Testing and Validation of Peer-to-peer Systems

Eduardo Cunha de Almeida

École Centrale de Nantes Université de Nantes École des Mines de Nantes

ÉCOLE DOCTORALE STIM

« SCIENCES ET TECHNOLOGIES DE L'INFORMATION ET DES MATÉRIAUX »

Année 2009

Test et Validation des Systèmes Pair-à-pair

THÈSE DE DOCTORAT

Discipline : INFORMATIQUE

Présentée
et soutenue publiquement par

Eduardo Cunha DE ALMEIDA

*Le 25 février 2009 à l'*UFR *Sciences & Techniques, Université de Nantes,*
devant le jury ci-dessous

Président	:	Yves Le Traon, Professeur	ENST Bretagne
Rapporteurs	:	Pierre Sens, Professeur	Université Paris 6
		Thierry Jéron, Directeur de recherche	INRIA, Rennes
Examinateurs	:	Patrick Valduriez, Directeur de recherche	INRIA, Nantes
		Gerson Sunyé, Maître de conférence	Université de Nantes

Directeur de thèse : Pr. Patrick Valduriez
Encadrant : Gerson Sunyé

Laboratoire: **L**ABORATOIRE D'**I**NFORMATIQUE DE **N**ANTES **A**TLANTIQUE.
UMR CNRS 6241. 2, rue de la Houssinière, BP 92 208 – 44 322 Nantes, CEDEX 3.
N° ED 503-036

TEST ET VALIDATION DES SYSTÈMES PAIR-À-PAIR

Testing and Validation of Peer-to-peer Systems

Eduardo Cunha DE ALMEIDA
⋈

favet neptunus eunti

Université de Nantes

Eduardo Cunha DE ALMEIDA

Test et Validation des Systèmes Pair-à-pair

IV+XVI+118 p.

This document was edited with `these-LINA` v. 2.7 LaTeX2e class of the "Association of Young Researchers on Computer Science (LOGIN)" from the University of Nantes (available on: `http://login.irin.sciences.univ-nantes.fr/`). This LaTeX2e class is under the recommendations of the National Education Ministry of Undergraduate and Graduate Studies (circulaire n° 05-094 du 29 March 2005) of the University of Nantes and the Doctoral School of « Technologies de l'Information et des Matériaux(ED-STIM) », et respecte les normes de l'association française de normalisation (AFNOR) suivantes:

- AFNOR NF Z44-005 (décembre 1987)
  Documentation – Références bibliographiques – Contenu, forme et structure;
- AFNOR NF Z44-005-2/ISO NF 690-2 (février 1998)
  Information et documentation – Références bibliographiques – Partie 2 : documents électroniques, documents complets ou parties de documents.

Print : these_eduardo.tex – 27/02/2009 – 15:58.

Last class review: these-LINA.cls,v 2.7 2006/09/12 17:18:53 mancheron Exp

Abstract

Peer-to-peer (P2P) offers good solutions for many applications such as large data sharing and collaboration in social networks. Thus, it appears as a powerful paradigm to develop scalable distributed applications, as reflected by the increasing number of emerging projects based on this technology.
However, building trustworthy P2P applications is difficult because they must be deployed on a large number of autonomous nodes, which may refuse to answer to some requests and even leave the system unexpectedly. This volatility of nodes is a common behavior in P2P systems and can be interpreted as a fault during tests.
In this thesis, we propose a framework and a methodology for testing and validating P2P applications. The framework is based on the individual control of nodes, allowing test cases to precisely control the volatility of nodes during their execution.
We also propose three different architectures to control the execution of test cases in distributed systems. The first approach extends the classical centralized test coordinator in order to handle the volatility of peers. The other two approaches avoids the central coordinator in order to scale up the test cases.
We validated the framework and the methodology through implementation and experimentation on two popular open-source P2P applications (i.e. FreePastry and OpenChord). The experimentation tests the behavior of the system on different conditions of volatility and shows how the tests were able to detect complex implementation problems.

Keywords: testing, peer-to-peer

Résumé

Le pair-à-pair (P2P) offre de bonnes solutions pour de nombreuses applications distribuées, comme le partage de grandes quantités de données et/ou le support de collaboration dans les réseaux sociaux. Il apparaît donc comme un puissant paradigme pour développer des applications distribuées évolutives, comme le montre le nombre croissant de nouveaux projets basés sur cette technologie.
Construire des applications P2P fiables est difficile, car elles doivent être déployées sur un grand nombre de noeuds, qui peuvent être autonomes, refuser de répondre à certaines demandes, et même quitter le système de manière inattendue. Cette volatilité des noeuds est un comportement commun dans les systèmes P2P et peut être interprétée comme une faute lors des tests.
Dans cette thèse, nous proposons un cadre et une méthodologie pour tester et valider des applications P2P. Ce cadre s'appuie sur le contrôle individuel des noeuds, permettant de contrôler précisément la volatilité des noeuds au cours de leur exécution.
Nous proposons également trois différentes approches de contrôle d'exécution de scénarios de test dans les systèmes distribués. La première approche étend le coordonnateur centralisé classique pour gérer la volatilité des pairs. Les deux autres approches permettent d'éviter le coordinateur central afin de faire passer à l'échelle l'exécution des cas de tests.
Nous avons validé le cadre et la méthodologie à travers la mise en oeuvre et l'expérimentation sur des applications P2P open-source bien connues (FreePastry et OpenChord). Les expérimentations ont permis de tester le comportement des systèmes sur différentes conditions de volatilité, et de détecter des problèmes d'implémentation complexes.

Mots-clés: testing, pair-à-pair

ACM Classification

Categories and Subject Descriptors : C. [**Computer Systems Organization**]: C.2 COMPUTER-COMMUNICATION NETWORKS; C.2.4 [**Distributed Systems**]: Distributed applications,Distributed databases; D. [**Software**]: D.2 SOFTWARE ENGINEERING; D.2.5 [**Testing and Debugging**]: Testing tools (e.g., data generators, coverage testing).

General Terms: Experimentation, Measurement, Performance, Reliability, Verification.

Acknowledgements

First of all, I would like to thank my advisor Patrick Valduriez and my co-advisor Gerson Sunyé who received me at the University of Nantes and gave me all the help I needed to carry out my research. I am very grateful for their remarkable patience, motivation and enthusiasm.

Thanks to the jury members : the referees Pierre Sens and Thierry Jéron for their valuable and detailed comments on my thesis and Yves Le Traon for his interest in my work. In special, Yves Le Traon, who accompanied my work along this thesis. Thank to the Programme Alban, the European Union Programme of High Level Scholarships for Latin America, scholarship no. E05D057478BR, that supported this work.

Thanks to Marcos Sunye, Ligia Setenareski, Marcos Castilho, Luis Bona, Fabiano Silva, Alex Direne e Izabella Fernandes, who believed in my work and motivated me to pursue a Ph.D. in the first place.

Thanks to my friends at LINA Sandra Lemp, Vidal Martins, Jorge Manjarrez, Rabab Hayek, Lorraine Goeuriot, Reza Akbarinia, Manal El-Dick, Wenceslao Palma, William "Dude" Kokou, Mounir Tlili, Fabrice Evan, Ricardo Soto, Jean-Marie Normand, Guillaume Raschia and Elodie Lize. Specially, Marcos Didonet for the beers, pizzas, advices, and sometimes just plain listening. Anthony Ventresque for the rugby injuries and Jorge "Ufo" Quiane, who shared every step I did along this journey.

Thanks to my friends outside LINA who made my life in a foreign country a happy and unforgettable experience : Thomas, Julie, Fabienne, Juca, Maryvonne, Mariana, all the guys from the volleyball Club and all the guys from the VSN Rugby.

Thanks to my friends in Brazil who had never forget us either coming in here or keeping in touch. The list is huge ! Fortunately !

My familly deserves more credit than I could ever express. Specially, my mother and my grandma for encouraging me all the time and for your unconditional support and love. You have enlighten my way. If I am able to provide the same level of love and support for my children someday, I will consider myself a successful parent. My father-in-law, Cido, and my mother-in-law Bela that spent a lot on phone bills and helped my out preserving my sanity ! My aunt Rô and uncles Luis and George that always encouraged me and were kind receiving my fancy postcards. I also thank, Sue, Eto and Bruno, for all the support and for visiting us here as well.

Finally, I dedicate this thesis to my loving wife, Siloé and my beautiful daughter, Juliana. Siloé, during how many difficult times in the whole process did you say to me "It’s worth it to be here, don’t worry" ? I needed to hear every time. You believed in me, supported me, and convinced me I had the strength to keep going. Juliana, I hope one day you read this, know that you are my source of inspiration, strength and happiness. Without you two, this thesis would not have been possible.

Versão brasileira

Inicialmente, agradeço meus orientadores Patrick Valduriez e Gerson Sunyé pelo apoio e ajuda durante minha pesquisa. Sou muito grato por sua paciência, motivação e entusiasmo.

Agradeço aos membros do júri : os relatores Pierre Sens e Thierry Jéron pelos valiosos e detalhados comentários sobre minha tese, e Yves Le Traon pelo seu interesse em meu trabalho. Em especial, Yves Le Traon que acompanhou de perto toda minha tese. Agradeço igualmente ao Programme Alban, programa de bolsas de alto nível da União Européia para América Latina, bolsa nº E05D057478BR, pelo apoio financeiro.

Agradeço meus amigos da UFPR Marcos Sunye, Ligia Setenareski, Marcos Castilho, Luis Bona, Fabiano Silva, Alex Direne e Izabella Fernandes que foram os primeiros a acreditar no meu trabalho e me motivar a fazer um doutorado.

Agradeço meus amigos aqui na França que fizeram de nossa estadia a mais agradável e inesquecível possível. Agradeço também meus amigos do Brasil que nunca nos esqueceram seja estando aqui ou mantendo contato : Fabio, Dauriane, Pirri, Christiane, Murilo, Jacheline, Sanfelice, pessoal do CMP, etc.

Minha família merece um crédito especial. Especialmente minha mãe e avó por terem sempre me encorajado e pelo seu amor incondicional. Vocês iluminaram meu caminho. Se eu for capaz de dar o mesmo nível de amor e apoio aos meus filhos, irei me considerar um pai muito bem sucedido. Meu sogro e sogra Cido e Bela que gastaram uma nota em telefone e me ajudaram a manter a sanidade até o fim. Minha tia Rô e meus tios Luis e George que nunca deixaram de me encorajar e de receber lindos cartões postais, e meus cunhados Sue, Eto e Bruno, que além disso tudo, estiveram aqui.

Finalmente, eu dedico esta tese a minha amada esposa Siloé e a minha linda filha, Juliana. Siloé, quantas vezes em tempos difíceis você me disse "Vale a pena estar aqui, não se preocupe" ? Eu precisava ouvir cada vez. Você acreditou em mim, me apoiou, e me convenceu que eu tinha forças para continuar. Juliana, quando ler esta dedicatória algum dia, saiba que você é a minha fonte de inspiração, força e alegria. Sem vocês duas, esta tese não seria possível.

Eduardo

Table of Contents

—Appendixes—

List of Tables

—Body of the Dissertation—

—Appendixes—

List of Figures

—Body of the Dissertation—

—*Appendixes*—

List of Examples

—Body of the Dissertation—

—Appendixes—

Résumé Étendu

Introduction

Motivation

Les systèmes pair-à-pair (P2P) mettent en œuvre une approche de partage de ressources complètement décentralisée. En distribuant le stockage et le traitement des données sur un grand nombre de pairs autonomes, ils permettent de passer à l'échelle sans avoir besoin de recourir à de grands serveurs. Des exemples célèbres de systèmes P2P comme Gnutella et Freenet [1] ont des millions d'utilisateurs qui partagent des petaoctets de données sur l'Internet. Bien que très utiles, ces systèmes sont très simples (par ex. il ne partagent que des fichiers) et offrent des fonctionnalités limitées (par ex. la recherche par mot-clé).

Les systèmes P2P n'offrent pas seulement des solutions pour gérer des grandes bases de données, mais aussi pour la plupart des systèmes existants dans différents domaines : affaires, médical ou même militaire. Les systèmes P2P apparaissent donc comme un puissant paradigme pour mettre au point des systèmes distribués qui passent à l'échelle, comme le démontre le nombre croissant de nouveaux projets basés sur cette technologie [12].

Parmi les nombreux aspects qui concernent développement des systèmes P2P, rendre les systèmes robustes est un objectif évident. Cela est d'autant plus critique lorsque les systèmes P2P sont largement utilisés. Alors, comme pour tous les autres systèmes, un système P2P doit être testé en fonction de ses besoins. Comme pour tout système réparti, la complexité de la communication et des échanges de messages doivent faire partie des objectifs de test. Classiquement, la validation d'un systèmes distribuée se compose d'une architecture centralisée composée d'un contrôleur de test, ou coordinateur, qui synchronise et coordonne les communications (les messages d'appel, la détection de blocages) et crée l'ensemble des verdicts locaux [101, 89, 76, 75, 37, 59, 5, 35]. Localement à chaque nœud, les séquences de test ou des automates de test peuvent exécuter partiellement ces tests à la demande et retourner leurs verdicts au coordinateur. Un testeur local par nœud ou par groupe de nœuds est produit à partir des objectifs de test. Le testeur observe ensuite les événements envoyés entre les différents nœuds du système et vérifie si la séquence des événements correspond à la machine d'état fini [25, 54, 24] ou aux systèmes de transition étiqueté [58, 85, 59].

Dans un système P2P, un pair joue le rôle d'un nœud actif, qui peut rejoindre ou quitter le réseau à tout moment, soit normalement (par exemple, par déconnexion) soit anormalement (par exemple, en raison d'une défaillance). Cette capacité, que nous appelons la volatilité, est une différence majeure avec les systèmes distribués. En effet, la volatilité modifie dynamiquement la taille du réseau et sa topologie, ce qui rend le test des systèmes P2P tout à fait différent. Ainsi, le comportement fonctionnel d'un système P2P (et les défauts fonctionnels) dépend fortement du nombre de pairs et de leur volatilité.

Certaines méthodes de test pour les systèmes P2P proposent d'arrêter aléatoirement l'exécution des pairs [13, 69], ou même d'insérer des erreurs dans le réseau [55, 76]. Bien que ces approches soient utiles pour observer le comportement de l'ensemble du système, elles ne sont pas totalement adaptées pour le tester. De plus, puisqu'elles se focalisent sur la tolérance aux perturbations du réseau, elles ne parviennent pas à détecter les défauts des logiciels, spécialement ceux qui se produisent en raison de la volatilité des pairs.

Comme nous avons dit précédemment, une des caractéristiques des systèmes P2P est le passage à l'échelle. Ainsi, l'architecture de test doit passer à l'échelle elle-aussi, ce qui n'est pas le cas de l'architecture de test centralisée (c'est-à-dire, du coordinateur central de test) [44, 64, 33, 71, 50, 4].

Il est cependant possible d'utiliser les testeurs distribués pour lesquels les cas de test sont coordonnées par des messages échangés dans des points de communication (PC) [101, 98, 83]. Toutefois, même avec ces PC, cette architecture de test perturbe la performance d'un système P2P. En effet, la performance (algorithmique) de cette architecture de test est linéaire avec le nombre de pairs, tandis qu'un système P2P typique peut passer à l'échelle logarithmiquement. Pour éviter que l'intrusion des testeurs menacent la validité des résultats des tests, l'architecture de test doit avoir une performance logarithmique dans le pire des cas.

Il y a aussi des architectures de test qui s'intéressent à la volatilité ainsi qu'au passage à l'échelle, comme par exemple, P2PTester présenté par Butnaru et al. [42], Pigeon présenté par Zhou et al.[110], et d'un cadre présenté par Hughes et al. [57]. Toutefois, toutes ces architectures demandent l'ajout de code supplémentaire dans le code source du système sous test. Cette inclusion peut être soit manuelle, par exemple en utilisant des interfaces spécifiques, comme dans P2PTester, ou automatique, en utilisant la réflexivité et la programmation par aspects, comme dans Pigeon.

Néanmoins, l'inclusion de code supplémentaire peut engendrer des erreurs et, par conséquent, contaminer les résultats des tests. En outre, il est difficile de vérifier si l'erreur provient du système sous test ou de l'architecture de test.

Finalement, il y a aussi des outils de vérification de modèle (par exemple, Bogor[14], SPIN[37, 108] et Cadence SMV [49]), qui permettent de tester un type de système P2P appelé système de publicat-souscript (PSS) [22]. D'une part, la vérification de modèle est une alternative attrayante pour trouver des bogues dans les systèmes en explorant tous ses états d'exécution[49, 60]. D'un autre part, elle est toujours basée sur les modèles, et les problèmes de mise en œuvre ne peuvent pas être trouvés. Même si en considérant la volatilité des PSS, le passage à l'échelle est une question ouverte. Tout au long d'un test, les pairs sont simulés en tant que "threads" et la taille du système à tester est limitée par les ressources de la machine de test. Par conséquent, les systèmes P2P à grande échelle ne peuvent pas être pleinement testés [33]. En outre, aucune de ces approches ne peut faire face à la volatilité des pairs qui peut conduire à un nombre exponentiel d'états d'exécution.

Contributions

Dans cette thèse, nous proposons une solution intégrée pour la création et le déploiement d'un environnement de validation des systèmes P2P, avec la possibilité de créer des pairs et de les faire rejoindre et quitter le système. Les objectifs de test peuvent combiner le contrôle du fonctionnement du système avec des variations de la volatilité (et aussi la passage à l'échelle). L'exactitude du système peut donc être validée sur la base de ces trois dimensions, c'est-à-dire les fonctions, le nombre de pairs et la volatilité des pairs.

Tout d'abord, nous proposons une méthode incrémentale pour prendre en compte ces trois dimensions. Cette méthode vise à couvrir les fonctions d'abord sur un petit système et progressivement de répondre aux questions liées à la volatilité et au passage à l'échelle [33].

Nous proposons également trois différentes architectures de coordination de test. Ces architectures sont basées sur deux aspects originaux : (i) le contrôle individuel de la volatilité des pairs et (ii) une architecture répartie de tests pour faire face à un grand nombre de pairs. Les avantages de ces architectures sont (1) d'automatiser l'exécution des tests localement à chaque pair, (2) de construire automatiquement un verdict global, (3) de permettre un contrôle explicite de la volatilité de chaque pair.

Nous avons mis en œuvre l'expérimentation sur deux systèmes P2P open-source. Durant l'expérimentation, nous avons analysé le comportement des ces deux systèmes avec des conditions différentes de volatilité et nous avons montré que nos contributions sont capables de détecter des problèmes de mise en œuvre. Nous avons également validé notre méthodologie progressive de test, montrant comment les trois aspects dimensionnels de test augmentent la couverture du code et par conséquent, la qualité des tests.

Validation des systèmes Pair-à-pair

Par rapport aux systèmes distribués classiques, les systèmes P2P sont totalement décentralisés. Un pair se comporte à la fois comme le client et le serveur d'une architecture client-serveur [16]. Dans un système P2P, les pairs communiquent les uns avec les autres directement dans une façon décentralisé sans la nécessité d'un serveur central. En raison de la décentralisation, les pairs sont fonctionnellement identiques, mais avec la capacité d'intégrer ou de quitter le système à tout moment. Cette particularité ajoute la possibilité de modifier dynamiquement la taille du réseau et la topologie résultant en systèmes à grande échelle.

Cette décentralisation se traduit aussi par une haute tolérance aux pannes car il n'existe pas de point individuel de défaillance. Sans un serveur central, n'importe quel pair dans le système peut assumer la responsabilité d'un autre pair en échec.

En outre, la décentralisation signifie qu'il est impossible de contrôler la propagation des pairs à travers le réseau, et que chaque pair peut décider de partager ou non ses ressources.

Nous pouvons classer les principales caractéristiques des systèmes P2P comme suit :

- les pairs sont décentralisés. Il n'y a pas de serveur central, les pairs sont répartis sur le réseau. Cela assure la tolérance aux pannes, ainsi que le passage à l'échelle.
- les pairs sont volatiles. Un pair peut joindre ou quitter le système à tout moment. Deshpande et al.[39] décrivent que la compréhension de la volatilité peut être un facteur clé dans la conception efficace des architectures P2P.
- les pairs sont autonomes. Un pair décide de partager son contenu et/ou ces ressources avec d'autres pairs.

Ces caractéristiques sont directement affectées par les différentes topologies et les structures d'un système P2P. Ces topologies et structures forment un réseau P2P également dénommé réseau logique (network overlay). En fait, un système P2P s'appuie sur un réseau P2P pour fonctionner. Les trois topologies principales de réseaux P2P sont :

- Non structurées. Les pairs s'organisent dans un cadre non-déterministe sans une structure particulière [77, 7, 80]. Chaque fois qu'un nouveau pair se connecte au réseau, il contacte d'autres pairs de façon aléatoire. Ces pairs deviennent voisins et commencent à échanger des messages pour annoncer leur connexions. Les messages sont propagés récursivement par inondation du réseau. Les messages de recherche sont également envoyés par inondation du réseau (par exemple, Gnutella, FreeHaven[40], and Kazaa[65]) ;
- Structuré. Ce type de réseau apparaît comme une solution capable de résoudre les questions de passage à l'échelle des réseaux non structurées. Dans les réseaux structurés, les pairs peuvent s'organiser dans différents types de structures telles que l'anneau[92], l'arbre binaire[6], l'espace multidimensionnel[86], etc. Les structures sont maintenues par un mécanisme de routage utilisé pour organiser le système, et également d'envoyer et de transmettre des messages (par exemple,

FreeNet[1, 27], Chord[92, 31], CAN[86], Pastry[88], PAST[43], Tapestry[109], Kademlia[78], PIER[56], and P-Grid[6]) ;

- Super-pair. Dans une architecture typique, un super-pair conserve un index des méta-données pour tous les fichiers dans le réseau. Une fois qu'un pair p interroge le super-pair pour un fichier donné, il reçoit une liste des pairs qui le possèdent. Ensuite, p ouvre une connexion directe avec un ou plusieurs pairs afin de télécharger le fichier (par exemple, Publius[74], JXTA[2], Edutella[81, 82] and Napster[3]).

Une particularité d'un système P2P, c'est que son interface est répartie sur le réseau. Par exemple, une table de hachage distribuée (DHT) [88, 92, 86] fournit une interface locale simple qui ne contient que trois opérations : insertion d'objet, récupération d'objet et recherche d'une clé. Toutefois, l'interface distante est plus complexe. Elle fournit des opérations pour le transfert de données et l'entretien de la table de routage, c'est-à-dire la table de correspondance entre les clés et les pairs, utilisée pour déterminer qui est le responsable d'une clé donnée.

Vu la simplicité de l'interface locale, tester une DHT dans un système stable est assez simple, mais le test n'assure aucune confiance dans la robustesse de la mise en œuvre des mécanismes spécifiques aux systèmes P2P. Par exemple, lorsque les pairs quittent et rejoignent le système, un test doit vérifier que la table de routage est correctement mise à jour et que les demandes sont correctement acheminés.

Lors du test de passage à l'échelle d'un système distribué, les aspects fonctionnels ne sont généralement pas prises en compte. Le même scénario de test est simplement répété sur un grand nombre de nœuds [45]. Nous pouvons utiliser cette même approche, mais cela nous conduirait à tester la volatilité séparément de l'aspect fonctionnel. Pour un système P2P, nous prétendons que les défauts techniques sont étroitement liés à la taille du système et la volatilité. Par conséquent, il est crucial de combiner la taille du système et la volatilité avec des aspects significatifs de test. Ainsi, nous avons également proposer une méthode incrémentale pour aborder ces trois aspects [33].

Notre méthodologie vise à couvrir les fonctions d'abord sur un petit système et ensuite passer le système sous test à l'échelle pour finalement introduire la volatilité.

Les architectures de test proposés ici sont basées sur la architecture CTMF ("multi-party") [5] de coordination des séquences de test, soit par un composant centralisé ou par de composants distribués.

La première architecture étend l'architecture centralisée classique de test avec le contrôle de la volatilité. Le but est de démontrer que cette volatilité est un paramètre-clé lors de la validation d'un système P2P [34]. Cette architecture a deux composants principaux : le testeur et le coordinateur. Les testeurs contiennent les suites de tests et sont déployés sur plusieurs nœuds logiques. Le coordinateur est déployé dans un seul nœud et est utilisé pour synchroniser l'exécution des cas de test. Accessoirement, il agit comme un médiateur (*broker* [21]) pour les testeurs déployés.

Toutefois, la performance de cette architecture centralisée est au mieux linéaire, alors que des tests à grande échelle requièrent une performance logarithmique. Par conséquent, nous proposons deux architectures pleinement distribuées pour faire face à l'échelle des systèmes P2P [36]. La deuxième architecture organise les testeurs dans un B-arbre [17] où la synchronisation est effectuée à partir de la racine vers les feuilles. Ensuite, les testeurs communiquent les uns avec les autres dans l'ensemble du B-arbre pour éviter d'utiliser un coordinateur centralisé. La troisième architecture utilise la propagation de rumeurs entre les testeurs pour réduire les communications entre les testeurs chargés d'exécuter des actions de test. Comme les deux architectures distribuées ne reposent pas sur un coordinateur central, elles passent à l'échelle correctement.

Definitions

Soit P l'ensemble des pairs représentant le système P2P, qui est le système sous test (SUT). Nous désignons par T, où $|T| = |P|$ l'ensemble des testeurs qui contrôlent le SUT, par DTS la suite de tests qui vérifie P, et par A l'ensemble des actions exécutées par DTS sur P.

Malgré que tous les pairs aient exactement la même interface, tester l'interface d'un seul pair n'est pas suffisant pour tester le système dans son ensemble. C'est pourquoi nous introduisons la notion de cas de test distribué, c'est-à-dire, les cas de test qui s'appliquent à l'ensemble du système et dont les actions peuvent être exécutées par les différents pairs.

Definition 1 (Cas de test distribué)**.** *Un cas de test distribué noté τ est un N-uplet $\tau = (A^\tau, T^\tau, L^\tau, S^\tau, V^\tau, \varphi^\tau)$ où $A^\tau \subseteq A$ est un ensemble ordonné d'actions $\{a_0^\tau, \ldots, a_n^\tau\}$, $T^\tau \subseteq T$ est une séquence de testeurs, L^τ est un ensemble de verdicts locaux, S^τ est un table de correspondance, V^τ est un ensemble de variables et φ^τ est le niveau acceptable de verdicts $inconclusive$.*

La table de correspondance permet de mettre en relation les actions et les ensembles de testeurs, où chaque action correspond à un ensemble de testeurs que l'exécute.

Definition 2 (La table de correspondance)**.** *La table de correspondance $S = A \mapsto \Pi$, où Π est une séquence d'ensembles de testeurs $\Pi = \{T_0, \ldots, T_n\}$, et $\forall T_i \in \Pi : T_i \subseteq T$*

Dans les systèmes P2P, l'autonomie et l'hétérogénéité des pairs interfèrent directement dans l'exécution des demandes de service. Alors que les pairs plus proches peuvent répondre rapidement, les pairs plus distant ou surchargés peuvent présenter un retard considérable pour répondre aux demandes. Par conséquent, les clients ne s'attendent pas à recevoir un résultat complet, mais les résultats disponibles qui peuvent être récupérées pour un délai donné. Ainsi, les actions de cas de test ne doivent pas attendre indéfiniment pour obtenir des résultats, mais spécifier un délai maximum (*timeout*) d'exécution.

Definition 3 (Action)**.** *Un cas de test est un N-uplet $a_i^\tau = (\Psi, \iota, T')$ où Ψ est un ensemble d'instructions, ι est l'intervalle de temps dans lequel Ψ doit être exécuté et $T' \subseteq T$ est une séquence de testeurs qui exécutent l'action.*

Les instructions sont généralement des appels à l'interface de l'application P2P, ainsi que toute déclaration dans la langage de programmation du cas de test.

Definition 4 (Verdict local)**.** . *Un verdict local est calculé en comparant le résultat de test attendu, noté E, avec le résultat de test lui-même, noté R. E et R peuvent être constitués d'une valeur unique ou d'un ensemble de valeurs de n'importe quel type. Toutefois, ces valeurs doivent être comparables. Le verdict local v de τ sur ι est définie comme suit :*

$$l_\iota^\tau = \begin{cases} pass & \text{si } R = E \\ fail & \text{si } R \neq E \text{ mais } R \neq \emptyset \\ inconclusive & \text{si } R = \emptyset \end{cases}$$

Motivating test case example

Nous illustrons ces définitions avec un cas de test distribué simple (voir l'exemple 3.1). Ce test sera également utilisé comme exemple par la suite. L'objectif de ce test est de détecter des erreurs sur le mise en œuvre d'une DHT. Plus précisément, elle vérifie si de nouveaux pairs sont en mesure de récupérer les données insérées avant leur arrivée.

Example 1 (Simple cas de test distribué).

Action	Testeurs	Action
(a_1)	0,1,2	join()
(a_2)	2	Insérer la chaîne de caractères "un" associé à la clé 1 Insérer la chaîne de caractères "deux" associé à la clé 2 ;
(a_3)	3,4	join() ;
(a_4)	3,4	Récupération de données associées à la clé 1 ; Récupération de données associées à la clé 2 ;
(a_5)	*	leave() ;
(v_0)	*	Calculer un verdict ;

Ce test comprend cinq testeurs $T^\tau = \{t_0 \dots t_4\}$ qui contrôlent cinq pairs $P = \{p_0 \dots p_4\}$ et cinq actions $A^\tau = \{a_1^\tau, ..., a_5^\tau\}$. Si les données récupérées dans a_4 sont les mêmes que celles insérées dans a_2, alors le verdict est $pass$. Si les données ne sont pas les mêmes, le verdict est $fail$. Si t_3 ou t_4 ne sont pas en mesure de récupérer toutes les données, alors le verdict est $inconclusive$.

Méthodologie des tests

Plusieurs caractéristiques d'un système P2P démontrent le besoin d'une méthodologie précise de test, où la simplicité des interfaces contraste avec la complexité des facteurs qui peuvent affecter les tests : la volatilité, le nombre de pairs, la taille des données, la quantité de données, le nombre de demandes concurrentes, etc. Ainsi, la difficulté de tester ne réside pas seulement dans le choix des données d'entrée, mais aussi dans le choix des facteurs qui doivent varier, leurs valeurs et leurs associations.

Les aspects d'un système P2P en trois dimensions

Comme indiqué dans l'introduction, le test d'un système P2P aborde les questions classique de test des systèmes distribués, mais avec une dimension spécifique que nous appelons volatilité, qui doit être un paramètre explicite dans les objectifs de test. Deux solutions possibles peuvent être utilisées pour obtenir une séquence de test qui prend en compte la volatilité : (i) un profil de simulation en attribuant une probabilité pour chaque pair de quitter ou de rejoindre le système à chaque étape de la séquence d'exécution, (ii) de façon explicite à l'intérieur d'une séquence de test. La première solution est la plus facile à mettre en œuvre, par contre, elle rend l'interprétation des résultats difficile, car nous ne pouvons pas découvrir pourquoi la séquence de test a échoué. En outre, elle peut créer une distorsion avec les réponses tardives de certains pairs au cours de l'exécution de la séquence de test. En conséquence, cette méthodologie ne peut pas être utilisée pour combiner un comportement de test sémantiquement riche avec le paramètre de volatilité.

Dans cette thèse, nous recommandons de contrôler pleinement la volatilité dans la définition de la séquence de test. Ainsi, un des pairs, du point de vue de test, peut avoir à quitter ou rejoindre le système à un moment donné dans une séquence de test. Cette action est spécifiée dans la séquence de test dans une manière déterministe.

