes to annotations *only* (Figure 4.4).

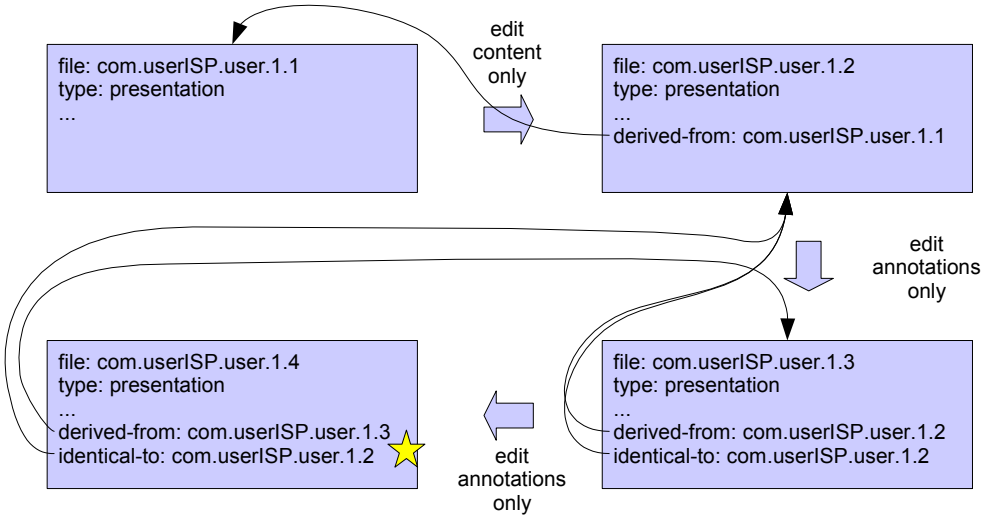


Figure 4.4: Maintaining file revision history

4.4 Infrastructure-based functionality

The repository manages and provides access to the user's reliable storage service in the infrastructure. It serves as a data collection point, where all files created by the user's portables are collected. In addition, it allows applications to interact with storage on portables in an indirect way, by submitting requests to send files from the repository to specific devices. Finally, it supports semantic-based lookup of files, using the annotations attached to them. We next discuss each of these

services in detail.

4.4.1 Automated archival

With storage density increasing and storage costs dropping, it is increasingly being advocated that *deep archival* [SFH+99] be employed for personal computing systems storage. Deep archival postulates that all user files be kept in storage, without ever being deleted, an approach which combines well with semantically organized file-systems, as discussed in [MTX03]. OmniStore extends the deep archival property across all devices owned by the user. We achieve this by having device daemons transfer all files created on portables to the data repository.

Device storage daemons maintain a backup queue of files which have not been transported to the repository. This queue is populated behind the scenes by the device library, when applications create new files. Specifically, as part of the `close()` operation (when the file's content and annotations are frozen), the device library will append the file being closed to the backup queue if either: (a) it is a newly created file, or (b) if it was a file resulting from an `open()` in read-write mode and its content and/or annotations have been modified.

A system annotation identified by the key `dirty` and the value `true` (Table 4.2) is added to unarchived files. Once a file has been backed up, the `dirty` annotation is set to `false`, to prevent the file from being re-appended to the backup queue. The `dirty` annotation is used to persist the backup queue even when devices are powered off: the daemon scans the device's storage medium during initialization, in order to find files in which it is set to `true`, recreating the backup queue accordingly.

The storage daemon periodically (Section 4.4.3) tries to contact the repository. When a connection is available, backup queue entries are processed as follows (Figure 4.5): First, the device sends the file's annotations[4] to the repository, which thus records the file's existence, its annotations and size, along with the fact that its entire content is missing. The repository then sends a list of offset-length pairs to indicate missing file content . The device uploads the requested fragments and repeatedly asks for others. When the repository responds that no more fragments are missing the device sets the file's `dirty` annotation to false and dequeues it. Other portables having copies of this file are guaranteed to eventually mark them as clean when they contact the repository.

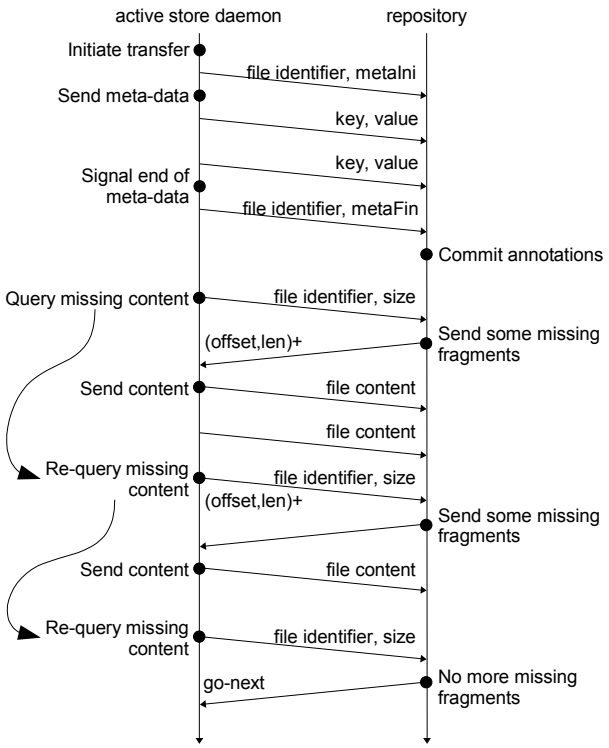


Figure 4.5: OmniStore backup protocol

[4]Device-local system annotations such as `dirty` and `replicate-count` are not uploaded.

This protocol makes it possible to progressively backup files, using opportunistic intermittent connections. In addition, backup can be performed collaboratively by multiple devices that have copies of the same file[5]. Notably, if two devices concurrently try to upload the the same file, the repository will distribute differing offset-length pairs to each of them.

4.4.2 Push-caching

The repository provides a service for sending files to portables which we refer to as *push-caching*. Clients may connect to the repository and submit push-caching requests, stating the desired file and target device identifiers (this is done using the repository library, as discussed in detail in Section 4.4.4).

When a device daemon contacts the repository, it is notified of pending push-caching transfers and adds corresponding entries to its *fetch queue*. This queue is processed in a similar manner to the backup queue (Section 4.4.1), resulting in the respective files being downloaded from the repository. Notably, backup is performed before push-caching. The file transfer protocol used is analogous to the backup protocol (Figure 4.5) and tolerates intermittent connectivity.

Push-caching requests can optionally be submitted with a *schedule*: a starting and ending point in time during which the file should reside on the target device. When the device daemon processes a push-caching request with a schedule attached to it, it records the desired availability time span (as received from the repository) using two system annotations with the keys `avail-start` and `avail-end` (Table 4.2). This allows for some flexibility in sending the file to

[5]File copies may arise automatically via the system's mechanisms (see replication in Section 4.5.2) or by explicit user requests.

the device, as download can be deferred if the desired time frame is far into the future. In addition, such files are excluded from the storage reclaim process (Section 4.5.2) until the end of the scheduled caching period, preventing accidental deletion.

Push-caching requests can optionally be submitted with the *live-update* option enabled, which instructs the repository to send the *latest version* of a file to the device. The repository is able to track the latest revision using the *derived-from* annotation. Live-update requests accompanied by a schedule are automatically removed when they expire. If no schedule exists, the repository updates the device indefinitely, until the push-caching request is removed. Notably, device-daemons are unaware of the live-update option. The repository alone determines if an updated file is to be sent to a device and enqueues a new push-caching request for it, when it is contacted by the respective device daemon. To the device daemon this simply appears to be a new push-caching request.

Push-caching allows any client application to place files on a specific devices. The optional scheduling feature allows one to specify the caching period in a very conscious way, depending on the expected usage of a certain file. Additional versatility is provided when combined with the live-update option, allowing one to request the placement of *future* revisions of a file to a device. It should be noted that the target device does not have to be reachable when the user (or application) submits such requests. Evidently, one can not guarantee that scheduled transfers will succeed, since portables may not be able to contact the repository in time. Having said that, this facility provides a convenient way for users to manage (future) data transfer operations, without the need to have the target device, nor the file to be transferred, present at the time.

4.4.3 The synchronization process

Synchronization of a device's contents with the repository occurs periodically, as specified by the `sync-period` parameter (Section 4.2.2), with the help of network access gateways. A passive discovery request for such services is registered by the device daemon for as long as the device is running, so that the process is not delayed unnecessarily by having to search for gateways. However, if none are known, active discovery is enabled to start probing the PAN. As soon as a gateway is found, the repository is contacted to perform synchronization, which may progress intermittently using multiple gateways (e.g. because the user is on the move).

Whenever a gateway is available, the following actions take place (assume the initial state shown in Figure 4.6): First, the fetch queue is populated, using file identifiers from registered push-cache requests targeting the device (Figure 4.7). Then, the backup queue is processed to archive files (Figure 4.8). Finally, the files in the fetch queue are retrieved and cached locally (Figure 4.9). As soon as this process completes without interruption, the daemon reverts to passive discovery of access points and schedules the next occurrence for `sync-period` time later.

Some optimizations are used to avoid unnecessary file transfers. Specifically, the `identical-content` annotation is used to avoid transferring the content of files that are new revisions of already archived files, which only differ in their annotations. As soon as the annotations are received, the repository checks to see whether the ancestor file has already been backed up and instructs the device to proceed with the next file. If this is not the case, the transfer takes place normally. This same check is performed when fetching a file from the repository, to avoid

