Prototype et expérimentations

8.1 Motivations

Nos recherches sur la mobilité de service ont requis une étude approfondie de l'état de l'art afin d'analyser les approches existantes, et identifier les problématiques de continuité dans des environnements hétérogènes. Un long travail de redéfinition des nombreux concepts sous un angle nouveau, plus éloigné de l'implémentation des services, a rendu possible l'esquisse d'un modèle générique qui réponde à ces problématiques. Le *distributed Service Manager* est une implémentation de ce modèle ; avec le dSM nous apportons une solution complète, basée sur de nombreuses fonctions originales qui assurent une mobilité de bout en bout.

Nous arrivons ici à la dernière étape de cette démarche qui consiste à évaluer le modèle proposé, estimer si les contraintes établies ont été respectées, si les objectifs fixés ont été atteints et si la solution proposée est satisfaisante. Nous avons pour cela développé un prototype simple, qui assure l'essentiel des fonctions présentées dans les chapitres précédents. Cette preuve de concept permettra d'estimer l'impact de notre modèle sur son environnement d'un point de vue qualitatif et quantitatif.

8.2 Description du prototype

Le prototype du distributed Service Manager (dSM) a été développé pour prouver la faisabilité du modèle défini dans le chapitre 6, et réaliser des tests de per-

formance et d'utilisabilité. Le système central (dSM *core*), une interface utilisateur (UI), une interface de programmation (API) et enfin deux applications de test ont en tout été développés et publiés sous forme de projet *open source* LGPL sur le site Launchpad [90] (le projet s'appelle dSM, nom de code « *Chameleon* » rapport à sa propriété d'adaptation).

Le projet est entièrement écrit en Java et organisé en trois parties de la manière suivante.

- **Un système central**. Les principaux mécanismes du dSM font partie du système central (*core*) qui gère entre autres le PSE, le réseau *overlay* sous-jacent (découverte, communication et transferts de données entre terminaux via un protocole pair-à-pair implémenté par la bibliothèque Java JXTA [91][92]), les capacités du terminal, les services, les ressources ainsi que les messages entrant et sortant.
- **Une interface utilisateur**. Une interface graphique minimale (GUI) qui permet à l'utilisateur d'interagir avec le système et qui peut (et devrait) être remplacée par une interface tierce plus évoluée grâce aux évènements générés par le *core*. À noter qu'une console est également disponible.
- **Une interface de programmation**. Cette API est essentielle, intégrée aux applications elle leur permet d'interagir avec le dSM, facilitant considérablement le travail des développeurs de solutions compatibles.

En plus de ces trois grands axes de développement, deux applications tierces ont été modifiées afin de les rendre compatibles avec le dSM dans le cadre de nos tests. Pour cela nous avons intégré l'API du dSM à des éditeurs de texte open source Java relativement basiques. Ces applications sont également disponibles dans le cadre du projet open source, plus de détails seront fournis en section 8.3.2. La Figure 8.1 est une représentation fonctionnelle de l'architecture du distributed Service Manager qui positionne les différents blocs du prototype, l'annexe B propose également un diagramme de classes du système, enfin vous trouverez plus d'informations sur le site du projet [90]. Ces travaux ont été publiés dans [93].

Le système complet a été conçu afin d'être indépendant du SE sous-jacent. Le choix de Java comme langage de programmation a été fait grâce à ses qualités de portabilité mais également pour des questions de vitesse de développement. De plus, l'ensemble des communications entre le dSM et les applications se font via

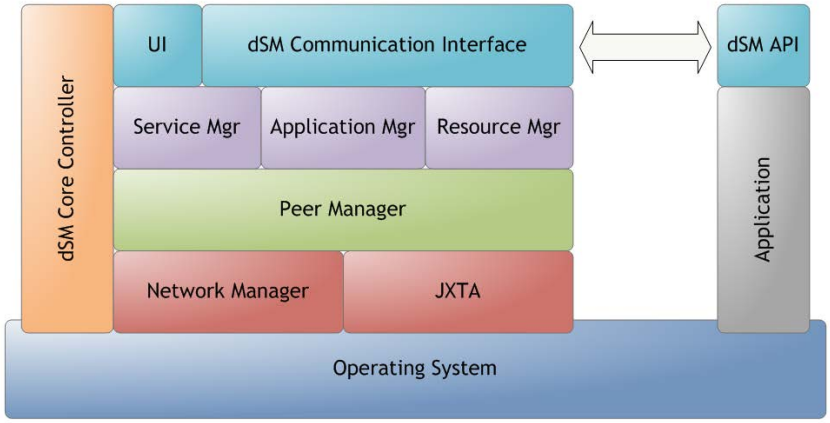

FIGURE 8.1 – Architecture fonctionnelle du prototype dSM.

des ports réseaux (le dSM utilise par exemple le port UDP 20100) pour les mêmes raisons d'indépendance avec l'environnement.

L'interface graphique n'est pas un axe actif du projet, en contrepartie un mécanisme d'évènements est fourni afin de permettre le développement d'interfaces plus évoluées sans avoir à modifier le système central. Une interface minimale de type « console » a également été développée pour les tests, permettant un accès rapide et exhaustif aux fonctions du dSM tout en limitant les mesures aux performances intrinsèques du modèle (dont le GUI ne fait pas partie).