Etant donné que nous devons être en mesure de gérer un grand nombre de pairs, la deuxième dimension de tests d'applications P2P est le passage à l'échelle. Parce qu'il accomplit un traitement, ces deux

dimensions doivent être testés avec du comportement fonctionnel correct. Par conséquent, une méthodologie des tests et son cadre devrait prévoir la possibilité de contrôler les aspects d'un système P2P en trois dimensions. Nous les résumons comme suit :

- Les fonctionnalités, exprimé par un comportement à être exercée.
- Le passage à l'échelle, exprimé par le nombre de pairs dans le système.
- La volatilité, exprimé par le nombre de pairs qui se joindre ou quitter l'application après son initialisation au cours du test.

La méthodologie incrémentale de test

Pour prendre en compte les aspects d'un système P2P en trois dimensions, nous proposons une méthodologie qui combine le contrôle du fonctionnement d'un système avec les variations des deux autres aspects. En effet, nous augmentons progressivement l'échelle du SUT avec ou sans la simulation de la volatilité.

Notre méthode incrémentale est composée par les étapes suivantes :

1. test d'applications à petite échelle sans volatilité ;
2. test d'applications à petite échelle avec la volatilité ;
3. test d'applications à grande échelle sans volatilité ;
4. test d'applications à grande échelle avec la volatilité.

L'étape 1 se compose de test de conformité, avec un minimum de configuration. L'objectif est de fournir une séquence de test (TS) suffisamment efficace pour parvenir à un test de critères pré-définis. Cette séquence de test TS doit être paramétrée par le nombre de pairs, de sorte qu'ils puissent être adapté pour les tests à grande échelle. Une séquence de test est désignée par $ts(ps)$, où ps désigne un ensemble de pairs. Les séquences de test peuvent également être combinées pour construire un scénario complexe de test en utilisant un langage de tests, tel que Tela [84].
L'étape 2 se compose de la réutilisation de la séquence de test initiale et en y ajoutant la volatilité (TSV).
L'étape 3 réutilise la séquence de test de l'étape 1 et la combine pour gérer un grand nombre de pairs. Nous avons donc obtenu un scénario de test (GTS). Un scénario de test peut être composé de plusieurs séquences de test.
L'étape 4 réutilise les scénarios de test de l'étape 3 avec les séquences de test de l'étape 2, et un scénario de test avec la volatilité ($GTSV$) est construit et exécuté.

L'avantage de ce processus est de se concentrer sur la génération des séquences de test pertinentes, d'un point de vue fonctionnel, et puis la réutilisation de ces séquences en y ajoutant progressivement la volatilité et le passage à l'échelle.

En termes de diagnostic, cette méthode permet de déterminer la nature des problèmes détectés. En effet, les problèmes peuvent être liés à une cause purement fonctionnelle (Etape 1), à une question de volatilité (étape 2), au passage à l'échelle (étape 3) ou une combinaison de ces trois aspects (étape 4). Les erreurs les plus complexes sont les derniers depuis leur analyse est liée à une combinaison de ces trois aspects. Les étapes 2 et 4 pourraient aussi être précédées par deux autres étapes (la réduction et l'expansion du système), afin d'aider le diagnostic d'erreurs, soit en raison de l'absence de ressources ou de l'arrivée de nouveaux ressources. Plusieurs taux de volatilité peuvent être explorées afin de vérifier la manière dont elles affectent l'aspect fonctionnel du SUT (par exemple, 10 % des pairs rejoignent, 20 % des pairs quittent).

Cadre de test P2P

Dans cette section, nous présentons le cadre de test P2P y compris trois différentes architectures de coordination de test.

- La première architecture repose sur un coordinateur de test centralisé, où un contrôleur central est chargé de synchroniser l'exécution des actions de test distribué à travers des testeurs pour assurer la bonne exécution de leur séquence [35, 34, 33]. En fait, cette architecture étend l'architecture classique de test des systèmes distribués avec la capacité de gérer la volatilité.
- La deuxième architecture évite le coordinateur centralisée organisant les testeurs dans un B-arbre. La synchronisation est faite par le biais de messages échangés entre les parents et les enfants [36].
- La troisième architecture évite également le coordinateur centralisé, mais par le biais de propagation de rumeurs échangés entre testeurs [36].

Toutes ces architectures ont été mises en œuvre en Java (version 1.5) et font appel à deux fonctionnalités de ce langage : la réflexion dynamique et les annotations. Comme nous le verrons par la suite, ces fonctionnalités sont utilisées pour sélectionner et exécuter les actions qui composent un scénario de test.

Testeurs

La suite de tests est mise en œuvre en tant que classe, qui est la classe principale de l'application de test. Une suite de test contient plusieurs cas de test, qui sont mis en œuvre comme un ensemble d'actions. Un cas de test est composé par des actions mises en œuvre en tant que des méthodes annotées, c'est-à-dire, les méthodes décorées par un méta-tag, ou *(*annotation), qui informe que la méthode est une action de test. Les annotations peuvent être attachées aux méthodes ou à d'autres éléments (des paquets, les types, etc), en donnant des informations supplémentaires concernant un élément : la classe est obsolète, une méthode est redéfinie, etc. En outre, des nouvelles annotations peuvent être spécifiées par les développeurs si nécessaire. Les annotations sont utilisées pour décrire le comportement des actions de test : où il doit exécuter, dans lequel testeur, si oui ou non la durée doit être mesuré. Les annotations sont semblables à celles utilisées par JUnit[1], bien que leur sémantique ne soit pas exactement la même.

Le choix d'utiliser les annotations pour la synchronisation et l'exécution conditionnelle a été motivée par deux raisons principales. Tout d'abord, de séparer le contrôle d'exécution du code de test. Deuxièmement, pour simplifier le déploiement des cas de tests : tous les testeurs reçoivent le même cas de test. Toutefois, les testeurs exécutent seulement les actions qui leur sont assignées. Les annotations sont ci-dessous :

Test. Spécifie que la méthode est en fait un action de cas de test. Cette annotation a quatre attributs qui sont utilisés pour le contrôle de son exécution : le nom du cas de test, le lieu où il doit être exécuté, son ordre à l'intérieur du cas de test et le délai d'exécution.

Before. Spécifie que la méthode est exécutée avant chaque test. Son but est de mettre en place un cadre commun pour tous les cas de test. La méthode joue le rôle d'un préambule.

After. Spécifie que la méthode est exécuté après chaque test. Son but est de mettre en place l'environnement de test dans le même état qu'avant l'exécution du test.

Chaque action est un point de synchronisation : à un moment donné, seulement les méthodes avec la même signature peuvent être exécutées sur les différents testeurs, c'est-à-dire, les actions ne sont pas toujours exécuté sur tous les testeurs. Au fait, les annotations sont utilisés pour restreindre les testeurs où

[1] http ://www.junit.org

une action peut être exécutée. Ainsi, les testeurs partagent le même code de test mais n'ont pas le même comportement. Le but est de séparer le code de test à partir d'autres aspects, tels que synchronisation ou exécution conditionnelle. Le testeur offre deux interfaces, pour l'exécution des mesures de contrôle et de la volatilité :

1. **execute(**a_n**)** : exécute une action donnée.
2. **leave(**P**), fail(**P**), join(**P**)** : fait une série de pairs quitter le système, quitter anormalement ou rejoindre le système.

Quand un testeur exécute un test, il procède comme décrit ci-dessous :

1. Il prie le coordinateur pour l'identification qui sera utilisée pour filtrer les actions qu'il doit exécuter.
2. Il emploie réflexion java pour découvrir toutes les actions, lire les annotations et créer un ensemble de descriptions des méthodes.
3. Il passe au coordinateur l'ensemble des descriptions juste avec les méthodes qu'il convient d'exécuter y compris leurs priorités d'exécution.
4. Il attend pour le coordinateur d'invoquer d'une de ses méthodes. Après l'exécution, il en informe au coordinateur que cette méthode a été correctement exécutée.

Architecture centralisée

Dans cette architecture, un composant centralisé (coordinateur) contrôle et synchronise une séquence de test.

En effet, le coordinateur contrôle plusieurs testeurs et chaque testeur fonctionne sur un nœud logique différent (le même que celui du pair qu'il contrôle). Le rôle d'un testeur est d'exécuter les actions de test et de contrôler la volatilité d'un seul pair. Le rôle du coordinateur est d'envoyer les actions d'un cas de test (A^τ) aux testeurs (T^τ) et de maintenir une liste des pairs indisponibles. Dans la pratique, chaque testeur reçoit la description de la séquence de test et est donc en mesure de savoir à quel moment exécuter cette séquence.

The Coordinator

Le rôle du coordinateur est de contrôler quand chaque action de test doit être exécutée. Quand le test commence, le coordinateur reçoit un ensemble des descriptions des méthodes de chaque testeur, en associant à chaque action un niveau hiérarchique. Ensuite, il utilise un compteur, qui représente le niveau hiérarchique qui peut être exécuté, pour synchroniser l'exécution. Du point de vue du coordinateur, un cas de test est composé d'un ensemble d'actions A, où chaque action a_n^A a un niveau hiérarchique $h_{a_n}^A$. Actions avec les niveaux inférieurs sont exécutés avant que les actions à des niveaux plus élevés.

L'exécution des actions suit l'idée de la validation à deux phases (2pc). Dans la première phase, le coordinateur informe tous les intéressés qui l'action a_n peut être exécutée et produit un verrou. Une fois que tous les pairs annoncent la fin de leur exécution, le verrou est libéré et l'exécution de la prochaine action commence. Si un délai d'action est atteint, le cas de test est terminée.

Le coordinateur propose trois interfaces, pour l'exécution des actions, le contrôle de la volatilité et l'accès aux variables de test :

1. **register(**t_i, A^t**), ok(**a_n**), fail(**a_n**), error(**a_n**)** : action de l'enregistrement (effectué tous les tests avant) et les réponses d'exécution d'une action, appelée par les testeurs, une fois l'exécution d'une action est terminée.

2. **set(key,value), get(key)** : accesseurs pour les variables de test.
3. **leave(P), fail(P), join(P)** : fait une série de pairs quitter le système, anormalement quitter ou rejoindre le système.

Architectures distribuées

Dans cette section, nous présentons deux alternatives à l'architecture de test centralisée.

B-arbre

La première architecture présentée ici se compose des testeurs en organisant dans un structure B-arbre [17], comme le réseau P2P par GFS-Btree [68]. L'idée est d'abandonner le coordinateur centralisé de test et d'utiliser la racine de l'arbre pour commencer l'exécution des cas de test et de calculer un verdict. Lors de l'exécution d'un test, la racine envoie les actions à ses enfants testeurs, qui envoient ces actions à leurs enfants. Une fois qu'une action est exécutée, les feuilles envoient leurs résultats à leurs parents, qui font le même jusqu'à ce que la racine reçoive tous les résultats. Ensuite, la racine peut envoyer les prochaines actions.

L'ordre du B-arbre n'est pas fixe, elle peut varier selon le nombre de testeurs, qui est connu au début de l'exécution. Le but est d'avoir un arbre bien proportionné, où la profondeur est équivalent à son ordre.

Propagation de rumeurs

La propagation de rumeurs est une autre solution pour synchroniser l'exécution des actions d'une manière distribuée. Dans la propagation de rumeurs, nous utilisons la même architecture utilisée par l'architecture en arbre avec un testeur par nœud, toutefois, la synchronisation des actions est exécutée par des changement des messages de coordination (des rumeurs) entre les testeurs.

L'exécution des tests se fait en plusieurs étapes. Tout d'abord, un nœud p du système P est désigné comme premier testeur t_0. Ce testeur va fournir des identifiants à tous les autres testeurs t_n qui rejoignent le système. L'identification est calculée en suivant une séquence de 1 à n et est utilisée pour sélectionner quelles actions un nœud doit exécuter. Deuxièmement, t_0 crée une adresse de multicast pour chaque action d'un cas de test. Troisièmement, les testeurs emploient réflexion java pour découvrir toutes les actions. Ensuite, chaque testeur vérifie les actions qu'il doit exécuter et souscrit à l'adresse de multicast appropriée. Enfin, les testeurs responsables pour exécuter la première action commencent l'exécution.

Un testeur peut jouer deux rôles différents au cours d'un test :

- *Testeur actif*. Ce testeur effectue une action a_i en propageant son achèvement à l'adresse de multicast de la prochaine action a_{i+1}. Une fois que le message a été envoyé, ce testeur devient un *Testeur inactif*.
- *Testeur inactif*. Ce testeur reste inactif en attendent les messages de tous les testeurs actifs. Une fois qu'il reçoit l'ensemble de messages, il devient un *Testeur actif*.

La propagation de rumeurs entre ces deux types de testeurs garantit la séquence d'exécution de la séquence de test.

Nous utilisons l'exemple 3.1 pour illustrer cette approche. Au départ, un nœud est choisi pour être testeur t_0. Ensuite, les autres nœuds contactent avec t_0 pour recevoir un identifiant n et abonner à un adresse de multicast appropriée. Par exemple, si un testeur reçoit $n = 1$, il souscrit aux adresses de a_1, a_3 et a_5. Ensuite à l'identification, la première action a_1 est exécutée par les testeurs $\{t_0, t_1, t_2\}$. Une fois l'exécution de a_1 est terminée, les testeurs envoient ces messages à l'adresse multicast de la prochaine

action a_2. Une fois testeur t_2 reçoit trois messages, il exécute a_2 propageant son achèvement à l'adresse de multicast de la prochaine action. Cela se produit consécutivement jusqu'à la dernière action a_7. Enfin, chaque testeur calcule un verdict local et l'envoie à t_0, qui attribue un verdict global de l'ensemble des cas de test.

Conclusion

Dans ce chapitre, nous présentons les conclusions générales de cette thèse. Tout d'abord, nous abordons les difficultés du test des systèmes P2P. Ensuite, nous résumons nos principales contributions. Enfin, nous discutons des futures orientations de recherche dans le test des systèmes P2P.

Difficultés du test des systèmes P2P

Dans le cadre des systèmes distribués, le pair-à-pair (P2P) apparaît comme un puissant paradigme pour développer des systèmes distribués qui passent à l'échelle. L'absence de serveur central pour gérer les nœuds distribués fait qu'il n'existe ni point individuel de défaillance, ni goulet d'étranglement des performances dans le système.

Dans un système P2P, chaque pair joue le rôle d'un nœud actif avec la possibilité de rejoindre ou de quitter le réseau à tout moment, soit normalement (par exemple, par déconnexion), soit anormalement (par exemple, par échec). Cette capacité, appelée volatilité, est une différence majeure d'avec les systèmes distribués classiques. En effet, la volatilité permet de modifier dynamiquement la taille et la topologie du réseau, ce qui rend ainsi les tests des systèmes P2P tout à fait uniques. Par conséquent, le comportement fonctionnel d'un système P2P (ainsi que les défauts fonctionnels) dépend fortement du nombre de pairs et de leur volatilité.

En résumé, un cadre de test P2P devrait prévoir la possibilité de contrôler trois aspects :

- La fonctionnalité, exprimée par le comportement du système ;
- Le passage à l'échelle, exprimé par le nombre de pairs dans le système ;
- La volatilité, exprimée par le nombre de pairs qui se joignent ou quittent le système après leur initialisation au cours du test.

Enfin, ces aspects font ressortir la nécessité d'une méthodologie précise de test des systèmes P2P, où la simplicité des interfaces contraste avec la complexité des facteurs qui peuvent affecter un test. Ils font également ressortir la nécessité d'une architecture de test robuste et efficace : robuste pour comprendre la volatilité comme un comportement normal et efficace pour subir des tests à grande échelle.

Contributions

Dans cette thèse, nous avons proposé une solution intégrée pour la création et le déploiement d'un environnement de validation des systèmes P2P, avec la possibilité de créer des pairs et de les faire rejoindre et quitter le système. Les objectifs de test peuvent combiner le contrôle du fonctionnement du système avec des variations de la volatilité et aussi avec le passage à l'échelle. L'exactitude du système peut donc être validée sur la base de ces trois dimensions, c'est-à-dire les fonctions, le nombre de pairs et la volatilité des pairs.

Méthodologie incrémentale

Dans cette thèse, nous avons recommandé de contrôler la volatilité dans la définition d'une séquence de test. Ainsi, chaque pair, du point de vue du test, peut être amené à quitter ou à rejoindre le système à un moment donné dans une séquence de test.

Devant être en mesure de gérer un grand nombre de pairs, la deuxième dimension d'un test de système P2P est le passage à l'échelle. Ces deux dimensions étant une partie intégrante des systèmes P2P, elles doivent être combinées avec le test des fonctionnalités.

Par conséquent, nous avons proposé une méthodologie de test incrémentale où nous testons les fonctionnalités du système en augmentant progressivement son échelle avec ou sans la simulation de la volatilité.

Nous avons validé notre méthodologie incrémentale de test en montrant comment les trois aspects dimensionnels de test augmentent la couverture de code, et par conséquent, la qualité des tests.

Les architectures de coordination de test

Nous avons proposé également trois différentes architectures de coordination de test [35, 36, 34, 33]. Ces architectures sont basées sur deux aspects originaux : (i) le contrôle individuel de la volatilité des pairs et (ii) la distribution de l'architecture de test pour faire face à un grand nombre de pairs. Les avantages de ces architectures sont (1) d'automatiser l'exécution des tests localement à chaque pair, (2) de construire automatiquement un verdict global, (3) de permettre un contrôle explicite de la volatilité de chaque pair.

Nous avons mis en œuvre ces architectures et nous les avons utilisées pour tester deux systèmes P2P open-source. Durant ces expérimentations, nous avons analysé le comportement des ces deux systèmes avec des conditions différentes de volatilité et nous avons montré que notre contribution est capable de détecter des problèmes de mise en œuvre.

Orientations futures de recherche

Notre prochain défi est de proposer une solution capable de générer des scénarios de test en assurant une meilleure couverture des fonctionnalités par la variation de la volatilité et du nombre de noeuds (passage à l'échelle). Cette combinaison de paramètres pourrait encore améliorer la couverture du code durant les tests.

Nous avons également l'intention de tester les systèmes P2P dans des situations plus extrêmes telles que l'exécution d'insertion et/ou de récupération de très grands volumes de données. Ces tests pourraient augmenter de manière significative la confiance sur les systèmes sous test. En outre, ces différentes situations pourraient être utiles pour réaliser des tests de stress, dont l'objectif est de vérifier comment se comporte un système aux conditions extrêmes mais valables : un grand nombre d'utilisateurs simultanés, peu de mémoire, etc.

Dans un autre contexte, les systèmes P2P génèrent une quantité importante de fichiers de *log* tout au long de leur exécution. Ainsi, l'analyse de ces fichiers pourrait être un thème intéressant de recherche, tout comme la mise en œuvre d'un oracle capable d'analyser ces fichiers et qui serait complémentaire à l'approche des assertions que nous avons présentée. Une solution possible serait de charger ces fichiers de *log* dans une base de données et de tirer parti des capacités du SGBD à construire des rapports complexes d'analyse (par exemple, des rapports d'anomalie) et de gérer de grandes quantités de données.

Enfin, la traçabilité [32, 97] pourrait être utilisée en combinaison avec l'analyse du *log*. L'idée est

d’identifier les traces des scénarios de faute pour guider le développeur à trouver l’origine d’une défaillance du système.

CHAPTER 1
Introduction

1.1 Motivation

Peer-to-peer (P2P) systems connect a set of distributed nodes (i.e., peers) across a network without any centralized control or any hierarchical system. Every new peer shares with the system its resources (e.g., CPU, storage, memory, etc) and/or its data (e.g., music files, web pages, etc). In the context of resource sharing, a P2P system yields high scalability since the amount of resources scales up with the number of peers. A typical P2P system can deal with heavy loads, for instance large data sets. In the context of data sharing, a P2P system yields availability and redundancy of data since a request for the same data can be treated by multiple peers. Furthermore, if some peers share the same data, they can build together a social network (i.e., a group of peers tied by one or more types of interdependency).

Indeed, P2P offers good solutions for many applications such as large data sharing and collaboration in social networks. Thus, it appears as a powerful paradigm to develop scalable distributed applications, as reflected by the increasing number of emerging projects based on this technology [12].

Among the many aspects of the P2P development, making systems reliable is an obvious target. This is even more critical when P2P systems are to be widely used. So, as for any system, a P2P system has to be tested with respect to its requirements. As for any distributed system, the communication and message exchanges complexity must be a part of the testing objectives. Testing of distributed systems consists of a centralized test architecture composed of a test controller, or coordinator, which synchronizes and coordinates the communications (message calls, detection of deadlocks) and creates the overall verdicts from the local ones [37, 59]. Locally to each node, test sequences or test automata can be executed which run these partial tests on demand and send their local verdicts to the coordinator. One local tester per node or group of nodes is generated from the testing objectives. The tester specifies the system using Finite State Machines [25, 54, 24], Labeled Transition Systems [58, 85, 59] and uses this specification to generate a test suite that is able to verify (totally or partially) whether each specified transition is correctly implemented. The tester then observes the events sent among the different nodes of the system and verifies that the sequence of events corresponds to the state machine (or to the transition system).

In a P2P system, a peer plays the role of an active node with the ability to join or leave the network at any time, either normally (e.g., disconnection) or abnormally (e.g., failure). This ability, which we call volatility, is a major difference with distributed systems. Furthermore, volatility yields the possibility of dynamically modifying the network size and topology, which makes P2P testing quite different. Thus, the functional behavior of a P2P system (and functional flaws) strongly depends on the number of peers, which impacts the scalability of the system, and their volatility.

A Distributed Hash Table (DHT) [88, 92, 86] is an example of a P2P system, where each peer is responsible for the storage of values corresponding to a range of keys. It has a simple local interface that only provides three operations: value insertion, value retrieval and key lookup. The remote interface is more complex, providing operations for data transfer and maintenance of the routing table, i.e., the correspondence table between keys and peers, used to determine which peer is responsible for a given key. Testing these interfaces in a stable system is rather simple. However, it is hard testing that the

routing table is correctly updated and requests are correctly routed while peers leave and join the system. Furthermore, such testing requires a mechanism to simulate volatility and ensures that this simulation does not interfere with the test, i.e., that unanswered requests are not interpreted as faults.

Some approaches for P2P testing propose to randomly stop the execution of peers [13, 69], or to insert faults in the network [55, 76]. While these approaches are useful to observe the behavior of the whole system, they are not totally adapted for testing a P2P system. Since they focus on tolerance to network perturbations, they fail in detecting software faults, specially those which occurs due to peers' volatility. For instance, if one wants to test if a peer is able to rebuild its routing table when all peers it knows leave the system, then one needs to specify precisely the peers to drop since the random volatility is not precise enough. Increasing significantly the rate of volatile peers is not desirable either since the high number of messages necessary to rebuild the routing tables from the remaining peers may interfere with the test.

A P2P system must also scale up to large numbers of peers. It means the testing architecture must scale up either. Indeed, a typical P2P system may have a high number of peers which makes the centralized testing architecture (i.e., central coordinator managing distributed testers) not scalable [44, 64, 33, 71]. It is also possible to use distributed testers where test cases are coordinated through messages exchanged at points of communication (PC) [101, 98]. Even with such PCs, the testing architecture perturbs the performance of P2P systems. Indeed, the algorithmic performance with such testing architecture scales up linearly with the number of peers, while a typical P2P system scales up logarithmically. To avoid testers intrusiveness to threaten the validity of test results, the testing architecture should thus have logarithmic performance in the worse case.

1.2 Contributions

In this thesis, we propose an integrated solution for the creation and deployment of a P2P test environment with the ability to create peers and make them join and leave the system. Then, the test objectives can combine the functional testing of the system with the volatility variations (and also scalability). The correctness of the system can thus be checked based on these three dimensions, i.e. functions, number of peers and peers volatility.

In detail, we propose an incremental methodology to deal with these dimensions, which aims at covering functions first on a small system and then incrementally addressing the scalability and volatility issues [33].

As stated in the previous section, P2P testing tackles the classical issue of testing a distributed system, but with a specific dimension we call volatility, which has to be an explicit parameter of the test objectives. Two possible solutions may be used to obtain a test sequence which includes volatility. It can either be simulated with a simulation profile or be explicitly and deterministically decided in the test sequence. The first solution is the easiest to implement, by assigning a given probability for each peer to leave or join the system at each step of the test sequence execution [39, 91, 69]. The problem with this approach is that it makes the interpretation of the results difficult, since we cannot guess why the test sequence failed. Moreover, it creates a bias with the possible late responses of some peers during the execution of the test sequence. As a result, it cannot be used to combine a semantically rich behavioral test with the volatility parameter. In this thesis, we recommend to fully control volatility in the definition of the test sequence. A peer thus, from a testing point of view, can have to leave or join the system at a given time in a test sequence. This action is specified in the test sequence in a deterministic way.

Since it has the objective to deal with a large number of peers, the second dimension of P2P sys-

tem testing is scalability. Then, because it is accomplishing a treatment, the scalability and volatility dimensions have to be tested with the behavioral and functional correctness. In summary, a P2P testing framework should provide the possibility to control:

- functionality captured by the test sequence which enables a given behavior to be exercised,
- scalability captured by the number of peers in the system,
- volatility captured by the number of peers which leave or join the system after its initialization during the test sequence.

When performing scalability testing for a distributed system, the functional aspects are barely taken into account. The same basic test scenario is simply repeated on a large number of nodes [45]. The same approach may be used, but would also lead to test volatility separately from the functional aspect. For a P2P system, we claim that the functional flaws are strongly related to the scalability and volatility issues. As a consequence, it is crucial to combine the scalability and volatility aspects with meaningful test sequences.

We also propose three different architectures to coordinate the test sequence. These architectures are based on two original aspects: (i) the individual control of peers' volatility and (ii) a distributed testing architecture to cope with large numbers of peers. The capabilities of these architectures are (1) to automate the execution of each local-to-a-peer test case, (2) to build automatically the global verdict, (3) to allow the explicit control of each peer volatility.

The first architecture extends the classical centralized testing architecture (i.e., central coordinator managing distributed testers) with volatility control to demonstrate that such volatility is a key-parameter when testing a P2P system [34]. Basically, this architecture has two main components: the tester and the coordinator. The tester is composed of the test suites that are deployed on several logical nodes. The coordinator is deployed in only one node and is used to synchronize the execution of test cases. It acts as a *broker* [21] for the deployed testers.

However, the performance of such centralized architecture is linear while testing with large numbers of peers requires logarithmic. Therefore, we propose two fully distributed architectures to cope with large-scale P2P systems [36]. The second architecture organizes the testers in a balanced tree [17] (B-Tree) manner where the synchronization is performed from the root to the leaves. Then, the testers communicate with each other across the B-tree to avoid using a centralized coordination. The third architecture uses gossiping messages among testers reducing communications among the testers responsible to execute consecutive test case actions. Since both distributed architectures do not rely on a central coordinator they scale up correctly.

These two distributed architectures only have one component, the tester. The tester is the application that executes in the same logical node as peers, and controls their execution and their volatility, making them leave and join the system at any time, according to the needs of a test. Thus, the volatility of peers can be controlled at a very precise level. These architectures do not address the issue of test cases generation but is a first element towards an automated P2P testing process. It can be considered analogous to the JUnit[1] testing framework for Java unit tests.

We validated them through implementation and experimentation on two open-source P2P systems. Through experimentation, we analyze the behavior of both systems on different conditions of volatility and show how they are able to detect implementation problems. We also validated our incremental testing methodology showing how the three dimensional aspects increased code coverage.

[1] http ://junit.org/

1.3 Outline

The rest of the thesis is organized as follows. In chapter 2, we introduce the P2P systems to establish the context of our work. Then, we survey the testing architectures comparing their perspectives from the context of distributed testing and we discuss why they are not suited to test P2P systems. Finally, we survey the test oracle approaches and discuss their advantages and drawbacks.

In chapter 3, we present our testing framework in two parts. First, we present an incremental methodology to deal with the dimensional key-aspects of P2P system testing. Second, we present three testing architectures to coordinate the execution of test cases across peers. In chapter 4, we present the PeerUnit tool which implements our testing framework.

In chapter 5, we describe the experimental validation of our contributions with three objectives: (i) validate the usability and efficiency of our testing framework based on experiments that verify two popular open-source DHTs, (ii) validate the feasibility of our incremental testing methodology and comparing it with classical coverage criteria and (iii) evaluate the coordination overhead of our testing architectures.

In chapter 6, we conclude this thesis and discuss future directions of research.

CHAPTER 2
State of the art

2.1 Introduction

In this chapter, we present initially an overview of the peer-to-peer (P2P) systems to establish the context of our work. Then, we survey the testing architectures comparing their perspectives from the context of distributed testing. Moreover, we discuss why they are not suited to test P2P systems. First, we show that they fail when dealing with the volatility, both interrupting the testing sequence and deadlocking the tester, then assigning false-negative verdicts to test cases (i.e., false fail verdict). Second, we show that most of them do not scale up due to centralized test coordination. Third, some of these architectures demand to include additional code into the system under test (SUT) source code. While this is a straightforward approach to control testing executions, it is error prone. In fact, we believe that an ideal testing architecture for P2P systems must consider the P2P features and also control the execution of tests without contaminating the SUT with additional code. Furthermore, a correctness mechanism must be provided to assign verdicts to tests. Finally, we survey two testing oracle approaches and their distinct ways to check correctness.

This chapter is organized as follows. In section 2.2, we present a background of P2P systems. In section 2.3, we describe several testing architectures and discuss why they are not able to test P2P systems. In section 2.4, we describe some testing oracles and discuss their advantages and drawbacks. In section 2.5, we conclude this chapter.

2.1.1 Software testing

Software testing is the process of finding evidences of errors in software systems. Tests can be implemented and executed in different ways along testing (e.g., Manual testing, Automated testing). Manual testing is still valuable, but testing is mainly about an automated system to implement, apply and evaluate tests [20]. To do so automatically, a test must be designed to work with interfaces and the runtime environment of the SUT.

Tests are designed to analyze and decide whether a SUT contains errors. To accomplish this task the test engineer must define a *test scenario*. A test scenario is a set of *test cases* that covers all possible ways of testing a set of conditions.

The aim of a test case is to validate one or more requirements from a system: it exercises the SUT and checks whether an erroneous behavior occurs. We roughly define a test case as being composed of a name, an intent, a sequence of actions, a sequence of input data and the expected outputs.

Along testing, an input helps to exercise the functional aspects of a system and aims to generate a result. To define an input, a test engineer must be based on some *test criteria*. Various test criteria can be used, such as control/based-flow based coverage criteria, specification-based coverage criteria, etc. Moreover, this input can be received from an external source, such as hardware, software or human [99].

By the end of a test execution, the exercised system generates some results in form of an output value (e.g., changes to data, reports, communication messages sent out, and output to screens). First, the actual result that is the result of the test case itself. Second, the expected result that is what the test engineer

expects to have by the end of a test. Then, these two results are used to assign a validation label, or a verdict to the test case. A verdict is normally assigned among three values: *pass*, *fail* and *inconclusive*.

The *oracle* is the mechanism responsible for assigning *verdicts*. This mechanism compares the actual result with the expected one and assign a verdict to the test case. If the values are the same, the verdict is pass. Otherwise, the verdict is fail. The verdict may also be inconclusive, meaning that the test case output is not precise enough to satisfy the test intent and the test must be done again. There are different sorts of oracles: assertions, value comparison, log file analysis, manual, etc. An entire testing technique thus includes test criteria, test cases generation techniques and mechanisms for obtaining the verdicts.