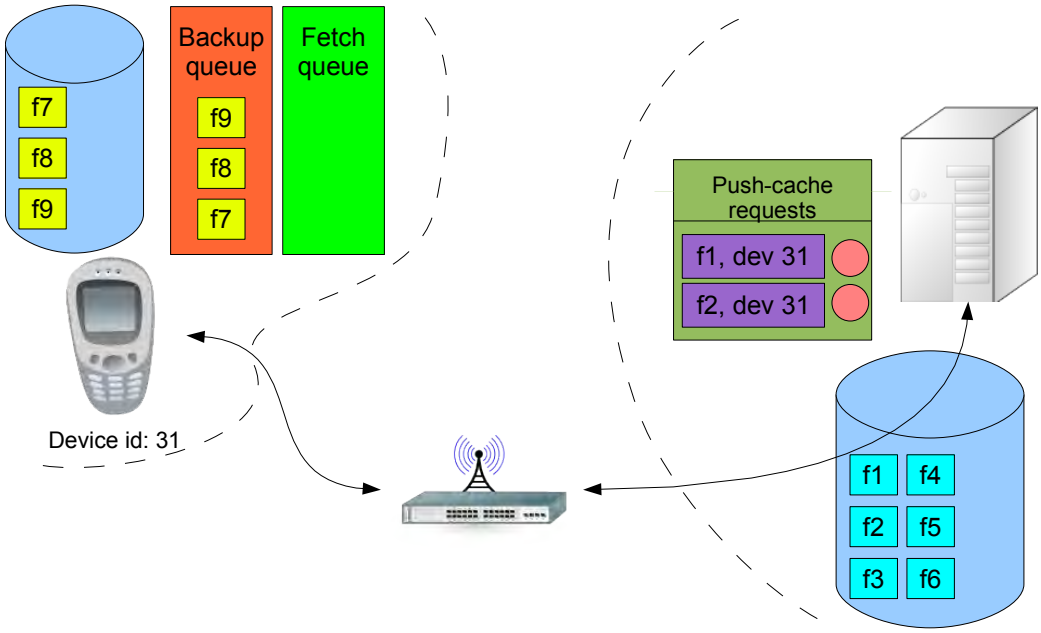


Figure 4.6: Device - Repository synchronization process, initial state

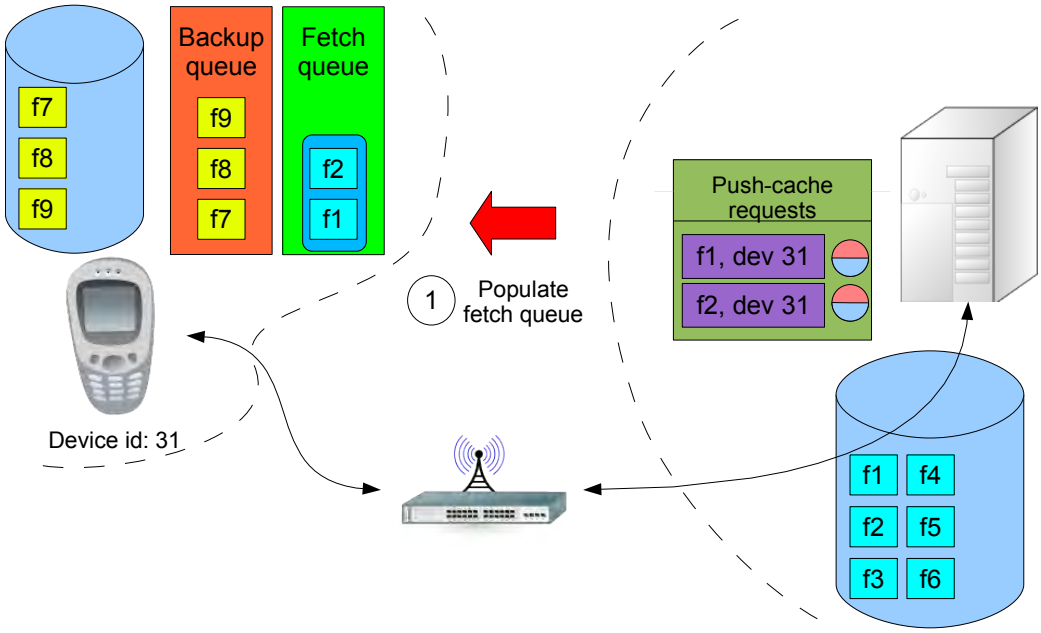


Figure 4.7: Device - Repository synchronization process, first step

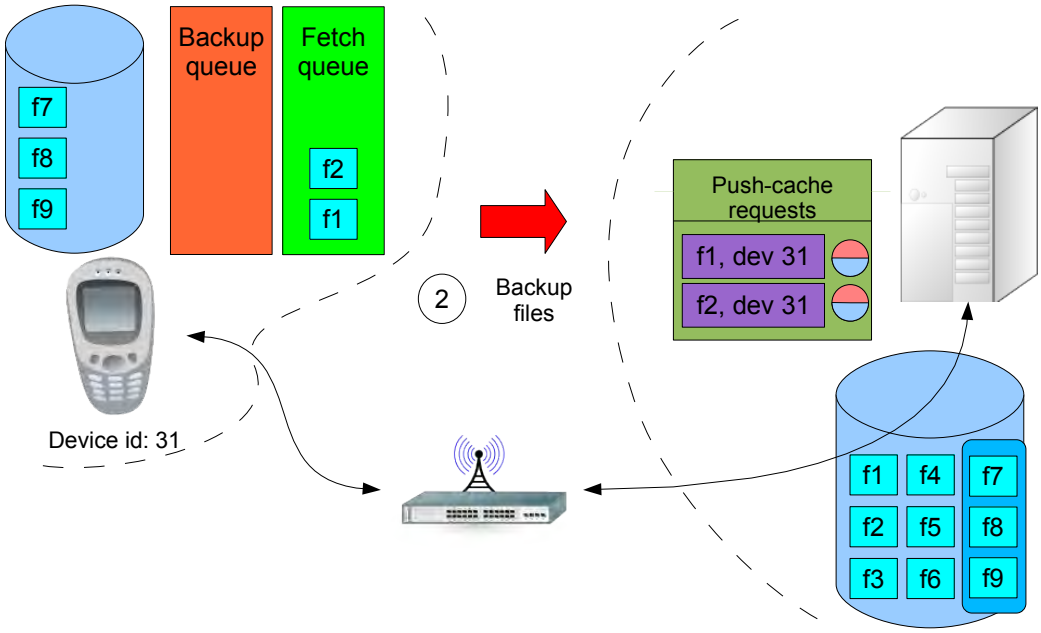


Figure 4.8: Device - Repository synchronization process, second step

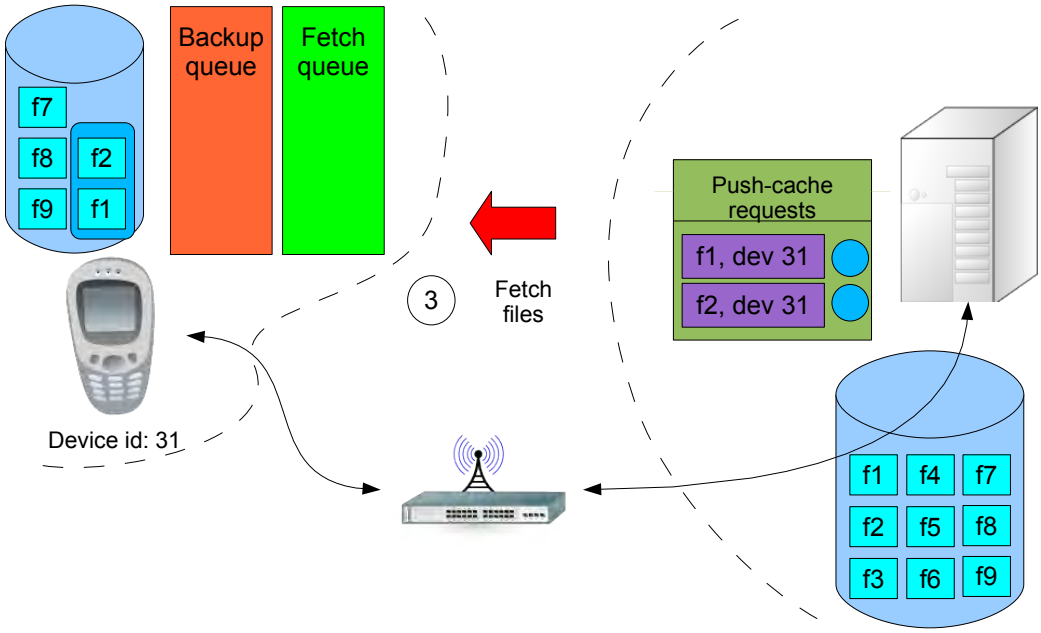


Figure 4.9: Device - Repository synchronization process, third step

superfluous transfers in case the file being downloaded has the same content as a previous version that already exists on the device.

Finally, an interesting optimization is that files in the fetch queue are not processed in a strict FIFO order. Instead, two passes are made on the fetch queue. In the first, files which were created on currently nearby devices[6] are skipped. These files are processed on the second pass made on the fetch queue. As a result, if the gateway becomes unavailable, there exists a chance that these files can be retrieved from the nearby devices that created them. This occurs by contacting the respective devices (which can be done even when no gateways are available) to make local copies of the files if they are found there. The devices are contacted in reverse "co-location probability" order, which is determined by the product of the average number of encounters and mean encounter duration (the smallest product has the least co-location probability). This reduces the likelihood that the device may dissappear prior to the file being fetched.

4.4.4 Application services

Applications may access repository services using the respective library, which wraps a convenient API (Listing 4.2) around the protocol via which these are made available.

File queries are submitted using the `lookupFiles()` method, which returns the files whose annotations match the search terms provided as a parameter. Queries can be restricted to either the key or the value field. For example, an application may request all files which have an annotation with its key matching the

[6]This is determined by checking the device component of the file's identifier and inspecting the co-location table to see if that device is currently in encounter.

term “location”, or all files which have an annotation with its value matching the term “location”. Queries may also be applied to both key or value, producing a match if any of the two fields matches the search term. For example, an application may request all files which have an annotation with key or value containing the term “location”. A query may use several search terms.

Push-cache requests can be scheduled using the `submitPCR()` method, with the target file and device identifiers supplied as parameters. The client must also specify the time period during which the push-caching should occur, along with whether the device should be updated with future revisions of the file.

4.5 Personal area network functionality

Given the ability of portable devices to communicate by means of ad-hoc networking, it is possible to have them collaborate among themselves to increase their utility and improve the functionality provided. OmniStore takes advantage of the ad-hoc networking capabilities in several ways:

- To automatically annotate files with semantic information from multiple sources.
- To collaboratively manage device contents by transferring files from one device to another for increased availability or to distribute storage load and reclaim used space.
- To support remote file access enabling applications to perform I/O operations on files anywhere in the PAN.

4.5.1 Context-based annotation

Cognitive psychology has identified that people often recall significant events contextually. For instance, we may often fail to recall a person's name even though we may easily recall the time or place where we met that person, what we were doing there or even what the weather was like at the time. In other words, the situation under which an event occurs plays an important role in our ability to recollect that event.

This observation also extends to file management [SG03]. Consider taking a photograph with a camera under the following circumstances: it is a hot, but rather cloudy day, on which the user is outdoors in some park at the city of Volos, Greece. This situation could be documented through the collection of sensory inputs shown in Figure 4.10. By recording such context information when a file is being created, it becomes possible to locate the file using whatever facts can be recalled regarding the situation at the time.

OmniStore automatically creates annotations using the context sensing capabilities of the devices in the PAN in order to facilitate browsing and lookup operations. The annotations are derived from the tuples maintained by our runtime's context component (Section 3.5). The goal is to capture the contextual situation, as recorded by the devices present in the PAN, at the time when files are created. It should be pointed out that in this manner. The annotation process (depicted in Figure 4.10) is as follows. When a new file is being created, via the `create` operation, the device library retrieves the entire tuple space contents of the runtime's context component. The key-value pairs of the context tuples are then used to create an annotation list. If multiple tuples have the same key *and* value, they

are merged into a single annotation. Tuples with the same key but different values are kept as separate annotations.

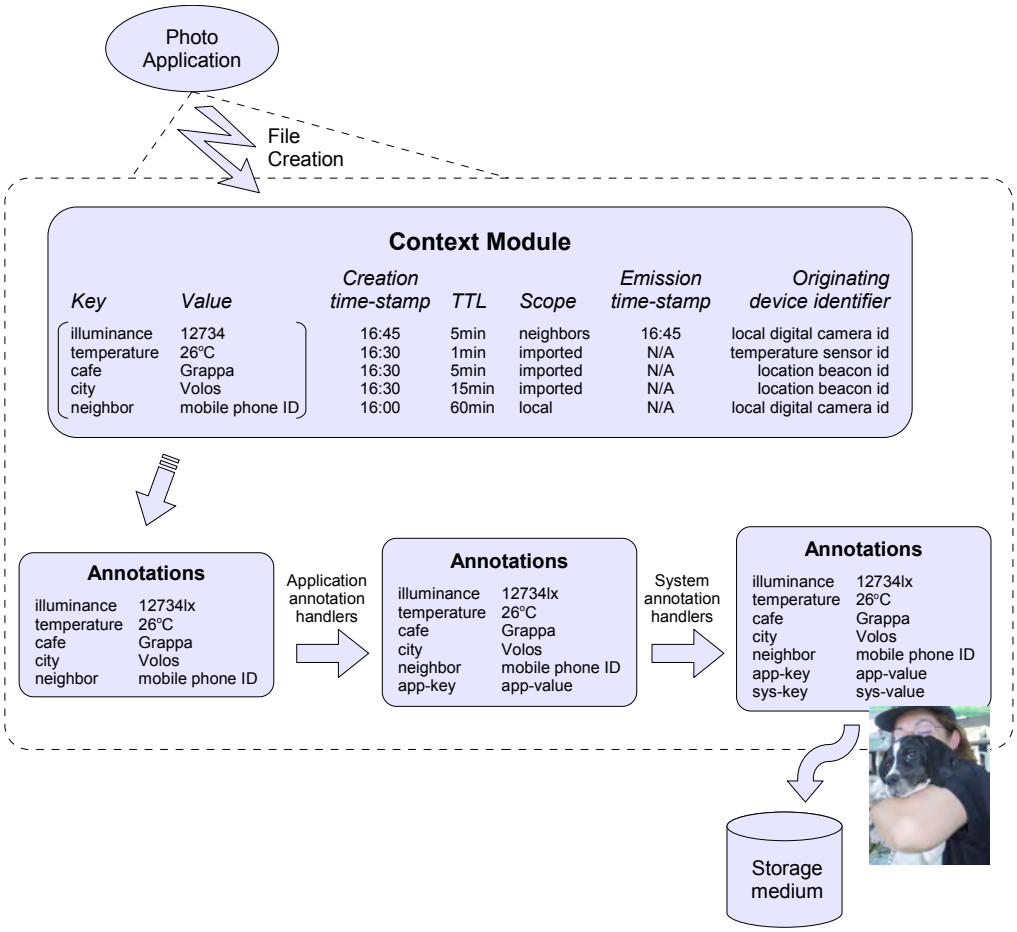


Figure 4.10: File annotation process

For increased flexibility, the meta-information attached to a file need not be an exact projection of the current contextual information assembled by the context component. Application logic can filter annotations, deducing new entries (e.g. inferring higher-level context from the raw context data gathered) and removing others. To achieve the desirable flexibility, applications are given control over the annotation process so that they can add new entries, alter values of existing

entries, or remove entries of the annotation list.

In terms of programming, this is accomplished via annotation handlers that can be dynamically installed in the storage component. These are up-called right after the default annotation process completes, taking as an in-out parameter the annotation list which can be modified as needed. As an example, Listing 4.3 shows the code of a handler which prevents files from being decorated with `location` annotations. This facility can also be exploited by the system itself to process the annotations to be added to the file, by installing system annotation handlers. Obviously, the user (or application) is free to inspect and modify file annotations at any point in time after the file has been created.

4.5.2 Off-loading and replication

OmniStore automatically creates free space on devices whose storage is filling up and replicates important files for increased availability. We discuss these features in the following.

Storage space reclamation is driven by two configuration settings of the device daemon (Section 4.2.2): the minimum free space ratio (MFSR) and desired free space ratio (DFSR), which are stored in the device registry as `min-free-ratio` and `desired-free-ratio` respectively. When the free space on a device drops below its MFSR, garbage collection is activated in order to remove local files. Candidates for deletion are the least recently accessed files that (i) have been successfully backed up in the repository and (ii) are not expected to be available on the local device in the near future. The former can be determined by inspecting the `dirty` annotation whereas the latter can be determined by in-

specting the `avail-start` and `avail-end` annotations (Section 4.3.2). The garbage collection process stops when the DFSR is reached, or when no more space can be recovered.

Though unlikely, it is possible that at some point during garbage collection, prior to reaching the DFSR, that none of the conditions hold for any (more) files, making it impossible to meet the DFSR. As a last-resort option a portable may still delete a file, but must first create a copy on another device, which then assumes responsibility for backing it up to the repository. The off-loading protocol is almost identical to the repository backup protocol (Figure 4.5) with minor deviations. Just as in the case of repository backup, the communicating devices may go out of range before transfer completes. Because partially downloaded files are included as storage reclaim candidates, they will eventually be deleted by the device. In any case, should the devices meet again prior to the incomplete file being deleted, the protocol allows file transfer to resume. To prevent data loss, a sender will not delete a file until *after* it has been completely off-loaded to another device.