Le fichier exécutable du distributed Service Manager (.jar) représente moins de 100ko (sans compter les bibliothèques externes et l'interface graphique). L'API, elle, fait moins de 40ko, offrant aux applications tous les mécanismes requis pour interagir avec le dSM. Il reste à l'application compatible d'initialiser l'API et d'assurer les fonctions *pause* et *resume* (cf. 7.2.2) qui ne représentent que quelques lignes de code par ressource à gérer.

Pour exemple, l'implémentation de ces fonctions dans les éditeurs de texte que nous avons modifiés a requis environ 50 lignes de code supplémentaires, soit moins de 2 à 5% de la taille totale de ces applications. Nous avons estimé que l'ajout d'une ressource supplémentaire ne coûterait qu'approximativement 5 lignes de code en plus.

8.3 Expérimentations

Évaluer la valeur ajoutée apportée par un modèle générique de mobilité de service n'est pas une simple tâche. En effet, nous ne proposons pas une solution qui accélère les applications ou augmente le débit d'un réseau, en réalité nous cherchons à permettre à l'utilisateur d'interagir de manière plus intuitive et plus libre avec ses terminaux, rendant la consommation des services en mobilité plus naturelle dans des environnements hétérogènes. Or il est difficile d'évaluer ces valeurs abstraites qui ont pourtant un impact bien concret au niveau de l'expérience utilisateur. Comment mesurer l'utilisabilité et la praticité ? D'un autre côté, le modèle que nous proposons se doit de respecter certaines contraintes pour prouver sa faisabilité, typiquement en terme de délai de transfert comparé aux approches existantes.

L'approche originale de mobilité de service que nous proposons doit convaincre sur deux plans afin de garantir une expérience satisfaisante : offrir à l'utilisateur une interaction intuitive avec ses services via des mécanismes de continuités temporelle et contextuelle performants. Nous avons ainsi réalisé deux expérimentations pour évaluer les deux aspects de notre modèle : utilisabilité et efficacité via une approche qualitative et quantitative de notre solution.

Dans la première expérimentation, nous avons étudié la relation des utilisateurs avec leurs services dans des situations de mobilité. Nous avons interrogé un échantillon d'utilisateurs afin de répertorier et analyser les méthodes de mobilité existantes. Dans la seconde expérience, nous avons mesuré la performance de notre prototype durant un transfert et analysé l'impact sur son environnement selon différentes métriques.

8.3.1 Évaluation qualitative

Dans cette première expérimentation nous avons voulu étudier le comportement des utilisateurs exposés à des contraintes de mobilité et plus particulièrement comment ils gèrent leurs services dans une telle situation. L'objectif est d'identifier les méthodes de transfert dans un cas de mobilité de service simple et réaliste. Cette première approche est une étape clé dans notre étude dans la mesure où l'expérience utilisateur est la métrique la plus importante dans l'évaluation de la mobilité de service. Ainsi, comprendre qu'est ce qui gêne ou empêche les utilisateurs de gérer

de manière simple, naturelle et intuitive leurs services parmi plusieurs terminaux nous aidera a identifier la réelle problématique.

Méthode.

Lors de cette expérience, nous avons demandé à des utilisateurs de répondre à des questions concernant une situation simple et leur expérience passée (questionnaire disponible en annexe C). Nous avons sélectionné un échantillon de personnes qui possèdent un ordinateur et l'utilisent au moins une fois par jour. Ainsi nous avons supposé que ces personnes avaient, de manière consciente ou non été confrontées au moins une fois à une situation de mobilité de service. Les participants étaient 33 personnes de différents âges (de 12 à 55 ans) et possédant des connaissances informatiques variées. Nous les avons classés en deux catégories selon leur niveau dans le domaine des Technologies de l'Information et de la Communication : les *spécialistes* (ingénieurs, chercheurs, étudiants) représentant 80% de l'échantillon et les *non-spécialistes* représentant les 20% restants. L'hétérogénéité de l'échantillon était recherché et nécessaire pour garantir des résultats de qualité et une vision consistante de la méthodologie utilisateur adoptée.

Nous avons présenté à chaque participant une situation simple d'utilisation d'un service informatique, puis nous avons introduit une contrainte de mobilité et nous leur avons demandé de décrire leur solution selon leur expérience. La situation proposée était basique : un utilisateur qui édite un document sur son ordinateur doit subitement interrompre son travail et le poursuivre sur un autre terminal. Les participants étaient alors invités à décrire en détails comment ils ont (ou auraient) réalisé un tel transfert.

Nous avions sciemment fourni peu de détails dans la description de la situation afin de faciliter l'adoption du cas par chaque participant. Des questions additionnelles étaient ensuite posées afin d'obtenir des informations plus précises en rapport avec leur expérience de la mobilité de service : besoins, praticité, simplicité, sentiment d'interruption, etc. Les données recueillies ont ensuite été analysées et rapprochées afin d'extraire des informations et des tendances sur les utilisateurs et leur perception des contraintes de mobilité. Les résultats sont présentés dans la section suivante.

Résultats.