Figure2.1 illustrates the whole software testing process. Initially, some test data is used as input to be executed by the SUT. Then, such execution generates some results also called as actual output. This actual output is compared with an expected output with respect to some specification. Such comparison is made by an oracle and results in a verdict. If the verdict is false, the error is localized, corrected and the process starts over. The process stops when the verdict is true and a certain stop criterion is reached.

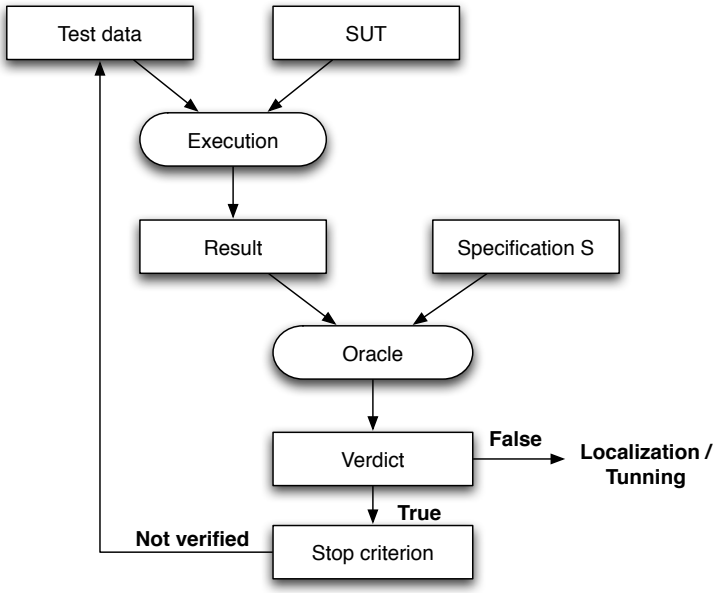


Figure 2.1 – Software Testing Process

2.1.2 Testing peer-to-peer systems

In this thesis, our goal is to build an automated system for the creation and deployment of a testing environment for P2P systems. Compared to classical distributed systems, P2P systems are completely decentralized. A peer behaves as a client computer in the client-server model, but also provides server functions [16]. In a P2P system, peers interact with each other directly in a decentralized way without the need of a central server. Due to the decentralization, peers are functionally identical, but with the particular capacity to join or leave the system at any time. This simple change adds the possibility to modify dynamically the network size and topology resulting in high scalability.

This decentralization also results in high fault-tolerance since there are no single points of failure.

Without a central server, any peer in the system may assume the responsibility of a failed one. Moreover, decentralization means that nobody/nothing can control the peers spread across the network. Thus, any peer can decide to share or not its resources.

P2P systems are distributed systems, and should be firstly tested using appropriate tools dedicated to distributed system testing. Distributed systems are commonly tested using conformance testing [90, 101, 89, 76, 75, 59]. The purpose of conformance testing is to determine to what extent the implementation of a system conforms to its specification. The tester specifies the system using Finite State Machines [25, 54, 24] or Labeled Transition Systems [58, 85, 59] and uses this specification to generate a test suite that is able to verify (totally or partially) whether each specified transition is correctly implemented. The tester then observes the events sent among the different nodes of the system and verifies that the sequence of events corresponds to the state machine (or to the transition system). In general, a tester, also called test driver[20, 99], is a component that applies test cases to a SUT. The classical architecture for testing a distributed system consists of a centralized tester which sends the test inputs, controls the synchronization of the distributed system and receives the outputs (or local verdicts) of each node of the SUT. In many cases, the distributed SUT is perceived as a single application and it is tested using its external functionality, without considering its components (i.e., black-box testing). The tester in that case must interpret results, which include non-determinism since several input/outputs orderings can be considered as correct.

The observation of the outputs for a distributed system can also be achieved using the traces (i.e., logs) produced by each node. The integration of the traces of all nodes is used to generate an event time line for the entire system. Most of these techniques do not deal with large scale systems, in the sense they target a small number of communicating nodes. In the case of P2P systems, the tester must observe the remote interface of peers to observe their behavior and also deal with a potentially large number of peers. Writing test cases is then particularly difficult, because non-trivial test cases must execute actions on different peers. Consequently, synchronization among actions is necessary to control the execution sequence of the whole test case.

Analyzing the specific features of P2P system, we remark that they are distributed systems, but the existing testing techniques for distributed systems do not address the issue of synchronization when a large number of nodes are involved. Thus, the typical centralized tester architecture can be reused for building a testing framework for P2P systems. However, Ulrich et al. [98] describe that such centralized architecture may not scale up and becomes a bottleneck. Another dimension is not addressed by distributed system testing, which is the problem of nodes volatility of P2P systems. P2P systems must be robust and work even if peers are volatile (under limits which have to be determined). Volatility thus interferes with the system functionality and may cause failures to occur. This aspect of dealing with volatility is usually addressed using stress and load testing techniques.

2.2 P2P Systems

Peer-to-peer (P2P) systems are distributed systems designed for resource sharing (content, storage, CPU cycles) by direct exchange, rather than requiring the support of a centralized server or authority. These systems are characterized by their ability to adapt to failures and accommodate transient populations of nodes while maintaining acceptable connectivity and performance [12].

We can classify the main characteristics of P2P systems as follows :

- Peers are decentralized. There are no central server, and peers are distributed over a wide-area network.

- Peers are volatile. A peer may join and leave the system at any time. Deshpande et al. [39] describes that understanding volatility can be a key factor in designing effective and efficient P2P architectures.
- Peers are autonomous. A peer decides to share its contents and/or resources with other peers.

These characteristics are directly impacted by different topologies and structures that a P2P system may have. These topologies and structures form a P2P network also called overlay network. In fact, a P2P system relies on a P2P network in order to operate. We organized the rest of the section based on the three main structures of P2P networks: unstructured, structured and super-peer.

2.2.1 Unstructured

In unstructured networks, peers organize themselves in a non-deterministic manner without any particular structure (e.g., ring, binary tree, multidimensional space) [77, 7, 80]. Gnutella, FreeHaven [40], and Kazaa [65] are examples of unstructured P2P networks.

Whenever a new peer connects to the network, it contacts other peers randomly. These peers become neighbors and start to exchange *Ping* messages to announce the connection. These messages are propagated recursively through the entire neighborhood, flooding the network. Query messages are also sent by flooding. However, such approach is not scalable. In a very large system, if all peers query by flooding, than the network becomes overloaded reducing the performance of the P2P system as well.

To reduce the flooding, some new mechanisms were presented. The Gnutella search mechanism is based on constrained flooding, where messages are passed between neighbors with a time-to-live (TTL). For instance, in a query, a peer forwards messages recursively to all of its neighbors until locate the desired data. These messages decrement their TTL field at each hop until such TTL reaches zero to be dropped.

Despite the flooding have been constrained by a TTL, the traffic load continued to be expensive [91] and the P2P systems continue to have difficulty to scale up. Thus, more sophisticated techniques were presented to reduce such traffic. For instance, Kalogeraki et al. [63] propose to forward messages to selected neighbors based on the past behavior of the network. Lv et al. [72] propose to forward messages to random neighbors, that also forward them randomly: periodically, these messages contact the query originator to ask whether the termination condition is held or not (e.g., TTL). For other techniques refer to [23, 29, 107].

A drawback of the unstructured networks is the absence of guarantee to query completeness [48]. This happens because some peers containing relevant data may be too far from the query originator and the query messages are dropped by TTL before they reach data.

2.2.2 Structured

Structured networks appear to solve the scalability issues of the unstructured networks. In structured networks peers may organize themselves in different types of structures such as ring [92], binary trees [6], multidimensional space [86], or others. The structures are maintained by a routing mechanism used to organize the system, and to send and forward messages.

A Distributed Hash Table (DHT) is an example of a structured network. In a DHT, each peer is responsible for the storage of values corresponding to a range of keys. A DHT has a simple local interface that provides three operations: value insertion, value retrieval and key look up. The remote interface is more complex, providing operations for data transfer and maintenance of the routing table, i.e., the

correspondence table between keys and peers, used to determine which peer is responsible for a given key.

Figure 2.2 illustrates a Chord DHT where a peer is responsible to keep a set of pairs "(key, value)" mapped to its ID. Once a peer joins the system, it acknowledges its neighborhood and acquires the pairs with the keys it is mapped for. When such peer decides to leave the system, it passes its pairs to its neighbors. Each peer also keeps a routing table with the IDs of its neighbors used to route messages. For instance, peer $P10$ decides to look up for key 54. It sends a message directly to its last neighbor $P29$ since it has the closest ID to the key. Then, the look up is routed in the same way until find the object keeper, which is peer $P52$.

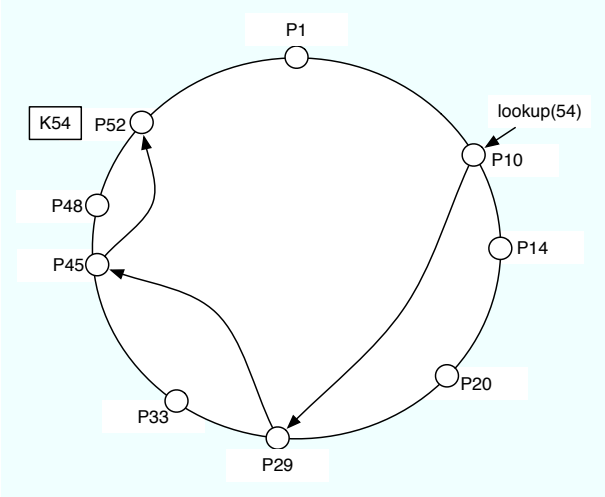


Figure 2.2 – Chord ring with three on-line nodes

The routing mechanism provides two advantages compared to unstructured networks. First, it avoids message flooding resulting in high scalability due to low network traffic. Yet, the routing ensures an upper bound of hops necessary to look up for a key. A typical upper bound is $O(log(N))$ where N is the number of peers in the network. Second, it efficiently routes query messages to a responsible peer up to the upper bound limit resulting in high quality of service (e.g., completeness, data availability, query response time, etc).

A drawback of the structured networks is the limited autonomy of peers since each peer is only responsible to keep the keys closer to its ID. Examples of P2P systems based on structured networks include FreeNet [1, 27], Chord [92, 31], CAN [86], Pastry [88], PAST [43], Tapestry [109], Kademlia [78], PIER [56], and P-Grid [6].

2.2.3 Super-peer

Super-peer network is a hybrid of unstructured networks and client/server systems. In super-peer networks some functionalities are still centralized in special peers classified as super peers due to their high amount of resources. Figure 2.3 illustrates a typical architecture, where a super-peer keeps an index of meta-data for all files in the network. Once a peer p queries the super-peer for a given file, it receives a list of peers that hold such file. Then, p opens direct connections with one or more of those peers in order to download the file.

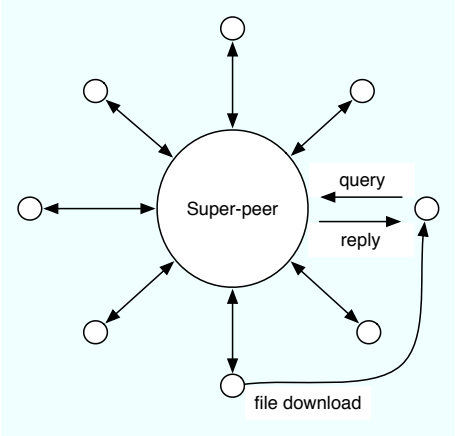


Figure 2.3 – Super-peer architecture

Yang and Garcia-Molina [106] argue that super-peer networks have better object location than pure P2P networks because the indices of files remains centralized. Furthermore, super-peer networks are relative simple to implement and the quality of the service can be perceived by users (i.e., completeness of query results, response time).

However, super-peer networks have an important disadvantage. It can be vulnerable to malicious attack, legal action, and technical failure since the content description and the ability to access it is controlled by a single institution [12]. Examples of P2P systems based on super-peer networks include Publius [74], JXTA [2], Edutella [81, 82] and Napster [3].

2.2.4 Comparing P2P Systems

In the previous subsections, we presented characteristics of different structures of P2P network. A comparison of these networks can be based on the ability of with which they can adapt to volatility, scalability and autonomy. Table 2.1 summarizes the comparison.

Characteristic	**Unstructured**	**Structured**	**Super-peer**
Fault-tolerance	high	high	low
Scalability	low	high	low
Autonomy	high	low	moderate
QoS	low	high	high

Table 2.1 – Comparing P2P networks

Initially, we compare the ability of the networks to deal with the volatility of peers (i.e., fault-tolerance). While super-peer networks rely on powerful but fail-prone peers (super-peers), unstructured and structured networks are completely decentralized with high degree of fault-tolerance. In super-peer networks, whenever a super peer leaves the system, the rest of the system suffer. For instance, centralized

index can not be reached when a super-peer fails and it becomes impossible to find data. However, in other networks a failed peer may have its data replicated and another peer takes its place.

Concerning scalability, super-peer networks can not scale up since new powerful peers can not be easily added. Unstructured networks have also some scalability issues, but related to their expensive traffic load (e.g., flooding, random walk), besides some efforts to reduce it. In contrast, structured networks are completely scalable. They are completely decentralized with low network traffic due to their message routing mechanism.

An issue in structured networks is the autonomy, which is low because the peers can not control the data they store. In fact, the storage is divided across all peers, where a peer is responsible to a range of keys. Super-peer networks also have autonomy issues, but related to connection restrictions that can be imposed by the super-peers. In contrast, unstructured networks have high degree of autonomy since peers may store their objects freely.

Finally, we compare the quality of service (QoS), e.g., completeness, data availability, query response time, etc. Indeed, both structured and super-peer networks guarantee high QoS, but in different approaches. While structured networks guarantees a high QoS based on the message routing mechanism, super-peer networks rely on centralized peers that keep the indices of the object location. In contrast, unstructured networks does not provide guarantee to completeness, because queries may not reach the object keeper.

2.3 Testing Architectures for Distributed Systems

We classify three possible approaches to execute test cases across a distributed system. Consider a simple example to illustrate each approach. A DHT stores pairs “(key,value)” across the nodes in system, where each node is responsible to keep a certain set of pairs. Three operations are provided to manipulate these pairs: lookup, put and retrieve. A test case is written to verify the retrieval operation. This test case is composed by the put of some value, then the retrieval of it, and finally the retrieved value is compared to the put one in order to assign a verdict.

In the first approach, it is considered that a single node represents the whole system. Figure 2.4 illustrates a node receiving a test case to be executed. Although, this approach may find implementation problems, it is not capable to find issues triggered by the communication between nodes. In fact, this approach will execute like an unit test performed by Junit [62].

In the second approach, multiple nodes can be used along testing. This approach aims to reduce the time cost of the testing process by deploying different test cases across different nodes in parallel. Figure 2.5(a) illustrates this approach where each node receives a copy of the same test case. While this approach tests a distributed system more properly, it is limited to simple tests since each node executes the entire test case. For instance, it is possible to perform a stress test with several nodes putting pairs in parallel, however, it is impossible to test if a node retrieves a pair put by another one.

As we mentioned, Distributed systems are particularly difficult to test. The system may have several possible sources of input and output spread across the network. In this case, a good approach to test is the decomposition of test cases in actions. Then, different nodes can be responsible to execute an action or a set of them. Yet, the nodes that compose the system may be heterogeneous, meaning that the execution time of test case actions varies for each node. Consequently, synchronization among test case actions is necessary.

In the third approach, again all nodes of the system participate in the test. However, the test case is decomposed in actions, also called test case events or test case actions, in order to be deployed across the

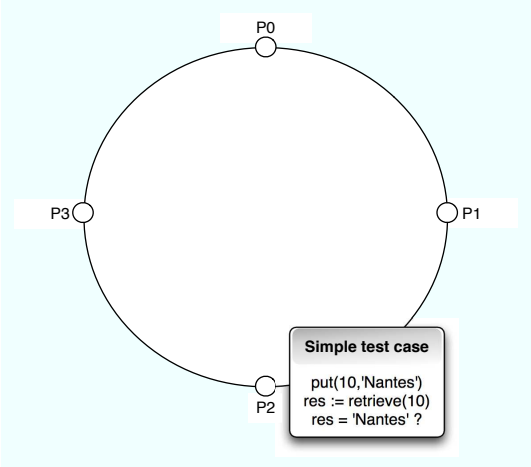
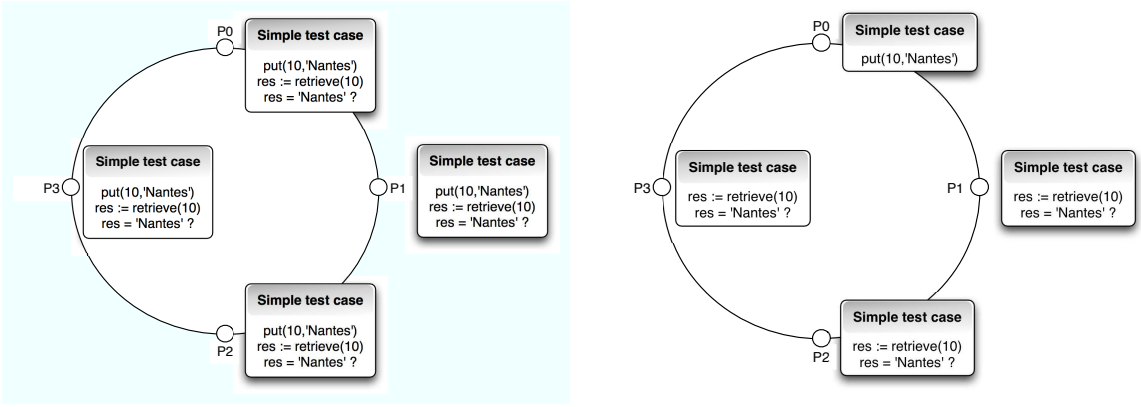


Figure 2.4 – One node representing the whole system

nodes. Figure 2.5(b) illustrates this approach where each node receives a set of actions to be executed. This approach can execute complex tests since different actions can be executed by different nodes. For instance, node $P0$ puts a pair into the DHT, then the other nodes will retrieve such pair and assign a verdict afterward. We believe this approach is more suited to P2P system testing, thus, we base our contribution on it.



(a) Entire Test case (b) Decomposed Test case

Figure 2.5 – Distributed approaches

2.3.1 Conformance Testing Methodology and Framework (CTMF)

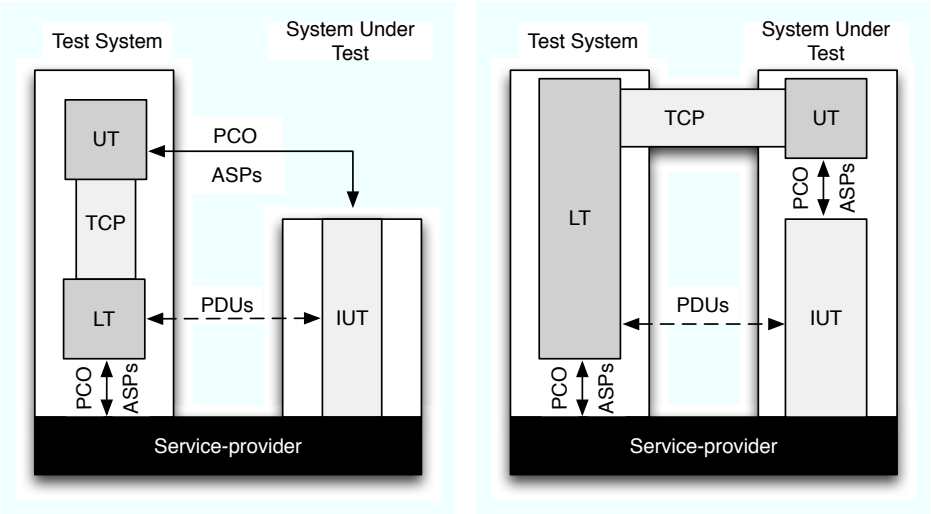
In this section, we describe some general architectures presented by the international standard ISO/9646 “Conformance Testing Methodology and Framework”(CTMF). Then, we describe the architectures beyond the CTMF showing their advantages and drawbacks focused on P2P system testing.

The international standard 9646 “Conformance Testing Methodology and Framework”(CTMF) [5] has been considered as the reference of testing architectures [101, 89, 76, 75, 59]. The CTMF defines

conceptual architectures to support test execution. A test architecture is a description of how an Implementation Under Test (IUT) is to be tested, in particular, what inputs can be controlled and what outputs can be observed. Yet, the CTMF defines a System Under Test (SUT) as a set of IUTs.

Figure 2.6 illustrates the CTMF single-party architectures. Single-party is aimed at testing an IUT that communicates in a point-to-point connection, with a single node. An IUT is composed by upper and lower interfaces through which is tested. Typically, the lower interfaces are accessible only from remote. These interfaces are tested by two components called the Upper Tester (UT) and the Lower Tester (LT). The tests are performed in the Points of Control and Observation (PCO) through control and observation of the inputs and outputs from/to the IUT.

Conceptually the IUT and the testing system communicate using different components. While IUT and UT communicate by means of Abstract Service Primitives (ASPs), IUT and Lower Tester (LT) exchange Protocol Data Units (PDUs). In practice, Schieferdecker et al. [89] describe that it is not necessary to distinguish between ASP and PDU since they are encoded together. Therefore, only the term PDU is used. Yet, both UT and LT can be realized in the same tester process [89, 96, 38], otherwise they need to exchange coordination messages using the Test Coordination Procedure (TCP).



(a) Local method (b) Distributed method

Figure 2.6 – CTMF Single-party Architectures

Figure 2.6 also illustrates the local and the distributed methods. In fact, the difference between them is the management of the PCOs. While in the local method the PCOs are managed from within the testing system, in the distributed method they are managed from within the SUT.

Figure 2.7 illustrates the CTMF multi-party architecture. In this setting, an IUT can have several nodes running at the same time. Thereby, more than one LTs and UTs are active to control and observe the IUT. The coordination between LTs is performed by a particular entity called Lower Tester Control Function (LTCF) through Coordination Points (CP). Besides the coordination, such entity performs two procedures. First, it starts all the LTs. Second, it determines the final test verdict gathering information from all the LTs after they stopped the test case execution.

Several architectures were based on the CTMF, however, most of them consider the CTMF too restrictive with respect to the types of systems that can be tested. Furthermore, they consider that the

up-coming distributed systems requires a flexible and adaptive test architecture. As a result, Walter et al. [101] present a flexible and generic architecture beyond conformance testing (i.e., Interoperability, Performance, Quality-of-service, etc). Further authors also described new architectures. Thus, we divided these architectures in the next subsections in order to survey their advantages and drawbacks. The first subsection describes the single-party architecture since it has one tester (LT+UT) communicating with the IUT. The second subsection describes the multi-party architecture since it has more than one tester (LT+UT) communicating with the IUT.

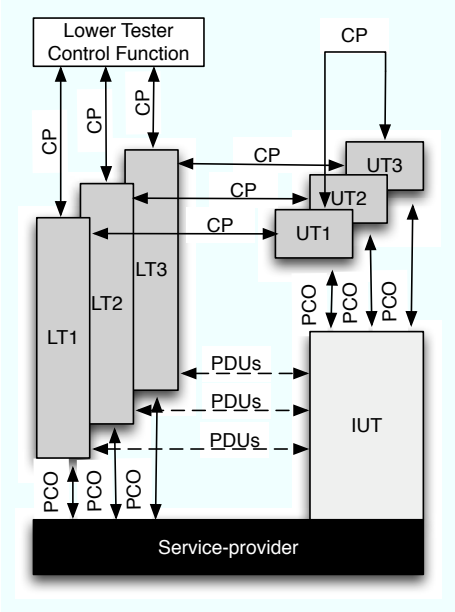


Figure 2.7 – CTMF Multi-party Architecture

2.3.2 Single-party architecture

The single-party architecture is used when the IUT communicates in a point-to-point connection, with a single node. Basically, this architecture has one tester (LT+UT) communicating with the IUT. This architecture is used by some tools such as ATIFS (a testing toolset with software fault injection) [75] comprising its fault injection mechanism FSoFIST (Ferry-clip with Software Fault Injection Support Tool) [76], and TorX [18, 96].

Figure 2.8 illustrates the TorX architecture. One tester is used to communicate with the IUT, in their case the Conference Protocol Entity (CPE), while the control and observation of the CPE is done through its interfaces defined as CSAP and USAP.

A test execution in TorX works as follows. Each time the tester decides to trigger the IUT with an input, it looks into the system specification module (a module of TorX) for a valid stimulus and offers it to the IUT. Whenever the tester observes an output, it checks whether this response is valid according to the specification module. This process of offering inputs and observing outputs can continue until an output is considered incorrect according to the specification, then resulting in a fail verdict. Moreover,

the output observation has a timeout which returns an empty if expired. In this case, the verdict will be also assigned fail.

Another testing tool called FSoFIST has its architecture similar to the TorX's. It includes a supplementary module to inject environment faults during a test execution. Yet, this module also aims at observing the IUT behavior in the presence of the injected faults.

These architectures are not able to test P2P systems for three reasons. First, they rely in only one tester that allows to control only one peer. In this case, one must assume that this peer represents the entire system. However, flaws related to P2P features (e.g., communication, lookup, etc) cannot be detected. Then, a better testing architecture requires multiple testers to control the distributed peers. Second, false-negatives verdicts can be assigned due to communication delay. For instance, in TorX an expired timeout returns an empty output resulting in a false fail verdict. Third, the volatility is not managed at all either returning an empty output or considering an error from the IUT. In both cases the results will be false-negative verdicts (i.e., false fail verdicts).

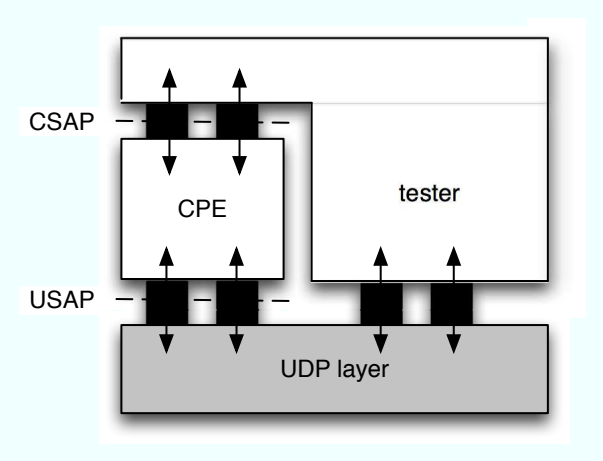


Figure 2.8 – TorX Architecture

2.3.3 Multi-party architecture

In the multi-party architecture context more than one tester controls and observes the IUT. These distributed testers collect partial information about the execution progress and use them to synchronize the rest of the execution. Synchronization can be managed either by a centralized test coordinator or by distributed testers. Ulrich et al. [98] describe that the distributed testers is of interest since it makes better use of system resources than the centralized test coordinator that can be a bottleneck during the test run.

In this section, we first discuss some architectures that synchronize the test sequence with a centralized test coordinator. Then, we discuss some other architectures that do the same in distributed manner. Finally, we survey the architectures focused on P2P system testing.

2.3.3.1 Centralized testing coordination

The classical architecture for testing a distributed system consists of a centralized tester which sends the test inputs, controls the synchronization of the distributed system and receives the outputs (or local verdicts) of each node of the SUT.

Several solutions based on this approach have been presented. Some of them focus on selecting representative subsets of test suites [52, 103, 66, 100, 105] prioritizing the execution of the most significant test cases. Some others distribute the execution of test cases simultaneously over a set of machines like SysUnit [4], Joshua presented by Kapfhammer et al. [64], GridUnit presented by Duarte et al. [44, 45], FIONA presented by Gerchman et al. [50] and BlastServer presented by Long and Strooper [71].

Figure 2.9 illustrates the three components of the Joshua architecture. The first component is the *TestController* which is responsible to prepare the test cases and to write them into the second component called *TestSpace*, that is a storage area. The third component, called *TestExecutor*, is responsible to consume the test cases from the *TestSpace*, to execute them, and to write the results back into the *TestSpace*. Finally, the *TestController* monitors the *TestSpace* for the results and updates the Joshua's interface with them.

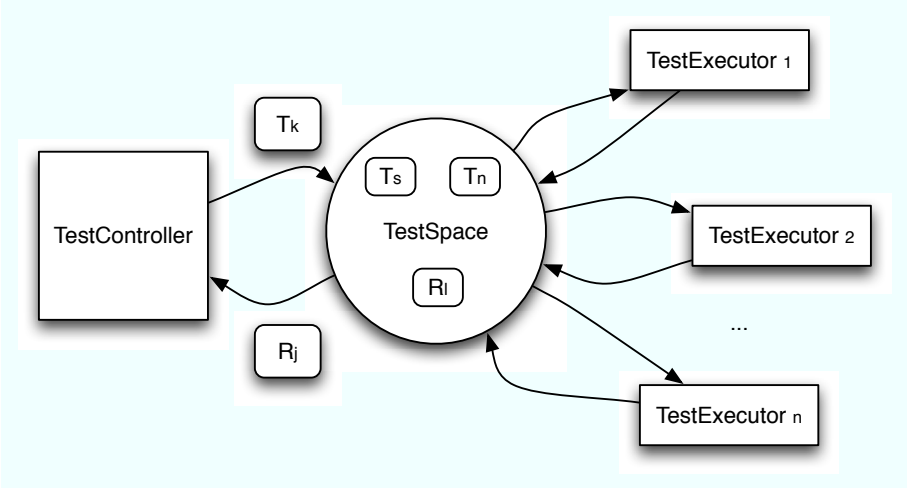


Figure 2.9 – The Joshua Architecture

The GridUnit architecture is based on Joshua. The main goal of GridUnit is to deploy and to control unit tests over a grid with minimum user intervention in order to distribute the execution of tests and speed up the testing process. To distribute the execution, each test case is transformed in a grid task. The control and scheduling of the tasks are provided by the OurGrid platform [30, 26]. Therefore, different test cases can be executed by different nodes. However, a single test case is only executed by a single node. Yet, it is not possible to decompose a test case in actions where different nodes execute different actions. Moreover, both architectures do not handle node volatility.

BlastServer and FIONA are other architectures similar to Joshua. They use the client/server approach. A server component is responsible to synchronize events, while a client component provides the communication conduit between the server component and the client application. The execution of tests is based on a queue controlled by the server component. Clients requests are queued, then consumed when nee-

ded. This approach ensures that concurrent test sequences run to completion. However, they can not be used to test P2P systems due to scalability issue. The issue is that centralized testing coordination could not scale up to a large system and – due to action synchronization tasks – impacted the performance of the SUT. Yet, BlastServer does not address the issues of network failures.

2.3.3.2 Distributed testing coordination

While the test case decomposition allows the execution of complex test cases, it requires actions synchronization to guarantee the correct sequence of execution. Furthermore, such kind of synchronization requires an architecture to coordinate when and where to dispatch the actions.

Ulrich et al. [98, 83] present a testing architecture for distributed systems, called Test and Monitoring Tool (TMT), that coordinates actions by synchronization of events. This architecture uses a global tester and a distributed tester. The global tester divides the test cases in small parts called *partial test cases* (PTC). Each PTC is assigned to a distributed tester and can be executed in parallel to another PTC with respect to a function that controls mutual exclusivity. The behavior of the distributed testers is controlled by a *Test Coordination Procedure* (TCP) which coordinates the PTCs execution by synchronization events. By using synchronization events, the distributed testers do not need to control the execution of the entire test case. Each tester runs independently its partial test case

Another testing architecture was presented by Walter et al. [101] for conformance, interoperability, performance and real-time testing. This architecture works as a toolbox of components (e.g., communication, test coordination, etc) which can be combined to develop a specific testing architecture. Figure 2.10 illustrates a specific architecture for interoperability testing. Three points of communication (PC) A, B and C are monitored by their respective test components (TC) which also provide the verdicts and the control of the SUT.