A related activity which occurs among portables is the replication of important files. To activate replication for a file, the `replicate-count` system annotation must be added to it, with an integer value indicating the number of desired replicas. Applications on portables may freely set this annotation when they decide that some file is of particular importance. This annotation indirectly prompts the device daemon to create additional copies of the file on other devices in the PAN. The `replicas-made` system annotation is used to keep track of the number of replicas created. Again, as with off-loading, the same file transfer protocol is used, only this time the file is not deleted after the transfer.

A usage example of our replication facility can be found in the OmniStore

system itself. The device daemon adds this annotation to files received due to push-cache requests which have a schedule (Section 4.4.2), in order to decrease the probability of unavailability. The replication of files, combined with the support for transparent failover when accessing remote files (Section 4.5.3), comprise a significant fault-tolerance system for mobile ad-hoc environments.

It should be noted that, in both the case of replication and off-loading, the device daemon does *not* randomly choose devices in the PAN on which to transfer files. Instead, it refers to the co-location history statistics maintained by the underlying runtime (Section 3.2), allowing it to intelligently select target devices. Specifically, devices with the highest probability of being co-located with the device from which the file originates are selected (the ones for whom the product of the average number of encounters and mean encounter duration is largest). This increases the likelihood that the file will still be accessible from the applications of the original device.

4.5.3 Distributed lookup and access

File lookup in semantic file-systems is not restricted to the use of file names; any annotation may be used. The device library allows applications to create *annotation lookup tasks* which are registered with the device daemon. Lookup tasks are populated with search results using local *and* remote files from nearby devices in the PAN. Lookup can use specific keys, or generic terms. Listing 4.4 shows an application that creates a task, matching files that are photographs (the key-value pair [file type, photograph] exists) and are also annotated with a key or value that contains the term `cafeteria` (e.g. both [location, Grappa cafeteria]

and [cafeteria, Jam] would match).

Device daemons are notified of the arrival and departure of nearby storage devices by the core runtime. New devices are inquired about files matching the search terms defined for registered lookup tasks. When a device leaves the PAN, matches from it are removed. Applications may register a notification listener to be called when the active result set changes. The application can then review matching file identifiers, along with the device on which they reside. This process is depicted in Figure 4.11, where a projector device uses the system to locate nearby presentations.

File access operations are similar with those of typical file-systems. An exception is that data locality is required for altering files. Opening a remote file in read-write mode results in a copy being created on the local device, using a locally-generated file identifier. Conversely, read-only access is allowed directly from remote devices. A notable feature for files opened in read-only mode is that – should one device used to access the file suddenly become unavailable – the device library can transparently fail-over to another device which carries that file. This is made possible due to the employed WORM model and naming scheme, which guarantee that files with the same file identifier necessarily have identical content and annotations. If a file is present on multiple devices in the PAN, all of its providers will match lookup tasks used to locate it (since the files necessarily have identical annotations). Therefore, the device daemon will be aware of all sources in the PAN from which to access the file. The device library thus switches to a different provider whenever a device used to access the file becomes unavailable (Figure 4.11).

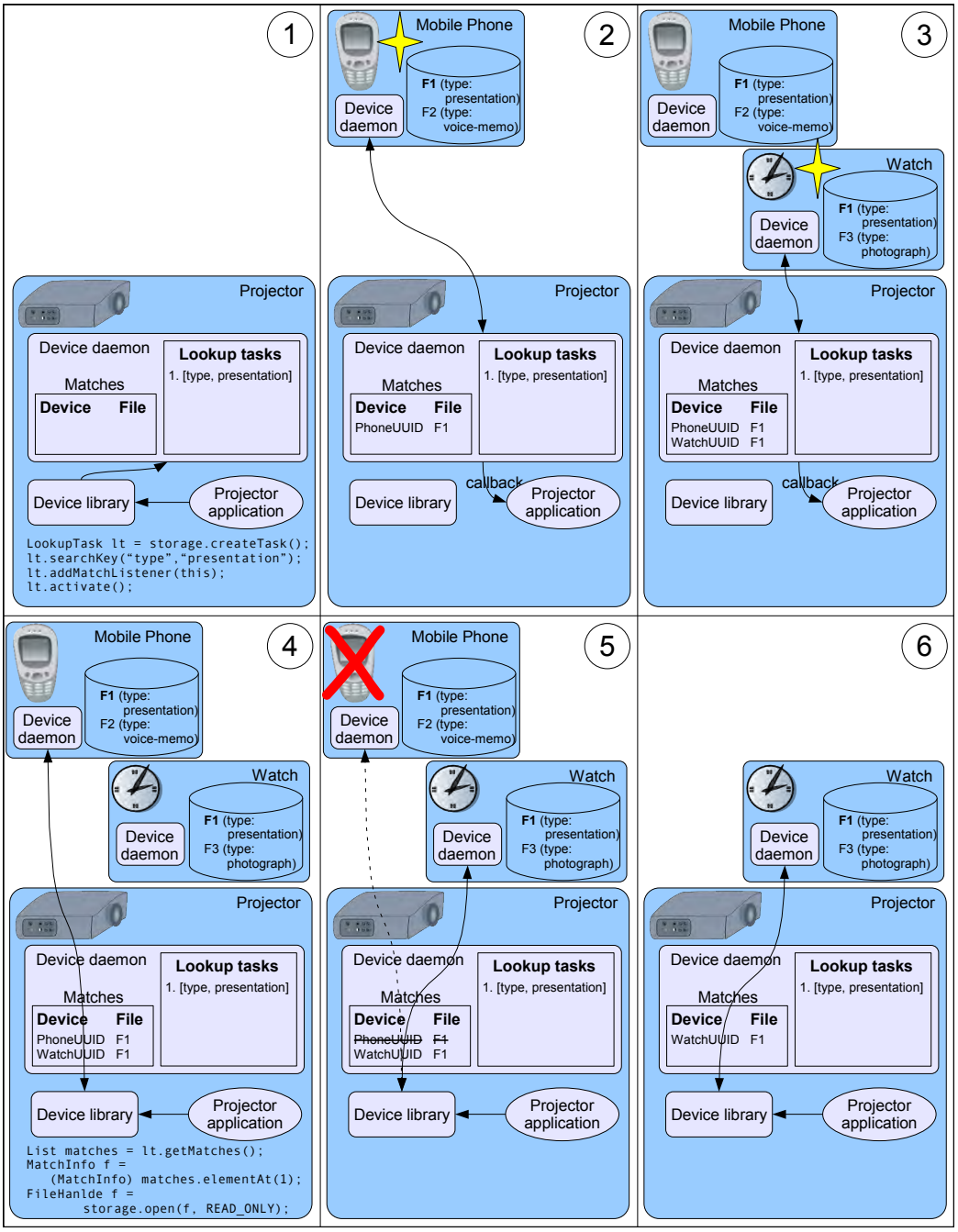


Figure 4.11: Locating files in the PAN

4.6 Security aspects

This work's focus does not include the security of ubiquitous computing systems. In the interest of completeness however, we provide some basic protection mechanisms, to demonstrate that the risk of having data compromised when using OmniStore is reasonable (i.e. within current user expectations).

The cornerstone of our security system is the device registry, which issues the certificates for the devices owned by the user (Section 4.2.1), also providing its public key to devices so that they may validate any certificate issued by the same registry. To establish trust, devices must present their certificates[7] to each other, checking to see if they were indeed issued by the same registry.

As a somewhat extreme measure, we perform this validation in every connection among devices, refusing to interact with devices that do not belong to the same user. Devices will thus only interact with other devices that the user owns, confining data access to the user's domain.

This scheme can be extended to implement functionality such as providing limited access to specific files from other users' devices, revoking certificates of compromised devices, etc. However, designing a complete security system is an open research issue that can be better addressed by the respective community.

[7] To prevent a malicious party from playing back the certificate, a device should not provide its certificate for validation in cleartext, but rather a signed (by the device) copy with a session token that will prevent its reuse.

```
public interface IStorage {
    // local file access methods only
    public FileHandle create();
    public FileHandle open(
        String domain, int devId,   int fileId,
        boolean readWrite) ;
    // ...
}

public class FileHandle {
    // access file content
    public void seek(long ofs);
    public int read(byte[] buffer, int ofs, int len);
    public int write(byte[] buffer, int ofs, int len);
    public void close();

    // access file annotations
    public void addAnnotation(String key, String value);
    public void removeAnnotation(String key, String value);
    public void updateAnnotation(String key,
        String oldValue, String newValue);
    public List getAllKeyValues(String key);
    public List listKeys();

    // obtain file identifier info
    public String getRegistryURL();
    public int getDeviceId();
    public int getFileId();
}
```

Listing 4.1: The device library's file access API

```
public class RepositoryClient {
    public RepositoryClient(String dnsAddress, int port);

    // "wild characters" for lookupFiles() method
    public static final int ALL_SCOPE = 0,
        KEY_SCOPE = 1, VALUE_SCOPE = 2;
    // "wild characters" for getPCRIds() method
    public static final int ANY_DEVICE = -1;
    public static final String ANY_FILE = "file://";
    // constants for statusMask in getPCRIds()
    public static final int DEVICE_UNAWARE = 1,
        DEVICE_NOTIFIED = 2, DEVICE_UPTODATE = 4,
        DEVICE_HAS_OLDER_REVISION = 8;

    // file lookup
    public List lookupFiles(String[] terms, int scope);
    // submit push-cache request -- returns PCR id
    public int submitPCR(String fileId, int devId,
        Date start, Date stop, boolean liveUpdate);
    // locate PCRs
    public int getPCRIds(String fileId, int devId,
        int statusMask);
    // review specific PCR status
    public int get PCRStatus(int PCRId);
    // cancel a pending PCR
    public int cancelPCR(int PCRId);
}
```

Listing 4.2: The repository library's API

```
public void annotationHandler (List annotationList) {
    Iterator it = annotationList.iterator();
    while (it.hasNext()) {
        // iterate through the
        // "to be attached" annotations
        Annotation an = (Annotation) it.next();
        if (an.getKey().equals("location"))
            it.remove(); // remove "neighbor" keys
    }
}
```

Listing 4.3: A sample file annotation handler

```
LookupTask lt = storage.createTask();
lt.searchKey ("file␣type", "photograph");
lt.searchTerm ("cafeteria");
lt.addMatchListener(this);
lt.activate();
```

Listing 4.4: A sample file lookup task

Chapter 5

Evaluation

5.1 Usage

The laboratory setup used to test the system is as follows: The repository and registry daemons are both[1] installed on a server with a wired connection to the Internet. The Internet access daemon is installed on several PCs equipped with both wired and wireless adapters, acting as network access points. PDAs are used as portable devices, running different applications that simulate more purposeful devices such as music players, digital cameras, digital photo frames and mobile phones.

We have created two applications for demonstrating access to the repository and registry services from hosts in the infrastructure. The registry management application (Section 5.1.2) is a web application for accessing device registry services from anywhere, using a web browser. The repository management application (Section 5.1.3) is its equivalent for repository services.

[1] It is not a requirement that the registry and repository daemons be co-hosted.

In addition, we have created some mock-up devices (Section 5.1.1). The mobile phone, digital camera, and digital photo frame have OmniStore's storage component installed and are used to demonstrate the system's functionality. The digital context provision device (Section 5.1.1) is intended to be placed in the infrastructure as a source of contextual information for devices.

We next discuss each of these elements.

5.1.1 Mock-up devices

Mobile phone

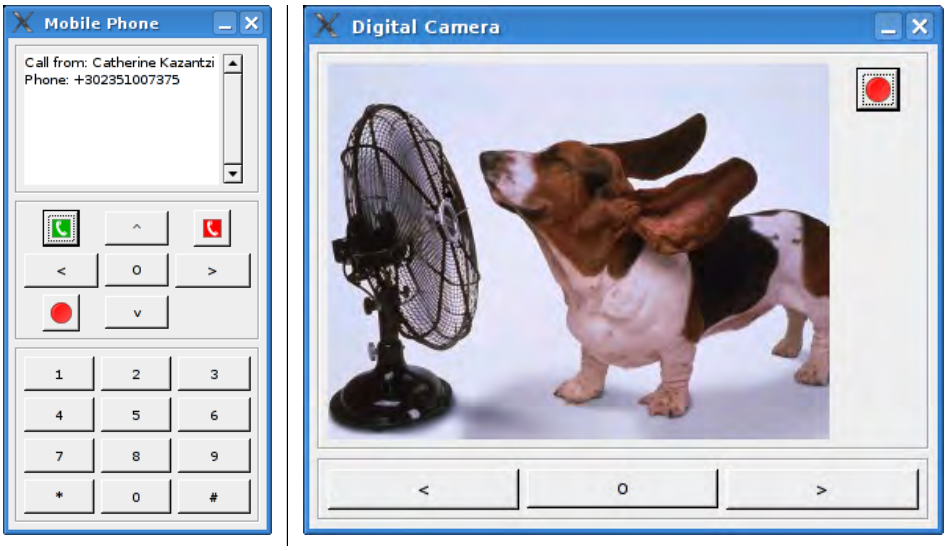
This program displays a GUI resembling a mobile phone device and is installed on PDAs to experiment with OmniStore usage scenarios. The standard pick up, hang up and dialing number buttons are displayed, along with a record button which allows one to record voice-memos or phone-calls.

When the pickup button is pressed, the device's state switches to "call in progress" mode, in which a random phone number is selected from the phone book and used to publish local entries in the context tuple space (Section 3.5.1). The hangup button clears the state returning the phone to idle mode and removing those context tuples. Figure 5.1 shows the phone with a call in progress.

When the record button is pressed, an audio file is created by our application, which undergoes all of OmniStore's processing operations: automated annotation with context information (Section 4.5.1) and asynchronous archival to the repository (Section 4.4.1).