L'objectif premier de cette expérience était d'analyser la méthodologie de mobilité de service des utilisateurs. Les participants ont donc été invités à exprimer toutes les approches qu'ils ont expérimentées ou auraient pu utiliser. Les résultats collectés sont présentés dans la Figure 8.2 qui résume les différentes méthodes de transfert par catégorie (spécialistes et non-spécialistes).

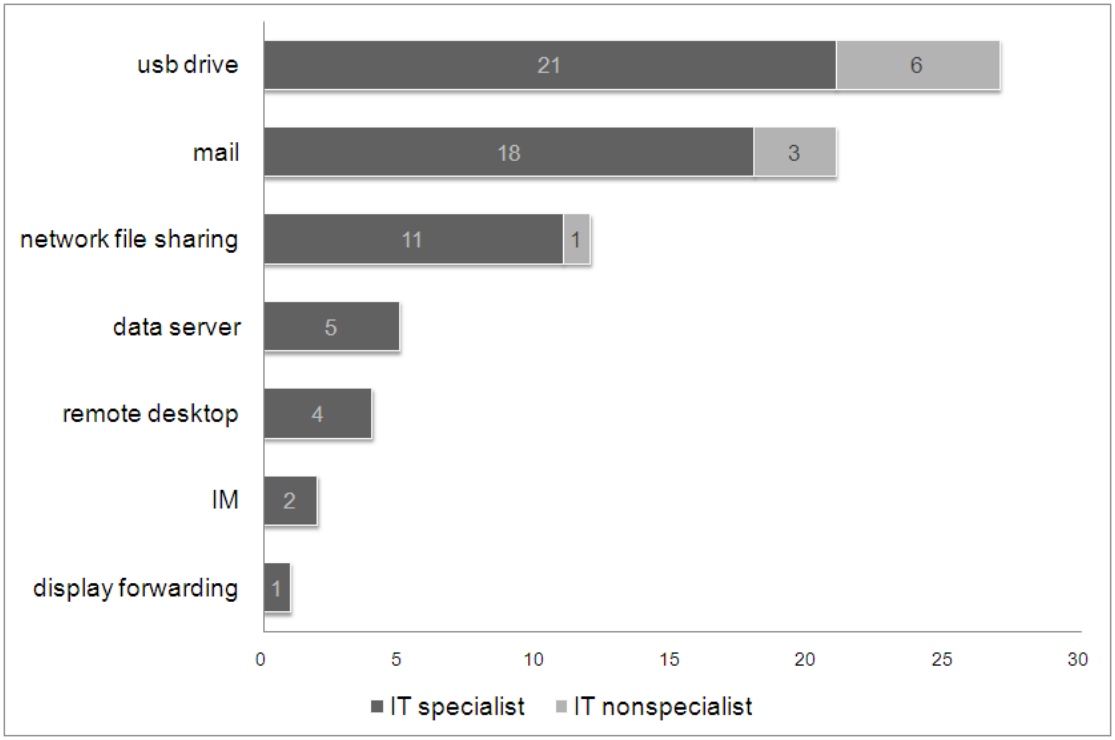

FIGURE 8.2 – Méthodes de mobilité de service expérimentées.

Le premier constat est le nombre de solutions différentes pour répondre à une même problème et ce uniquement sur un échantillon de 33 personnes (7 méthodes distinctes pour les spécialistes, 3 pour les non-spécialistes). Il n'y a pas d'approche commune, un modèle générique pour gérer une contrainte de mobilité dans un cas relativement basique. De plus, les trois approches les plus utilisées : stockage USB, courrier électronique et lecteur réseau sont partagées par les deux catégories de l'échantillon, ce qui indique que c'est bien une problématique globale et qu'il n'existe pas de solution miracle réservée aux initiés.

On peut également remarquer que ces solutions sont simples : USB et *email* sont manipulés presque quotidiennement et le partage réseau est intégré nativement aux systèmes d'exploitations, souvent prêt à l'emploi dans le milieu de travail des utilisateurs. La popularité de ces solutions familières est liée à leur simplicité d'utilisation qui en font un choix naturel bien que non prévues a priori à la mobilité de service. Cependant, simplicité n'est pas synonyme de praticité.

Nous avons également demandé aux participants d'évaluer la simplicité et la praticité de leur propres solutions, nous avons ensuite traduit leurs réponses en valeurs numériques (1, 2 et 3) selon le barème suivant :

1. **non** simple/pratique,
2. simple/pratique,
3. **très** simple/pratique.

Finalement non avons calculé la moyenne pour chacune des propriétés et de manière finalement peu surprenante, leurs méthodes sont évaluées comme assez simples (2.58) mais tout juste pratiques (1.91). En fait, par manque de solution dédiée, les approches proposées sont des alternatives « bricolées » basées sur des applications connues, maîtrisées par l'utilisateur et donc simples mais qui ne peuvent en aucun cas offrir des mécanismes efficaces et pratiques de continuité.

Pour illustrer ce manque d'efficacité, 66% des participants prétendent se sentir interrompus dans leur tâche en employant leur solution de mobilité. De plus, ils ont décrit les problèmes et les contraintes liées à leur solution, celles-ci sont résumées, par approche, ci-dessous.