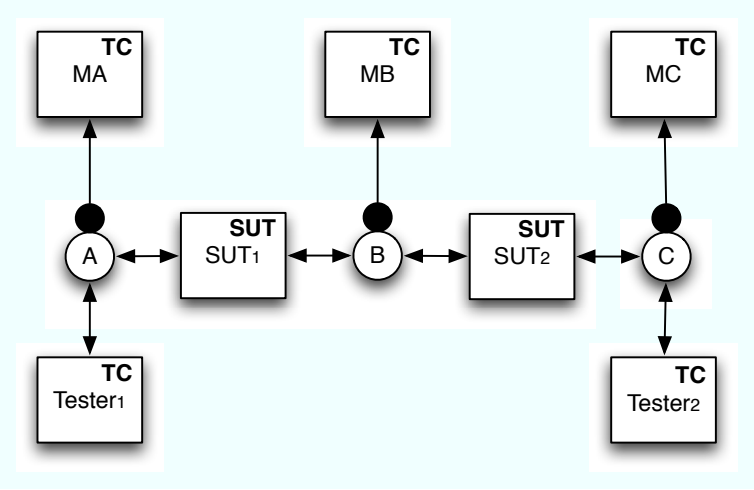


Figure 2.10 – A specific interoperability testing architecture based on Walter et al. [101].

The components can be also used to simulate failures along the tests like network delay or noise. For instance, it can be possible to create disturbs in the network to simulate volatility. However, such a disturb may be considered a failure.

The departure of nodes can also prevent the synchronization of events in the architecture of Ulrich et al. [98, 83]. Consequently, the whole test architecture deadlocks. These architectures have also an issue in the presence of result incompleteness, since the result sets are expected to be complete when testing traditional distributed systems.

Furthermore, scalability is an issue. Despite this architecture synchronize the test sequence through distributed testers exchanging messages at PCs, the performance of the P2P SUT may be perturbed. Indeed, the performance of the synchronization algorithm within such testing architecture scales up linearly with the number of peers, while a typical P2P system scales up logarithmically. To avoid testers intrusiveness to threaten the validity of test results, the testing architecture should thus have logarithmic performance in the worse case.

2.3.3.3 P2P testing architectures

Due to the specific characteristics of P2P Systems (e.g., volatility and scalability), the former distributed testing architectures may not detect P2P implementation flaws. The main problem to build a proper P2P testing architecture is how to support the volatility of nodes without considering it as an error and also scale up along with the SUT. Moreover, how to control the execution of tests, but avoid the inclusion of additional code inside the SUT.

Some testing architectures address the volatility and scalability issues like P2PTester presented by Butnaru et al. [42], Pigeon presented by Zhou et al. [110], and the framework presented by Hughes et al. [57]. However, all of them requires the inclusion of additional code in the SUT source code. This inclusion can be either manually, for instance using specific interfaces, like in P2PTester, or automatically, using reflection and aspect-oriented programming, like in Pigeon.

The inclusion of additional code is error-prone since the added code may produce errors and contaminate the test results. Furthermore, it is hard to verify if the error came from the SUT or the testing architecture.

In fact, all these architectures are used to measure the performance of P2P systems. For instance, P2PTester measures the time and costs of processing jobs such as data indexing or querying. The platform also logs the communication spawned from processing each specific query or search issued by a node. The same approach is used by the framework of Hughes et al. [57].

The Pigeon framework is another platform focused on performance testing for massively multiplayer online games (MMGs). The authors believe that the performance of MMGs is the most crucial problem that should be addressed since the network latency is impacted by the frequent propagation of updates during a game. Then, a layer called P2P Communication Model (PCM) is charged to monitor the network latency and log the exchanged messages.

While measuring the performance of the systems can be interesting to execute both performance testing and benchmarks, it is not enough to other kinds of tests. Moreover, all of these platforms rely on logging while testing. Assuming that these platforms aim to verify correctness, they must check the log files after the test execution using an oracle. However, none of the platforms provide any oracle mechanism to do it. The use of logging to verify correctness is discussed later on in this chapter.

2.3.4 Other approaches

There are also some model-checking tools (e.g., Bogor [14], SPIN [37, 108] and Cadence SMV [49]) to check a kind of P2P system called publish-subscribe system (PSS) [22]. In one hand, model-checking is an attractive alternative to find bugs in systems by exploring all possible execution states [49, 60].

In the other hand, it is still based on models and implementation problems can not be found. While they consider the volatility nature of PSS, scalability is an open issue. Along model-checking, peers are simulated as threads and the size of the SUT may be bounded by the checking machine resources. Therefore, large-scale P2P systems cannot be fully checked since implementation flaws are strongly related to such large-scale [33]. Furthermore, none of these approaches address the volatility of peers that may lead to an exponential number of execution states.

2.3.5 Comparing Testing Architectures

We divide the comparison of testing architectures in two perspectives. In the first perspective, we compare from the testing point of view. In the second perspective, we compare from the P2P point of view by discussing architectural choices and the treatment of the P2P characteristics.

From the testing perspective, a testing architecture must fulfill some requirements such as a mechanism to control the execution of tests, a manner to develop meaningful tests and a mechanism to verify correctness. Table 2.2 describes that some of the architectures surveyed in this section do not address all these requirements.

Architecture Name	**Testing Type**	**Action Decomposition**	**Correctness Treatment**	**SUT Code Contamination**
TorX	generic	no	logging	no
FSoFIST	fault injection	no	logging	no
SysUnit	unit	no	assertions	no
Joshua	unit	no	assertions	no
GridUnit	unit	no	assertions	no
BlastServer	generic	no	assertions	no
FIONA	fault injection	no	logging	no
TMT	conformance	yes	logging	yes
Walter et al. [101]	generic	yes	-	-
P2PTester	performance	no	logging	yes
Pigeon	performance	no	logging	yes
Hughes et al. [57]	performance	no	logging	yes

Table 2.2 – Comparing testing requisites

Each architecture is focused on a different type of test. However, we are interested on a generic type of test that can be easily adapted to any other type. For instance, using a Junit's like unit test we can decompose test cases in actions providing the possibility to build a conformance test like the example illustrated in figure 2.5(b).

Concerning the test control requirement, four architectures do so including supplementary code inside the SUT. While such approach provides a straightforward way to control the execution of the tests, errors from the supplementary code can contaminate both the SUT and the test itself, and lead to unpredictable results. A better approach in this case is provided by GridUnit, which implements unit tests accessing the SUT's public interfaces. Concerning the correctness treatment requirement, the advantages and drawbacks will be discussed later on in this chapter.

From the P2P perspective, a testing architecture must be clearly in multi-party manner. Table 2.3 describes that several architectures address this requirement, but when testing a P2P system the architecture must also scale up along with the SUT, which means the use of a central coordinator becomes a bottleneck. In fact, only three architectures can scale up along with a P2P SUT by avoiding a central coordination. They are P2PTester, Pigeon and the framework presented by Hughes et al. [57]. Since these architectures are focused on testing P2P systems, they also handle the volatility of peers. Yet, FSoFIST and FIONA handle the volatility of nodes during tests as well, however, they consider it as an injected fail and do not consider it as a common behavior of the SUT.

While the P2P perspective points out in favor to the last three architectures, the testing perspective pointed out that some flaws (e.g., SUT contamination and no action decomposition) may restrict their use to test P2P systems.

Architecture Name	**Architecture type**	**Test case coordination**	**Volatility aware**
TorX	single-party	no	no
FSoFIST	single-party	no	yes
SysUnit	multi-party	centralized	no
Joshua	multi-party	centralized	no
GridUnit	multi-party	centralized	no
BlastServer	multi-party	centralized	no
FIONA	multi-party	centralized	yes
TMT	multi-party	centralized	no
Walter et al. [101]	multi-party	centralized	no
P2PTester	multi-party	distributed	yes
Pigeon	multi-party	distributed	yes
Hughes et al. [57]	multi-party	distributed	yes

Table 2.3 – Comparing P2P requisites

2.4 Test Oracles

Test oracles are used to check the SUT correctness given by means of verdicts. A verdict is the result of a test case execution and is given by comparing the actual result with an expected one provided by a trusted source. A *pass* verdict is given when the SUT produces an acceptable result, otherwise it will be *fail*.

Baresi et al. [15] describe that in industry the oracle is often a human being and point out two drawbacks to this approach: accuracy and cost. While the accuracy of the human "eyeball oracle" drops with the large volume of tests to be evaluated, the cost becomes too expansive. Moreover, humans are prone to error when checking complex behaviors.

The oracle is a trusted source of expected results that would be behaviorally equivalent to the implementation under test [20]. However, a perfect oracle can not be guaranteed and it is impossible to decide that an output is correct in all possible cases [73]. In case of a *pass* verdict, a stopping criteria may define

the end of the testing process. However, if a *fail* verdict is assigned, then a bug was found. As a further process, a diagnosis process can be executed to localize the exact character of such bug.

In this section we survey the oracle approaches more related to test implementations like assertions and log file analysis. Other oracle approaches related to model-checking, FSM and LTS are discussed by Baresi et al. [15].

2.4.1 Assertions

Tao Xie [104] describes that oracles are often in form of runtime assertions. Assertions [95] allow either to check required constraints directly in the program source or build test applications. In fact, an assertion is a boolean expression that defines required constraints.

Assertions are coded during development and enabled before runtime. Along runtime a program execution may reach an assertion, if this assertion is satisfied, then it is evaluated to *true*, otherwise *false*.

For instance, consider the following code in java :

```
public Double fahrenheitToCelsius (Double fahrenheit ){
    assert fahrenheit != null : "Unacceptable temperature";
    // now return value
    return ( fahrenheit .doubleValue() − 32) / 1.8;
}
```

This code converts the temperature from fahrenheit to celsius, but it does not accept *null* values. In this case, an assertion checks the entered values. If the assertion predicate becomes false, meaning that a necessary condition was not met, then an assertion violation occurs. This violation is also called as “assertion failure”. In fact, the assertion has not failed, but it has detected an unacceptable situation.

Assertions can be provided either by a programming language (e.g., Java, Eiffel, C++) or by a testing framework (e.g., JUnit [62], JTiger [61], TestNG [94], GNU Nana [51] and APP [87]). The difference is that a testing framework can be used to develop a test application, while the assertions of programming languages are built inside the SUT’s source code and used to check required constraints.

The following code is a JUnit test application used to test the fahrenheit to celsius conversion method.

```
@Test
public void testFahrenheitToCelsius (){
    Double celsius = fahrenheitToCelsius (32);
    assertEquals (0.0, celsius .doubleValue );
}
```

In the above test application, the input value is 32° Fahrenheit and the expected output value corresponds to 0° Celsius. If the values are the same, then the verdict is *pass*, otherwise it is *fail* and a bug was detected.

Assertions can provide many advantages. Some of them include :

- The condition checking can easily detect interface faults (e.g., incorrect parameters, inconsistent messages) that can escape during the system development. Still, an assertion violation is a bug [79].
- Errors and omissions are caught early and automatically. This means, assertions prevent errors [20].
- It offers a cheap and effective tool to testing effectiveness [20].
- Powerful tool for automatic detection of faults during debugging, testing and maintenance [87].

A practical use of assertions is described by Berg et al. [19] and Robert Binder [20]. During the development of the IBM OS/400 operating system (OS), the assertions were widely used. They report that the entire OS has 20 millions lines of code, where 2 millions are C++ code. About half of this C++ code was dedicated testing code where roughly 40% were assertions.

However, assertions have some drawbacks including :

- Assertions are written by humans, therefore, prone to errors as well. Specially for built-in assertions.
- Assertions are source code. This means more maintenance effort from the developers.
- Assertions can not detect all errors. They can not express all valid conditions. Yet, human semantics can create confusion [20].
- An incorrect assertion will not report itself as defective [20].

Nevertheless, assertions provide a powerful and straightforward test oracle approach. Yet, both build-in assertions and assertions in test applications can be combined to improve testing effectiveness.

2.4.2 Log File Analysis

Log files are the real-world manifestation of mathematical report traces [10]. A report trace is a set of reports and a report is a set of report elements. In a log file each report is a log entry with a set of elements. For instance, Figure 2.11 illustrates the entries of a log file where each log entry contains its set of elements. In this case, the elements are the entry time-stamp, the log level and the log message.

Log file analysis is the inspection of log files to identify and diagnose problems. This inspection can be done informally, by the programmer, or formally, by using formal methods and log file analysis languages (LFAL) [9, 15].

```
16:32:16.439|FINEST|start Action: BeforeClass Hierarchy: -2147483648
16:32:16.441|FINEST|startingNetwork Action: action1 Hierarchy: 0
16:32:16.442|FINEST|joiningNet Action: action2 Hierarchy: 0
```

Figure 2.11 – Log file

The effective use of log file analysis is based on some hypotheses [11]. Initially, the SUT records its events to an output called log file. These events can be messages exchanged, returned values, or any other input/output. Then, a log file policy must be defined to state precisely the outputs from the SUT. In fact, both log file and its policy can be easily built using some available logging frameworks [53, 70]. Finally, an analyzer must take the log files as input either accepting it or rejecting it with an error message.

Andrews et al. [9, 10, 11] used an analyzer for both unit and system testing. Such analyzer compares each log entry with a dictionary to decide either *pass* or *fail*.

The log files may also be loaded in databases. This approach takes advantage of database capabilities to generate reports (i.e., database queries). However, it has some issues: how to translate verification conditions into queries, generate reports and load large amounts of data. The Planchecker system [46, 47] follows this approach (see Figure 2.12). The steps of log file analysis on Planchecker are :

1. Database setup. Database structures are created to hold the content of the log files. Each log message is represented as an object in the database.
2. Data loading. Loading data is a straightforward process whether the log files are well-structured. Moreover, the loading of very large data files into databases is not a novel challenge and is also an object of study in data warehousing [8, 67].
3. Translation. This step determines the validation conditions, and expresses them as database queries. It is also a straightforward process since query languages (e.g., SQL, AP5) are very mature and well developed.
4. Log analysis. Databases have rules capabilities [93, 102] that can be used to check violations. Yet, the results from queries are organized into confirmation and anomaly reports.

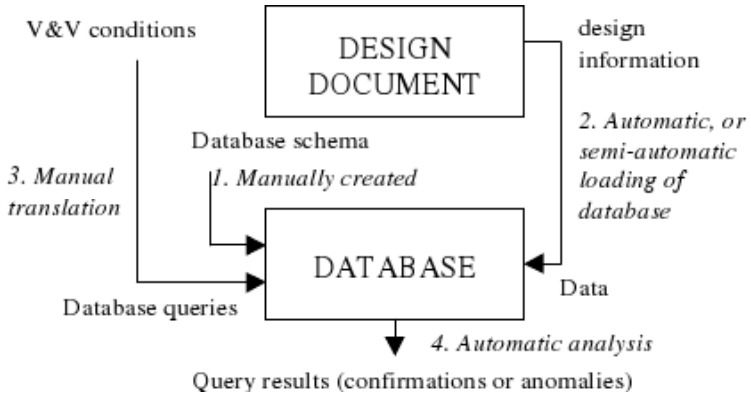


Figure 2.12 – Planchecker Architecture

Another important advantage of this approach is the querying of very large volume of data. Most of the actual Database Management Systems (DBMS) can process terabytes of information to produce analysis reports [28].

Log file analysis arises some issues if used in P2P system testing. First, log files are spread across distributed nodes and must be gathered to be loaded into a database. Second, each log entry has a timestamp registering the time a given event happened. However, each node in the network has a different clock. Thereby, a global clock must be defined in order to build an overall log file. Once this global clock is defined, then every log entry must be recalculated.

2.4.3 Comparing Test Oracles

In the previous subsections, we presented the characteristics of different oracle approaches. A comparison of these oracles can be based on the approach type, the oracle approach and the extension of the SUT source code with supplementary code. Table 2.4 summarizes the comparison.

Some programming languages provide embedded assertion facilities, but there are also external assertion facilities that can be added (i.e., APP, GNU Nana). While embedded assertion facilities are very useful during the development of a system. Testing frameworks can be used to create testing applications that can be applied in different types of tests (e.g., unit test, system test, etc).

Moreover, embedded assertion facilities extend the SUT source code with supplementary assertion code. Besides its advantages to detect several types of errors during the system development, it has two main drawbacks. First, the added code must be maintained as long as the SUT code evolves. Second, the added code is also error prone since it is added by humans. Nevertheless, such approach showed effective in detect errors and can be used complementary to the testing framework approach.

In contrast, log file analysis does not provide a framework to develop testing applications, neither extends the SUT source code. In fact, it uses the SUT outputs to identify and diagnose problems. However, if a distributed SUT is tested, some issues arise. First, a distributed SUT generates log files spread across distributed nodes that must be gathered. Second, a global clock must be defined to manage the clock difference between distributed nodes. This global clock is useful whether to build a global log file.

Approach Name	**Approach Type**	**Oracle Approach**	**SUT Code Extension**
Java assertion facility	Language embedded	Assertions	yes
C++ assertion facility	Language embedded	Assertions	yes
Eiffel assertion facility	Language embedded	Assertions	yes
JUnit	Testing Framework	Assertions	no
JTiger	Testing Framework	Assertions	no
TestNG	Testing Framework	Assertions	no
APP	External assertion facility	Assertions	yes
GNU Nana	External assertion facility	Assertions	yes
Andrews et al.	Testing Framework	Log file analysis	no
Planchecker	Testing Framework	Log file analysis	no

Table 2.4 – Comparing testing oracles

In conclusion, we can state that the log file analysis and both assertion approaches are complementary, and can be combined in order to improve testing effectiveness.

2.5 Conclusion

In this chapter, we discussed the testing approaches for distributed systems. We presented an overview of the P2P systems to establish the context of our work. We compared the differences between P2P systems and traditional distributed systems by presenting the main characteristics of P2P systems. Furthermore, we surveyed the main structures of P2P networks: unstructured, structured and super-peer.

We also surveyed the testing architectures comparing their perspectives from the context of distributed testing. Then, we discussed why they are not suited to test P2P systems. We showed that they fail when dealing with the volatility, either interrupting the testing sequence and deadlocking the tester or assigning false-negative verdicts to test cases (i.e., false fail verdict). Then, we showed that most of them do not scale up due to centralized test coordination. In fact, the ones that scale up demand to include additional code into the SUT source code. While this approach supports the volatility control, it is error prone.

We also described some oracle mechanisms. We focused on assertions and log file analysis since they are closely related to test implementations, rather than formal models. The assertion approach can be used in two distinct ways. First, embedded into the source code. In one hand, it can detect several types of errors during the system development. In the other hand, the additional code must be also considered during the source code maintenance. Yet, additional code is error prone since it is added by humans. Second, built in testing applications. Several testing frameworks provide assertion facilities that can be used to build testing application.

The log file analysis approach can also be used in two distinct ways. First, using an analyzer that takes the log files as input either accepting it or rejecting it with an error message. Second, using a database. Log files are loaded into databases to take advantage of its capabilities. Nowadays query languages are very mature and provide a powerful tool to build reports (e.g., anomaly reports). Yet, databases structures like rules and triggers can be used to check violations. Moreover, today's DBMS can load very large amounts of data quickly. Therefore, the load of log files is no longer an issue.

Besides their different characteristics, both assertion and log file analysis showed complementary and can be combined in order to improve testing effectiveness.

CHAPTER 3
Framework for Testing Peer-to-peer Systems

In this chapter, we present an integrated solution for testing large-scale P2P systems. This solution is based on a framework that allows the creation and deployment of a P2P test environment. From the P2P perspective, a test designer can instantiate peers and make them join and leave the system (i.e., peers' volatility). From the testing perspective, a test designer can combine the functional testing of a system with volatility variations, and also different system sizes (i.e., system scalability). The correctness of such a system can thus be checked based on three-dimensional aspects, i.e., functionality, scalability and volatility.

While testing the scalability of a distributed system, the functional aspect is typically not taken into account. The same basic test scenario is simply repeated on a large number of nodes [45]. This also happens while testing volatility. The basic test scenario is repeated upon different rates of volatility. For a P2P system, we claim that the functional flaws are strongly related to scalability and volatility. Therefore, it is crucial to combine the scalability and volatility aspects with meaningful test sequences. Thus, we present an incremental methodology to deal with these three-dimensional aspects [33]. Our methodology aims at covering functions first on a small system and then incrementally scaling up this system introducing the volatility later on.

We also present three different architectures to coordinate the tests. These architectures are based on the multi-party CTMF architecture coordinating the tests either by a centralized component or by distributed ones.

The capabilities of these architectures are (1) to automate the execution of each local-to-a-peer test case, (2) to build automatically the global verdict, (3) to allow the explicit control of each peer volatility. These architectures are based on two original aspects : (i) the individual control of peers' volatility and (ii) the distribution of the coordination components to cope with large numbers of peers.

The first architecture extends the multi-party centralized testing architecture, centralized architecture for short, in order to handle the volatility of peers [34]. Basically, this architecture has two main components : the tester and the coordinator. The tester is the application that executes in the same logical node as peers, and controls their execution and volatility, making them leave and join the system at any time, according to the needs of a test. Thus, the volatility of peers can be controlled at a very precise level. The tester is composed of test suites that are deployed across several logical nodes.

The coordinator is deployed in only one node and is used to coordinate the execution of test cases. It acts as a *broker* [21] for the deployed testers.

However, the algorithmic performance of such architecture scales up linearly with the number of peers, while a typical P2P system scales up logarithmically. To avoid testers intrusiveness to threaten the validity of test results, the testing architecture should thus have logarithmic performance in the worse case. Therefore, we present two multi-party architectures that are fully distributed, distributed architectures for short, to cope with large-scale P2P systems [36]. These architectures replace the central

coordinator with distributed testers that coordinate the execution of test cases in distinct manners.

The second architecture organizes the testers in a balanced tree [17] (B-Tree) manner where the coordination is performed from the root to the leaves. Then, the testers communicate with each other across the B-tree to avoid using a central coordinator. The third architecture uses gossiping messages among testers. This reduces communication among testers that are responsible to execute consecutive test case actions. Since both distributed architectures do not rely on a central coordinator they scale up correctly. These architectures do not address the issue of test cases generation but is a first element towards an automated P2P testing process. It can be considered analogous to the JUnit[1] testing framework for Java unit tests.

This chapter is organized as follows. Section 3.1 introduces the basic definitions used along this thesis. Section 3.2 presents the incremental testing methodology including a discussion of the three dimensional aspects. Section 3.3 introduces the framework including a documentation to develop test cases. Within the framework we present the three testing architectures.

3.1 Definitions

Let us denote by P the set of peers representing the P2P system, which is the system under test (SUT). We denote by T, where $|T| = |P|$ the set of testers that controls the SUT, by DTS the suite of tests that verifies P, and by A the set of actions executed by DTS on P.

Considering that all peers have exactly the same interface, testing the interface of a single peer is not sufficient to test the whole system. Actually, to test the whole system several peers have to run together the same test case. However, they may not execute such test case in the same way. Each peer executes a different part of the test case in order to reproduce a precise behavior. This is why we introduce the notion of distributed test cases, i.e., test cases that apply to the whole system and whose actions may be executed by different peers.

Definition 5 (Distributed test case)**.** *A distributed test case noted τ is a tuple $\tau = (A^\tau, T^\tau, L^\tau, S^\tau, V^\tau, \varphi^\tau)$ where $A^\tau \subseteq A$ is a sequence of actions $\{a_0^\tau, \ldots, a_n^\tau\}$, $T^\tau \subseteq T$ a set of testers, L^τ is a set of local verdicts, where $|L^\tau| = |P|$, S^τ is a schedule, V^τ is a set of variables and φ^τ is the level of acceptable $inconclusive$ verdicts.*

The Schedule is a map between actions and sets of testers, where each action corresponds to the set of testers that execute it.

Definition 6 (Schedule)**.** *A schedule is a map $S = A \mapsto \Pi$, where Π is a collection of tester sets $\Pi = \{T_0, \ldots, T_n\}$, and $\forall T_i \in \Pi : T_i \subseteq T$.*

A test sequence is a sequence of messages (e.g., service requests) exchanged among peers along testing.

Definition 7 (Test sequence)**.** *A test sequence is a tuple $TS \in (P)$*

In P2P systems, the autonomy and the heterogeneity of peers interfere directly in the execution of service requests. While close peers may answer quickly, distant or overloaded peers may need a considerable delay to answer. Consequently, clients do not expect to receive a complete result, but the available results that can be retrieved during a given time. Thus, test case actions (Definition 8) must not wait indefinitely for results, but specify a maximum delay (*timeout*) for an execution.

[1] http ://junit.org/

Definition 8 (Action)**.** *A test case action is a tuple $a_i^\tau = (\Psi, \iota, T')$ where Ψ is a set of instructions, ι is the interval of time in which Ψ should be executed and $T' \subseteq T$ is a set of testers that executes the action.*

The instructions are typically calls to the peer application interface as well as any statement in the test case programming language.

Definition 9 (Local verdict)**.** *A local verdict is given by comparing the expected result, noted E, with the result itself, noted R. E and R may be a single value or a set of values from any type. However, these values must be comparable. The local verdict v of τ within ι is defined as follows :*

$$l_\iota^\tau = \begin{cases} pass & \text{if } R = E \\ fail & \text{if } R \neq E \text{ but } R \neq \emptyset \\ inconclusive & \text{if } R = \emptyset \end{cases}$$

3.1.1 Motivating test case example

Let us illustrate these definitions with a simple distributed test case (see Example 3.1). This test case will be also used as a motivating example through the chapters to come. The aim of this test case is to detect errors on a DHT implementation. More precisely, it verifies whether new peers are able to retrieve data inserted before their arrival.

Example 3.1 (Simple test case).

Action	Testers	Action
(a_1)	0,1,2	join()
(a_2)	2	Insert the string "One" at key 1 ; Insert the string "Two" at key 2 ;
(a_3)	3,4	join() ;
(a_4)	3,4	Retrieve data at key 1 ; Retrieve data at key 2 ;
(a_5)	*	leave() ;
(v_0)	*	Calculate a verdict ;

This test case involves five testers $T^\tau = \{t_0 \dots t_4\}$ that control five peers $P = \{p_0 \dots p_4\}$ and five actions $A^\tau = \{a_1^\tau, ..., a_5^\tau\}$. If the data retrieved in a_4 is the same as the one inserted in a_2, then the verdict is $pass$. If the data is not the same, the verdict is $fail$. If t_3 or t_4 are not able to retrieve any data, then the verdict is $inconclusive$.

3.2 Testing methodology

Several aspects of a P2P system expose the need for a precise testing methodology, such as : leave/-join of peers, number of peers, data size, amount of data, number of concurrent requests, etc. Thus, the difficulty of testing is not only in choosing the relevant input data, but also in choosing the aspects that should vary, their values and their association.

3.2.1 The three dimensional aspects of a P2P system

As stated in the introduction, P2P testing tackles the classical issue of testing a distributed application, but with a specific dimension which we call volatility, which has to be an explicit parameter of the test objectives. Two possible solutions may be used to obtain a test sequence which includes volatility. It can either be simulated with a simulation profile or be explicitly and deterministically decided in the test sequence. The first solution is the easiest to implement, by assigning a given probability for each peer to leave or join the system at each step of the test sequence execution. The problem with this approach is that it makes the interpretation of the results difficult, since we cannot guess why the test sequence failed. Moreover, it creates a bias with the possible late responses of some peers during the execution of the test sequence. As a result, it cannot be used to combine a semantically rich behavioral test with the volatility parameter. For instance, if we test the recovery from peer isolation where it is verified if a peer correctly updates its routing table when all peers it knows have left the system, then such random volatility simulation is not useful.

In this thesis, we recommend to fully control volatility in the definition of the test sequence. Thus, a peer, from a testing point of view, can leave or join the system at a given time in a test sequence. This action is specified in the test sequence in a deterministic way. For instance, if we want to perform the same recovery from isolation test we only need a system containing enough peers to fill the routing table of a peer. The drawback of any volatility simulation is that test cases must be aware of the arrival/departure of peers, otherwise they may wait indefinitely for messages from peers that are no longer available.

The second dimension of P2P application testing is scalability since we can deal with large numbers of peers. Because it is accomplishing a treatment, the scalability and volatility dimensions must be tested with behavioral and functional correctness. In conclusion, a P2P testing methodology and framework should provide the possibility to control the three dimensional aspects of a P2P system. We summarize them as follows :

- Functionality, captured by a test sequence which enables a given behavior to be exercised.
- Scalability, captured by the number of peers in the system.
- Volatility, captured by the number of peers which leave or join the application after its initialization during the test sequence.

3.2.2 The incremental methodology

To take into account the three dimensional aspects of P2P systems, we present a methodology that combines the functional testing of a system with the variations of the other two aspects. Indeed, we incrementally scale up the SUT either simulating or not volatility. This simulation can be executed with different workloads, such as : shrinking the system, expanding it or both at the same time. These different workloads may exercise different behaviors of the SUT and possibly reveal different flaws.

Our incremental methodology is composed by the following steps :

1. small scale application testing without volatility ;
2. small scale application testing with volatility ;
3. large scale application testing without volatility ;
4. large scale application testing with volatility.

Step 1 consists of conformance testing, with a minimum configuration. The goal is to provide a test sequence set efficient enough to reach a predefined test criteria. These test sequences must be paramete-

rized by the number of peers $TS(P)$, so that they can be extended for large scale testing. Test sequences can also be combined to build a complex test scenario using a test language such as Tela [84].

In our motivating example, we start a stable system with all the peers set as illustrated in Figure 3.1. The peer p_2 will insert some data into a DHT, then the peers p_3 and p_4 will retrieve them. This first step aims to verify pure functional problems without interference with the size of the system and/or volatility. In the case of a stable and small scale DHT, all the peers probably know each other representing minimal or even nonexistent routing table updates. Thus, messages may be exchanged directly between peers.

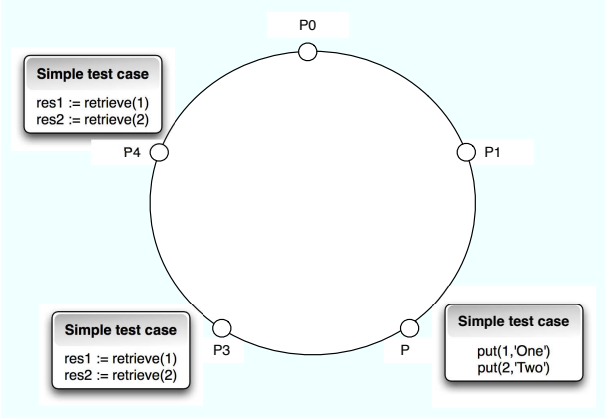


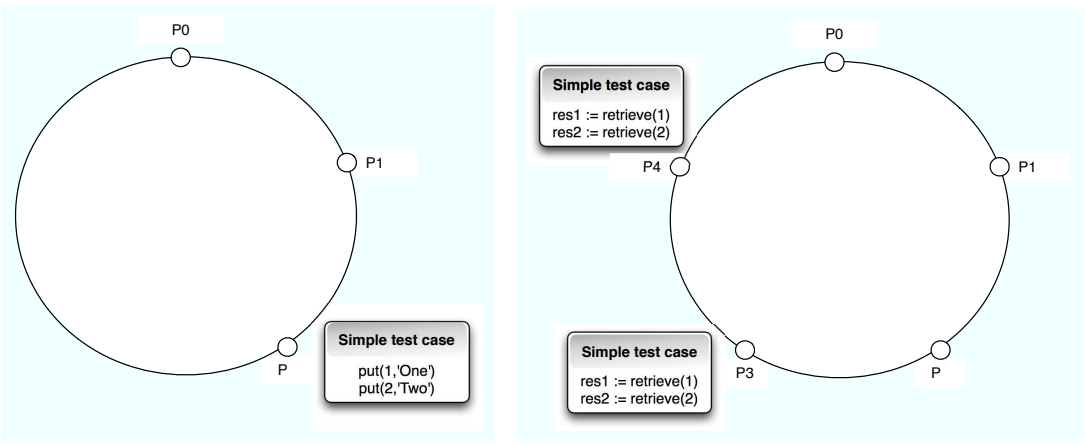
Figure 3.1 – Small scale application testing without volatility (Step 1)

Step 2 consists of reusing the initial test sequences and adding the volatility dimension. The result is a set of test sequences including explicit volatility (TSV). Figure 3.2(a) illustrates a DHT before volatility when data is inserted by peer p_2. Then, the peers p_3 and p_4 join the system and retrieve data as illustrated in Figure 3.2(b). This second step aims to verify functional problems related to volatility at a small scale considering that pure functional problems were isolated at Step 1. Indeed, testing inserts and retrieves upon volatility exercises both data forwarding and routing table update. Furthermore, a small scale system guarantees low forwarding since data tend to be sent to peers within the routing table.