As a result of the local context tuples that contain calling party information, recordings are annotated with the name and phone number of the person one

is presumably talking to. The phone will furthermore annotate the file with a `type` key whose value is set to `voice-memo recording` or `phone-call recording`, depending on whether or not a call was in progress when the recording was made. Additional information (such as location information from the context provision device described in Section 5.1.1) may be attached to the file, depending on the proximity of the mobile phone to other devices.



(a) (b)

Figure 5.1: Mobile phone (left) and digital camera (right) device

Digital camera

The digital camera application displays a GUI resembling such a device and can be installed on PDAs to demonstrate OmniStore scenarios. Figure 5.1 depicts the GUI.

When the record button is pressed, an image file is created by our application, which undergoes all of OmniStore's processing operations: automated annotation

with context information (Section 4.5.1) and asynchronous archival to the repository (Section 4.4.1). The file is annotated with an illuminance reading taken from the camera's light sensor. Additional information (such as location information from the context provision device described in Section 5.1.1) may be attached to the file, depending on the proximity of the camera PDA to other devices.

Context provision device

The context provision device is an application whose purpose is to generate context tuples. Presumably, such a device would be programmed to advertise some type of static context information and deployed in the infrastructure. For example, it can be configured to advertise room location information. The screenshot in Figure 5.4 shows two recordings created by the mobile phone device, one of which was recorded when the mobile phone PDA was in our laboratory, which has a context provision device advertising location context.

Alternatively, this program can be installed on a PDA and be configured with the identity of a person. It can then be used to simulate an active badge carried by that person.

Digital photo frame

A digital photo frame is a device that is used to decorate our living environments, on which photographs are placed and are used to display a slideshow. The digital photo frame application is installed on a tablet PC which is used to simulate this facility.

There is not much else to present regarding this device, as it is embarrassingly simple. Our intent is to demonstrate that even such a device can benefit from

running OmniStore. As described in detail in the usage scenarios of Section 5.1.4, users can take photographs with the digital camera (Section 5.1.1) and later use our repository management application (Section 5.1.2) to lookup and push photos onto the frame, altering the slideshow remotely.

5.1.2 Registry management application

The registry management application is a set of Java server pages (JSP) and servlets using the registry library to provide a web-based interface to the registry's services. This makes it possible to interact with the registry from any Internet-connected host, with the help of a web browser.

The main purpose of the registry management application is to handle device registration requests (Section 4.2.1). As discussed in Section 4.6, new devices must be registered in order to be recognized by the rest of the user's devices. Figure 5.2 shows the registry management application listing pending registration requests for three devices, which have completed the first stage of registration and are waiting for their requests to be approved. The user may accept them to complete the process, or reject them to abort it.

Figure 5.3 shows a sample of the registry management's applications output, when the user requests to see a list of registered devices: a digital camera, a mobile phone and a context provision device. Notably, the repository is shown in the registry's device list, as it is itself a "home appliance" purchased by the user. It has both Internet and ad-hoc networking adapters and may thus interact with other devices in an ad-hoc fashion when the user keeps them at home. Therefore it has been registered as well.

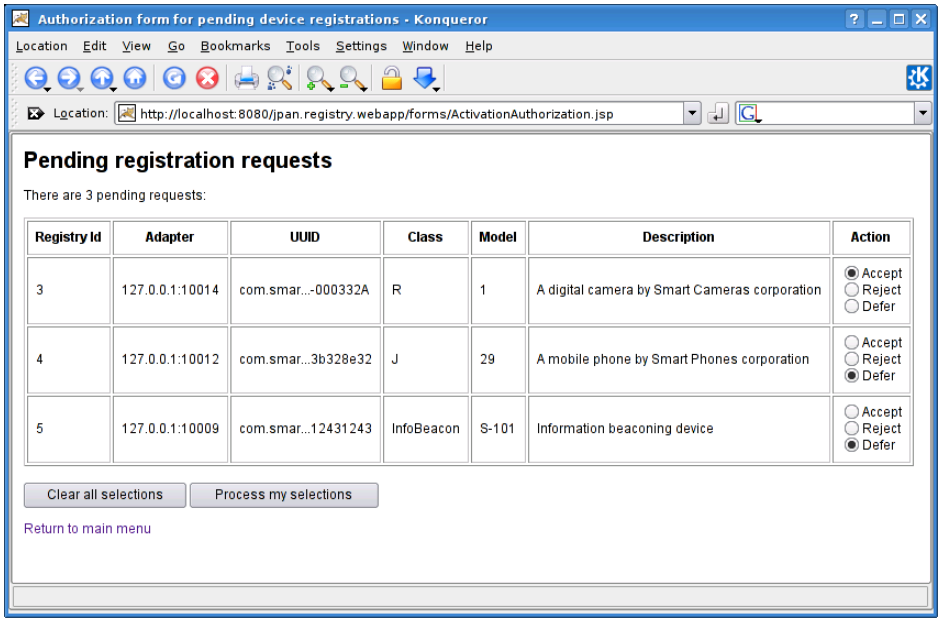


Figure 5.2: Registry management – pending registration requests

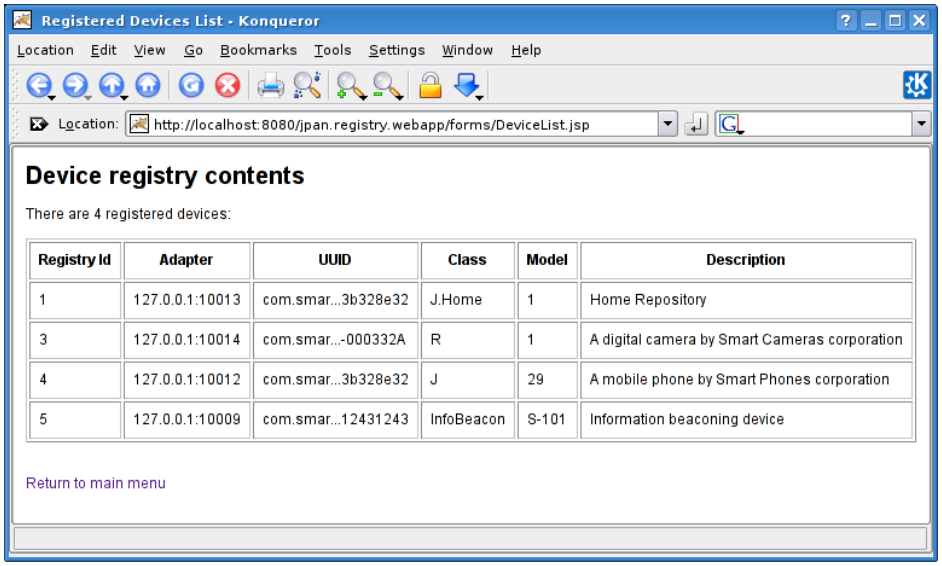


Figure 5.3: Registry management – listing registered devices

5.1.3 Repository management application

The repository management application is a set of Java server pages (JSP) and servlets built using the repository library, in order to make repository services accessible to users. The operations exported are annotation-based lookup and retrieval of files, as well as managing push-cache requests. They may be performed using any Internet-connected host with a web browser.

The web form of the repository management application for looking up files can receive any number search terms, optionally restricting the search operation to the `key` or `value` fields of annotations. Figure 5.4 shows the results of searching for the term `%recording%` (the `%` character is a wild-character) among the repository's files. The user may adjust the searh terms and resubmit the request, or double-click on a file to download it.

In this example, we have previously created two sample recordings using our mobile phone device (Section 5.1.1). One of the recordings was created when the mobile phone was near our context provision device (Section 5.1.1) which is placed in the laboratory. That file is thus annotated with the relevant location information.

A useful feature of the web-based application is that the user may at any time lookup a file and request that it be push-cached to a specific device. The device need not be carried by the user, or even turned on, at the time the request is made. Figure 5.5 shows the web form used to submit such requests. In the example, a phone call recording was selected to be sent to the mobile phone, for a certain period of time. The push-cache request's parameters are entered in the panel on the left. The panel on the right is used to lookup device identifiers and files,

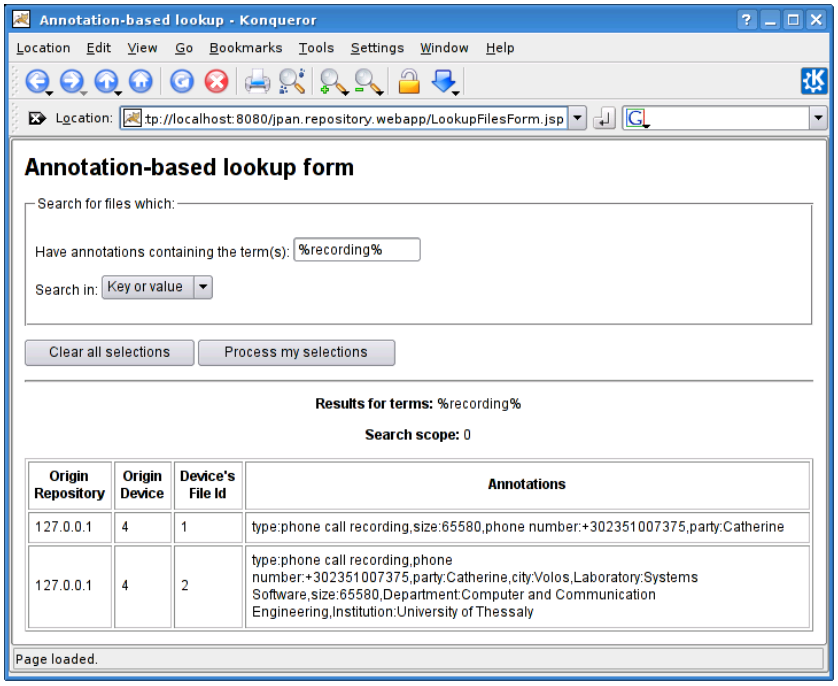


Figure 5.4: Repository management – annotation-based lookup

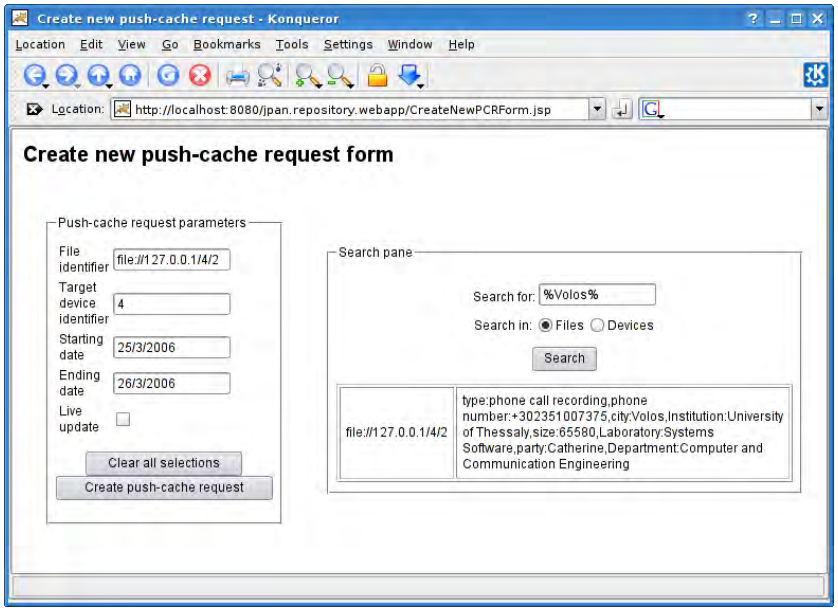


Figure 5.5: Repository management – creating a push cache request

using human-readable terms such as “phone” or “recording”. Double-clicking on a search result transfers the value to the respective field of the push-cache request’s data entry panel.

5.1.4 Usage scenarios

Using the mock-up devices presented in Section 5.1.1, we can demonstrate some interesting usage scenarios. In the following, we attempt to narratively present the potential of our system, by unfolding a story which can be realized using the mechanisms implemented in OmniStore.

Catherine receives an e-mail from the marketing manager, which informs her that, due to a colleague falling ill, she must travel to Berlin and visit some clients to give a presentation of her company’s new product. The flight’s itinerary, hotel reservation information, presentation schedule and presentation file, are all attached to the e-mail. The groupware suite automatically updates her schedule to reflect that she will be out of office and adds the presentation meeting to her calendar. In addition, it contacts her repository to upload the presentation and schedule a push-cache request[2] targetting her mobile phone, using the travel dates as the caching period and activating the live-update feature (Figure 5.6, steps 1 and 2) .

On the day prior to the meeting, Catherine leaves work to go to the airport. Her phone has already received the presentation file from an access point in her office (Figure 5.6, step 3). While commuting with the airport shuttle, a colleague calls her to discuss some ideas for improving the presentation. She records the conversation so that she may later refer to it (Figure 5.7). While waiting to board

[2]Since we have not written a groupware suite, we must use the repository management application to upload the file and submit the push-cache request.

the plane, an access point in the lounge is used to archive the recording to the repository (Figure 5.8).

In the evening, Catherine is reviewing her presentation on her laptop. She fires up the file browser and searches for the phone-call recording using her colleague's name and "airport shuttle" as search terms. She makes some changes to reflect what was said and goes to bed (Figure 5.9). During the night, her phone is updated with the latest version of the presentation (Figure 5.10) and also makes a replica of it on her watch (Figure 5.11).

Early in the morning, prior to going to the meeting, Catherine strolls around downtown Berlin and takes some photographs. She then proceeds to the client offices on time for her presentation. During the presentation, her digital camera is sending the photographs to the repository (Figure 5.12). Meanwhile, Catherine does not notice that her phone is running out of battery, causing the projector to switch to her watch for reading the subsequent slides (Figure 5.13).