- Stockage USB. Contraintes matérielles et logicielles (interface et lecteur USB requis, compatibilité du système d'exploitation, système de fichier, etc).
- Courrier électronique. accès à Internet nécessaire, limitations sur le taille et le type des données transférées (problèmes de sécurité). Délai de transfert important.
- Partage réseau. Problèmes de compatibilité avec le système d'exploitation, le protocole utilisé et le système de fichier. Solution moins triviale généralement pour les spécialistes.

Néanmoins, les utilisateurs sondés ont recours en moyenne à un transfert de service 2.73 fois par semaine, spécialistes et non-spécialistes confondus, ce qui

est particulièrement important considérant le manque de praticité des méthodes utilisées et l'absence d'un mécanisme dédié à la continuité de service. Cela prouve qu'un réel besoin existe et pas seulement pour les « geeks ». La table 8.1 résume les résultats obtenus dans le cadre de cette expérimentation.

TABLE 8.1 – Résultats de l'évaluation qualitative.

Metric	**Result**	
Sample	33 people	
	80% IT spec.	20% nonspec.
Service Mobility Frequency	2.73 per week	
Ease of Use	2.58	
Convenience	1.91	
Interruption Feeling	66% yes	

Enfin, nous avons proposé aux participants une autre situation, plus complexe cette fois-ci. Nous avons supposé qu'ils regardaient une vidéo sur Internet (service multimédia tel que proposé par *Youtube*) et qu'ils devaient cette fois encore changer de terminal. Nous leur avons demandé si leur approche de mobilité initiale était applicable à cette nouvelle situation de mobilité, et si non d'en proposer une. Bien entendu, aucune de leurs approches n'a pu être appliquée à ce service différent excepté l'approche d'*affichage déporté* proposée par un spécialiste et qui est en fait une solution relativement performante de continuité de service que nous avions identifiée dans l'état de l'art (cf. 2.3.1).

76% des participants ont proposé une méthode *manuelle* de mobilité qui consiste à transférer le service multimédia en plusieurs étapes (l'URL, le nom de la vidéo, la position de lecture, etc) via des mécanismes de communication classiques reliant les terminaux origine et destination (tels que ceux listé en Figure 8.2) ou simplement en les mémorisant mentalement (le cerveau reliant effectivement les deux interfaces). Il est intéressant de constater qu'inconsciemment les participants ont identifié les éléments importants du service et transmis ces ressources manuellement afin de rétablir le contexte sur le nouveau terminal ; exactement le principe du mécanisme de transfert du distributed Service Manager. De plus, le processus

d'adaptation a également été pensé, seules les ressources compatibles aux capacités de l'application destination ont été transférées. Uniquement le contexte a été considéré, le service a été déplacé et non son instance. Il est à noter que 12% de l'échantillon n'a pu trouver de solution pour cette nouvelle situation.

Discussion.

Avec de plus en plus de terminaux autour de l'utilisateur, celui-ci tend à être plus mobile. Cette première expérience démontre que le besoin de mobilité est bien réel et que pourtant il n'existe pas de modèle de continuité générique, ni d'ailleurs de solution standard pour les situations simples de la vie de tous les jours. Lorsque confrontés à des contraintes de mobilité, les utilisateurs bricolent un semblant de continuité à partir d'applications simples mais prévues à un tout autre dessein. Les approches utilisateurs imposent des contraintes de compatibilité logicielles et matérielles fortes, sans mécanisme de sécurité, ni d'adaptation, offrant une expérience pauvre (sentiment d'interruption, perte de contexte, etc) et dédiées à des scénarios spécifiques. Quant aux scénarios plus complexes tels que le streaming vidéo, ils ne sont simplement pas supportés.

Ces solutions alternatives « fait-maison » induisent intrinsèquement de nouvelles problématiques, de plus elles ne peuvent assurer le processus mobilité de bout en bout comme défini dans le modèle générique : « *découverte, capture, déclenchement, transfert et reprise* » (cf. 6). En effet, au mieux, seule l'étape de transfert est réalisée, les autres phases sont généralement assurées par l'utilisateur, dans la mesure du possible.

Il n'y a pas d'identification des services et des terminaux, en conséquence ils ne peuvent être transférés directement d'une interface à une autre, l'utilisateur doit alors passer par une étape intermédiaire : lecteur USB, boîte de courrier électronique, disque partagé, etc.

Les services « simples » tel que l'édition de texte proposé dans ce scénario sont gérés au niveau application via une fichier complet (généralement propriétaire) qui laisse peu de manœuvre à l'adaptation, totalement incompatible avec un environnement hétérogène. Lorsque l'on considère des services plus complexes (en termes de mobilité) tel que les services multimédias, les phases de *capture* et de *reprise* sont effectuées manuellement (et inconsciemment) par l'utilisateur.

Les solutions alternatives proposées par les participants et listées en Figure 8.2 permettent de transférer des données. Si le service est facilement identifiable en termes de ressources et permet à l'utilisateur de capturer une partie de son contexte (via un fichier par exemple), alors il peut essayer de bricoler une pseudo-mobilité de l'application. Mais un modèle de mobilité de service et plus que le transfert de données, il doit considérer l'environnement dans son ensemble, avec son hétérogénéité de terminaux, d'applications et de services. De plus, il doit fournir une solution unique, indépendante du type de service pour une simplicité et un praticité totale.