Step 3 reuses the initial test sequences of Step 1 combining them to deal with a large number of peers. We thus obtain a global test scenario GTS. A test scenario composes test sequences. This third step aims to verify functional problems related to scalability. To do so, we test the SUT without volatility in a large scale. As described in Step 1, a stable system represents minimal or even nonexistent routing table updates. Whenever we scale up the SUT, peers are obligated to perform some tasks like routing messages and forwarding data to unknown peers. Indeed, these tasks could be only tested in large scale systems since peers are unlikely to know all the others.

Figure 3.3 illustrates a large scale and stable DHT. In our motivating example, peer p_2 inserts some data into the DHT respectively at p_1 and p_2. Whenever the peers p_3 and p_4 try to retrieve data, they probably do not know p_1 and p_2, messages are routed until reaching such data. Therefore, aspects related to scalability, such as message routing, can be verified from this third step.

Step 4 reapplies the test scenarios of Step 3 with the test sequences of Step 2, and a global test scenario with volatility ($GTSV$) is built and executed. Figure 3.4 illustrates a large scale DHT upon volatility. In fact, this step aims to verify the problems related to all three dimensions. Therefore, after the insertion of data illustrated in Figure 3.4(a), peers come and go depending on the type of the volatility. For simplicity, Figure 3.4(b) illustrates the join of new peers p_3 and p_4. In our example, the successors



(a) DHT before volatility (b) DHT after volatility

Figure 3.2 – Small scale application testing with volatility (Step 2)

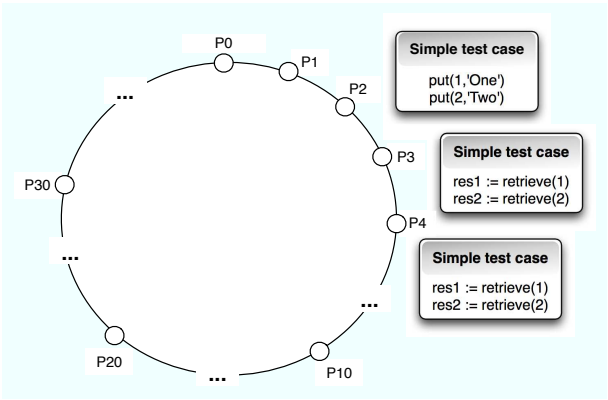
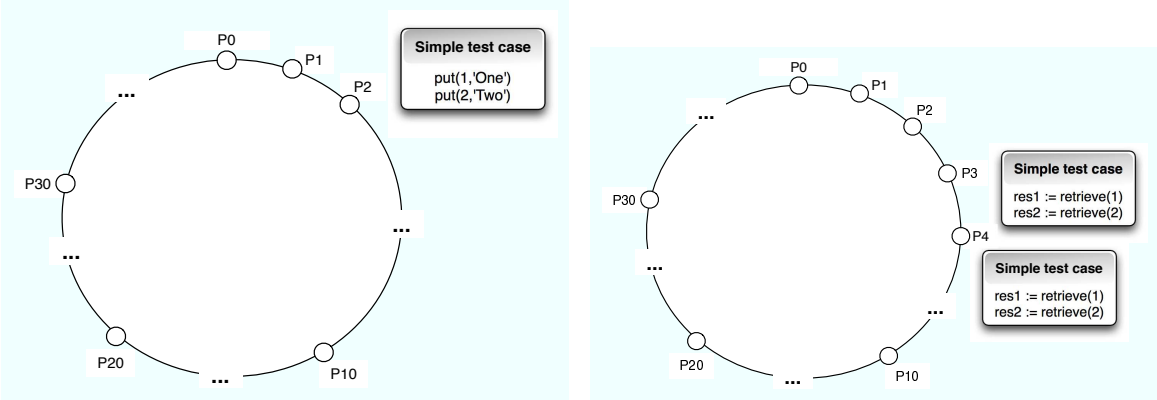


Figure 3.3 – Large scale application testing without volatility (Step 3)

of both p_3 and p_4 have to update their routing table and route messages. Eventually, the test case can be improved to store something at p_3 or p_4 in order to exercise data forwarding as well.



(a) DHT before volatility (b) DHT after volatility

Figure 3.4 – Large scale application testing with volatility (Step 4)

The advantage of this process is to focus on the generation of relevant test sequences, from a functional point of view, and then reuse these basic test sequences by including volatility and scalability. The test sequences of Step 1 satisfy test criteria (code coverage, interface coverage). When reused at large scale, the test coverage is thus ensured by the way all peers are systematically exercised with these basic test sequences.

In terms of diagnosis, this methodology allows to determine the nature of the detected erroneous behavior. Indeed, the problem can be linked to a purely functional cause (Step 1), a volatility issue (Step 2), a scalability issue (Step 3) or a combination of these three aspects (Step 4). The most complex errors are the last ones since their analysis is related to a combination of the three aspects. Steps 2 and 4 could also be preceded by two other steps (shrinkage and expansion), to help the diagnosis of errors due to either the unavailability of resources or arrival of new ones. Yet, several rates of volatility can be explored to verify how they affect the functionality aspect of the SUT (e.g., 10% joining, 20% leaving).

3.3 Framework

In this section, we present a testing framework which encompasses the basic features of distributed testing, volatility and deployment within three scalable coordination architectures.

- The first architecture relies on a centralized test coordinator, where a central controller is responsible to synchronize the execution of test case actions across distributed testers to ensure their correct execution sequence [35, 34, 33]. In fact, this architecture extends the classical architecture for testing distributed systems with the capacity to handle volatility.
- The second architecture drops the centralized coordinator organizing the testers in a B-Tree manner and synchronizing through messages exchanged among parents and children [36].
- The third architecture also drops the centralized coordinator synchronizing through gossiping messages exchanged among consecutive testers [36].

All these architectures were implemented in Java (version 1.5) and make extensive use of two Java features : dynamic reflection and annotations (the prototype is presented in Chapter 4). As we will see in

the following, these features are used to select and execute the actions that compose a test case.

3.3.1 Requirements for a P2P testing framework

Testing P2P systems thus relates to distributed system testing, however, such testing certainly includes both volatility and scalability features which make the existing distributed testing techniques not fully adapted.

We believe that volatility is a parameter which must be part of the functional tests. Thus, volatility must be integrated to functional testing since faulty behaviors may be detected due to variations in the number of online peers. Moreover, the existing approaches do not replace the generation of test sequences which involve volatility explicitly.

Concerning scalability, current testing architectures are not as scalable as P2P systems. In a testing campaign with a large size SUT, a large delay may happen between the execution of consecutive actions due to synchronization. However, such delay is not acceptable due to time out constraints. We believe that testing a P2P system is not only a matter of correctness. Peers exchange messages asynchronously and may not answer a complete result set to a query due to time out constraints or even volatility. Therefore, testing any P2P system is also a matter of time.

In summary, a testing framework for P2P systems should provide :

- distributed control/observation facilities to manage distributed test sequences,
- functions to deal with volatile peers at runtime,
- deployment facilities to deal with high number of peers.

3.3.2 Testers

In general, a tester is the control/observation component responsible to manage test suites (i.e., test cases and test sequences). A test suite is implemented as a class, which is the main class of the testing application. A test suite contains several test cases, which are implemented as a set of actions. Test case actions are implemented as *annotated* methods, i.e., methods adorned by a particular meta-tag, or *annotation*, that informs that the method is a test case action, among other information. Annotations can be attached to methods and to other elements (e.g., packages, types, etc.), giving additional information concerning an element : the class is deprecated, a method is redefined, etc. Furthermore, new annotations can be specified by developers. Method annotations are used to describe the behavior of test case actions : where it should execute, when, in which tester, whether or not the duration should be measured. The annotations are similar to those used by JUnit[2], although their semantics are not exactly the same.

The choice of using annotations for synchronization and conditional execution was motivated by three main reasons. First, to separate the execution control from the testing code. The execution control is a duty of the framework and should be done automatically upon certain rules or parameters. These parameters are provided by the annotations and filled by a test engineer while writing the testing code. Second, to simplify the deployment of the test cases. One approach would deploy the precise set of actions to each tester. However, such deployment could be expensive in a very large system since different testers manage different sets of actions. In our approach, all testers receive the same test case, however, they only manage the actions annotated to them. Third, to implement a testing system that supports effective and repeatable automated testing also called as test harness [20].

The test harness requires four steps :

[2]http ://www.junit.org

1. **Set up :** sets the SUT to be tested. This means the runtime environment is set. This step is also called as preamble scenario. For instance, loading databases, files or memory ; establishing the peer set within a DHT.
2. **Execute :** exercises the SUT by inputs and outputs that are stored to be compared afterward. For instance, inserting data into the DHT, then retrieving such data.
3. **Evaluate :** evaluates a test case execution. This step is the oracle implementation. Thus, the test case must compare the actual results with the expected ones and the results can be both displayed and/or logged.
4. **Clean up :** releases all the resources used within testing. The idea is to let the SUT in the same state as before the test case begins. Its purpose is to ensure that there will be no interference among different test cases. This step is also called as postamble scenario. For instance, stopping databases ; closing network connections ; making all the peers leave a DHT.

In our framework, all the testers receive the same test case to simplify the deployment of a distributed test case. However, the testers only execute the actions assigned to them in agreement with the annotations. The available annotations are listed below :

Test : specifies that the method is actually a test case action. This annotation has four attributes that are used to control its execution : the test case name, the place where it should be executed, its order inside the test case and the execution timeout.

Before : specifies that the method is executed before each test case. The purpose of this method is to set up a common context for all test cases. The method plays the role of a preamble scenario.

After : specifies that the method is executed after each test case. The purpose of this method is to clean up the resources used along testing. The method plays the role of a postamble.

Each action is a point of synchronization : at a given moment, only methods with the same signature can be executed on different testers. Actions are not always executed on all testers, since annotations are also used to restrain the testers where an action can be executed. Thus, testers may share the same testing code but do not have the same behavior. The purpose is to separate the testing code from other aspects, such as synchronization or conditional execution. The tester provides two interfaces, for action execution and volatility control :

1. **execute(**a_i**)** : executes a given action.
2. **leave(), fail(), join()** : makes a peer leave, abnormally quit or join the system.

When a tester executes a test case, it proceeds as described in Figure 3.5 :

1. It asks the coordinator for identification, used to filter the actions that it should execute.
2. It uses java reflection to sort out the actions, read their annotations and create a set of method descriptions (i.e., the signature of methods it should manage).
3. It passes to the coordinator the set of method descriptions that it should manage and their priorities.
4. It waits for the coordinator to invoke one of its methods. After the execution, it informs the coordinator that the method was correctly executed.

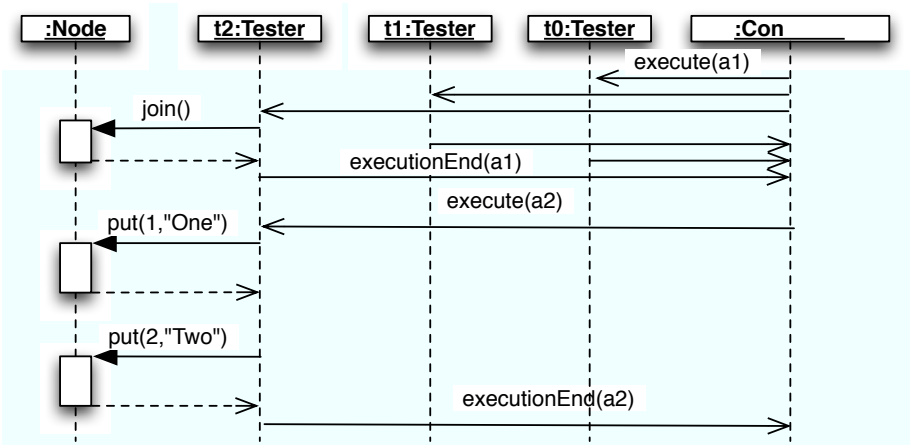


Figure 3.5 – Test case execution

3.3.3 Centralized testing architecture

Our objective when developing the framework was to make the implementation of test cases as simple as possible. The framework is not driven by a specific type of tests (e.g., conformance, functional, etc.) and can be used as a basis for developing different test cases.

A first approach for this framework relies on a centralized test controller architecture to synchronize the execution of the distributed test cases. Basically, this architecture has two main components : the tester and the coordinator. The coordinator is deployed in only one node, while the testers are deployed in one or more nodes depending on the size of the SUT. The deployment diagram of this architecture is illustrated in Figure 3.6.

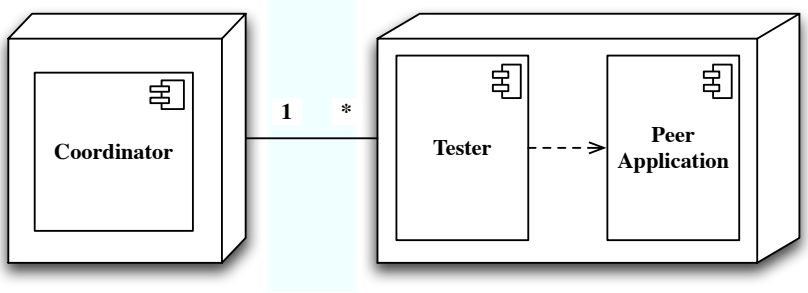


Figure 3.6 – Deployment diagram

The coordinator controls several testers and each tester runs on a different logical node (the same as the peer it controls). The coordinator acts as a *broker* [21] for the deployed testers. Along a test case execution, the role of the tester is to execute test case actions and to control the volatility of a single peer. The role of the coordinator is to dispatch the actions of a test case (A^τ) through the testers (T^τ) and to maintain a list of unavailable peers. In practice, each tester receives the description of the overall test sequence and is thus able to know when to apply a local execution sequence.

3.3.3.1 The coordinator

The coordinator is a central component that controls when each test action should be executed. When the test starts, the coordinator receives a set of method descriptions from each tester and associates each action to a hierarchical level. Then, it iterates over a counter representing the hierarchical level that can be executed, allowing the execution to be synchronized. From the coordinator point of view, a test suite consists of a set of actions A, where each action a_n^A has a hierarchical level $h_{a_n}^A$. Actions with lower levels are executed before actions with higher levels.

The execution of actions follows the idea of two phase commit (2PC). In the first phase, the coordinator informs all the concerned peers that an action a_n can be executed and produces a lock. Once all peers announce the end of their execution, the lock is released and the execution of the next action begins. If an action's timeout is reached, the test case is *inconclusive*.

The coordinator provides three different interfaces, for action execution, volatility and test case variables :

- **register(**t_i, A^t**), ok(**a_n**), fail(**a_n**), error(**a_n**)** : performs the actions registration (performed before all tests) and response for actions execution, called by testers once the execution of an action is finished.
- **set(key,value), get(key)** : provides the accessors for test case variables, that are described on section 3.3.6.
- **leave(**P**), fail(**P**), join(**P**)** : makes a set of peers leave, abnormally quit or join the system.

3.3.3.2 Test case execution

The synchronization algorithm has three steps (see Algorithm 1) : registration, action execution and verdict construction. Before the execution of a τ, each $t \in T$ registers its actions with the *coordinator*. For instance, in the motivating example (see Example 3.1), tester t_2 may register the actions $A' = \{a_1, a_2, a_5\}$.

The registration algorithm works as follows (see Algorithm 2). Initially, the *coordinator* identifies each *tester* with an integer identifier, for instance the first tester receives the identifier 0. The identifier is increased by 1 every time a new tester asks for it. This simple method simplifies the action definition that is made at the user level. The identifier is also used by a node to know whether it is allowed to execute a given action.

Once the registration is finished, the *coordinator* builds the schedule S, mapping the actions with their related subset of testers. In our example, action a^3 is mapped to $\{t_3, t_4\}$. Once S is built, the coordinator traverses all test cases $\tau \in DTS$ and then the actions of each τ. For each action a_i^τ, it uses $S^\tau(a_i^\tau)$ to find the set of testers that are related to it and sends the asynchronous message $execute(a_i) \forall t \in S^\tau(a_i^\tau)$. Then, the coordinator waits for the available testers to inform the end of their execution. The set of available testers corresponds to $S^\tau(a_i^\tau) - T_u$, where T_u is the set of unavailable testers. In our example, once a_1 is finished, testers $\{t_0, t_1, t_2\}$ inform the *coordinator* of the end of the execution.

Thus, the *coordinator* knows that a_1 is completed and the next action can start. When a tester $t \in T^\tau$ receives the message $execute(a_n^\tau)$, it executes the suitable action. Each tester t_i receives the asynchronous message $execute(a)$ and then performs action a as described in algorithm 3. If the execution succeeds, then a message ok is sent to the coordinator. Otherwise, if the action timeout is reached, then the message $error$ is sent.

Once the execution of τ finishes, the coordinator asks all testers for a local verdict. In the example, if t_3 gets the correct strings "One" and "Two" at a_4, then its local verdict is $pass$. Otherwise, it is $fail$.

Algorithm 1: Test suite execution

Input: T, a set of testers ; DTS, a distributed test suite
Output: $Verdict$

```
1  foreach t ∈ T do
2  |  register(t);
3  end
4  foreach τ ∈ DTS do
5  |  foreach a ∈ A^τ do
6  |  |  foreach t ∈ S^τ(a) do
7  |  |  |  send execute(a) to t;
8  |  |  end
9  |  |  wait for an answer from all t ∈ (S^τ(a) − T_u) ;
10 |  end
11 |  foreach t ∈ T^τ do
12 |  |  L^τ ← L^τ ∪ {l_t^τ} ;
13 |  end
14 |  return oracle(L^τ, φ) ;
15 end
```

Algorithm 2: Registration

Input: p, a node
Output: id

```
1  foreach a ∈ A^p do
2  |  S^τ(a) ← S^τ(a) ∪ p ;
3  end
4  id++ ;
5  return id ;
```

Algorithm 3: Action execution

Input: a, an action to be executed

```
1  invoke(a) ;
2  if ι^a is reached then
3  |  send error to Coordinator ;
4  else
5  |  send ok to Coordinator ;
6  end
```

After receiving all local verdicts, the coordinator is able to assign a verdict L^τ. If any local verdict is $fail$, then L^τ is also $fail$, otherwise the coordinator continues grouping each l_t^τ into L^τ. When L^τ is completed, it is analyzed to decide between verdicts $pass$ and $inconclusive$ as described in Algorithm 4. This algorithm has two inputs, a set of local verdicts (L) and an index of relaxation (φ), representing the level of acceptable $inconclusive$ verdicts (detailed in section 3.3.3.4). If the ratio between the number of $pass$ and the number of local verdicts is greater than φ, then the verdict is $pass$. Otherwise, the verdict is $inconclusive$.

Algorithm 4: Oracle

Input: L, a set of local verdicts ; φ an index of relaxation

```
1 if ∃l ∈ L, l = fail then
2 |   return fail
3 else if |{l ∈ L : l = pass}|/|L|) ≥ φ then
4 |   return pass
5 else
6 |   return inconclusive
7 end
```

3.3.3.3 Dealing with node volatility

The volatility of nodes can make testing difficult during the execution of a test case. If the coordinator is not informed that a node has left the system, then it is unable to follow the test sequence and is unable to proceed. We must then be able to control node volatility to forecast the next action that must be performed. Our algorithm treats the volatility of nodes as common actions, where the tester informs the coordinator that a node has joined/left the system.

Since the coordinator is informed by the testers of node departures or fails, it is able to update its schedule and does not wait for confirmation from these nodes. Therefore the next action is set for execution and the synchronization sequence continues.

3.3.3.4 Setting a global verdict

To set a global verdict, the algorithm should also take into account the autonomy of nodes, since it directly influences the result completeness of queries. Result completeness guarantees that a result set is complete with respect to the set of objects in the master source. For instance, in a distributed database system, query results are complete since all nodes are expected to answer.

In a P2P system, when a node queries the system a partial result set may satisfy the request. As some nodes may not answer (e.g., due to timeout constraints), there is no guarantee of completeness. During the verification of a P2P system, the lack of result completeness may engender $inconclusive$ verdicts, since the oracle can not state if the test case succeeded or not. This lack of completeness should not be interpreted as an error, it belongs to the normal behavior of P2P systems.

Thus, we introduce an index for completeness relaxation, which was showed in algorithm 4 as φ. This index is used to take into consideration $inconclusive$ verdicts engendered by lack of response to some node request. It represents the percentage of desirable $pass$ verdicts in V^τ. Such index is chosen by the testing designer, since it represents the human knowledge about the SUT completeness.

3.3.4 B-Tree distributed testing architecture

In this section, we present an alternative to the centralized test controller architecture. This architecture consists of organizing testers in a B-Tree manner, similarly to the overlay network used by GFS-Btree [68]. Briefly, a B-Tree stores a set of pairs (k, v) where at most one value v is associated with each key k.

In our B-Tree implementation we use the testers' ID as keys. Then, these IDs become communication addresses arranged in a B-Tree manner. In fact, we take advantage of the B-Tree structure for two reasons. First, drop the centralized coordinator to scale up the testing architecture. Second, optimize the exchange of messages since any message from the root to a leaf takes $log(n)$ hops where n is the number of elements in the tree.

The order of the B-Tree is not fixed, it may vary according to the number of testers, which is known at the beginning of the execution. The goal is to have a well-proportioned tree, where the depth is equivalent to its order.

Furthermore, the responsibility to start the execution of test cases and assign verdicts is left to the root. When managing an execution, the root dispatches actions to its children testers, which dispatch them to their children consecutively way down to the leaves. Once an action is received and executed, the leaves start sending the results back to their parents way up to the root. Whenever the root receives back the results, it dispatches the next action. Consecutively, all actions are executed up to the last one when the root assigns a verdict.

3.3.4.1 Preliminaries

The B-tree architecture has a main component, the tester. The role of the tester is to execute test case actions and control the volatility of a single peer. Moreover, it coordinates the dispatch of the test case actions (A^{τ}) through distributed testers (T^{τ}) arranged in a B-Tree manner. The UML diagram presented in Figure 3.7 illustrates the deployment of the framework : each tester runs on a logical node (the same as the peer it controls). Yet, a node may have several testers depending on the order of the tree.

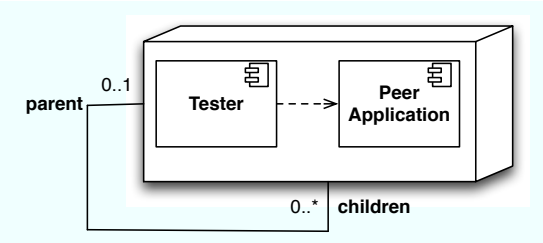


Figure 3.7 – Deployment diagram

Figure 3.8 illustrates a b-tree of order 1. Whether using the interfaces, the **root()** operation returns the ID of tester 3. The **children(**t_1**)** operation returns the IDs 0 and 2. The **parent(**t_1**)** returns the ID of tester 3.

In the B-tree architecture, the tester provides two additional interfaces compared with the previous centralized architecture. These additional interfaces are for action execution management and test case variables :

- **register(**t, A^t**), ok(**a_i**), fail(**a_i**), error(**a_i**)** : performs action registration (before all tests), action execution and responses to execution.
- **set(**$key, variable$**), get(**key**) :**$variable$: provides accessors for test case variables.

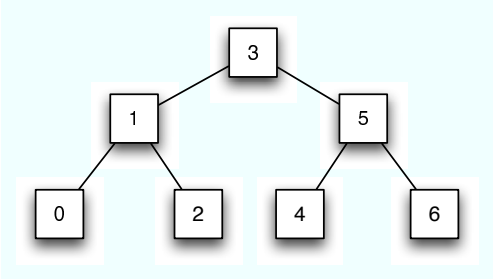


Figure 3.8 – B-Tree example

We assume the B-Tree provide us two different operations, for node insertion and structure information :

- **insert(**$tree, t_i$**)** : inserts a tester into the tree using the tester ID as key.
- **children(**t_i**), parent(**t_i**), root(**$tree$**)** : returns the children or the parent of a given tester, and returns the root of the tree.

3.3.4.2 Test case execution algorithms

In detail, a test case execution works as follows. Initially, a tester is randomly chosen as the bootstrap tester, or bootstrapper for short, and receives the ID 0. The role of the bootstrapper is to provide IDs to all the incoming testers and build a B-Tree.

Algorithm 5 describes the bootstrapper creating the B-tree and waiting for subscriptions from all the other testers. At each subscription, the B-Tree is populated with the tester's ID (lines 3-5). Once the B-Tree is built, the bootstrapper informs the root that the structure is ready to execute (line 6).

The execution of actions is managed by a dispatch function which is also responsible to send them to a set of testers (see Algorithm 6). To start, this function receives a tester and an action a_i. Then, it starts sending dispatch messages to its children (lines 1). If the tester is supposed to, it also executes a_i (lines 2-4). For each sent message, an answer must be returned (line 5). This ensures that the testers are always available and the communication goes along. It has to be noticed that both tasks of sending and receiving messages are asynchronous.

In fact, there are two different types of testers : the root and the "regular" testers. The role of the root is to start the test sequence way down the B-Tree and to assign a global verdict. The role of a regular tester is simply execute actions and route messages way up and down the B-Tree. Both testers are presented below.

Algorithm 7 describes the root tester. Briefly, it describes the actions dispatching and also the grouping of the local verdicts that will be used to assign the final verdict.

Initially, the root waits for a message from the bootstrapper (line 1). Once it receives, it decomposes a given test case in actions to start the test case execution (lines 2-6). In fact, the execution of actions is allowed through dispatch messages sent to children. However, sometimes not all the children must receive

Algorithm 5: Bootstrap tester

Input: A, a set of actions ; T, a set of testers

$tree \leftarrow$ **create** BTree;
receive $register(t, A^t) \forall t \in T$;
foreach $t \in T$ **do**
    $insert(tree, t)$;
end
send $start$ to $root(tree)$;

Algorithm 6: Dispatch function

Input: $this$, a tester ; a_i, an action to be managed

send $dispatch(t, a_i) \forall t \in children(this)$;
if $\exists a_i \in actions(this)$ **then**
    $execute(a_i)$;
end
receive $ok \forall t \in children(this)$;
send ok to $parent(this)$;

Algorithm 7: Root tester

Input: τ, a distributed test case

receive $start$ from t_0 ;
foreach $a_i \in A^\tau$ **do**
    $T' \leftarrow receivers(a)$;
    send $dispatch(a_i, t) \forall t \in T'$;
    receive $ok \forall t \in T'$;
end
send $result \forall t \in T^\tau$;
$L \leftarrow$ **receive** $verdict \forall t \in T^\tau$;
return $oracle(L, \varphi^\tau)$;

a given message. For instance, only one child in the left side of the tree. In order to avoid any message wasting, we use an optimization function (receivers) to know exactly which children must receive it (line 3). The optimization function works like the binary search, regardless it returns only the correct child (or children) instead of the entire branch. Then, the correct children are informed to the dispatch function (line 4).

Once all the actions are executed, the root asks testers for a result and waits for all verdicts. Then, the root tester groups them to compute the final verdict within the oracle function (lines 7-8). The oracle function has two inputs, the set of local verdicts (L) and an index of relaxation (φ^{τ}). If the ratio between the number of $pass$ and the number of local verdicts is greater than φ^{τ}, then the verdict is $pass$. Otherwise, the verdict is $inconclusive$.

Algorithm 8 describes how the verdict is computed. Once the execution of actions has finished, the root asks its children for their results. This request is sent the way down to the leaves (line 2). Once a tester receives the set of verdicts from its children, it adds its own verdict to this set and send it to its parent. Consecutively, all the messages are sent from children to parents the way up to the root.

Algorithm 8: Result

Input: $this$, a tester ;
1 **send** $result \forall t \in children(this)$;
2 $L' \leftarrow$ **receive** $verdict \forall t \in children(this)$;
3 $L' \cup verdict$;
4 **return** L';

3.3.4.3 The optimization function algorithm

In the B-Tree structure, when one wishes to send a message, starting at the root, the tree is traversed top to bottom, choosing the child pointer whose separation IDs are on either side of the ID that will receive the message. For instance, a test case is written to be executed by seven testers where an action must be executed specifically by testers t_1 and t_3.

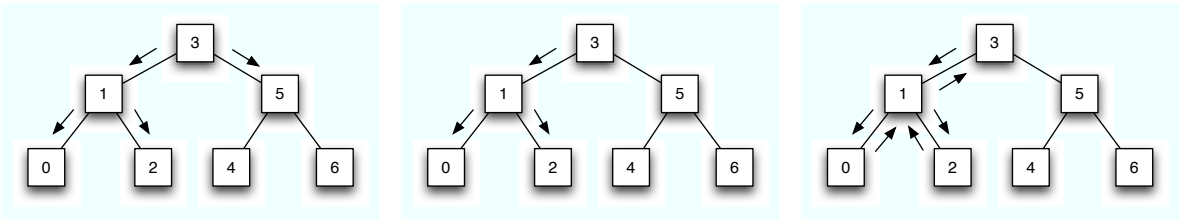
Figure 3.9(a) illustrates an execute message being sent to all the root's children, however, one message is wasted to t_5. Figure 3.9(b) illustrates the binary search where the proper child is chosen and the search goes the way down to the leaf. Yet, the binary search will be executed to every tester since its ID is the key to be sought. For instance, to seek for IDs 0 and 2 the binary search performs twice and returns the paths "3-1-0" and "3-1-2".

Thus, in our optimization function we perform in the same way of the binary search choosing the child pointer that fits the best. However, we are interested only in the correct child rather than the entire path followed by the binary search.

In fact, we aim to send an action only once verifying the best child for it, then the child does the same avoiding any message wasting. Algorithm 9 describes the optimization function. Briefly, this function receives the current action and returns the children that must receive the execute message.

Recalling that an action is a tuple $a_i^{\tau} = (\Psi, \iota, T')$ as described in section 3.1, it is possible to know exactly which testers T' will execute it (line 1). However, we want to know only the children of the actual tester.

Then, we first verify if each tester $t \in T'$ is a child of the actual tester. If it is a child, then the subset of testers T'' will keep it (lines 4-5). If it is not a child, we verify in which side of the tree t is. If it is in



(a) Message wasting (b) Basic B-Tree search (c) Optimal message exchange

Figure 3.9 – B-Tree upon a message exchange

the left side, then T'' simply keeps the left child's ID (lines 6-7) otherwise T'' keeps the right child's ID (line 9). Finally, the correct children are returned within the subset T'' (line 12).