In the afternoon, Catherine relaxes at her hotel room. She logs into the Internet and pushes some photos to the digital frame in the living room. When the frame contacts the repository, it starts downloading the pushed photos (Figure 5.14). During this process, its free space drops dangerously, triggering garbage-collection. Some of the older photos are automatically deleted, in order to make enough space. Some time later, her husband calls and fills her in on the excitement the photos caused when the kids noticed them.

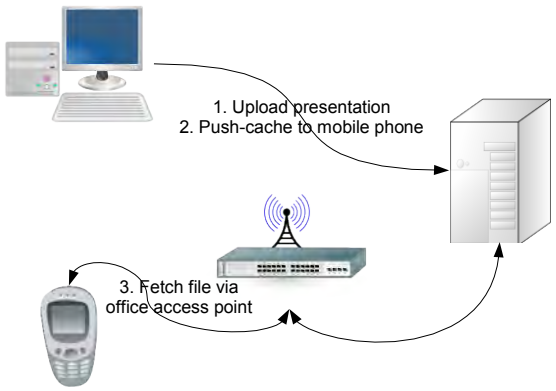
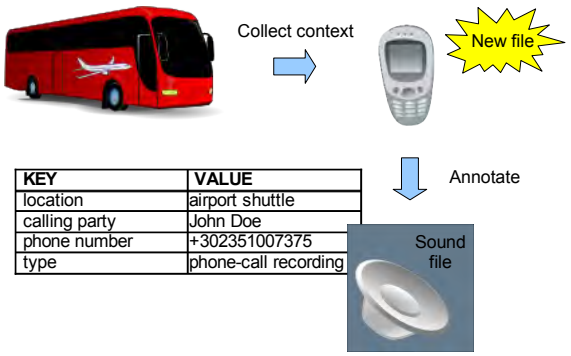


Figure 5.6: Request presentation to be sent to the phone



KEY	VALUE
location	airport shuttle
calling party	John Doe
phone number	+302351007375
type	phone-call recording

Figure 5.7: Annotate phone-call recording with context

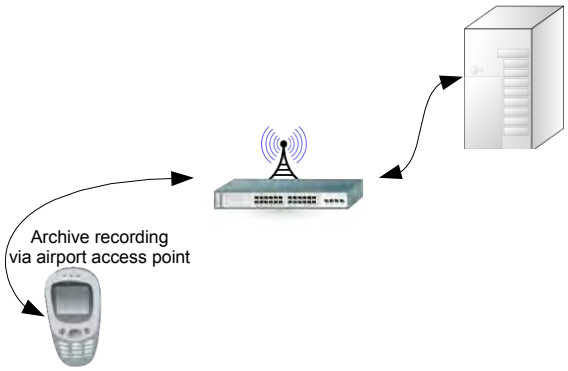


Figure 5.8: Archive phone-call recording via airport access point

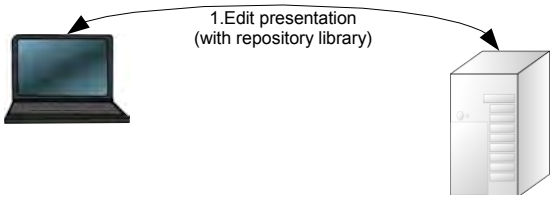


Figure 5.9: Edit the presentation using the laptop

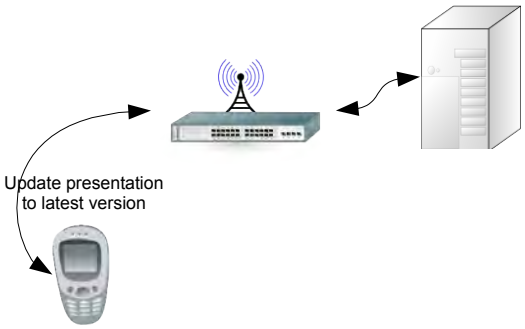


Figure 5.10: Live-update of the presentation on the phone

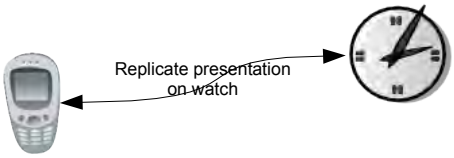


Figure 5.11: Replicate the presentation on the watch

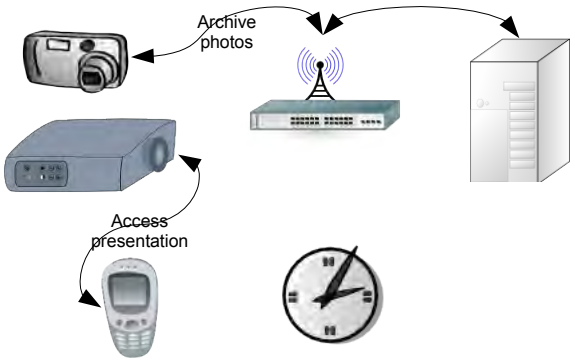


Figure 5.12: OmniStore activity during the presentation

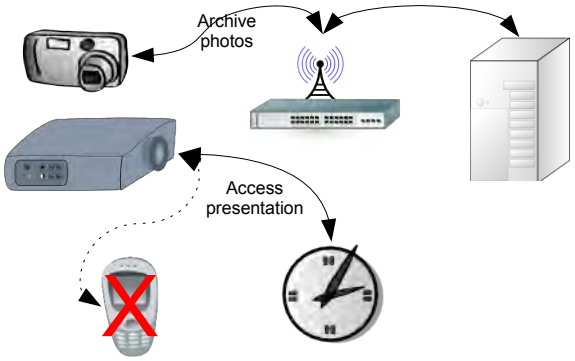


Figure 5.13: Transparent fail-over

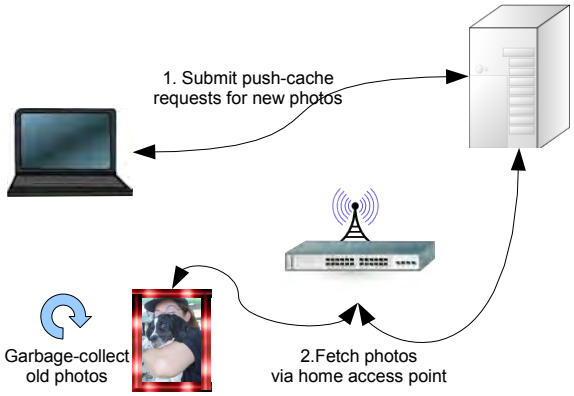


Figure 5.14: Pushing photos to the digital frame

5.2 Performance evaluation

We performed measurements to determine OmniStore's performance regarding execution speed, memory consumption, network protocols and storage access. This Section presents and discusses the results.

Our tests were performed using a PDA, a desktop and a laptop computer. The PDA ran our mock-up devices (Section 5.1.1), whereas the desktop and laptop computers hosted the repository/registry service and the network access point application respectively.

The PDA's processor was a 206MHz StrongARM SA-1110 with 16MB of RAM. The permanent storage device was a PCMCIA CompactFlash EPROM which was the added to the PDA using an appropriate sleeve. The adapter was capable of 1,3MB/sec read and 500KB/sec write throughput (direct device I/O without file-system overhead). All systems were running a JVM on top of Linux.

For technical reasons, we were unable to perform some of the measurements presented using the PDA. Specifically, the execution time and memory consumption figures (Appendix A.1, B.1, C.1) were obtained from the laptop computer. We took some measures to reduce the impact of measuring on a more capable device, by restricting the processor's (AMD Athlon XP-M) clock speed to 500MHz and using the PCMCIA CF card for storage. In addition, we restricted the memory available to the JVM to 16MB.

Evidently, the figures obtained from the laptop are better than those that would have been recorded had we used the PDA. We ran a popular benchmark[3] on the "slowed down" laptop and the PDA, obtaining performance indexes of 12.0 and

[3]The benchmark is written and used by a major-circulation global computer magazine.

3.52 respectively. However, because the PDA is an old model[4], we expect the performance of contemporary PDAs to closely match that of the laptop. Therefore, the execution speed timings are indicative of what can be expected from modern embedded systems.

In any case, we used the system on our old and slow PDA, without experiencing any performance-related usability issues (e.g. lagging interactivity). Thus, absolute performance does not constitute a problem, whereas the relative performance of the various operations can still be deduced from the laptop. Furthermore, memory consumption is not affected by the processor and the measurements should be almost identical for both the laptop and the PDA (the Java byte-code for both systems is produced from the same compiler and both systems use 32-bit word-addressable memory subsystems). Finally, we note that with the exception of these specific measurements, the rest of the tests and results were performed on the PDA.

5.2.1 Core services

Execution time statistics are given in Appendix A.1, whereas memory consumption statistics are given in Appendix B.1. We obtained these measurements by profiling the execution of our code on an information beaconing device. During the test, the (presumably newly purchased) device was first registered with the owner's device registry and then placed in the laboratory. A management application was then used to program the beacon to provide location information. In this section we discuss some focal points of the measurements obtained.

[4]The PDA's processor runs at 206MHz, whereas the latest model in this line runs at 533MHz and also features major architectural improvements.

Execution time

Appendix A.1 lists execution timing data gathered by the runtime. As can be seen in the data, the most heavily used package overall is `gr.jpan.datastructs`, which contains various data structures used throughout the runtime. The methods with the largest number of invocations belong to the linked list's iterator class (`LinkedList$LLIterator`), namely `hasNext()` and `next()`. The reason for this is because the main engine loop processes tasks in a FIFO fashion, which are retrieved from a linked list. Every packet received / sent by our runtime causes such a task to be processed. However, the runtime's services generally use hash maps (`HashMap`) to achieve better performance in searching.

We next focus on the discovery system, the important methods of which (from an application's perspective) are: `registerInterest()`, `registerService()`, `setActiveDiscovery()`, `setActiveAdvertize()` and `getProviders()`. Service registration occurs in roughly 3.3ms, whereas interest registration requires almost twice as much. Turning 'active' mode on in each case requires 0.24ms and 0.46ms respectively. The increased cost in searching for services as opposed to exporting them, is due to the fact that more complex data structures are employed to track application interest. Finally, a list of matching providers for a service can be retrieved with `getProviders()` in just two tenths of a millisecond.

The methods with the most impact on discovery performance are `processReceivedLookup()` and `processReceivedAdvertisement()`. These are called every time a beacon with discovery information is received, to extract and process the discovery data. The system requires 0.21ms to process lookups

and 0.54ms to process advertisements. These operations are based on hash map lookups.

Moving on to context manipulation operations, one can see that posting a new tuple with `post()` requires 0.73ms, whereas retrieving a tuple with `read()` requires 0.3ms. Incoming beacons with context information are processed in just 0.076ms by `processFrame()`. Again, as is the case with service discovery, these operations are backed by hash maps.

Memory consumption

Section B.1 presents memory consumption information. The discovery system uses 344 bytes to manage one advertized service (the beacon management interface) and one tracked service (the device registry which is looked up to perform registration). The context manager occupies 264 bytes, with its three entries accounting for 144 bytes of those. The hash maps and linked lists used to manage this information make up of another 8.5kb. It should be noted that the memory consumption of both service discovery and context management grows linearly, with the number and the length of the relevant strings: the name representing the service tracked / advertized in the first case, or the key and value pair for each context entry in the latter case.

The total active *data* memory footprint of the runtime is 176024 bytes. The runtime's compiled code amounts to 351445 bytes. The total memory usage during execution is thus 527469 bytes (515KB).

Context dissemination

Context dissemination performance depends on the beaconing period used by the device exporting the information. As was discussed in Section 3.1, the runtime uses three beaconing rates: idle, normal and fast. When a device contains new context information (i.e. a new key was posted or the value of an existing key was updated), the fast beaconing rate is used to emit beacons. Assume the period among beacons for a device are T_{idle}, T_{normal} and T_{fast}. Furthermore, assume that M discovery tasks (active lookups and/or advertisements) are registered with the system. If an application posts N tuples immediately after a beacon has just been emitted (which is the worst case scenario), the first of these tuples will be exported in at most $2 * T_{fast}$ time and the last in $(M + N) * T_{fast}$ time.

In the general case however, a device will export beacons using the T_{normal} period. Our trial uses have shown that fast beaconing periods (T_{fast}) can be as large as 2 seconds, whereas normal periods (T_{normal}) can be as large as 10 seconds and still accommodate the real-time requirements of users. For example, using these value, a user entering a room containing an information beaconing device will receive up to 6 context entries within a minute (provided the device maintains the normal emission rate and does not accelerate).

These calculations do not take into account the time required to transmit the beacon, as well as the time required for the packet to travel from one node to another, as these are orders of magnitude smaller than the beaconing periods used. They also do not consider the effect of collisions and packet loss which we discuss next. Beacons are relatively small packets (up to 256 bytes) transmitted at very low rates (6 packets per minute), causing negligible load (a $6*256*8/60 = 205$bps

stream) on the network. A standard 54Mbps WLAN could theoretically accommodate hundreds of thousands of such streams. Practically, any congestion that arises will be due to other (application) traffic in the PAN, or due to unrealistically (at this time) large numbers of nodes in a small area. In a congested network, performance will depend on the type of radio modulation / demodulation and medium access control mechanism employed by the wireless network technology in use. In any case, one should not forget that perfect dissemination of context information is not critical for the operation of our a system.

Service discovery performance

Service discovery performance is similar to that of context dissemination, as it shares the same underlying mechanism. There are two ways to discover a service: by actively polling the PAN (emitting lookup beacons), or by passively monitoring the PAN with the hope of receiving a relevant advertisement. The latter method is preferable for infrastructure services (e.g. network access points) when an asynchronous task which does not require user interaction (e.g. data backup) is to be performed. The former method is preferable when searching for services to perform an interactive (e.g. list files in the PAN for the user) or an urgent (e.g. find a device with free space to offload files) task. We discuss the time required in each case.