8.3.2 Évaluation quantitative

Les résultats de l'expérience précédente démontrent qu'une meilleure solution aurait certainement sa place pour offrir aux utilisateurs un moyen pratique et efficace de gérer leurs services dans un environnement hétérogène. Cependant, si notre système offre déjà de nouveaux usages, nous devons garantir qu'il est techniquement réalisable et qu'il répond de manière satisfaisante aux contraintes de continuité, notamment dans la perception du transfert.

L'objectif de cette expérimentation est d'analyser l'impact de notre modèle sur son environnement : terminal hôte, réseau, mais aussi application du point de vue de la continuité temporelle du transfert. Les mesures ont été effectuées sur le dSM, un prototype expérimental relativement instable et non optimisé. De plus nous avons dû modifier des applications pour les rendre compatibles, les tests seront donc uniquement basés sur ces services d'édition de texte, la performance d'autres types ne pouvant qu'être estimée théoriquement. En conséquence, les résultats obtenus n'offriront qu'une approximation mais qui apporteront déjà un aperçu des performances de notre modèle.

Méthode.

Pendant ces tests, nous avons réalisé un transfert entre deux terminaux. Nous avons choisi un service de type édition de texte qui est commun, simple, et dont l'implémentation n'est pas basée sur des sessions. Choisir un service « simple » était important car nous avons dû intégrer l'API dSM à deux applications de ce type en modifiant leur code source. De plus, un service local (sans session) était

préférable afin d'étudier un cas qui s'écarte des nombreuses solutions existantes basées sur des mécanismes de continuité de session (cf. 2.3.2).

Le transfert a été réalisé dans un environnement hétérogène, entre un ordinateur de bureau (PC) et un ordinateur portable (*Laptop*), chacun possédant des caractéristiques logicielles et matérielles différentes : CPU, mémoire, interface réseau, système d'exploitation (MS Windows et Linux) et application d'édition de texte (Java Notepad et TerpWord). Les applications origine et destination sont donc différentes : d'un côté (Java) Notepad, un éditeur basique (de type bloc notes) distribué comme exemple avec l'environnement de développement Java (*Java Development Kit*, JDK [94]) ; de l'autre TerWord, un éditeur plus riche, également open source et écrit en Java, qui fait partie de la suite bureautique TerpOffice développée par l'Université du Maryland [95]. Les deux applications ont été choisies pour leur simplicité et l'ouverture de leur code Java permettant une intégration aisée de notre API. Les caractéristiques des terminaux de test sont résumées dans la Table 8.2.

TABLE 8.2 – Environnement de test de l'évaluation quantitative.

	Laptop	**Personal Computer**
Operating System	MS Windows	Linux (kernel 2.6)
Text application	TerpWord	(Java) Notepad
Processor	Dual Core 1.2GHz	Single Core 2.5Ghz
Memory	1.5GB	2.5GB
Network	WiFi 11Mb/s	Ethernet 100Mb/s

Le scénario de l'expérience est le suivant (cf. Figures 8.3 et 8.4). Initialement le PC et le *Laptop* sont connectés via un réseau local par leurs interfaces respectives. Il est à noter que le réseau est capable de gérer des paquets en IP Multicast ([96]). Une instance du prototype, le DSM est lancée sur chaque terminal. Une fois que les deux terminaux se sont mutuellement découverts via le DSM (1), l'application TerpWord est démarrée sur le *Laptop* (2), résultant d'une notification immédiate de « nouveau service » sur le PC et affichée sur l'interface DSM de ce terminal.

Un texte quelconque est alors entré dans l'éditeur et le curseur est positionné à un endroit précis afin de vérifier la continuité après le transfert (a). Alors, depuis l'interface dSM du PC, nous sélectionnons l'unique service affiché, c'est à dire celui fourni par l'application TerpWord, et nous déclenchons le transfert vers le terminal courant, le PC (3). TerpWord entre alors en mode *pause*.

L'application locale du PC associée au service d'édition de texte est alors automatiquement exécutée en mode *resume* par le dSM (4). Le (Java) Notepad est alors lancé avec comme paramètre le contexte du service distant (un *snapshot* plus exactement, cf. 7.2.3) et il demande immédiatement au dSM les ressources qu'il désire (5). Ces ressources sont transférées par le distributed Service Manager du terminal origine jusqu'au Notepad qui restitue le service d'édition de texte de l'utilisateur (6). Finalement, l'application TerpWord est arrêtée automatiquement, le service étant maintenant disponible sur le PC, via l'application Notepad (b).

Durant ce scénario, nous avons relevé diverses informations. Pour estimer l'impact du dSM sur son hôte, nous avons mesuré les consommations CPU et mémoire grâce à un outil système, le *Reliability and Performance Monitor* [97], une application Microsoft qui fournit des informations précises sur les processus (ici le dSM). Nous nous sommes également intéressés aux performances du transfert, nous avons donc mesuré l'impact de notre solution sur le réseau en termes de délai et d'*overhead* engendrés par les mécanismes de mobilité. Ces mesures ont été réalisées à partir de l'analyse des paquets de données (grâce à l'outil *Wireshark* [98]) et les logs générés par les dSM des deux terminaux (les horloges internes ayant été synchronisées préalablement).