Algorithm 9: Receivers optimization function

Input: a_i, an action
Output: T'', a set of testers

```
1  T' ← testers(a_i) ;
2  T'' ← ∅ ;
3  foreach t ∈ T' do
4      if t ∈ children(this) then
5          T'' ∪ {t} ;
6      else if t.ID < this.ID then
7          T'' ∪ {left(children(this))} ;
8      else
9          T'' ∪ {right(children(this))} ;
10     end
11 end
12 return T'' ;
```

3.3.4.4 B-Tree example

Let us illustrate the execution of a distributed test case. This test case improves the motivating example to better explain this architecture. Like the previous motivating example 3.1, the objective is to detect errors on a DHT implementation.

This test case involves seven testers $T^\tau = \{t_0 \dots t_6\}$ to control seven peers $P = \{p_0 \dots p_6\}$ and six actions $A^\tau = \{a_1, ..., a_6\}$. According to Example 3.2, the first action may be executed by testers t_0, t_1 and t_2. This means that the root (see Figure 3.8), now t_3, is responsible to dispatch the action a_1 to its child t_1. Then, t_1 dispatch a_1 to its children t_0 and t_2, and execute it.

After the execution of a_1, t_1 receives the finalization message from t_0 and t_2, and informs its parent the finalization as well. Then, t_3 dispatches the next action a_2 to t_1 to be executed by t_2. This action is executed and sent back way up to the root. The next action a_3 has a call to the volatility interface. This makes the testers aware that their peers will join the system. Thus, the test sequence can run without volatility interference.

In the end, t_3 receives all the local verdicts to compute the final verdict. If the data retrieved in a_4 is

the same as the one inserted in a_2, then the verdict is $pass$. If the data is not the same, the verdict is $fail$. If the testers were not able to retrieve any data, then the verdict is $inconclusive$.

Example 3.2 (Simple test case).

Action	Testers	Instructions
(a_1)	0,1,2	join()
(a_2)	2	put(14, "fourteen") ;
(a_3)	3,4,5,6	join() ;
(a_4)	3,4,5,6	data := retrieve(14) ;
(a_5)	3,4,5,6	assert(data = "fourteen") ;
(a_6)	*	leave() ;

3.3.5 Gossiping distributed testing architecture

The Gossiping is another solution to synchronize the execution of actions in a distributed manner. In Gossiping we use the same architecture used by the B-Tree approach with a tester per node, however, the synchronization of actions is executed by gossiping coordination messages through testers.

The Gossiping approach has the following steps. First, any node p in the system P is designated to execute the first tester t_0. This tester will act as an identifier to all the other testers $t_n \in T$ that join the system. The identification follows an incremental sequence from 0 up to n and is used to select the actions a node should execute. Second, t_0 creates a multicast address for each test case action. Third, the decomposed test case is deployed through P and stored at each tester. Then, each tester verifies which actions it should execute and subscribes to the suitable multicast addresses. Finally, the testers responsible for the first action start the execution.

A tester can play two different roles during the test case execution :

- *Busy tester*. This tester executes an action a_i and gossips its completion to the multicast address of the next action a_{i+1}. Once it has sent a gossip, it becomes an *Idle tester*.
- *Idle tester*. This tester remains idle waiting the gossips from all the *Busy testers*. Once it receives all their gossips, then it becomes a *Busy tester*.

The gossiping between these two types of testers guarantees the execution sequence of the whole test case.

3.3.5.1 Interfaces

Indeed, the gossiping architecture deploys testers like the b-tree architecture and also uses a bootstrapper to identify testers and compute verdicts. This bootstrapper provides the same two additional interfaces of the B-Tree bootstrap tester, for action execution and test case variables.

The “regular” testers also provide the same interfaces used by the B-Tree “regular” testers to execute actions, control the volatility of peers, and access test case variables. However, an interface to communicate with the multicast structure is provided. The operations of this interface are used to subscribe to a multicast address and both send and receive messages from it. The operations are :

- **send(**$address$**,**$message$**), receive(**$message$**), subscribe(**$address$**)** : provides communication with the multicast addresses.

3.3.5.2 The algorithm

The algorithm works as follows. Initially, a bootstrap tester, or bootstrapper for short, is randomly chosen. It will be responsible to identify the other testers and create the gossiping structure. This structure holds a multicast address to each test case action $a \in A^\tau$. Algorithm 10 details the creation of these addresses by the bootstrapper before the execution starts (lines 1-3).

Algorithm 10: Bootstrap tester

```
   Input: A^τ, a set of actions
1  foreach a ∈ A^τ do
2  |  create(multicast(a)) ;
3  end
4  foreach t ∈ T^τ do
5  |  register(t);
6  end
7  send(multicast(a_0),ok) ;
8  wait for l_t from all t ∈ T^τ ;
9  foreach t ∈ T^τ do
10 |  L^τ ← L^τ ∪ {l_t^τ} ;
11 end
12 return oracle(L^τ, φ) ;
```

Then, the bootstrapper registers each new tester returning an ID to them (lines 4-6). Once the registration phase is done, the bootstrapper send a ok message to the first multicast address in order to start the test case execution (line 7).

Algorithm 11 details the non-bootstrapper testers. Initially, the testers subscribe themselves to the multicast addresses of the actions they are set to execute (lines 1-3). Once the subscriptions are finished, the registration phase is terminated and the testers wait for the start message (line 4). This guarantees the communication among testers is ready to execute the test case.

Once the fist action is received, the testers subscribed to it (e.g., T'_{a_0}) receive the ok message and start its execution (line 5). As soon as the execution is terminated, the testers of this first action (e.g., T'_{a_0}) send a message to the multicast address of the next one (line 6). This happens consecutively up to the last action a_n.

Once all the actions are executed, the testers send their local verdicts to the bootstrapper (line 8). Then, the bootstrapper gather all the local verdicts in the set L^τ (see Algorithm 10 lines 8-12). In fact, the final verdict is computed in the same way described at the centralized controller architecture and the B-Tree architecture.

Algorithm 11: Tester

Input: A', a subset of actions
1 **foreach** $a \in A'$ **do**
2 | subscribe(multicast(a)) ;
3 **end**
4 **foreach** *receive(ok)* **do**
5 | execute(a_i);
6 | send(multicast(a_{i+1}), ok) ;
7 **end**
8 send l_t^T to t_0 ;

3.3.5.3 Gossiping example

To illustrate this approach we use the original motivating example (see Example 3.1). Initially any node is chosen to be tester t_0. In fact, this node decomposes the actions and creates the multicast addresses (see Figure 3.10(a)). Then, the other nodes contact t_0 to receive an identifier n and subscribe to the suitable multicast addresses. For instance, if a tester receives $n = 1$, it subscribes to the addresses of a_1 and a_5.

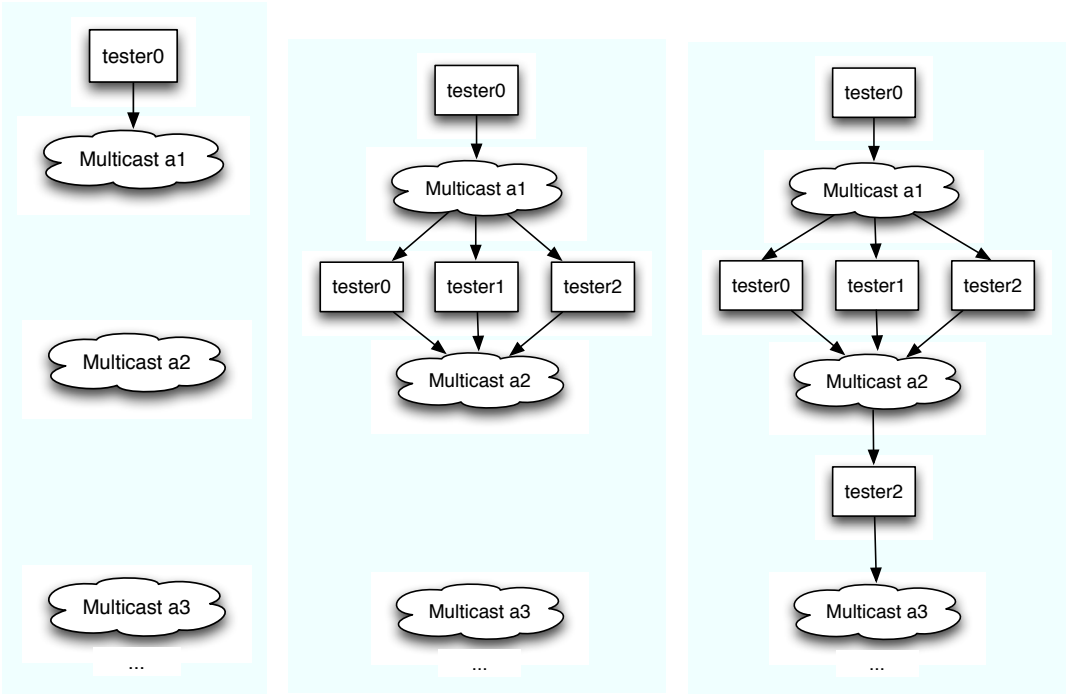
Figure 3.10(b) presents the first action a_1 being executed by the testers t_0, t_1 and t_2. Once the execution of a_1 is finished, they gossip the completion to the multicast address of the next action a_2. Then, the testers subscribed to a_2, that is only t_2, receive the multicast message and execute a_2 gossiping its completion in the end (see Figure 3.10(c)). Consecutively, execution, then gossiping happen up to the last action a_5. Finally, each tester computes a local verdict and sends it to t_0, which assigns a verdict of the entire test case.

3.3.6 Test case variables

We call *Test Case Variables* a structure that keeps values used along testing and cannot be predicted when a test case is written. This means that values generated on-the-fly can be stored and used at runtime. To access this structure, the framework provides an interface to insert and retrieve variables as described at each architecture interface section.

The structure is kept by the central coordinator when using the centralized architecture, or by the bootstrapper when using the distributed architecture. Several testing scenarios can take advantage of this structure. For instance, testers must analyze the routing table of their peers to verify if it was correctly updated. More precisely, testers must compare the peer IDs from a routing table with the ID of peers that leave or join the system. This comparison is not trivial, because each tester only knows the ID of its peer, which is dynamically assigned. For instance, a given tester may be aware that the peers controlled by testers t_0 and t_1 left the system, but does not know that they correspond to peers $20BD8AB6$ and $1780BB16$. To simplify the analysis of routing tables, we use test case variables to map tester IDs to peer IDs, as shown in Table 3.1.

The variables are stored only at runtime and must be purged by the end of each test case execution. This avoids interference with further executions and happens within the postamble scenario.



(a) Creating multicast addresses (b) Executing the first action (c) Executing the second action

Figure 3.10 – Gossiping architecture

Tester ID	Peer ID
0	20 BD 8A B6
1	17 80 BB 16
2	A0 36 02 F7
3	40 E4 DE A2
...	...
15	FA 09 EE 90
16	21 3A C2 58

Table 3.1 – ID mapping

3.4 Conclusion

Let us summarize the contributions presented in this section.

When testing P2P systems, we claim that three dimensional aspects must be combined to check for correctness. These aspects are : volatility, scalability and functionality.

A peer, from a testing point of view, can have to leave or join the system at a given time in a test sequence. This action is specified in the test sequence in a deterministic way. Since it has the objective to deal with a large number of peers, the second dimension of P2P system testing is scalability. Then, because it is accomplishing a treatment, the scalability and volatility dimensions have to be tested with the behavioral and functional correctness.

To take into account the three dimensional aspects of P2P systems, we presented a methodology that combines the functional testing of an application with the variations of the other two aspects. Indeed, we incrementally scale up the SUT either simulating or not volatility.

In terms of diagnosis, this methodology allows to determine the nature of the detected erroneous behavior. Indeed, the problem can be linked to a purely functional cause (Step 1), a volatility issue (Step 2), a scalability issue (Step 3) or a combination of these three aspects (Step 4). The most complex errors are the last ones since their analysis is related to a combination of the three aspects. Steps 2 and 4 could also be preceded by two other steps (shrinkage and expansion), to help the diagnosis of errors due to either the unavailability of resources or arrival of new ones. Yet, several rates of volatility can be explored to verify how they affect the functionality aspect of the SUT (e.g., 10% joining, 20% leaving).

We also presented three different architectures to coordinate the test sequence. These architectures are based on two original aspects : (i) the individual control of peers' volatility and (ii) a distributed testing architecture to cope with large numbers of peers. The capabilities of these architectures are (1) to automate the execution of each local-to-a-peer test case, (2) to build automatically the global verdict, (3) to allow the explicit control of each peer volatility.

The first architecture extends the classical centralized testing architecture (i.e., central coordinator managing distributed testers) with volatility control to demonstrate that such volatility is a key-parameter when testing a P2P system [34]. Basically, this architecture has two main components : the tester and the coordinator. The tester is composed of the test suites that are deployed on several logical nodes. The coordinator is deployed in only one node and is used to synchronize the execution of test cases. It acts as a *broker* [21] for the deployed testers.

However, the performance of such centralized architecture is linear while testing with large numbers of peers requires logarithmic. Therefore, we presented two fully distributed architectures to cope with large-scale P2P systems [36]. The second architecture organizes the testers in a balanced tree [17] (B-Tree) manner where the synchronization is performed from the root to the leaves. Then, the testers communicate with each other across the B-tree to avoid using a centralized coordination. The third architecture uses gossiping messages among testers reducing communications among the testers responsible to execute consecutive test case actions.

The distributed architectures are composed of a distributed component, the tester. The tester is the application that executes in the same logical node as peers, and controls their execution and their volatility, making them leave and join the system at any time, according to the needs of a test. Thus, the volatility of peers can be controlled at a very precise level.

CHAPTER 4
PeerUnit: a Testing Tool for Peer-to-peer Systems

In this chapter, we present the PeerUnit tool which implements the P2P testing framework presented in the previous section. PeerUnit provides a generic interface to control tests using any of the three testing architectures. As mentioned, PeerUnit was developed in Java (version 1.5) and make extensive use of two Java features: dynamic reflection and annotations.

This chapter is organized as follows. Section 4.1 describe the interfaces and the interactions of the main components of PeerUnit. Section 4.2 details the Coordinator component, its classes and behavior. Section 4.3 details the Tester component. Section 4.4 presents the testing elements provided by PeerUnit, such as annotations and assertions. It also presents how to write a test case using PeerUnit. This test case is written based on the motivating Example 3.1.

4.1 Main Components

In this section, we describe the two main components of PeerUnit. We present the interfaces of these components: the Coordinator that is used whenever testing with the centralized architecture and the Tester. We also detail the interactions between them: registration of test case actions and execution.

Figure 4.1 illustrates the main components of PeerUnit. As mentioned, the *Coordinator* is responsible to deploy the test cases across distributed testers and synchronize the execution of these test cases. To communicate with each tester, every tester address is stored on a list which is read as the execution goes on. This list is populated when the testers register their actions and get their identifiers. This list also keeps the actions which are mapped to the testers responsible to execute them. The communication is done by Java Remote Method Invocation (Java RMI), in which the methods of remote Java objects can be invoked from other Java virtual machines, possibly on different hosts.

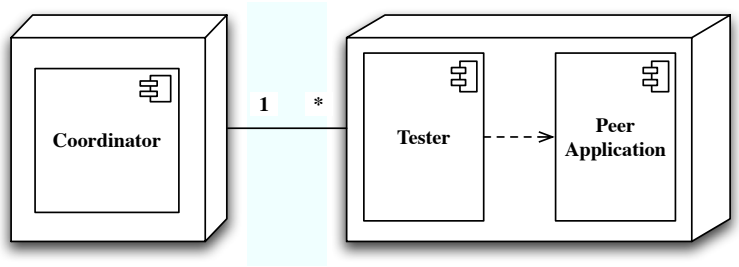


Figure 4.1 – Main components

Figure 4.2 illustrates the main interfaces and data types of PeerUnit. The most important operations provided by the *Coordinator* interface are:

- **register()**: performs the actions registration (performed before all tests). An action is an instance of the *MethodDescription* data type.
- **put(), get()**: provides the accessors for test case variables.
- **quit()**: accessed by a tester either to inform a peer it controls abnormally quit the system or to finish a test case execution. In both cases, a local verdict is informed.
- **getNewId()**: accessed by Testers to get their identifiers.

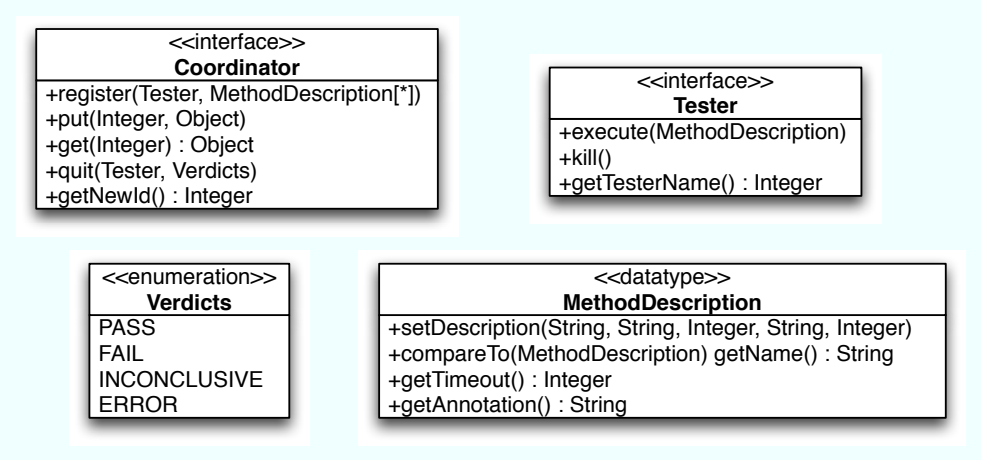


Figure 4.2 – Main interfaces

The Tester component was implemented to be independent of the architecture. This allows us the execution of tests without changing the testing code. Obviously, the difference between architectures happens in the communication between components. Although the communication at the centralized architecture has a central coordinator, all the architectures implement the same operations.

The most important operations provided by the *Tester* interface are:

- **execute()**: executes an action. This happens when the Coordinator allows the execution of the next action and the tester is set to execute it. Whenever testing in a distributed architecture, the execution is allowed by another tester (e.g., parent tester in the B-Tree architecture).
- **getTesterName()**: returns the tester's identifier.
- **kill()**: is accessed to make a peer quit the P2P system.

Since the communication happens by Java RMI, these operation are invoked remotely. This means either the Coordinator or a parent tester synchronizes the execution of test cases by calling the **execute()** operation.

Figure 4.3 illustrates the registration at the centralized architecture. Two testers register their actions at the coordinator receiving their identifiers as soon as the coordinator computes the identifiers and stores the actions in the list of actions.

The execution of test cases was previously presented in Figure 3.5.

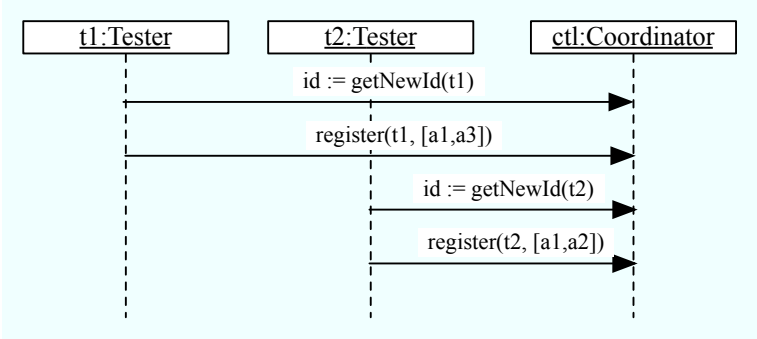


Figure 4.3 – Registration sequence

4.2 Coordinator

In this section, we describe the Coordinator component. First, we present the main classes and their operations. Then, we present the various behaviors of each class.

4.2.1 Main Classes

Figure 4.4 illustrates the Coordinator component in detail. As mentioned, the Coordinator returns an identifier to each tester at the registration. An identifier is an integer that is incremented at each registration, however, a P2P testing may have a large number of testers. Therefore, there is possibly high-degree of concurrency to access the **register(Tester, MethodDescription[*])** operation remotely and get an identifier. To manage this degree of concurrency, we use the *java.util.concurrent* package. This package provides utility classes commonly useful in concurrent programming[1].

Along testing, the Coordinator is capable to measure the test case execution which is done by calling the *Chronometer* class. Actually, the *Chronometer* is called depending on the setup of the “measure” annotation (see Section 4.4.1).

The Coordinator also stores the local verdicts sent by the testers along the execution. As soon as the Coordinator receives all the local verdicts, the Oracle package computes a final verdict to a test case. The Oracle package is detailed later in this chapter.

4.2.2 Behavior

Figure 4.5 illustrates the states of the Coordinator during the test case execution. Initially, the Coordinator waits for the registration ending. This happens whenever all the testers register their actions and get an identifier. Then, it reads the list of actions and submits to testers each action at a time. Once all the actions are submitted, the list of actions is cleaned and the final verdict can be assigned.

Basically, the Coordinator component has two main classes: the *CoordinatorImpl* that implements the interface and the *Chronometer* that measures the execution of test cases. The operations provided by the *CoordinatorImpl* class are:

- **run()**: starts the coordinator.

[1] http: //java.sun.com/j2se/1.5.0/docs/guide/concurrency/

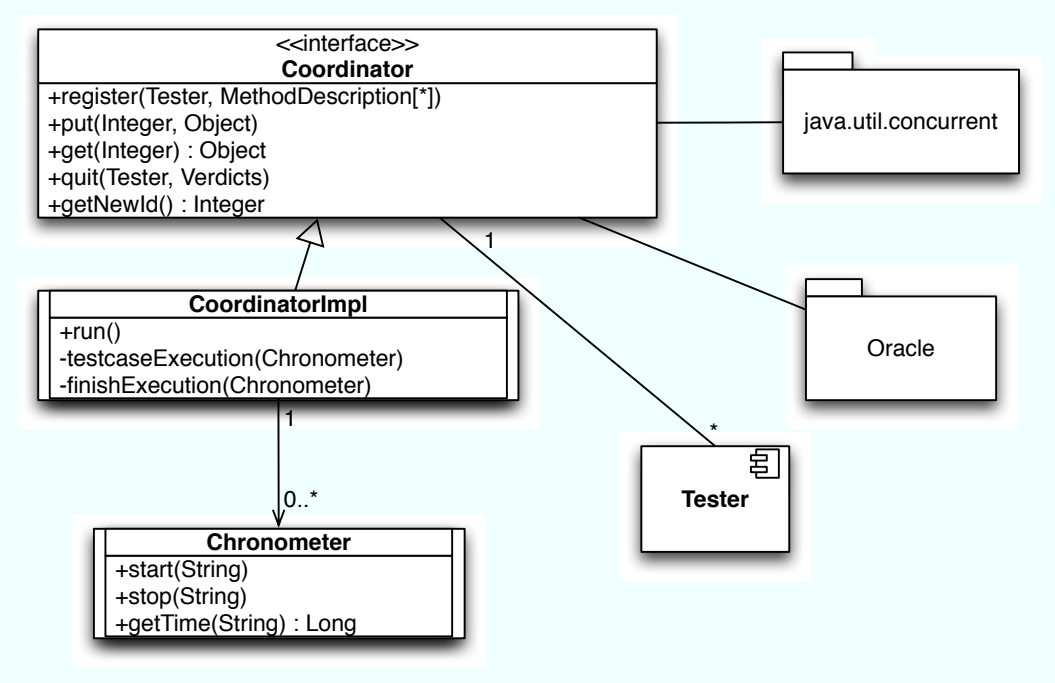


Figure 4.4 – Coordinator component classes

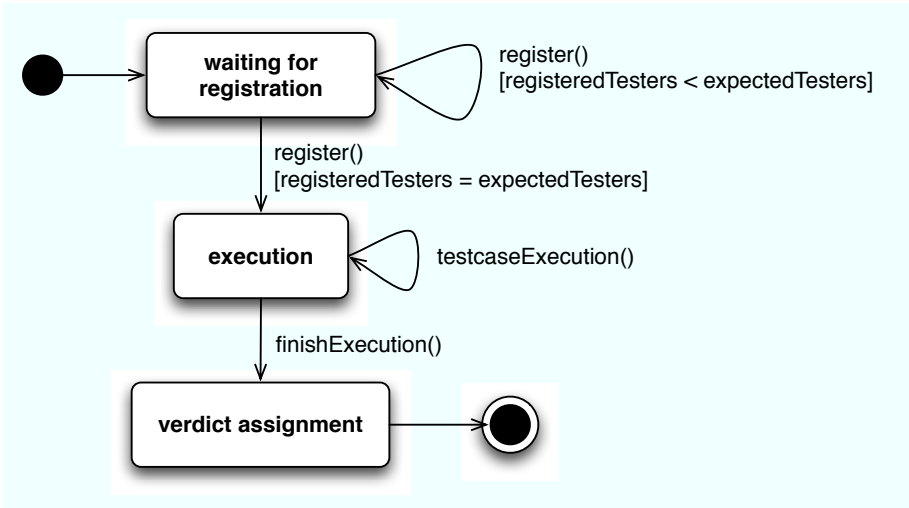


Figure 4.5 – Coordinator state chart

- **testcaseExecution()**: submit the actions to testers. It receives as parameter an instance of Chronometer to measure the execution of each action.
- **finishExecution()**: cleans the list of actions, closes all the connections and stops the coordinator. It receives as parameter an instance of Chronometer to measure the execution of the entire test case.

The operations provided by the *Chronometer* class are:
- **start()**: starts the measurement of a given action.
- **stop()**: stops the measurement.
- **getTime()**: get the time in milliseconds.

4.3 Tester

In this section, we describe the Tester component. First, we present the main packages and their classes. Then, we present the behavior of each class.

4.3.1 Main Classes

We divided the main classes in three packages: (i) the main package that implements the core activities of the testers, (ii) the assertions that implements the oracle and (iii) the exceptions.

4.3.1.1 Main Package

Figure 4.6 illustrates the Tester component in detail. Initially, a tester is instantiated according to the testing architecture. This can be configurated in the setup file (see Appendix B). For instance, the *TreeTester* instantiates a tester in a B-Tree architecture.

Then, the tester keeps the list of actions that will be executed. This list is generated by the *Parser* which is responsible to read and sort the annotations in order to guide the test case execution (i.e., points of synchronization, conditional execution and the testing code). In the centralized architecture, such list is used at the registration phase and stored at the *Coordinator*. In the distributed architectures, it is used to constraint the test case execution. Finally, the tester reads the list of actions to synchronize with the *Coordinator* or another testers during the execution of test cases.

To access the *Tester* interface, a tester engineer has to inherit the *TestCase* interface when writing the testing code. The *TestCase* interface provides generic operations to manage test cases and take advantage of the features provided by PeerUnit (see Figure 4.7). Then, the *TestRunner* class uses dynamic reflection to read the testing code and send it to be parsed and executed. Furthermore, the *TestCase* interface allows the testing engineer to switch the testing architecture without changing the testing code.

4.3.1.2 Assertion Package

Figure 4.8 illustrates the assertion package. The *GlobalVerdict* class groups the local verdicts assigned by the *Oracle* class. Actually, the *Oracle* class assigns a local verdict based on the assertions provided by the *Assert* class. During testing, the assertions may throw a range of failures. Then, a verdict is computed based on the throwable object. Furthermore, two circumstances lead the *Assert* class to assign *inconclusive* verdicts automatically (as detailed in Section 3.3.3.4): execution timeout and incompleteness of a result set.

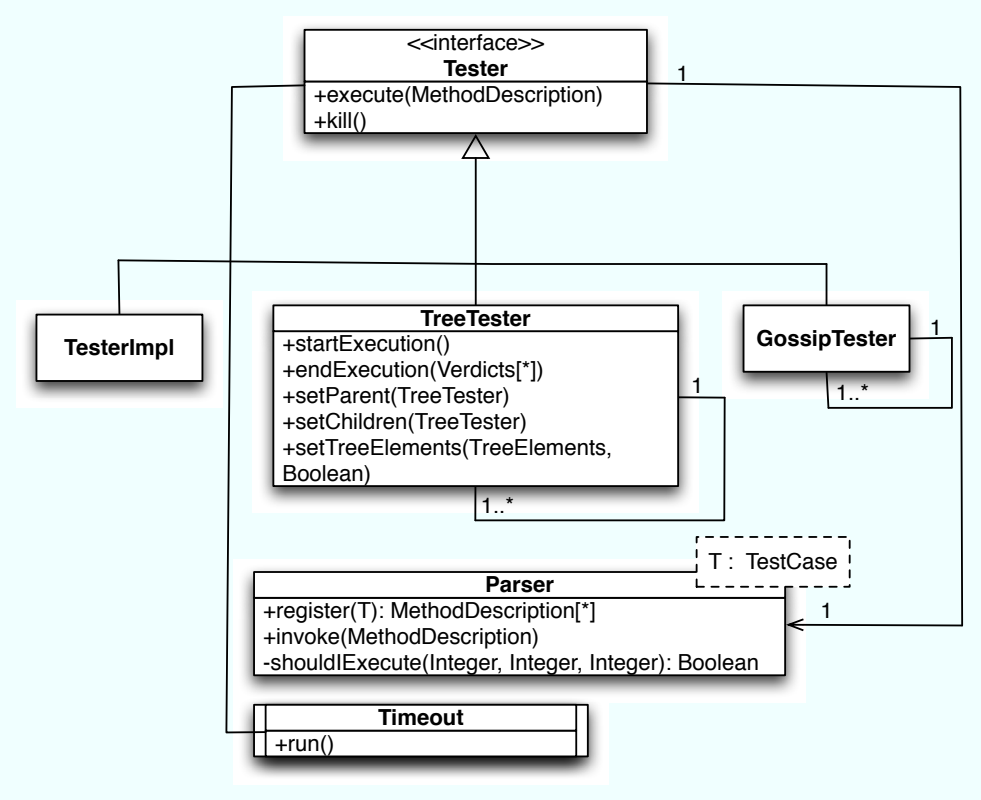


Figure 4.6 – Tester component classes

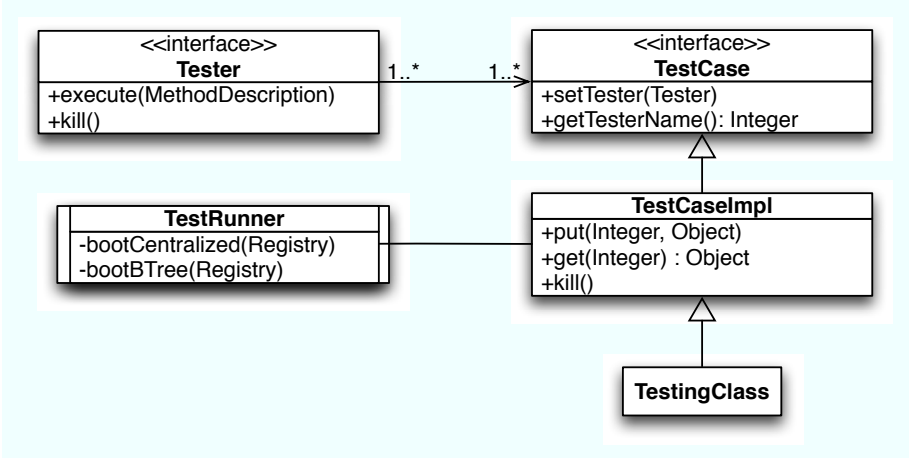


Figure 4.7 – Test Case Execution

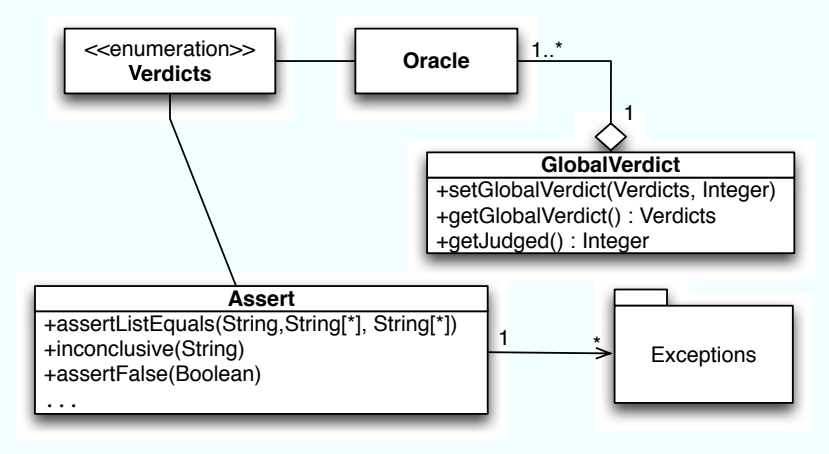


Figure 4.8 – Assertion package

Assertions provide a powerful and straightforward test oracle approach as described in section 2.4.1. Our framework inherits the default assertion methods from the JUnit framework's TestCase class. Some of the most important assertions are:

- **assertTrue(boolean condition)** asserts if a condition is true.
- **assertFalse(boolean condition)** asserts if a condition is false.
- **fail(String message)** fails a test with the given message.
- **assertEquals(String message, Object expected, Object actual)** asserts that two objects are equal.
- **assertArrayEquals(String message, Object[] expecteds, Object[] actuals)** asserts that two object arrays are equal.