Assume an active discovery lookup is submitted at time T_1 and that the beaconing periods for the device are T_{idle}, T_{normal} and T_{fast}. The device will immediately emit a lookup beacon, followed by another one at $T_1 + T_{fast}$ and then more beacons at $T_1 + T_{fast} + I * T_{normal}$, where I is the number of the next beacon to be emitted. Devices in the PAN reply almost immediately, as lookup beacons are

processed in 5ms (see Appendix A.1), which is negligible compared to T_{fast} and T_{normal}. Therefore, a T_{normal} period of around 10 seconds, with a T_{fast} period of around 2 seconds can yield satisfactory interactiveness, as a nearby service will be detected in 2 seconds, or at most in 10 seconds after the user has performed some action (moved to another location / turned on a device), in response to it not being detected[5].

Passive discovery depends on the normal beaconing rate of a device. With a T_{normal} period of 10 seconds, devices carried into a room where the user spends some time, will easily pick up any services within it.

5.2.2 Storage system operations

Local device access

Appendix C.1 lists execution timings related to storage access methods on a portable device. We perform 50 invocations of `read()` and `write()` using a 1KB buffer, which require an average of less than 1ms to process (the storage medium's cache causes the methods to return immediately prior to actually writing the data to the medium). The time required to list all annotation keys with `listKeys()` 37ms, whereas retrieving all values for a specific key with `getAllKeyValues()` costs 6ms. Creating / modifying an annotation with `updateAnnotation()` costs 5ms.

[5]Compare this to the 12.8 seconds in which – as described in the Bluetooth specification [The] – a device is discovered in the PAN, which is considered acceptable.

File transfers

OmniStore performs two types of file transfers: one (from one device to another one in the PAN) occurs strictly within the wireless network, whereas the other uses network gateways and transcends both the wireless network and the infrastructure network (from a portable device to the repository or vice-versa). The protocol used to transfer the data is the same. However, when transferring files from / to the repository, the transfer rate is affected by network conditions among the network access point and the repository server. We therefore installed the infrastructure server and network gateway in the same isolated 100MBps LAN, to ensure that outside traffic does not affect our measurements. The wireless network used was an 11MBps WLAN in ad-hoc mode. Our testbed consists of two PDAs, a network gateway and a server running the repository and registry services.

We had the device perform file backups using the gateway. Four files – each 256KB in size, totaling 1MB of data – along with 6 annotations per file, were sent to the repository. The transfer's duration was 6525ms, which translates into a rate of 156.935KB/sec. We transferred those same files from the PDA to the repository using the UNIX rcp command (without their annotations) in 4233ms; this translates into a 241.909KB/sec throughput. The reduced rate can be attributed to: (i) the fact that we update annotations on the CF card after each file is transferred, in order to change the `dirty` annotation, (ii) the fact that our backup occurs through a network *tunnel* to the repository, (iii) the fact that we transfer slightly more data (e.g. the file annotations, or the handshake information to support incremental file transfers) during backup, and (iv) because our TCP implementation is less optimized[6]. than that of the Linux kernel. One must also consider that our network

[6]The small packet MTU (512 bytes compared to 1500 normally used in WLAN/IP) and the

stack runs in a virtual machine in user space.

We then sent these files to the PDA, using push-cache requests. Transfers from the repository to the PDA yielded similar transfer rates. This was to be expected, since the CF card's write throughput (500KB/sec) does not constitute a bottleneck. Finally, direct OmniStore-to-OmniStore transfers (e.g. for replication) did not exhibit any significant deviation as well.

As a final note we point out that, since transfers occur incrementally in the background, file transfer rates are not critical to the system's operation.

fixed window size have a big impact in the ideal network conditions of our testbed.

Chapter 6

Related work

Portable storage does have advantages over distributed file systems. For this reason, its popularity has not been diminished, in spite of the increasing level of network coverage. Portable storage offers guaranteed performance; in contrast, distributed file system performance is subject to network congestion. Availability is also guaranteed, provided the storage medium does not fail. Distributed file systems – besides server failure – are also subject to other types of failure (such as network outage, operator errors, etc). An overview of the advantages and disadvantages of the two fronts can be found in [THKS04], which justifies the place of portable storage in upcoming ubiquitous computing environments.

The concept of an"ultimate" portable storage device, which holds all of a user's data, is investigated in [WPD+02]. The idea is that all user data resides in the personal server, a single portable storage device with ad-hoc networking capability and no U/I elements. The user accesses the data through terminals in the environment which form ad-hoc connections to the personal server. This approach is interesting as it takes the active store concept to the extreme, although

admittedly one is unlikely to discard of all backend storage, if only to avoid disaster scenarios in which the personal server fails and all user data is lost. Such a device nicely fits into our model as a more capable portable store that is carried by the user more often than others. In fact, we have implemented a similar device, called the electronic wallet, which was used as the main data and application store of our first prototype system [LKSS03].

Rather than centering the ultimate storage solution around one portable device, the other end of the spectrum is the unified management of both portable and backend storage in which portable storage acts merely as a cache. This is also the approach taken by OmniStore. A similar concept can be found in [THKS04], where the "lookaside caching" technique is presented. Lookaside caching allows updating the files on a portable storage medium when it is mounted, by means of a hash function (the authors use SHA-1) which triggers updating of files whose hash has changed on the server. While this work was devised with passive stores (non-network-capable storage devices) in mind, it is equally applicable to PAN-based stores: a dismounted passive store corresponds to a device that cannot communicate with the repository. The provided functionality is similar to push-caching technique with live-update enabled on all files. However, by using the WORM approach [MTX03] for deep archival, combined with our naming scheme, we are able to detect different file versions through simple comparison of file identifiers.

Moving on to infrastructure-based approaches, Coda [Sat02] is the most well-known system addressing mobile computing. It uses optimistic caching to replicate the working set of user files on laptops and keep them in sync with the server. The UbiData [HH04] system builds upon Coda to address the existence of resource-limited clients such as PDAs. It supports transcoding of data in combi-

nation with a system for "format-independent change detection and propagation", which allows consolidating changes made to transcoded versions using different applications. This is aimed to enable data editing via applications which employ different formats, a common case in stripped-down PDA versions of desktop applications. Both Coda and UbiData mainly address relatively rich clients such as laptops and PDAs, which have much in common with the personal computer model. Our design targets less capable, specific-purpose devices and assumes that users will want to carry and use several such devices at the same time. We also introduce significant collaborative functionality among portable devices.

An improvement over a centralized repository is the use of a distributed backend storage service. The Roam system [RRP04] which uses peer-to-peer communication and can perform synchronization among any two replicas seems well-suited for deploying such an infrastructure. Another equally sophisticated system is OceanStore [KBC+00], which can further exploit untrusted servers for storing information. Needless to say, OmniStore's repository would benefit from a distributed approach, in which multiple backend servers are used to hold user data.

The treatment of files as immutable objects was first introduced in Cedar [SGN85, Hag87]. Deep archival offers several advantages and its use has been employed by systems such as the Elephant [SFH+99] file system and more recently Sedar [MTX03]. The later is closer to our design as it combines both deep archival and semantic organization. Again, these systems do not address mobile devices. We point out that since deep archival keeps all revisions of all files in the repository, data reconciliation may occur at a higher level using approaches such as [Lin03] to generate merged copies.

Semantic annotations have been the target of research for a while [GJSO91,

GM99] and researchers seem to agree that they are a more flexible and powerful way of managing files. Recently, their use in combination with deep archival [XKTK03] was suggested to further simplify storage management. The core issue now is the automatic generation of annotations [SG03] in order to relieve users from manual entry. Our method [KL06a] of automatically annotating files generated on portable active stores using context information (e.g. sensor data) of various devices which are present in a PAN. A significant amount of meta-information can be generated using our technique with absolutely no user input.

In terms of meta-data management, our system shares some analogy with the Roma [SKW+02] personal meta-data service. Roma uses a server to store meta-information regarding user files. It operates at higher level than OmniStore, using URIs to refer to the files indexed. However, Roma does not deal with the files themselves and makes no provisions for automating storage management operations.

Our portable device runtime can be seen as the realization of the vision described by Shivers in [Shi93], the "BodyNet". In it, a short-range hardware communications system connects a set of personal devices with a common interface language: "BodyTalk". Our underlying runtime system supports such a distributed approach.

A similar architecture for collaborating wearables is MEX [LHSA99], where a single component, the *post-office*, implements both a service directory and an event router. Services are thus undetectable and inaccessible without this intermediary. With our approach any service can be discovered and accessed directly from any client device. The MOCA [BGI99] framework is also targeted towards mobile devices, advocating service-based decomposition of applications and sub-

sequent dynamic binding to discovered services. A limitation is that MOCA is strictly Java-based as services are accessed via Java proxy objects. For this same reason, MOCA exhibits a high degree of location transparency as it does not differentiate among local and remote services. Our approach bases interoperability on the neutral communications layer allowing for implementations in various languages. It allows applications to communicate with services directly, to facilitate mobile-aware development. Transparency is optional via libraries for masking the developer from the dynamic execution setting.

Most of mobile computing middleware does not address the issue of service selection. Focus is typically either on the technicalities of enabling spontaneous interaction [HKSSR97, BGI99, RNP03], or on specific application scenarios and requirements [LHSA99, KS03] where the high granularity of functionality will generally not give rise to association issues. We introduce co-location statistics as a metric is used to efficiently perform this task.

Some analogy exists among our approach to exploit co-location history to infer device proximity (and thus availability) and work in P2P networks for achieving fault-tolerance. In [CY04], nodes gather history regarding faulty behavior of peers, over some observation period. Services are then replicated on nodes which fail within different time-slots, thereby decreasing the probability of service unavailability due to concurrent failure. This work is however not applicable to our MANET setting, nor to resource-constrained devices.

Further insight into the potential use of co-location history can be found in comparison to the analysis of user location information, e.g. see [AS03]. In this case GPS time and location stamps are recorded using a wearable device, and are subsequently used to infer discrete locations (in the higher level sense of the

word) based on the areas where a person spends significant time. This data is further exploited to compute the correlation among such locations as a function of time in order to predict user movement. Such elaborate analysis could also be applied to study device co-location patterns, provided that a device has enough memory resources to be able to keep more detailed information.

The importance of context information has led to toolkits for collecting and processing it both by using infrastructure [Dey01] and also by aggregation from multiple sources [GSB02]. The latter is the natural choice for PAN-based computing environments and used in our work. Context is used to create context-aware applications [Rho03], user interaction mechanisms [Sch00] and even context-sensitive middleware [Yau02] layers. In our work, context information is exploited to create semantic annotations for files created on portable devices in a generic way. The facility is generic and is available for use by developers to create any kind of context-aware applications.

Chapter 7

Discussion

OmniStore is an attempt to take personal data management a step forward in the direction of the ubiquitous computing vision. Its combination of features is designed to remove the burdens associated with storage management from the user. As a result, a significant source of distraction is eliminated, allowing people to focus on their goals rather than on operating computers.

The advantages enjoyed due to the automated and generic annotation of files with context information can be summarized in the following points: (i) users may flexibly navigate through their files or lookup specific files using contextual information regarding their creation, (ii) users may review contextual information regarding a file's creation for their own information, and (iii) users are greatly relieved from having to explicitly manage or annotate the files being created on the move via their wearable or portable devices.

The rest of OmniStore's features facilitate data placement activities. The automated archival of files to the repository means that users need not worry about the location of files created using portable devices; they know that ultimately they

can be reached using the repository. The push-caching facility on the other hand means that users may schedule file transfers when they so desire; the file and device need not be at hand at the time the request is issued. This is important because we allow the user to decide when to interact with the system and not vice-versa. In addition, applications can exploit implicit input to schedule file transfers without the intervention of the user. Meanwhile, the automated garbage collection feature relieves the user from having to manually make space for new data all the time. Lastly, we point out that the replication of files, combined with transparent fail-over in the event of sudden unavailability during file access, can mask mischievous events.

On the other hand, our system has several limitations that need to be addressed. The most striking one is that we have not considered operations involving multiple users. Due to the importance of this area, have already made provisions for such future work: our naming scheme guarantees the avoidance of naming conflicts, device registries can be used to track permissions and data repositories can be exploited to assist in performing relevant operations at the infrastructure side. However, we currently have not investigated, nor implemented, any support for carrying out common tasks such as file exchange, collaborative work, etc. Work needs to be done in this direction, to determine what can be accomplished and what additions / modifications are required to implement the envisaged features.

Another area that must be considered in future work, is that of heuristic operation. At the moment, system actions are driven by simple rules: backup is initiated at fixed intervals; automatic replication is initiated for push-caching operations that have an expiration date; the process of replicating such files is triggered immediately upon their receipt; free space reclamation is controlled by fixed

pre-configured ratios, etc. The only source of dynamic, auto-adjustable input in the system are co-location statistics, but these are only taken into account for data placement decisions. The possibility of using this input in additional aspects, along with the applicability of other heuristic methods (such as reputation-based schemes or machine-learing) needs to be considered.

Other areas of great importance that require in-depth investigation are security and power-consumption . Although we have kept these in mind, we have not in fact focused on providing overall solutions, but rather limited ourselves to basic features. While this oversight was intentional in order to remain focused on one problem, it is now necessary to pinpoint and address the issues related to these sensitive fields.