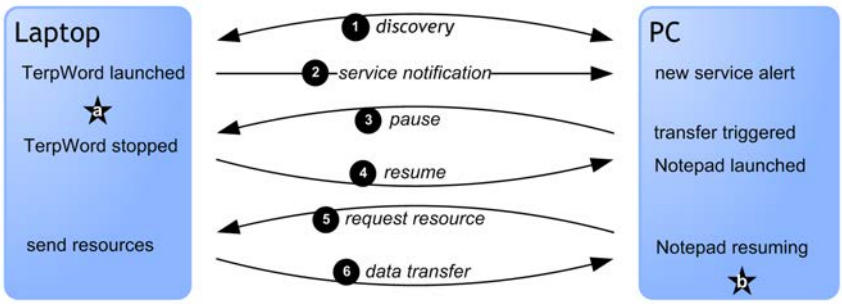

FIGURE 8.3 – Le scénario expérimental.

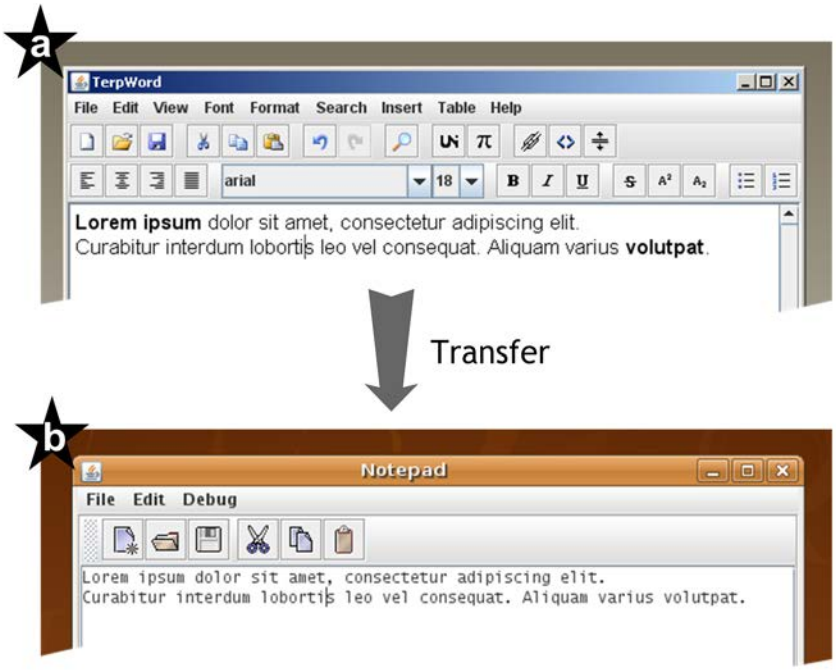

FIGURE 8.4 – Le service avant et après le transfert.

Résultats.

Nous avons mesuré l'impact du DSM sur son hôte pendant toute la durée du scénario décrit ci-dessus et selon trois métriques : processeur, mémoire et réseau. Les données acquises sont reportées dans la Figure 8.5 qui montre l'utilisation du CPU (pourcentage de la capacité totale du terminal), la quantité de mémoire allouée (en Mégaoctets) et l'activité réseau (en octets) pendant environ 50 secondes. Essayons d'analyser ces résultats.

D'après le graphique présenté en Figure 8.5, les consommations CPU et mémoire augmentent très fortement dans les premières secondes (33% de CPU et 26Mo de mémoire), correspondant logiquement au démarrage et à l'initialisation du DSM. On notera par la même occasion que le prototype est instantannément opérationnel, la charge CPU chutant très rapidement.

Après 12 secondes (noté T12), un autre pic d'activité CPU (15%) et réseau (700o) est observable, accompagné d'une légère augmentation de la consommation

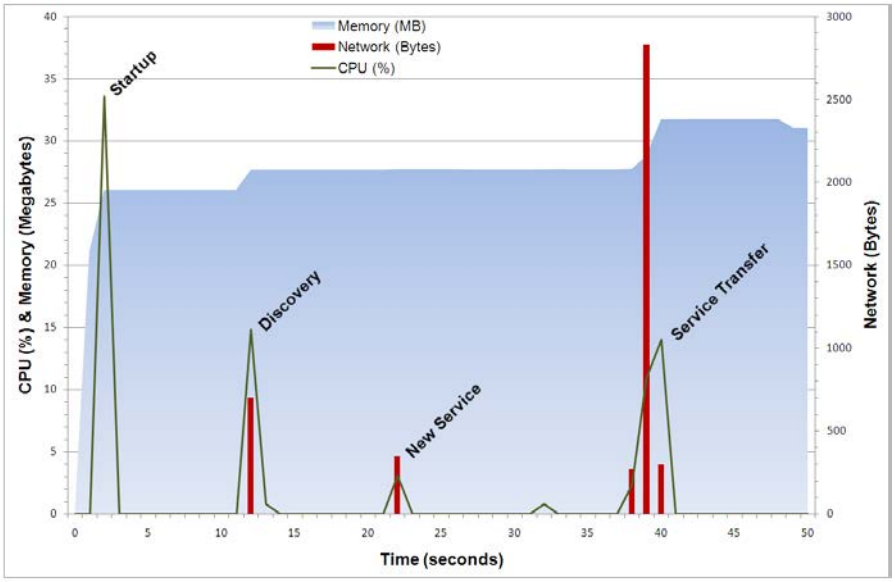

FIGURE 8.5 – Performance du DSM pendant un transfert.

mémoire (+2Mo). Cela correspond à la découverte d'un nouveau terminal, le DSM du PC distant ayant été démarré à T11, un échange de messages « hello » permettant l'identification mutuelle des terminaux a créé cette activité (cf. étape (1) dans la Figure 8.3).