Our framework also offers two additional test assertions:

- **inconclusive(String message)** indicates an *inconclusive* verdict. This assertion is also used internally by the framework in case of a test case execution's timeout.
- **assertCollectionEquals(String message,Collection expecteds, Collection actuals)** asserts that two collections are equal.

In the assertions presented above except for **fail(String message)** and **inconclusive(String message)**, if the condition is not met, the assertion throws an error. In this case, the oracle assigns the *fail* verdict.

The code below was extracted from the source code presented in the appendix section B.3. This piece of code verifies if a peer can join the system during the start up. If such peer cannot join the system, then the peer cannot proceed the test case execution. In this case, it is not possible to indicate that the system is buggy. In fact, it only indicates that an unmanageable situation occurs, then an *inconclusive* verdict is assigned. This piece of code also verifies if a collection of objects inserted by one peer was correctly retrieved by the others.

```
public class SimpleTest extends TestCaseImpl{
    ...
    if (! net.joinNetwork(peer, bootaddress, false, log)){
        inconclusive ("I couldn't join, sorry");
    }
    ...
```

```
    @Test(place=0,timeout=1000000, name = "tc1", step = 3)
    public void put(){
     ...
        List<PastContent> expecteds= new ArrayList<PastContent>();
        for (PastContent content : peer.getInsertedContent()) {
            log.info("Expected so far : "+content.toString());
            expecteds.add(content);
        }
     ...
    @Test(place=-1,timeout=1000000, name = "tc1", step = 4)
    public void get(){
     ...
        List<PastContent> actuals = new ArrayList<PastContent>();
        for (PastContent actual : peer.getResultSet()) {
            actuals.add(actual);
        }
        Assert.assertCollectionEquals("[Local verdict] ",expecteds, actuals);
     ...
}
```

4.3.1.3 Exceptions Package

An exception is an event that occurs during the execution of a program that disrupts the normal flow of instructions[2]. The Exception package and its classes are a form of *java.lang.Throwable* that indicate conditions that a reasonable application might want to catch.

We provide some new exceptions in order to catch disruptions from P2P features (see Figure 4.9), such as: actions timeout and incompleteness. As mentioned, these features cannot be considered as failures, thus, we provide the *InconclusiveFailure* exception that indicates an *inconclusive* verdict to the oracle.

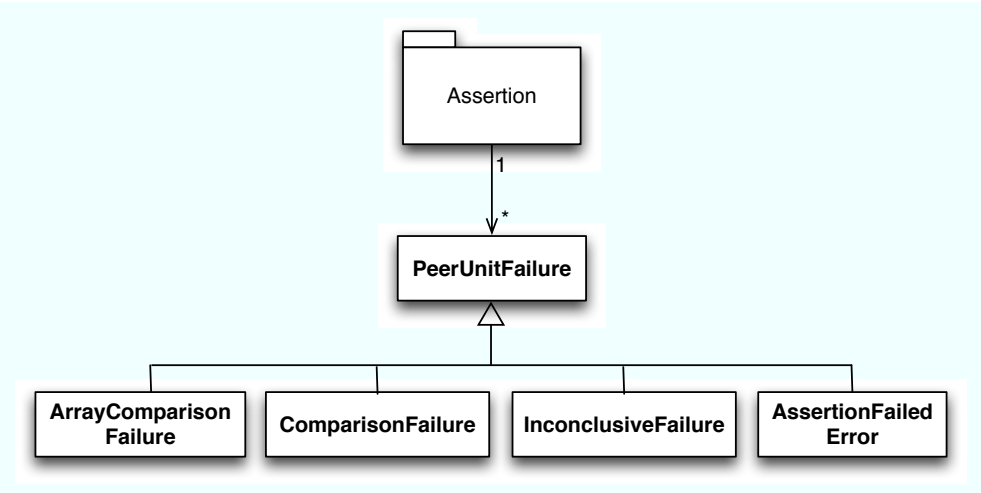


Figure 4.9 – Exceptions package

[2]http: //java.sun.com/docs/books/tutorial/essential/exceptions/

4.3.2 Behavior

Figure 4.10 illustrates the states of a tester during the test case execution. Initially, the testing code is parsed to allow the registration of the actions. Once the actions are registered, the tester waits to start their execution. When all actions are completely executed, a local verdict is assigned and sent to the coordinator.

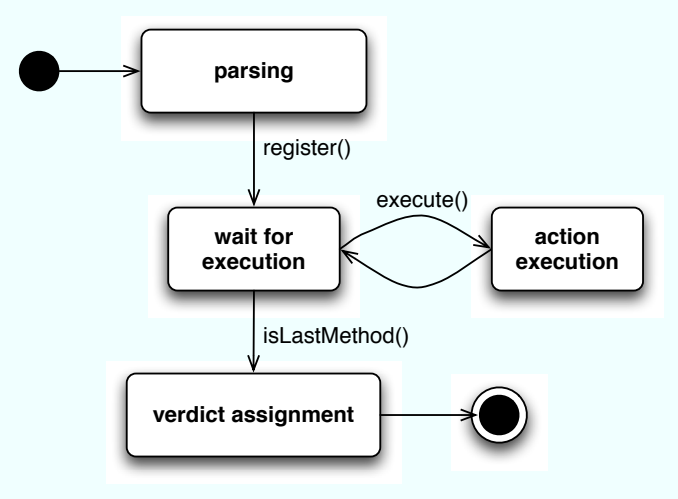


Figure 4.10 – Tester state chart

The behavior of the Tester component can be explained basically through the following classes and interfaces. Starting from the *TreeTester* class that is the implementation of the Tester interface for the B-Tree architecture, such implementation provides operations to manage the execution of test cases and manage the tree structure as well. The operations provided by the *TreeTester* class are:

- **startExecution()**: starts the execution of the tester.
- **endExecution()**: ends the execution of the tester receiving as a parameter the verdict list that is passed by the children testers. Then, the tester eventually adds its local verdict to the list since it goes up towards the root tester.
- **setParent()**: keeps informed the tester about its parent in the tree.
- **setChildren()**: keeps informed the tester about its children in the tree.

The *TestCase* interface provides the operations to access the *Tester* interface and its implementations (e.g., *TreeTester*). As mentioned, this provides us architecture independency. The operations provided by the *TestCase* interface are:

- **setTester()**: sets an instance of *Tester*.
- **getTesterName()**: is accessed to acknowledge the tester's identifier.
- **kill()**: is accessed to make a peer quit the P2P system.
- **put(), get()**: provides the accessors for test cases.

As mentioned the *Parser* class is responsible to read the testing code and sort out the annotations in order to guide the testing sequence. Then, it provides some operations to read the testing code, verify if the tester is supposed to execute an action and eventually execute it. The operations provided by the

Parser class are:

- **register()**: returns a list of actions to be executed. It receives as parameter the testing code that inherits the *TestCase* interface.
- **shouldIExecute()**: returns true if an action has to be executed by a tester. It receives as parameter the identifiers of testers set to execute such action. These parameters can be a range of testers identifiers or only one identifier.
- **invoke()**: receives as parameter an action and invokes it.

The *GlobalVerdict* class computes the local verdicts sent by the testers during the tests. This computation also considers the relaxation index that is used to handle inconclusive verdicts as discussed in Section 3.3.3.4. The operations provided by the *GlobalVerdict* class are:

- **setGlobalVerdict()**: sets a local verdict and the relaxation index.
- **getGlobalVerdict()**: returns the final verdict as soon as all local verdicts were computed.
- **getJudged()**: returns the quantity of testers already judged. This is used to acknowledge how many testers remains without a local verdict.

4.4 Writing test cases

In this section, we present how to write a test case using PeerUnit. First, we describe the annotations used by the testers to manage the test execution (i.e., synchronization and deployment). Then, we describe a simple test case wrote over the motivating example (see Example 3.1).

4.4.1 Annotations

PeerUnit provides three annotations. The first annotation specifies that a method is a test case action. As mentioned, a test case action is a point of synchronization used to manage the test sequence. This annotation, called @Test, provides the following parameters :

- “place” informs the id of a tester responsible to manage the action on a peer. Note that the testers’ id are dynamically assigned during the test case registration, and are not related to the peers’ id, which are proper to the SUT. If set to “-1”, then all the testers will manage the action.
- “from” informs the id of the first tester within a set of testers. This parameter indicates that the action will be managed by more than one tester. Furthermore, it cannot be used together with the “place” parameter.
- “to” informs the id of the last tester within a set of testers. This parameter also indicates that the action will be managed by more than one tester. It cannot be used together with the “place” parameter either.
- “name” informs the name of the test case. Different methods may be part of the same test case.
- “step” informs that a test case has different steps during the execution.
- “measure” measures the execution time in milliseconds.
- “timeout” indicates a milliseconds time frame to execute the action. If this time expires, then it is assigned a local *inconclusive* verdict.

The code below shows the annotation usage. This action, that belongs to test case “tc1”, will be managed by testers 0 to 20. The other parameters indicate that the execution will be measured and it has 1000 milliseconds to finish.

```
@Test(from=0, to=20, name="tc1", step=1, measure=true, timeout=1000)
public void action (){
    ...
}
```

The second and third annotations specify the test case set up and clean up respectively. The objective is to comply with the test harness described in section 3.3.2. These annotations, called @Before and @After, provide some of the same parameters used by @Test, such as : “place”, “from”, “to” and “timeout”.

The code below shows the @Before and @After usage. These methods will be managed by all the testers since the “place” parameter value is set to “-1”. The execution of each method has a time frame of 1000 milliseconds.

```
@Before(place=−1, timeout=1000)
public void setupAction (){
    ...
}
...
@After(place=−1,timeout=1000)
public void cleanAction (){
    ...
}
```

4.4.2 A test case example

We present a simple test case to illustrate the elements presented in this section. This test case was wrote over the motivating example (see Example 3.1). It checks the correctness of the insertion of two pairs *{(1, “One”),(2, “Two”)}* , by peers that join the system after its storage.

The test suite is implemented by a class named **TestSample**, which is a subclass of **TestCaseImpl**. The class **TestCaseImpl** is the implementation of the interfaces introduced into the architecture sections. The **TestSample** class contains an attribute named **peer**, an instance of the peer application. In this way, we can reach the SUT since peer is the implementation of a SUT peer (e.r., Chord, Pastry, etc).

```
public class TestSample extends TestCaseImpl {
    private Peer peer;
    private Integer key[] = {1,2};
    private String data[] = {"One","Two"};
}
```

The objective of the method *start()* below is to initialize the peer application. Since this initialization is rather expensive, it is executed only once. The *@Before* annotation ensures that this method is performed at all peers at most one time. This annotation attributes specify that the method can be executed everywhere and that its timeout is 100 milliseconds.

```
@Before(place=−1, timeout=100)
public void start (){
    peer = new Peer(); }
```

The method **join()** asks the peer instance to join the system. The annotation *@Test* specifies that this method belongs to the test case "tc1", that it is executed by testers 0, 1 and 2, and that the execution time is measured.

```
@Test(from=0, to=2, name="tc1", step=1, measure=true)
public void join(){
    peer.join();
}
```

The method **put()** stores some data in the DHT. The annotation ensures that only tester 2 executes this method.

```
@Test(place=2, timeout=100, name="tc1", step = 2)
public void put(){
    for(int i=0;i<2;i++){
        peer.put(key[i], data[i]);
    }
}
```

The method **joinOthers()** is similar to the method **join()**, presented above. The annotation *@Test* specifies that this method is the third action of the test case, and that only testers 3 and 4 execute it.

```
@Test(from=3, to=4, name = "tc1", step=3)
public void joinOthers(){
    peer.join();
}
```

The method **retrieve()** tries to retrieve the values stored at keys 1 and 2. If the retrieved objects correspond to the ones previously stored, the test case passes, otherwise it fails.

```
@Test(from=3, to=4, name = "tc1", step=4)
public void retrieve(){
    String actual[];
    for(int i=0;i<2;i++){
        actual[i] = peer.get(key[i]);
    }

    assertEquals(data, actual);
}
```

Finally, the method **stop()** asks the peer instance to leave the system. This method is executed by all the peers.

```
@After(place=-1,timeout=100)
public void stop(){
    peer.leave();
}
```

CHAPTER 5
Experimental Validation

In this chapter, we present an experimental validation of our framework with three objectives : (i) evaluate the coordination overhead of our testing architectures, (ii) validate the feasibility of our incremental testing methodology (iii) validate the usability and efficiency of the P2P testing framework based on experiments that test two popular open-source P2P systems. The first P2P system, FreePastry[1] from Rice University, is an implementation of Pastry [88]. The second P2P system, OpenChord[2] from Bamberg University, is an implementation of Chord [92]. Furthermore, testing these two P2P systems allows us to exercise the framework in different scenarios. Additionally, it allows us to compare both P2P systems as a side effect.

We tested two structures that are strongly impacted by volatility. The first structure is the routing table, which stores the addresses of the successor peers. Briefly, we tested the ability of a peer to update its routing table in three different test cases. In the first test case, a peer is isolated, i.e., all peers present in its routing table leave the system at the same time. In the second test case, some peers join the system. In the third test case, some peers leave the system.

The second structure is the DHT, which is responsible to store the data. We tested the implementation of the insert and retrieve operations. To do so, we wrote four test cases. The first test case validates these operations on a stable system and provides a base for the experiments that simulate volatility. The other three test cases validate these operations upon different volatility workloads. All test cases use randomly generated data.

5.1 Experimental environment

We validated our framework over the Grid5000 platform[3]. The Grid'5000 project aims at building a highly reconfigurable, controlable and monitorable experimental Grid platform gathering 9 sites geographically distributed in France featuring a total of 5000 processors.

We used two clusters of 64 nodes from the Grid5000 platform. Both clusters run GNU/Linux operating system. In the first cluster, each node has 2 Intel Xeon 2.33GHz dual-core processors. In the second cluster, each node has 2 AMD Opteron 248 2.2GHz processors. Since we can have full control over these clusters during experimentation, our experiments are reproducible. We allocate the peers equally through the nodes in the clusters up to 16 peers per node. In experiments with up to 64 peers, we use only one cluster.

In all experiments reported in this thesis, each peer is configured to run in its own Java VM. The implementation and tests can be found in our web page.[4] A complete test case with instructions to install and execute our framework is available at the appendix section.

[1] http ://freepastry.rice.edu/FreePastry/
[2] http ://open-chord.sourceforge.net/
[3] Grid5000 platform : http ://www.grid5000.fr/
[4] Peerunit project, http ://peerunit.gforge.inria.fr

5.2 Coordination overhead evaluation

We first evaluated the overhead of the coordination of actions. We measured this overhead by the response time of a test case execution. To measure the response time, we submitted a fake test case, composed of empty actions through a different range of testers. Then, for each action, we measured the whole execution time, which comprises remote invocations, execution of the empty actions and confirmations.

The fake test case contains 8 empty actions (we choose this number arbitrarily) and is executed up to a limit of 2048 testers running in parallel. The java code described below shows the source code of the fake test case. Figure 5.1 presents the response time for action coordination for a varying number of testers. Results are shown for two different testing architectures. Using the centralized architecture (see Section 3.3.3), the response time grows linearly with the number of testers as expected for an algorithmic complexity of $O(n)$. In contrast, the B-Tree architecture (see Section 3.3.4) had logarithmic response time.

```
/**
 * The fake test case
 */
public class SimpleTest extends TestCaseImpl{

    /**
     * This method starts the test
     */
    @BeforeClass(place=-1,timeout=1000)
    public void begin(){

    }
    @Test(place=-1,timeout=1000, name = "action2", step = 1)
    public void action2 (){

    }
    @Test(place=-1,timeout=1000, name = "action3", step = 1)
    public void action3 (){

    }

     ...

    @Test(place=-1,timeout=1000, name = "action7", step = 1)
    public void action7 (){

    }
    @AfterClass(timeout=1000,place=-1)
    public void end() {

    }
}
```

The result shows that the B-Tree architecture addressed scalability by managing actions across distributed testers. However, the result does not consider the construction of the tree in order to compare only the coordination times. The construction of the tree, performed by the bootstrapper, took 306 milliseconds for 2048 testers. Even if added to the coordination time, such increase yielded 72% better response time than the centralized architecture.

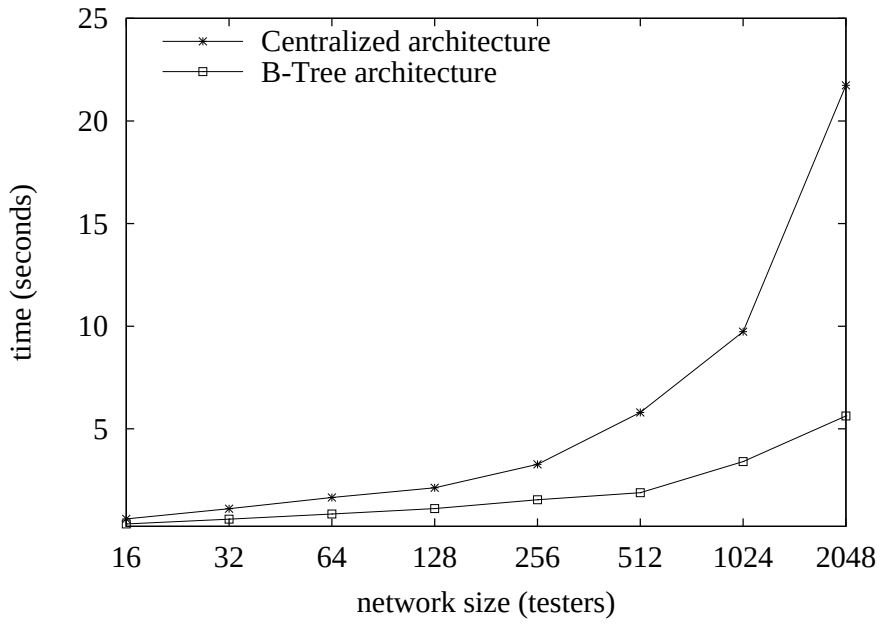


Figure 5.1 – Coordination Overhead Evaluation

5.3 Test Cases Summary

In this section, we describe the test cases used to test the routing table and the DHT. Initially, we describe the test sequences that the test cases are based on. Then, we detail the test cases.

5.3.1 The routing table test sequence

In the routing table test cases, testers must analyze the routing table of their peers to verify if it was correctly updated. More precisely, testers must compare the ID of peers from a routing table with the ID of peers that leave or join the system. This comparison is not trivial, because each tester only knows the ID of its peer, which is dynamically assigned. To simplify the analysis of routing tables, we use test case variables to map tester IDs to peer IDs, as shown in section 3.3.6.

We implemented the test case as follows.

Name : Routing Table Test.

Objective : Test the update of the routing table.

Parameters :

- P : the set of peers that form the SUT ;
- P_{init} : the initial set of peers ;
- P_{in} : the set of peers that join the system during the execution ;
- P_{out} : the set of peers that leave the system during the execution.

Actions :

1. System creation.
2. Volatile peers are stored in test case variables.
3. Volatility simulation.

4. Routing table verification and verdict assignment.

In the first action, a system is created and joined by all peers in P_{init}. In the second action, the IDs of P_{in} and/or P_{out} are stored in test case variables. In the third action, volatility is simulated : peers from P_{in} join the system and/or peers from P_{out} leave the system by comparing their IDs with the test case variables. In the fourth action, each remaining peer ($p \in P_{init} + P_{in} - P_{out}$) verifies its routing table, waiting for ι seconds. Then, the routing table is analyzed whether it has references to the test case variables and a verdict is assigned. Three different test cases are based on this test sequence :

1. Recovery from peer isolation.
2. Expanding system.
3. Shrinking system.

5.3.1.1 Recovery from peer isolation

The first test case consists in the departure of all peers that are present in the routing table of a given peer p. Then, we test if the routing table of p is updated within a time limit.

In the first action, the P2P system is created and a set of peers $P_{init} = P$ joins the system. In the second action, a peer $p \in P$ (randomly chosen) stores the contents of its routing table in the test case variable RT. In the third action, the peers whose IDs are in RT leave the system. In the fourth action, the routing table of p is periodically analyzed within a delay. At the end of this action, the routing table of p is analyzed a last time to assign a verdict. The values of RT are compared with the updated routing table of p. If the intersection of these two sets of IDs is empty, the verdict is $pass$.

5.3.1.2 Expanding system

In the second test case, we test if the peers that join a stable system are taken into account by the older peers. To do so, we analyze the routing table of each peer that belongs to a set of peers P_{init} to test if it is correctly updated within a time limit, after the joining of a set of new peers P_{in}.

In the first action, the P2P system is started and peers that belong to P_{init} join the system. The second action is a waiting time, allowing the SUT to reach a stable state. In the third action, the new peers (P_{in}) join the system and their IDs are stored in the test case variable, NID. In the fourth action, the routing table of each peer from P_{init} is compared with the values of NID. If all routing tables are updated, the verdict is $pass$.

5.3.1.3 Shrinking system

In this third test case, we test if the peers that leave a stable system are correctly removed from the routing tables of the remaining peers, within a time limit.

In the first action, the P2P system is created and all peers ($P = P_{init} \cup P_{out}$) join the system. In the second action, the IDs of peers from P_{out} are stored in the test case variable LID. In the third action, peers whose IDs belong to LID leave the system. The fourth and last action verify if the routing table of peers from $P_{init} - P_{out}$ does not have references to peers IDs belonging to LID.

5.3.2 The DHT test sequence

Name : DHT Test.

Objective : Test the insert/retrieve operations.

Parameters :

- P : the set of peers that form the SUT ;
- P_{init} : the initial set of peers ;
- P_{in} : the set of peers that join the system during the execution ;
- P_{out} : the set of peers that leave the system during the execution ;
- $Data$ the input data, corresponding to set of pairs (key, value).

Actions :

1. System creation.
2. Insertion of $Data$.
3. Volatility simulation.
4. Data retrieval and verdict assignment.

We describe the DHT test sequence as follows. In the first action, a system is created and joined by all peers in P_{init}. In the second action, a peer $p \in P_{init}$ inserts n pairs. In the third action, volatility is simulated : peers from P_{in} join the system and/or peers from P_{out} leave the system. In the fourth action, each remaining peer ($p \in P_{init} + P_{in} - P_{out}$) tries to retrieve all the inserted data, waiting for ι seconds. When the data retrieval is finished, the retrieved data is compared to the previously inserted data and a verdict is assigned. Four different test cases were extracted from this summary :

1. Stable system.
2. Expanding system.
3. Shrinking system.
4. Volatile system.

5.3.2.1 Insert/Retrieve in a stable system

In this test case, we set a system with two sets of peers P and P_1 where $P_1 \subset P$, and verify if all peers from P are able to retrieve k data previously inserted by peers from P_1. The test case is executed several times, for different sizes of P and of P_1.

Then, we set the system to execute 4 times. Each time testing with a different system size ($|P| = (16, 32, 64, 128)$). In all executions, no peer leaves or joins the system ($P_{in} = \emptyset$, $P_{out} = \emptyset$ and $P_{init} = P$). The same input data is used in all executions ($|Data| = 1,000$).

In the first action, all peers from P join the system. The second action waits for about 16 seconds in order to let the system stabilize. This delay was set based on the time to update the routing table presented in Section 5.6. In the third action, all peers from P_1 insert k data into the DHT. Since k is generated on-the-fly, we stored it in the test case variable ED. In the fourth action, each $p \in P$ tries to retrieve the data. The retrieval may be executed several times, the number of retrievals was defined before the execution of the test case. Once the data are retrieved, they are compared with the data of ED to assign a verdict.

5.3.2.2 Insert/Retrieve in an expanding system

In the second test case, we configured a system containing a set P_{init} of peers, where a peer $p \in P_{init}$ inserts k data. After the insertion, we configured a set P_{in} of new peers to join the system. Then, we test if all peers from P_{in} are able to retrieve the k inserted data. The size of P_{in} is computed by $(|P| * r)/100$ where r represents the rate of peers joining the system. We set this rate from 10% to 50% to test the system under different joining workloads.

We use a predefined number of peers ($|P| = 128$) and of input data ($|Data| = 1,000$). The test case uses different configurations, for different rates of peers joining the system. We set this rate from 10% to 50% ($|P_{init}| \times |P_{in}|$ = [(116, 12) ; (103,25) ; (90,38) ; (77,51) ; (64,64)]). No peer leaves the system ($P_{out} = \emptyset$).

In the first action, a system is created and joined by all peers in P_{init}. In the second action, a peer $p \in P_{init}$ inserts data, which is also stored in the test case variable ED. The number of peers inserting data is irrelevant, as presented in Section 5.6. In the third action, the peers from (P_{in}) join the system. In the fourth action, each $p \in P_{in}$ tries to retrieve all the inserted data, waiting for ι seconds. The retrieved data is stored in the local variable OD. When the data retrieval is finished, OD is compared to ED and a verdict is assigned.

5.3.2.3 Insert/Retrieve in a shrinking system

In this test case, we set a system containing a set P of peers, where a peer $p \in P$ inserts k data. Then, we test if a set of peers $P_1 \subset P$ is able to retrieve all inserted data when a set of peers $P_2 \subset P$ leaves the system. The size of P_2 is variable ; it is computed by $(|P| * r)/100$, where r corresponds to the rate of peers leaving the system. We set this rate from 10% to 50% to test the system under different departure workloads. We limited P to a 64-peer system.

We use a predefined number of peers ($|P| = 128$) and of input data ($|Data| = 1,000$). Initially, all peers join the system ($P_{init} = P$). After the data insertion, some peers leave the system. We set the rate of peers leaving the system from 10% to 50% ($|P_{out}|$ = (12, 25, 38, 51, 64)). No peer joins the system ($P_{in} = \emptyset$). Note that in Pastry, the data stored by a peer becomes unavailable when this peer leaves the system and remains unavailable until it comes back. Thus, in this test case we do not expect to retrieve all data, only the remaining data is retrieved to build the verdict.

In the first action, the system is created and all peers from P join the system. In the second action, a peer $p \in P$ inserts k data. In the third action, the IDs of all peers from P_2 (randomly chosen) are stored in the test case variable LP. Then, testers read LP and compare to their peers ID. If both IDs match, then the tester makes the peer leave the system.

In the fourth action, each $p \in P_1$ tries to retrieve all the data. After the departure, some data may be lost, especially with high rates of departure. Thus, a single retrieval by each $p \in P_1$ is enough to acknowledge the data available in the system. Such data is stored in the test case variable AD, and the data from the unavailable peers are removed. Then, each $p \in P_1$ tries to retrieve all the data waiting for a response within a time ι. The retrieval may be executed several times, the number of retrievals being defined before the execution of the test case. In the case of multiple retrievals, a delay is respected between two retrievals. Whenever the data arrives, it is stored in the local variable LI. Thus, LI is compared to AD at the end of this action and a verdict is assigned.

5.3.2.4 Insert/Retrieve in a volatile system

In this test case, we set a system containing a set $P_1 = P_{stable} \cup P_{out}$ of peers, where each peer $p \in P_1$ inserts k data. Then, we test if P_{stable} is able to retrieve all inserted data while a set of peers $P_2 = P_{in} \cup P_{out}$ joins/leaves the system. The size of P_2 is variable ; it is computed by $(|P| * r)/100$, where r corresponds to the rate of peers joining/leaving the system. We set this rate from 10% to 50% to test the system under different departure workloads. We limited P to a 64-peer system.

In this test case, we define a set of stable peers $P_{stable}, P_{stable} \subset P$ and $P = P_{stable} \cup P_{in} \cup P_{out}$. We set the rate of stable peers from 90% down to 50% ($|P_{stable}|$ = (116, 103, 90, 77, 64)). The initial set of peers is composed of the stable peers and the peers that will leave the system ($P_{init} = P_{stable} \cup P_{out}$). After the data insertion, all peers from P_{out} leave the system while all peers from P_{in} join the system.

5.4 Testing OpenChord

In this section, we briefly describe the routing table and the DHT of the Chord's algorithm. Then, we describe the result of the tests.

In the context of the routing table, Chord makes no effort to achieve good network locality. A peer's ID is chosen by hashing its IP address. Then, the peers are arranged ordered in an ID circle. Chord updates its routing table in two different ways : periodically and/or dynamically. To update its routing table periodically, the Chord algorithm uses an active process called stabilization. The update of the routing table also happens dynamically every time a peer joins or leaves the system. For instance, in a Chord system with peers p_0, p_1, p_3, and p_6, the routing table of p_1 stores the addresses of peers p_3 and p_6. These peers stored in the routing table are called successors. If a new peer p_4 joins the system, then p_1 will update its routing table with the address of p_4. Then, p_4 becomes a successor of p_1.

In the context of the DHT, Chord provides support for one operation : given a key, it maps the key onto a peer. Briefly, a key identifier is produced by hashing the key, while the peer identifier is chosen by hashing the peer's IP address. Then, both identifiers are used to assign keys to peers. For simplicity, we call "key" both the original key and the key identifier. For instance, in the same Chord system with peers p_0, p_1, p_3, and p_6, a key 1 would be located at p_1. Similarly, key 2 would be located at node p_3, and key 7 would be located at peer p_0.

5.4.1 Recovery from peer isolation

In this test case, we set the system to 64 peers. Indeed, creating a system with less than 64 peers can lead the test to an inconclusive result because p may know all the peers which are removed in the third action. In a larger system, the results would be similar since the update of the routing table is performed periodically.

OpenChord got a $pass$ verdict, however, 4 seconds after the removal of its successors. In fact, this delay represents a unique execution of the stabilization process (whose periodicity is set to 6 seconds). This test case showed that OpenChord peers were able to update their routing table correctly upon isolation.