Finally, this work limits itself to user-created personal data files. Few provisions are made for system and / or application data, such as configuration settings, user-defined preferences, history, etc. We have addressed this area with the device registry (see Section 4.2.2) in which we implemented support for storing configuration information and making it available through the backbone to all interested parties. We thus have a solid basis upon which one may build more sophisticated services. It would be very interesting to see how – and to what extent – OmniStore's approach towards data management can be applied in these other cases of data.

In any case, even with all its limitations, this work's most significant contribution is perhaps that both our runtime for developing collaborative services and applications in ad-hoc networking environments, as well as the storage management system, constitute *enabling technology*. The former provides the core mechanisms required for ad-hoc computing systems in a tight, clean design. Its

scope goes well beyond the implementation of OmniStore, as it enables the development a multidude of applications targeting such dynamic environments. This is also true for the latter – our storage management system – for it is possible to build upon its services to develop new ubiquitous computing applications. A lot of potential exists in server-side processing of the context information attached to files, by applying data-mining techniques and semantic reasoning. While we can not forsee all areas that our work can potentially impact, we believe that we have presented a convincing case for adopting personal data management solutions following our approach.

Appendix A

A.1 Core runtime execution speed analysis

This appendix presents execution speed statistics regarding our runtime. The meaning of each column of measurements is depicted in table A-1. It should be noted that due to the large number of methods, we provide aggregate statistics on a per-package basis. For selected packages, we provide detailed information on a per-class basis, or even on a per-method basis. The information outlines the core runtime's performance, with a focus on service discovery and context manipulation methods. The results are discussed Section 5.2.1.

Column name	**Interpretation**
Base Time	For any invocation, the base time is the time taken to execute the invocation, excluding the time spent in other methods that were called during the invocation.
Average Base Time	The base time divided by the number of calls.
Cumulative Time	For any invocation, the cumulative time is the time taken to execute all methods called from an invocation. If an invocation has no additional method calls, then the cumulative time will be equal to the base time.
Invocations	The number of calls made to the method.

Table A-1: The execution statistics table fields explained

Package / Class / Method	Base Time (seconds)	Average (seconds)	Cumulative (seconds)	Invocations
[+]gr.jpan.runtime.components	0,026379	0,000776	0,105279	34
[-]gr.jpan.runtime	64,080588	0,036102	267,617736	1775
[+]Engine	1,025179	0,102518	212,165162	10
[+]BaseSystem	0,140160	0,000118	0,140160	1192
[+]Engine$EngineControl	0,000074	0,000002	0,000074	45
[+]Engine$BasicInterface	0,005655	0,000013	0,261099	431
[+]Engine$PrivilegedInterface	0,000338	0,000007	0,028357	46
[+]Engine$SystemInterface	0,000527	0,000007	0,038767	75
[+]Engine$StartComponentTask	0,000065	0,000005	0,151353	14
[+]SecurityComponent	0,027714	0,005543	0,179399	5
[+]RegistrationComponent	0,055255	0,013814	0,055691	4
[+]RegistrationComponent$RegistrationThread	62,826831	4,188455	63,877439	15
[+]Globals	0,000188	0,000188	0,000199	1
[+]Globals$1	0,000009	0,000009	0,000011	1

Package / Class / Method	Base Time (seconds)	Average (seconds)	Cumulative (seconds)	Invocations
[+]gr.jpan.infoBeacon	0,028106	0,004015	203,928781	7
[+]gr.jpan.j2se.drivers.commIP	0,932501	0,001160	1,203453	804
[+]gr.jpan.runtime.comm	0,019298	0,000060	0,116667	322
[+]gr.jpan.bootloader	0,385136	0,011671	0,385136	33
[+]gr.jpan.crypto	0,010712	0,000249	0,096507	43
[-]gr.jpan.datastructs	0,251096	0,000048	0,251356	5272
[-]HashMap	0,141535	0,000426	0,151198	332
HashMap(int)	0,072625	0,012104	0,072625	6
get(...)	0,003998	0,000034	0,008818	117
put(...)	0,006353	0,000635	0,007021	10
iterator()	0,022332	0,000263	0,026804	85
lookupEntry(...)	0,000853	0,000009	0,004727	99
valuesIterator()	0,035374	0,002358	0,035929	15
[+]LinkedList	0,092991	0,000078	0,095948	1188

Package / Class / Method	Base Time (seconds)	Average (seconds)	Cumulative (seconds)	Invocations
[+]LinkedList$Entry	0,001558	0,000002	0,001682	746
[+]ArrayOccupancyMap	0,000117	0,000008	0,000117	15
[-]LinkedList$LLIterator	0,013890	0,000005	0,014050	2676
LinkedList$LLIterator(...)	0,001529	0,000002	0,001529	617
hasNext()	0,010209	0,000008	0,010209	1244
next()	0,001056	0,000001	0,001056	774
addBefore(...)	0,001082	0,000028	0,001225	38
remove()	0,000016	0,000005	0,000032	3
[+]Map$Entry	0,000034	0,000003	0,000034	10
[+]HashMap$MapIterator	0,003478	0,000007	0,010052	483
[+]HashMap$ValuesIterator	0,000154	0,000004	0,001274	42
[+]gr.jpan.runtime.taskQueues	201,904642	0,175722	202,616502	1149
[+]gr.jpan.concurrency	0,000014	0,000014	0,000014	1
[-]gr.jpan.runtime.discovery	0,146062	0,000263	0,266746	556

Package / Class / Method	Base Time (seconds)	Average (seconds)	Cumulative (seconds)	Invocations
[+]DeviceDetectorComponent	0,091318	0,000755	0,144034	121
[-]ServiceDiscoveryComponent	0,029356	0,000249	0,042426	118
ServiceDiscoveryComponent()	0,000112	0,000112	0,000308	1
piggyDiscoveryStuff()	0,005218	0,000193	0,006567	27
nextDiscoveryTask()	0,000395	0,000015	0,000395	27
registerInterest(...)	0,005995	0,005995	0,006468	1
setActiveDiscovery(...)	0,000097	0,000049	0,000920	2
getProviders(...)	0,001991	0,000181	0,005317	11
processReceivedLookup(...)	0,000295	0,000012	0,005066	24
processReceivedAdvertisement(...)	0,009297	0,000465	0,010712	20
unregisterInterest(...)	0,000016	0,000016	0,000081	1
registerService(...) void	0,005904	0,002952	0,006613	2
setActiveAdvertize(...)	0,000021	0,000021	0,000238	1
unregisterService(...)	0,000016	0,000016	0,000137	1

Package / Class / Method	Base Time (seconds)	Average (seconds)	Cumulative (seconds)	Invocations
[+]DeviceDetectorComponent$BeaconTask	0,001288	0,000020	0,030672	65
[+]DeviceDetectorComponent$PruneColocDataTask	0,002493	0,000156	0,081561	16
[+]ServiceDiscoveryComponent$ServiceInterestList	0,000058	0,000007	0,000288	8
[+]ServiceDiscoveryComponent$ServiceInterestInfoNode	0,000036	0,000006	0,000036	6
[+]DeviceHistoryInfo	0,023445	0,000115	0,031771	204
[+]ServiceDiscoveryComponent$EndpointDiscoveryInfo	0,000089	0,000002	0,000089	45
[-]gr.jpan.runtime.context	0,008865	0,000097	0,027889	91
[-]ContextManagerComponent	0,003256	0,000099	0,005149	33
-clinit-()	0,000043	0,000043	0,000043	1
ContextManagerComponent()	0,000009	0,000009	0,000020	1
startup()	0,001197	0,001197	0,001650	1
access$0(...)	0,000004	0,000004	0,000004	1
access$1(...)	0,000004	0,000004	0,000004	1
access$2(...)	0,000005	0,000005	0,000005	1

Package / Class / Method	Base Time (seconds)	Average (seconds)	Cumulative (seconds)	Invocations
post(...)	0,000031	0,000010	0,002184	3
internalPost(...)	0,001591	0,000530	0,002139	3
cleanupStaleTuples()	0,000127	0,000032	0,000318	4
notifyAdded(...)	0,000010	0,000003	0,000062	3
processFrame(...)	0,000176	0,000015	0,000911	12
read(...)	0,000049	0,000049	0,000300	1
notifyRemoved(...)	0,000005	0,000005	0,000025	1
[+]ContextManagerComponent$ContextBeaconTask	0,002872	0,000115	0,023019	25
[+]ContextTuple	0,002823	0,000076	0,004586	37
[+]gr.jpan.exceptions	0,000124	0,000062	0,000124	2
[+]gr.jpan.utils	0,009213	0,000709	0,009777	13

Appendix B

B.1 Core runtime memory consumption analysis

This appendix presents statistics regarding the memory usage of our runtime. The meaning of each column of measurements is depicted in table B-1. Again, due to the large number of classes, we provide aggregate statistics on a per-package basis. For selected packages, we provide detailed information on a per-class basis. The information outlines the memory usage of the runtime system with a focus on service discovery and context manipulation. The results are discussed in Section 5.2.1.

Column name	**Interpretation**
Total instances	The total number of instances that had been created of the selected package or class.
Live instances	The number of instances of the selected package or class, where no garbage collection has taken place.
Collected	The number of instances of the selected package or class, that were removed during garbage collection.
Total size	The total size (in bytes) of the selected package or class, of all instances that were created for it, including whatever has been removed through garbage collection.
Active size	The summed size of all live instances.

Table B-1: The memory consumption statistics table fields explained

Package / Class	Total Instances	Live	Collected	Total Size (bytes)	Active Size (bytes)
[+](default package)	1750	1612	138	775624	176024
[+]gr.jpan.bootloader	1	1	0	40	40
[+]gr.jpan.concurrency	1	1	0	16	16
[+]gr.jpan.crypto	7	7	0	168	168
[-]gr.jpan.datastructs	327	327	0	8416	8416
[LinkedList	6	6	0	808	808
ArrayOccupancyMap	1	1	0	24	24
HashMap	6	6	0	96	96
HashMap$MapIterator	24	24	0	576	576
HashMap$ValuesIterator	5	5	0	120	120
LinkedList	16	16	0	384	384
LinkedList$Entry	105	105	0	2520	2520
LinkedList$LLIterator	185	185	0	4440	4440
Map$Entry	6	6	0	96	96

Package / Class	Total Instances	Live	Collected	Total Size (bytes)	Active Size (bytes)
[+]gr.jpan.infoBeacon	1	1	0	40	40
[+]gr.jpan.j2se.drivers.commIP	45	45	0	984	984
[+]gr.jpan.runtime	13	13	0	368	368
[+]gr.jpan.runtime.comm	12	12	0	240	240
[+]gr.jpan.runtime.components	1	1	0	16	16
[-]gr.jpan.runtime.context	5	5	0	264	264
ContextManagerComponent	1	1	0	56	56
ContextManagerComponent$ContextBeaconTask	1	1	0	64	64
ContextTuple	3	3	0	144	144
[-]gr.jpan.runtime.discovery	5	5	0	344	344
DeviceDetectorComponent	0	0	0	0	0
DeviceDetectorComponent$BeaconTask	1	1	0	64	64
DeviceDetectorComponent$PruneColocDataTask	1	1	0	64	64
DeviceHistoryInfo	1	1	0	88	88

Package / Class	Total Instances	Live	Collected	Total Size (bytes)	Active Size (bytes)
ServiceDiscoveryComponent	1	1	0	96	96
ServiceDiscoveryComponent$EndpointDiscoveryInfo	1	1	0	32	32
[+]gr.jpan.runtime.taskQueues	3	3	0	64	64
[+]gr.jpan.utils	0	0	0	0	0
[+]java.lang	65	65	0	5744	5744

Appendix C

C.1 Storage system execution time analysis

This appendix presents execution time statiistics regarding the storage system. The meaning of each column of measurements are the same as those of Section A.1 and are depicted in table A-1. Yet again, due to the large number of methods, we provide aggregate statistics on a per-package basis. For selected packages, we provide detailed information on a per-class basis, or even on a per-method basis. The information provides an overview of the performance of storage-related methods and is discussed in Section 5.2.2.

Package / Class / Method	Base Time (seconds)	Average (seconds)	Cumulative (seconds)	Invocations
[+](default package)	0,000000	0,000000	0,000000	0
[+]gr.jpan.bootloader	0,038258	0,002733	0,038258	14
[+]gr.jpan.comm	0,000300	0,000001	0,000300	217
[+]gr.jpan.components.gatewayClient	0,014306	0,001590	0,082451	9
[+]gr.jpan.components.registry	0,000194	0,000097	0,000622	2
[+]gr.jpan.concurrency	0,000026	0,000026	0,000026	1
[+]gr.jpan.crypto	0,095456	0,010606	2,501784	9
[+]gr.jpan.datastructs	1,954963	0,000046	1,954973	42924
[+]gr.jpan.jni.drivers.commIP	0,294297	0,000257	0,478927	1143
[+]gr.jpan.jni.drivers.storage	1,099043	0,000029	1,099276	38246
[+]gr.jpan.omnistore	0,028335	0,000727	0,438613	39
[-]gr.jpan.omnistore.drivers.jni	26,113380	0,000521	32,583020	50168
[+]AnnotationsFile	6,593798	0,000534	24,093936	12340
[+]AnnotationsFile$AnnotationRecord	15,071270	0,000406	15,071270	37118

Package / Class / Method	Base Time (seconds)	Average (seconds)	Cumulative (seconds)	Invocations
[-]OFileHandleJNI	4,263265	0,006610	32,284041	645
close() void	0,000111	0,000111	0,061853	1
getAllKeyValues(...) List	2,108839	0,006202	23,837768	340
listKeys() List	1,881901	0,037638	5,512696	50
OFileHandleJNI(...)	0,013649	0,013649	0,016318	1
read(...) int	0,000642	0,000013	0,000642	50
seek(long) void	0,000685	0,000007	0,000685	100
size() long	0,000027	0,000027	0,000027	1
updateAnnotation(...) void	0,256138	0,004926	2,852805	52
write(...) int	0,001273	0,000025	0,001273	50
[+]OmnistoreDriverJNI	0,168459	0,005809	0,329213	29
[+]OmnistoreDriverJNI$1	0,016589	0,000461	0,016589	36
[+]gr.jpan.runtime	0,286727	0,000227	2.128,790	1264
[+]gr.jpan.runtime.comm	0,130401	0,000574	0,252291	227

Package / Class / Method	Base Time (seconds)	Average (seconds)	Cumulative (seconds)	Invocations
[+]gr.jpan.runtime.components	0,034963	0,000795	0,076783	44
[+]gr.jpan.runtime.context	0,020664	0,000093	0,027455	222
[+]gr.jpan.runtime.discovery	0,140144	0,000138	0,330196	1019
[+]gr.jpan.runtime.taskQueues	2.124,677863	0,319982	2.125,527	6640
[+]gr.jpan.storage	3,677003	0,000144	4,874877	25520
[+]gr.jpan.storageBox	0,243268	0,060817	2.161,482	4
[+]gr.jpan.utils	2,789436	0,000641	2,789436	4352
[+]java.lang	0,000000	0,000000	0,000000	0

Bibliography

[AS03] Daniel Ashbrook and Thad Starner. Using GPS to learn significant locations and predict movement across multiple users. *Personal Ubiquitous Comput.*, 7(5):275–286, 2003.