L'application TerpWord est démarrée sur le *Laptop* à T21, en conséquence de nouveaux messages de signalisation sont échangés entre le DSM et l'application (processus d'enregistrement) et entre les DSM des deux terminaux (processus de notification). On observe un léger pic d'activité CPU (3%) et réseau (348o) à cette occasion à T22 (cf. étape (2) dans la Figure 8.3).

Enfin, l'activité principale observée de T37 à T41 correspond au mécanisme de transfert (cf. étapes (3-6) dans la Figure 8.3) déclenché depuis le PC à T36. Plusieurs actions sont réalisées ici : l'application est mise en *pause* et quelques messages (*snapshot*, notifications, etc) sont envoyés pour initier la reprise du service sur le terminal distant. Ensuite, chaque ressource demandée est transférée par le DSM local. Pendant le transfert, l'activité CPU atteint brièvement 15%, l'allocation mémoire est augmentée de 4Mo (pour une total de 32Mo) dû à la gestion des ressources par le DSM et l'activité globale de l'interface réseau (trafic entrant

et sortant) est d'approximativement 3,4ko (à noter que le trafic interprocessus dSM⇔Terpword est inclus dans cette mesure, cf. 8.3.2).

Le service est effectivement transféré et repris sur le PC à T41, pour un délai total de *terminal handover* égal à 5 secondes. Un résultat tout à fait satisfaisant pour ce type de service. De plus, il faut prendre en considération que l'utilisateur ne percevra pas un délai de 5 secondes, l'application destination étant lancée instantanément lors du déclenchement, avant même que le contexte ne soit effectivement transféré et traité par la fonction *resume*. La Figure 8.4 présente deux captures d'écran du même service avant et après le transfert : d'abord instancié par l'application TerpWord (a), puis par l'application Java Notepad (b). Vous noterez que le service a été adapté à l'application destination, le curseur et le texte ont été correctement restitués, par contre le formatage de certains caractères (en gras) a été ignoré car non supporté par le Java Notepad. Si un transfert supplémentaire est réalisé depuis cette instance du service et que l'application destination supporte cette ressource (formatage), alors les caractères en gras seront de nouveau restitués (principe de transitivité, cf. 7.2.3).

Discussion.

Cette deuxième expérimentation nous a permis de quantifier l'impact du dSM sur son environnement et d'évaluer la performance du transfert. Or ces valeurs sont approximatives car basées sur un prototype, des précisions complémentaires sont nécessaires pour une compréhension correcte des résultats.

Nous pensons que les métriques les plus importantes dans l'évaluation quantitative de la mobilité sont l'activité réseau et le délai de transfert car elles ont un impact direct sur l'environnement et la perception de continuité de l'utilisateur. Les mesures que nous avons réalisées pendant le transfert (de T37 à T41 cf. 8.5) ont donné des valeurs très satisfaisantes, cependant elles pourraient être considérablement optimisées. En effet, durant cette période, l'activité réseau totale est estimée à 3.4ko, or la bande passante mesurée ici inclut deux interfaces, dont une qui n'a pas d'impact sur l'environnement car seulement locale : dSM⇔dSM (distant) et dSM⇔application (locale au terminal). De plus, notre prototype étant développé en Java pour des raisons de rapidité d'implémentation et de déploiement, l'ensemble des mesures effectuées incluent la Machine Virtuelle Java (JVM

[99]) qui exécute l'application dSM. Cette JVM permet une portabilité de notre système mais elle le pénalise largement en termes de performances d'exécution (CPU et mémoire notamment).

La JVM n'est pas le seul facteur limitant les performances de notre système, la gestion du réseau P2P sous-jacent est assurée par la bibliothèque Java JXTA (actuellement la seule API ouverte de ce type) qui induit également une charge supplémentaire au niveau CPU et réseau (cf. initialisation à T12 dans la Figure 8.5). De plus, JXTA implémente de nombreux mécanismes de contrôle basé sur TCP et inutiles dans le cadre de notre système (UDP aurait été préféré), un *overhead* excessif est produit par ces fonctions. Dans la mesure du possible, nous avons essayé de ne pas prendre compte cet *overhead* dans nos mesures car il fausse l'évaluation des mécanismes de mobilité. Pour information, l'activité réseau incluant cet *overhead* durant la phase de transfert s'élève à 180ko... pour seulement 3,4ko de données utiles.