[BGI99] James Beck, Alain Gefflaut, and Nayeem Islam. MOCA: a service framework for mobile computing devices. In *Proceedings of the 1st ACM international workshop on Data engineering for wireless and mobile access*, pages 62–68. ACM Press, 1999.

[BLR+04] Hari Balakrishnan, Karthik Lakshminarayanan, Sylvia Ratnasamy, Scott Shenker, Ion Stoica, and Michael Walfish. A layered naming architecture for the internet. In *SIGCOMM '04: Proceedings of the 2004 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 343–352, New York, NY, USA, 2004. ACM Press.

[Bus96] Vannevar Bush. As we may think (reprint). *Interactions*, 3(2):35–46, 1996.

[CBR04] Ian D. Chakeres and Elizabeth M. Belding-Royer. AODV Routing Protocol Implementation Design. In *Proceedings of the Inter-*

national Workshop on Wireless Ad hoc Networking (WWAN), pages 698–703, Tokyo, Japan, March 2004.

[CGS+02] Shang-Wen Cheng, David Garlan, Bradley R. Schmerl, João Pedro Sousa, Bridget Spitznagel, Peter Steenkiste, and Ningning Hu. Software architecture-based adaptation for pervasive systems. In *Proceedings of the International Conference on Architecture of Computing Systems*, pages 67–82. Springer-Verlag, 2002.

[Cha06] Chakraborty, Dipanjan and Joshi, Anupam and Yesha, Yelena and Finin, Tim. Toward distributed service discovery in pervasive computing environments. *IEEE Transactions on Mobile Computing*, 05(02):97–112, 2006.

[CY04] Fang-Yu Chen and Soe-Tsyr Yuan. A contextualized fault-tolerant infrastructure for P2P mobile service composition. In *IEEE International Conference on Services Computing (SCC 2004) Shanghai, China*. IEEE Computer Society Press, September 2004.

[Dey01] Anind K. Dey. Understanding and using context. *Personal Ubiquitous Comput.*, 5(1):4–7, 2001.

[GJSO91] David K. Gifford, Pierre Jouvelot, Mark A. Sheldon, and James W. O'Toole, Jr. Semantic file systems. In *Proceedings of the thirteenth ACM symposium on Operating systems principles*, pages 16–25. ACM Press, 1991.

[GM99] Burra Gopal and Udi Manber. Integrating content-based access mechanisms with hierarchical file systems. In *Proceedings of the*

third symposium on Operating systems design and implementation (OSDI '99), pages 265–278. USENIX Association, 1999.

[Gra03] Jim Gray. What next?: A dozen information-technology research goals. *J. ACM*, 50(1):41–57, 2003.

[GSB02] Hans W. Gellersen, Albercht Schmidt, and Michael Beigl. Multi-sensor context-awareness in mobile devices and smart artifacts. *Mobile Networks and Applications*, 7(5):341–351, 2002.

[Hag87] R. Hagmann. Reimplementing the Cedar file system using logging and group commit. In *SOSP '87: Proceedings of the eleventh ACM Symposium on Operating systems principles*, pages 155–162, New York, NY, USA, 1987. ACM Press.

[HH04] Abdelsalam Helal and Joachim Hammer. Ubidata: requirements and architecture for ubiquitous data access. *SIGMOD Rec.*, 33(4):71–76, 2004.

[HJ04] Yih-Chun Hu and David B. Johnson. Exploiting congestion information in network and higher layer protocols in multihop wireless ad hoc networks. In *Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS'04)*, pages 301–310, 2004.

[HKSSR97] Todd D. Hodes, Randy H. Katz, Edouard Servan-Schreiber, and Lawrence Rowe. Composable ad-hoc mobile services for universal interaction. In *Proceedings of the 3rd annual ACM/IEEE in-*

ternational conference on Mobile computing and networking, pages 1–12. ACM Press, 1997.

[ISI81] University of Southern California Information Sciences Institute. Transmission control protocol (TCP). RFC 739, Internet Engineering Task Force (IETF), September 1981.

[JM96] David B Johnson and David A Maltz. Dynamic source routing in ad hoc wireless networks. In Imielinski and Korth, editors, *Mobile Computing*, volume 353. Kluwer Academic Publishers, 1996.

[KBC+00] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Chris Wells, and Ben Zhao. Oceanstore: an architecture for global-scale persistent storage. In *ASPLOS-IX: Proceedings of the ninth international conference on Architectural support for programming languages and operating systems*, pages 190–201, New York, NY, USA, 2000. ACM Press.

[KL05] Alexandros Karypidis and Spyros Lalis. Exploiting co-location history for efficient service selection in ubiquitous computing systems. In *2nd International Conference on Mobile and Ubiquitous Systems (MobiQuitous 2005): Networking and Services*, pages 202–209. IEEE Computer Society Press, July 2005.

[KL06a] Alexandros Karypidis and Spyros Lalis. Automated context aggregation and file annotation for pan-based computing. *Personal Ubiquitous Comput.*, 11(1):33–44, 2006.

[KL06b] Alexandros Karypidis and Spyros Lalis. OmniStore: A system for ubiquitous personal storage management. In *Proceedings of the Fourth Annual IEEE International Conference on Pervasive Computing and Communications (PerCom'06)*, pages 136–147. IEEE, March 2006.

[KS03] Gerd Kortuem and Zary Segall. Wearable communities: Augmenting social networks with wearable computers. *IEEE Pervasive Computing*, 2(1):71–78, 2003.

[LHSA99] Juha Lehikoinen, Jussi Holopainen, Marja Salmimaa, and Angelo Aldrovandi. MEX: A distributed software architecture for wearable computers. In *Proceedings of the 3rd IEEE Internation Symposium on Wearable Computing*, pages 52–57, 1999.

[Lin03] Tancred Lindholm. XML three-way merge as a reconciliation engine for mobile data. In *MobiDe '03: Proceedings of the 3rd ACM international workshop on Data engineering for wireless and mobile access*, pages 93–97, New York, NY, USA, 2003. ACM Press.

[LKS05] Spyros Lalis, Alexandros Karypidis, and Anthony Savidis. Ad-hoc composition in wearable and mobile computing. *Commun. ACM*, 48(3):67–68, 2005.

[LKSS03] Spyros Lalis, Alexandros Karypidis, Anthony Savidis, and Constantine Stephanidis. Runtime support for a dynamically composable and adaptive wearable system. In *Proceedings of the 7th IEEE Inter-*

nation Symposium on Wearable Computing, pages 18–21, October 2003.

[LV03] Peter Lyman and Hal R. Varian. How much information. http://www.sims.berkeley.edu/how-much-info-2003 on 31 January 2006, 2003.

[MTX03] Mallik Mahalingam, Chunqiang Tang, and Zhichen Xu. Towards a semantic, deep archival file system. In *Proceedings of the 9th IEEE Workshop on Future Trends of Distributed Computing Systems (FTDCS'03)*, pages 115–121. IEEE Computer Society, May 2003.

[NKR+02] Chandra Narayanaswami, Noboru Kamijoh, Mandayam Raghunath, Tadanobu Inoue, Thomas Cipolla, Jim Sanford, Eugene Schlig, Sreekrishnan Venkiteswaran, Dinakar Guniguntala, Vishal Kulkarni, and Kazuhiko Yamazaki. IBM's Linux watch: The challenge of miniaturization. *IEEE Computer*, 35(1):33–41, January 2002.

[PB94] Charles E. Perkins and Pravin Bhagwat. Highly dynamic Destination-Sequenced Distance-Vector routing (DSDV) for mobile computers. In *SIGCOMM '94: Proceedings of the conference on Communications architectures, protocols and applications*, pages 234–244. ACM Press, 1994.

[Rho03] Bradley J. Rhodes. Using Physical Context for Just-in-Time Information Retrieval. *IEEE Transactions on Computer*, 52(8):1011–1014, 2003.

[RNP03] Mandayam Raghunath, Chandra Narayanaswami, and Claudio Pinhanez. Fostering a symbiotic handheld environment. *IEEE Computer*, 36(9):56–65, 2003.

[RRP04] David Ratner, Peter Reiher, and Gerald J. Popek. Roam: a scalable replication system for mobility. *Mob. Netw. Appl.*, 9(5):537–544, 2004.

[Sat02] M. Satyanarayanan. The evolution of coda. *ACM Transactions on Computer Systems (TOCS)*, 20(2):85–124, 2002.

[Sch00] A. Schmidt. Implicit human computer interaction through context. *Personal Technologies*, 4(2-3):191–199, June 2000.

[SFH+99] Douglas S. Santry, Michael J. Feeley, Norman C. Hutchinson, Alistair C. Veitch, Ross W. Carton, and Jacob Ofir. Deciding when to forget in the elephant file system. In *Proceedings of the seventeenth ACM symposium on Operating systems principles (SOSP '99)*, pages 110–123. ACM Press, 1999.

[SG03] Craig A. N. Soules and Gregory R. Ganger. Why can't I find my files? New methods for automating attribute assignment. In *Proceedings of HotOS IX: The 9th Workshop on Hot Topics in Operating Systems*, pages 181–186. USENIX Association, May 2003.

[SGN85] Michael D. Schroeder, David K. Gifford, and Roger M. Needham. A caching file system for a programmer's workstation. *SIGOPS Oper. Syst. Rev.*, 19(5):25–34, 1985.

[Shi93] O. Shivers. BodyTalk and the BodyNet: A Personal Information Infrastructure. Personal Information Architecture Note 1, MIT Laboratory for Computer Science, Cambridge, MA, December 1993.

[SKW+02] Edward Swierk, Emre Kiciman, Nathan C. Williams, Takashi Fukushima, Hideki Yoshida, Vince Laviano, and Mary Baker. The Roma personal metadata service. *Mobile Networks and Applications*, 7(5):407–418, 2002.

[The] The Bluetooth SIG. Bluetooth Specification. http://www.bluetooth.com.

[THKS04] Niraj Tolia, Jan Harkes, Michael Kozuch, and M. Satyanarayanan. Integrating portable and distributed storage. In *Proceedings of the 3rd Conference on File and Storage Technologies (FAST '04)*, pages 227–238, San Francisco, CA, March 31 - April 2 2004. USENIX.

[Tim02] Tim Kindberg and Armando Fox. System software for ubiquitous computing. *IEEE Pervasive Computing*, 1(01):70–81, 2002.

[Var05] Hal R. Varian. Universal access to information. *Commun. ACM*, 48(10):65–66, 2005.

[VRdL05] Alex Varshavsky, Bradley Reid, and Eyal de Lara. A cross-layer approach to service discovery and selection in MANETs. In *the Second International Conference on Mobile Ad-Hoc and Sensor Systems (MASS 2005)*, November 2005.

[WBS04] Micahel Walfish, Hali Balakrishnan, and Scott Shenker. Untangling the web from DNS. In *Proceedings of the 1st USENIX/ACM Symposium on Networked Systems Design (NSDI '04)*, pages 225–238, San Francisco, CA, March 2004. USENIX.

[Wei91] Mark Weiser. The computer for the 21st Century. *Scientific American*, 265(3):94–104, September 1991.

[WPD+02] Roy Want, Trevor Pering, Gunner Danneels, Muthu Kumar, Murali Sundar, and John Light. The Personal Server: Changing the Way We Think About Ubiquitous Computing. In *Proceedings of the 4th International Conference on Ubiquitous Computing*, pages 194–209, 2002.

[XKTK03] Zhichen Xu, Magnus Karlsson, Chunqiang Tang, and Christos Karamanolis. Towards a semantic-aware file store. In *Proceedings of HotOS IX: The 9th Workshop on Hot Topics in Operating Systems*, pages 115–120. USENIX Association, May 2003.

[Yau02] Yau Stephen S. and Karim Fariaz and Wang Yu and Wang Bin and Gupta Sandeep K. S. Reconfigurable context-sensitive middleware for pervasive computing. *IEEE Pervasive Computing*, 1(03):33–40, 2002.

[Yau03] Yau, Stephen S. and Karim, Fariaz. An energy-efficient object discovery protocol for context-sensitive middleware for ubiquitous computing. *IEEE Transactions on Parallel and Distributed Systems*, 14(11):1074–1085, 2003.