Nous avons indiqué au début de cette section que l'activité réseau mesurée correspond à la quantité de données échangées aux interfaces du dSM. Or le dSM possède des interfaces ouvertes vers l'extérieur qui ont un impact direct sur l'environnement mais également des interfaces internes permettant de communiquer avec les applications locales. Divers messages tels que des notifications pour l'interface utilisateur, qui n'interviennent pas dans la réalisation du transfert ont également été mesurés et surévaluent l'activité réseau inhérente au modèle. Pour être précis, seuls quatre types de messages sont échangés durant cette période : la requête PAUSE (déclenchement du transfert), la requête *Service Snapshot* (le contexte), les *Ressource Request* (demandes de ressource) et finalement les données (ressources). Ainsi, l'activité réseau engendrée par notre système pour la réalisation d'un transfert de service peut être exprimée par la formule :

$$\sum_{i=1}^{n} (size_i + \sigma_{res}) + \sigma_{tot} \tag{8.1}$$

Où n est le nombre de ressources effectivement *transférées* (uniquement celles supportées par l'application destination), $size_i$ la taille de la ressource i, σ_{res} l'*overhead* par ressource et σ_{tot} l'*overhead* global généré par l'ensemble des messages de signalisation. D'après notre scénario et les mesures effectuées, $n = 2$: le

corps du texte ($size_1 = 128$ octets) et la position du curseur ($size_2 = 2$ octets), l'*overhead* par ressource $\sigma_{res} = 90$ octets et l'*overhead* global $\sigma_{tot} = 990$ octets. Ainsi, appliqué à notre scénario de mobilité d'un service d'édition de texte, l'activité réseau exacte liée au transfert est de 1,3ko seulement, comparé aux 3,4ko donnés par une mesure globale.

Notons que cette valeur pourrait encore être optimisée, en effet les messages de signalisation du dSM sont en XML, un format de présentation de données particulièrement verbeux [100]. Or un algorithme de compression pourrait être appliqué pour réduire la quantité de données transmises en contrepartie d'une activité CPU revue à la hausse. Pour exemple, un ratio de compression de 1,4 peut être facilement atteint sur ce type de données grâce à l'outil *gzip* [101].

D'après la formule précédente (cf. Eq. (8.1)), nous pouvons à présent estimer l'impact réseau de notre système appliqué à tout type de service. Cependant, nous ne pouvons estimer la performance de transfert a priori sans connaître les détails d'implémentation du service pour une application donnée ; l'efficacité des mécanismes de continuité dépendant directement de cette implémentation : nombre, type et taille des ressources, complexité et durée des processus internes, volatilité du contexte, etc. Néanmoins, notre modèle assure la mobilité de tout service transférable grâce à un mécanisme original de gestion du contexte (*snapshots*, *anchors*) qui optimise le processus de continuité :

- priorités de transfert des ressources pour une adaptation et une reprise efficace du service,
- gestion de toutes les ressources, même les intransférables par redirection de flux,
- faible impact sur l'environnement, moins de 0.01% d'*overhead* par exemple pour un service constitué de trois ressources de 5Mo chacune.

Si la transférabilité de *tous* les services est fortement améliorée grâce au distributed Service Manager considérant l'expérience utilisateur ou d'un point de vue purement technique, la mobilité reste (et restera) dépendante des capacités du réseau sous-jacent. En d'autres termes, transférer un service qui manipule un contexte de 200Mo via un accès 3G (limité à 2Mb/s) sera réalisable si et seulement si il est possible d'extraire une quantité de données compatible aux capacités de

l'environnement (notamment les propriétés du réseau) et suffisante à l'application destination pour assurer un service minimal.

8.4 Conclusion

Dans ce chapitre, nous avons prouvé la faisabilité de notre modèle générique grâce à l'implémentation d'un prototype du distributed Service Manager. Nous avons essayé d'évaluer ce système dans un environnement hétérogène en s'intéressant aux contraintes de continuité contextuelle et temporelle.

Ainsi nous l'avons évalué d'un point de vue qualitatif afin de positionner notre solution par rapport aux besoins utilisateurs et à leur expérience de la mobilité. Nous avons identifié un réel besoin, des solutions inexistantes et une approche intuitive de mobilité qui se calque sur les mécanismes que nous avions conçus.

Ensuite nous avons évalué notre prototype d'un point de vue performance dans un scénario de mobilité classique. Nous avons mesuré l'impact de notre solution sur son environnement : l'application, le terminal hôte, le réseau. Nous avons estimé la performance des mécanismes de transfert, les délais et l'*overhead* induits. Enfin nous avons formulé de manière précise l'impact réel sur le réseau de notre système en fonction des caractéristiques d'un service à transférer ; cette formule permet de calculer la quantité de données effectivement transférées à partir d'un contexte donné. Elle pourra être appliquée à d'autre types de services que nous n'avons pas eu le temps de traiter ici.

Le bilan des tests réalisés sur notre prototype est satisfaisant. Tout d'abord, nous avons pu démontrer la faisabilité de notre modèle dans un environnement hétérogène. Le système a répondu avec succès aux contraintes quantitatives ; les transferts sont rapides, optimisés et l'impact du DSM sur son environnement est faible : *overhead* minimal, terminal peu affecté, implémentation triviale sur les applications compatibles... Les contraintes qualitatives sont également considérées ; la gestion de bout en bout de la mobilité de manière générique et transparente apporte la solution qui manque aujourd'hui à l'utilisateur afin qu'il gère de manière intuitive, simple et pratique ses services dans son environnement hétérogène.