Furthermore, we noticed that testing P2P systems is also a matter of time. Therefore, we also discuss the time to get a verdict in the further sections, in particular in Section 5.6.

5.4.2 Expanding system

In this test case, we increased the size of the system exponentially (2^n) up to 1024 peers[5] to test the update in different system sizes. We set a maximum time to limit the test execution. We also increased this time in exponential scale (2^n), starting from 8 seconds in order to perform at least one update in the routing table. Similar to the peer isolation test, OpenChord got a $pass$ verdict after the stabilization. However, the update took a little longer due to the time to execute the stabilization. In this test, we performed up to 25 different executions to validate the system at different sizes and upon different volatility rates. OpenChord got a $pass$ verdict in all executions.

5.4.3 Shrinking system

In this test case, we increased exponentially the size of the system and the time limit similarly to the expanding workload. OpenChord got also a $pass$ verdict. However, it took a longer period of time than the expanding system to get this verdict. This happened because the depart of peers were not informed (i.e., the peers were killed) and the stabilization could not contact all the bogus[6] peers to erase their addresses. Similar to the expanding workload, we performed up to 25 different executions to validate the system at different sizes and upon different volatility rates. OpenChord got a $pass$ verdict in all executions.

5.4.4 Insert/Retrieve in a stable system

Starting the test with a 16-peer system, all local verdicts were $pass$ (i.e. the data stored in ED matches with the retrieved objects). However, when testing with a higher number of peers, some local verdicts were $inconclusive$ due to a lack of response. For instance, in a 32-peer system, more than 9% of the testers did not get an answer of their peers (see Figure 5.2). When executing the tests in a larger system (e.g. 128 peers) the verdicts were frequently $fail$. At first sight, we thought that this verdict was due to an insufficient time to stabilize the system. Thus, we stated a maximum delay of 512 seconds. However, when executing with 128 peers, the test case got frequently $fail$ verdicts.

We concluded that OpenChord has a bug to retrieve data in systems larger than 64 peers. Peers may organize themselves in more than one DHT (i.e. Chord ring) and the peers of one DHT are not able to retrieve data from the others. This would mean that OpenChord is not able to merge the DHTs in order to retrieve all data within the configured period of time.

5.4.5 Insert/Retrieve in a volatile system

In this subsection, we describe the test results in a system up to 64 peers upon three different volatility workloads : expanding, shrinking and full volatility. Similar to the stable system, OpenChord got $fail$ verdicts in systems with more than 64 peers.

In an expanding system, OpenChord was able to retrieve data and get a $pass$ verdict. However, the time to retrieve data was longer if compared with the stable system. When a new peer joins, it becomes responsible for some inserted data, then such data must be transported to this peer consequently slowing down the retrieval. The data transportation is bigger as long as more peers join the system, therefore, retrieval was even slower at higher rates of volatility.

[5] 1024 peers correspond to 8 peers per node in the clusters.

[6] A peer that left the system, but with its address still valid in any routing table.

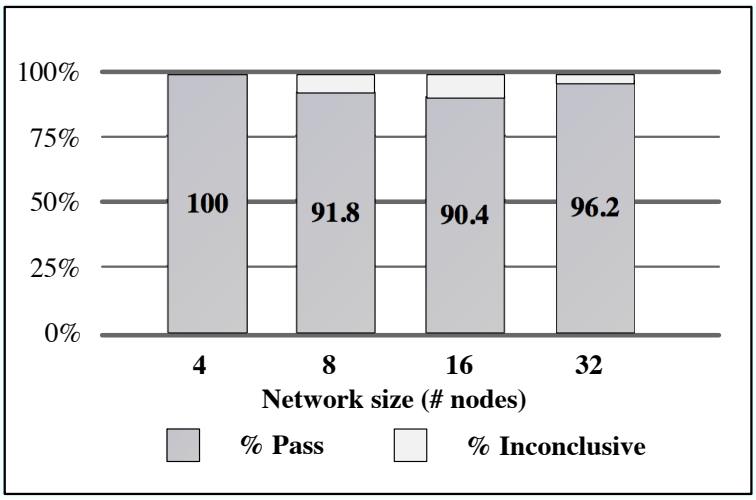


Figure 5.2 – Insert/Retrieve in a stable system

In a shrinking and full volatile system, OpenChord was able to retrieve data and get a $pass$ verdict. However, the time to retrieve data increases dramatically if compared with the expanding system. In OpenChord, once p notices that its successor has failed, p replaces its successor entry with the first live entry in the routing table. In high rates of departure, this replacement may be frequent, thus slowing down the retrieval.

5.5 Testing FreePastry

Like in the OpenChord section, we briefly describe the routing table and the DHT of the Pastry's algorithm. Then, we describe the result of the tests.

The Pastry routing table algorithm differs from Chord's mainly in its approach to achieve network locality. In Pastry, the ID of a peer is based on its local address. This means a peer is close to its successor geographically. To achieve this, a peer fills its routing table with the IDs of peers who shares the same prefix. Another difference is the update of the routing table. The Pastry algorithm uses a lazy approach to update its routing table. It means the routing table is updated only when a peer communicates with its successors (these are called neighbors in Pastry).

The Pastry DHT, called Past [43], is similar to Chord's DHT. A key is stored at the peer numerically closest to the key ID. Like Chord, the key ID is produced by hashing the key.

5.5.1 Recovery from peer isolation

Similar to the OpenChord test, we created a system of 64 peers to avoid the acknowledgment of the entire system. As mentioned, FreePastry uses a lazy approach to update the routing table. Then, we called a *ping* method to force the update of the routing table. We executed this test twice increasing the amount of calls to the *ping* method at each time. In the first time, we called the method just once and FreePastry got an $inconclusive$ verdict. Such verdict was assigned since we could not affirm that the routing table was not updated due to the laziness or to a bug. In the second time, we called the method twice, then FreePastry got a $pass$ verdict.

5.5.2 Expanding system

In this scenario, we used the same testing configuration of OpenChord's (i.e., system size and volatility rates). FreePastry had a similar result compared with the active process of OpenChord and also got a $pass$ verdict. This happened because when a new peer joins a FreePastry system, it needs to communicate with all its neighbors inducing the update of their routing tables.

5.5.3 Shrinking system

In this scenario, we used the same testing configuration of the expanding workload in terms of system size and volatility rates. FreePastry got a $pass$ verdict in all executions, however, the time to get such verdict increased dramatically compared with the expanding system due to laziness. Differently from the expanding workload, a peer does not contact any neighbor when leaving the system. Then, we had to call the *ping* method to force the update of the routing table, otherwise $inconclusive$ verdicts were assigned frequently.

5.5.4 Insert/Retrieve in a stable system

FreePastry passed all the tests in a stable system and could retrieve data with different system sizes. Concerning the retrieval response time, FreePastry got the same response time in all executions in a small-scale system. This happened because a peer knows a large portion of the system and in some cases it retrieves data directly (e.g., from a neighbor).

In a large-scale system, FreePastry also got $pass$ verdicts, however, the time to get this verdict increased motivated by the response time delay.

5.5.5 Insert/Retrieve in a volatile system

In this subsection, we describe the test results upon three different workloads : expanding, shrinking and full volatility. We executed each test 5 times to each workload. This represented different volatility rates starting from 10% to 50%.

In an expanding system, FreePastry retrieved data faster than in a stable system. A possible reason can be the neighbor locality. In FreePastry, the neighbors are geographically closer by IP address, which means the lookup may be faster, especially with the configured system size. Whenever a new peer p joins the system it needs to find and contact a neighbor. Then, Pastry updates the neighbor list of all the impacted peers. This update floods a large portion of the system and assists the retrievals. Moreover, FreePastry got $pass$ verdicts in all executions.

In a shrinking system, FreePastry retrieved data slower than any other workload. A possible reason can be the update of the routing table. As mentioned, FreePastry updates its routing table whenever a peer p needs to contact its neighbor. In the case of a retrieval, p contacts its neighbors immediately, and theirs neighbors do the same until reaching the peer which stores the data. This update may flood a large portion of the system and may assist other retrievals. However, this is slower than the expanding one also due to the laziness. The update of the neighbor list only happens when a peer tries to contact a neighbor, for instance, during a retrieval. FreePastry got $pass$ verdicts in all executions.

In a full volatile system, FreePastry could benefit from constant joins and leaves to update its routing table constantly, therefore, improving retrieval. Furthermore, FreePastry also passes this test case with any rate of volatility.

5.6 Comparing the P2P systems

Testing two P2P systems through the same test cases allows us the comparison of the test results and also of their approach to implement a DHT and a routing mechanism. However, we consider this comparison a side effect of our contribution. In this section, we discuss advantages and drawbacks of each approach based on the response time to execute the tests and get a verdict.

In the context of the routing table tests, OpenChord showed better results than FreePastry to update its routing table in terms of time to get a verdict. For instance, in the recovery from peer isolation FreePastry needed 29.9 seconds to get a $pass$ verdict while OpenChord needed only 4 seconds. In fact, OpenChord showed a faster routing table updating process than FreePastry due to its stabilization. Such stabilization can detect the departure of a peer quickly and may be a better approach compared with the lazy approach of FreePastry. This can be seen in the other results as well. Figure 5.3 shows the minimum time for a peer to update its routing table and get a $pass$ verdict in an expanding system. Figure 5.4 shows this minimun time in a shrinking system. In a shrinking system, FreePastry's laziness showed more evident and the time to finish any test increased dramatically.

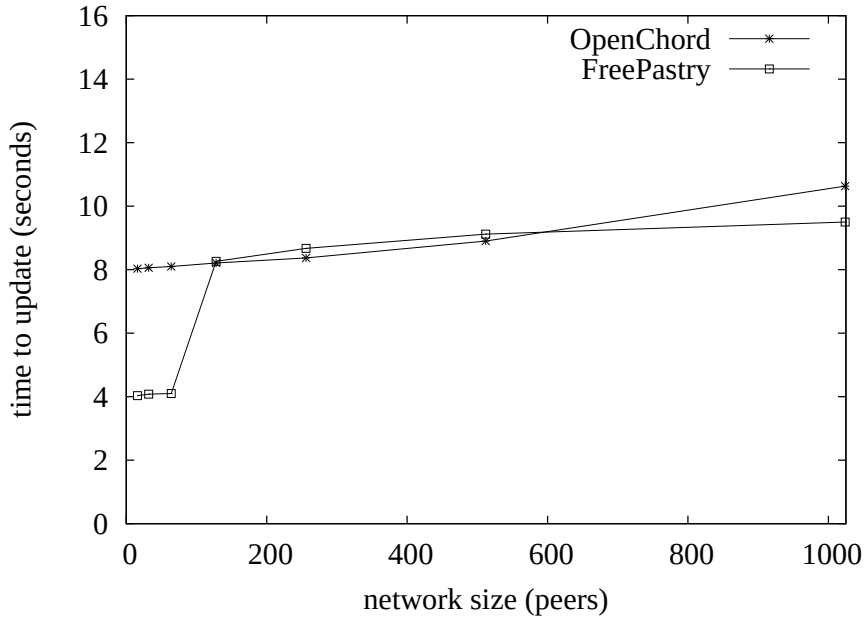


Figure 5.3 – Routing table update (expanding)

In the context of the DHT tests, Figure 5.5 shows that data retrieval in FreePastry is faster than in OpenChord for all system sizes. The retrieval time in FreePastry remains stable for any size of P_1. However, we did not test in larger systems due to the scalability limit of OpenChord.

In an expanding system limited to 64 peers, FreePastry and OpenChord were able to retrieve data and get a $pass$ verdict. Figure 5.6 shows that, with a joining rate of 30%, both DHTs need the same time to retrieve the data. However, FreePastry is faster than OpenChord in higher rates. As mentioned, a possible reason can be the successors locality. In FreePastry, they are geographically closer by IP address, which means the lookup may be faster, especially with the configured system size.

In a shrinking and full volatile system limited to 64 peers, Figure 5.7 shows that FreePastry was faster

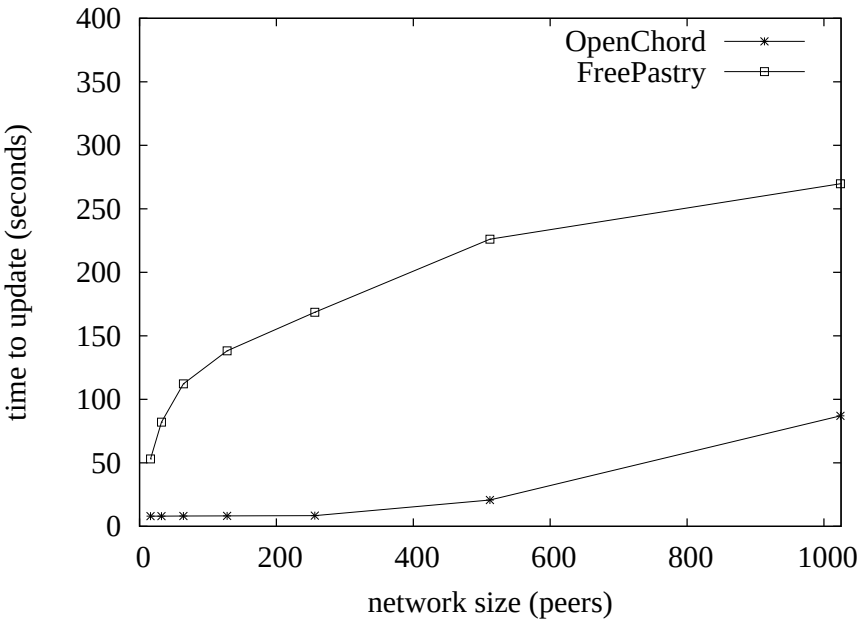


Figure 5.4 – Routing table update (shrinking)

than OpenChord in all rates. Due to retrieval, FreePastry peers are forced to update their routing tables. Therefore, the retrieval response time was better than in OpenChord. In OpenChord, the high rates of departure may reorganize the peers in multiples DHT (i.e., Chord ring), as argued in Section 5.4.4, thus slowing down data retrieval.

5.7 Code coverage

In this section, we present the code coverage of the test cases. We coupled code coverage with the experiments to measure the confidence of the test cases. Furthermore, we can measure whether using our incremental methodology increases coverage.

We focus the coverage analysis only on FreePasty since other DHTs, such as Chord [92] or CAN [86], have similar behavior for data storage and message routing. Therefore, a similar impact on code coverage of the size of the system and the number of data should be expected. Moreover, FreePastry has the most dynamic community among the DHTs with new packages releases available quite often.

To analyze the impact of volatility and scalability on the different test cases presented before, we conducted several experiments, using the insert/retrieve test case presented above, with different parameters. In these experiments, we use two Java code analysis tools for code coverage and code metrics, Emma[7] and Metrics[8], respectively.

According to these tools, FreePastry has 80,897 bytecode instructions and contains 130 packages. About 56 packages are directly concerned by the DHT implementation. The remaining packages deal with behaviors that are not relevant here : tutorials, NAT routing, unit testing, etc. In the code coverage

[7] http ://emma.sourceforge.net
[8] http ://metrics.sourceforge.net

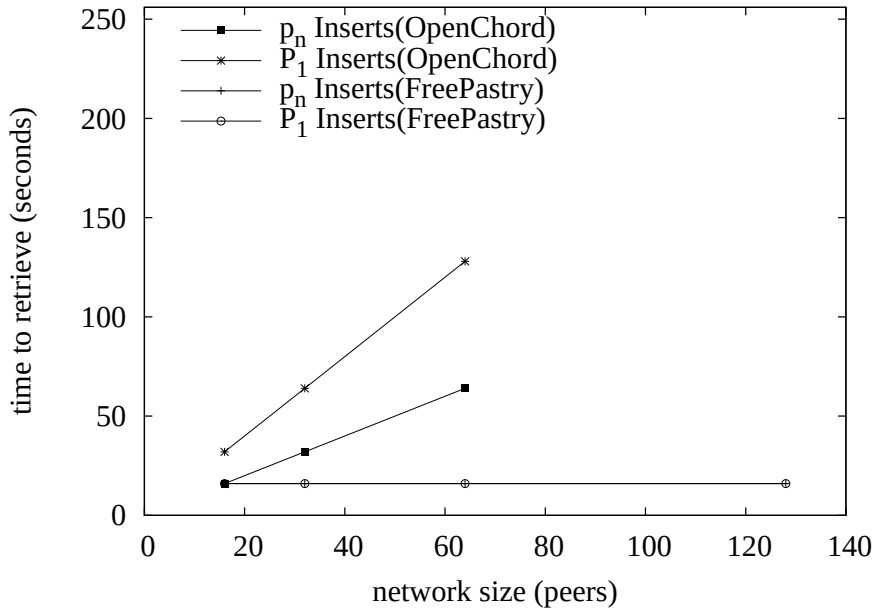


Figure 5.5 – Insert/Retrieve (stable system)

analysis presented in this section, we focus on 4 main packages and their sub-packages, which are summarized in Table 5.1. In all results presented here, the code coverage rate corresponds to a merge of the code covered by all peers.

Name	Qualified Name	Sub-packages	Instructions	Description
Past	rice.p2p.past	3	4,606	DHT service
Transport	org.mpisws.p2p.transport	16	19,582	Transport protocol (sockets/messages)
Pastry	rice.pastry	14	26,795	Routing network (nodes, join, routing)
Replication	rice.p2p.replication	4	2,429	Object replication

Table 5.1 – Main packages summary

For the first two experiments, we analyze the impact on the code coverage of two parameters, the size of the input data and the number of peers. As Figure 5.8 shows, the Past package is the most impacted by the growth of the cardinality of the input data, while the impact on the other packages is less significant. The reason for this is that the choice of the peer responsible for storing a given data depends on the data key. Thus, when a peer stores a large number of data, it must discover the responsible peers, i.e., using the **lookup()** operation. This operation behaves differently when communicating with known and unknown peers.

Figure 5.9 shows that the code coverage of the four packages grows when the system scales up. The explanation for this is that in small systems (e.g., 16 peers), peers know each other, and messages are not routed. When the system expands up to 128 peers, each peer only knows part of the system, making

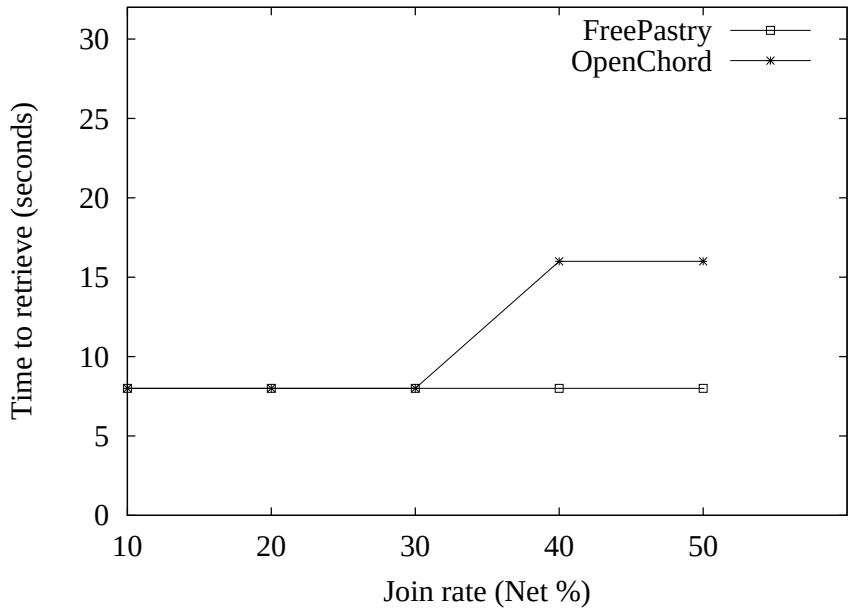


Figure 5.6 – Insert/Retrieve (expanding system)

communication more complex. However, there is a limit on the coverage gains, while scaling up from 128 peers to 256 peers.

In the other experiments, we analyze the impact of volatility on the code coverage, using the same test cases presented in Section 5.3.2. We compare these results with the coverage of the 14 original unit tests provided with FreePastry (noted OT), which are executed locally. Figure 5.10 presents a synopsis of the different code coverage results. As expected, our test cases cover more code than the original unit tests, especially on packages that implement the communication protocol.

At first glance, volatility seems to have a minor impact on code coverage, since the stable test case with 256 peers yields better results than some other test cases (e.g., shrinking 128). In fact, the impact is significant because the different test cases exercise different parts of the code and are complementary. This complementarity is noticeable for the Pastry and the Past packages, where the accumulated results are better than any other result. The total accumulated coverage (Accum.+OT) shows that our tests cases and the original unit tests are also complementary.

5.8 Conclusion

Let us summarize the experimental results.

We can notice by the results that testing a P2P system is not only a matter of correctness. Peers exchange messages asynchronously and may not answer a complete result set to a query due to timeout limitations or even volatility. Therefore, testing any P2P system is also a matter of time.

In the tests to update the routing table, FreePastry and OpenChord were able to update their routing table in all scenarios, therefore, getting $pass$ verdicts. The results also show that OpenChord was faster than FreePastry to update the routing table. This is because OpenChord updates it through an active process while FreePastry updates it only when a peer communicates with its successors.

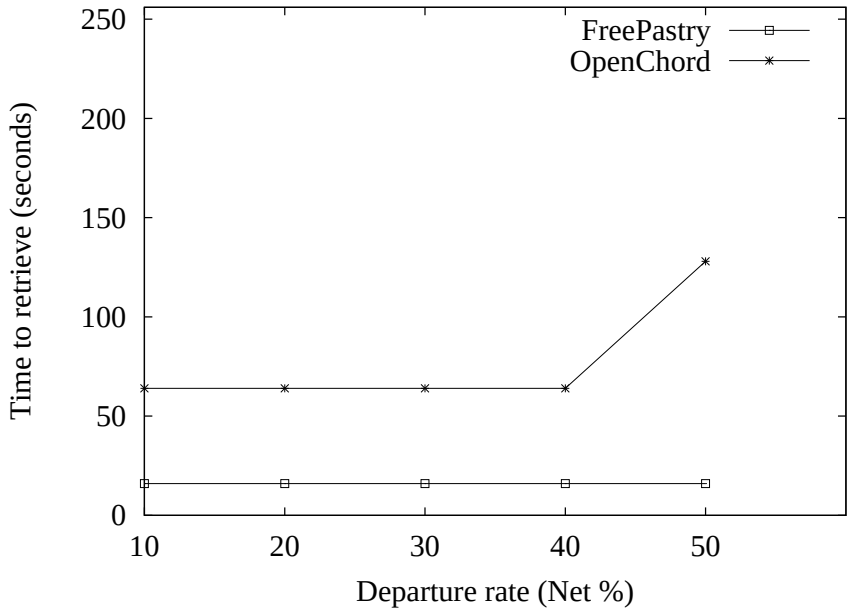


Figure 5.7 – Insert/Retrieve (shrinking system)

In the tests of the insert/retrieve operations, FreePastry was able to retrieve data upon different workloads and get $pass$ verdicts. However, OpenChord was not able to retrieve data in a system with more than 64 peers. In this size, OpenChord gets frequently $fail$ verdicts. A possible reason is the organization of OpenChord peers in multiple DHTs. The peers from one DHT are not able to retrieve data from the other.

In a system limited to 64 peers, the results show that FreePastry was faster than OpenChord to retrieve data upon different workloads. A possible reason is the organization of peers in the system. FreePastry organizes the peers by IP address which speeds up the lookup for data, especially with the configured system size. OpenChord organizes the peers logically by ID. This means that two peers running in the same machine may not be successors.

As expected, volatility increases code coverage. However, such increase has a limit due to some specific portions of the code (e.g., exceptions) that can be covered only by specific test cases. For instance, a test case that covers the exception threw by a look-up performed with the address of a bogus peer. This situation only happens when a peer address resides in the routing table after its volatility.

Other DHTs, such as Chord [92] or CAN [86], have similar behavior to FreePastry for data storage and message routing. Therefore, a similar impact on code coverage of the size of the system and the number of data should be expected.

In spite of the test cases simplicity, the ratio of code covered by all test cases is rather important, as showed in Figure 5.10. While the impact of volatility, the number of peers and the amount of input data on the code coverage are noticeable, the only variation of these parameters is not sufficient to improve code coverage on some packages, for instance, the transport package. A possible solution to improve the coverage of these packages is to alter some execution parameters from the FreePastry configuration file. Most of the parameters deal with communication timeouts and thread delays. Yet, the number of parameters (≈ 186) may lead to an unmanageable number of test cases.

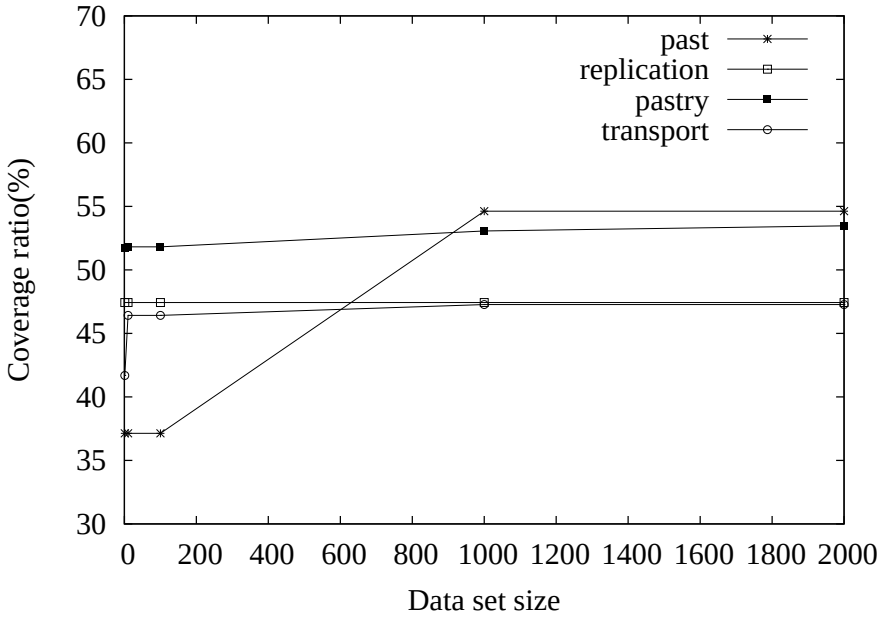


Figure 5.8 – Coverage on a 16 peers stable system

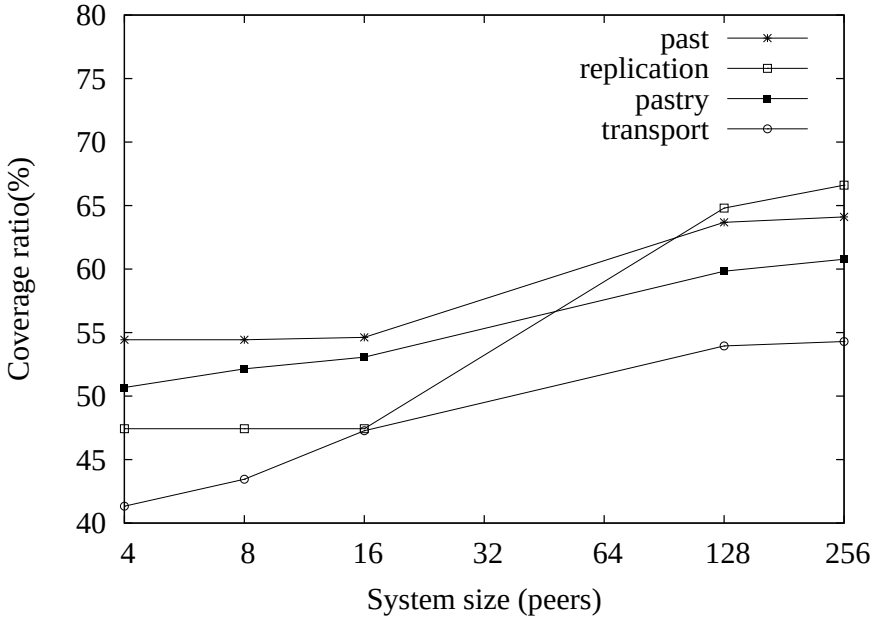


Figure 5.9 – Coverage inserting 1000 pairs

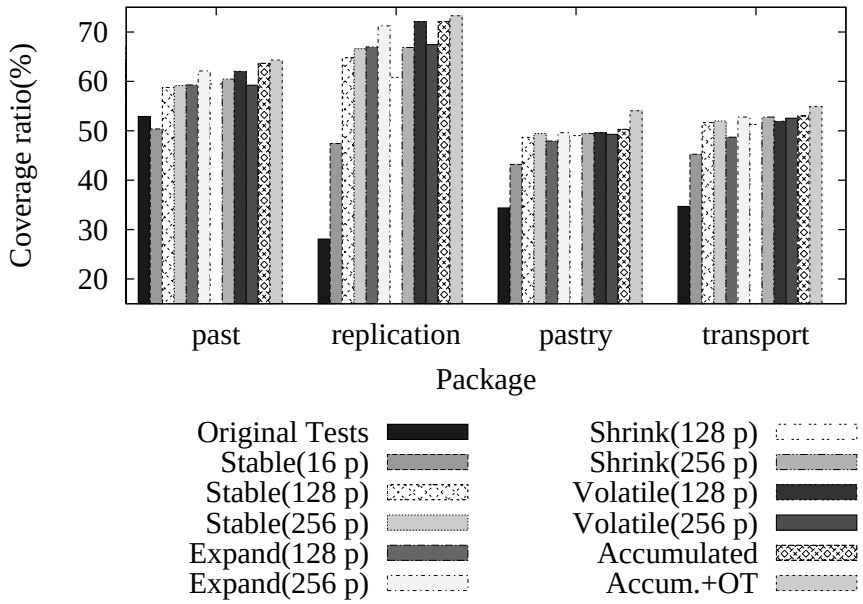


Figure 5.10 – Coverage by package

The experiments also revealed a bug on both systems, which was indirectly found along testing. Indeed, the bootstrap peer, i.e., the peer that is used by new peers to join the system, acting like a *Broker* [21, 41], were unable to treat a certain number of simultaneous requests (more than approximately 30). This bug was particularly annoying when setting up tests for large systems. Hopefully, the developers were able to correct the bug and release a new version of their software.

CHAPTER 6
Conclusion

In this chapter we present the general conclusions of this thesis. First, we revisit the major issues for testing P2P systems. Second, we summarize the related work. Then, we summarize our main contributions. Finally, we discuss future directions of research in testing of P2P systems.

6.1 Issues on testing P2P systems

In the context of distributed systems, peer-to-peer (P2P) appears as a powerful paradigm to develop scalable distributed systems. No central server manages the distributed nodes. Thus, there is neither a single point of failure nor a performance bottleneck in the system.

In a P2P system, a peer plays the role of an active node with the ability to join or leave the network at any time, either normally (e.g., disconnection) or abnormally (e.g., failure). This ability, which we call volatility, is a major difference with distributed systems. Furthermore, volatility yields the possibility of dynamically modifying the network size and topology, which makes P2P testing quite different.

In summary, a P2P testing framework should provide the possibility to control three aspects :

- functionality captured by the test sequence which enables a given behavior to be exercised,
- scalability captured by the number of peers in the system,
- volatility captured by the number of peers which leave or join the system after its initialization during the test sequence.

These aspects exposed the need for a precise methodology for P2P testing, where the simplicity of interfaces contrasts with the complexity of the factors that can affect the test : volatility, number of peers, data size, amount of data, number of concurrent requests, etc. Thus, the difficulty of testing is not only in choosing the relevant input data, but also in choosing the factors that should vary, their values and their association.

6.2 Survey of testing distributed systems

We surveyed the testing architectures comparing their perspectives from the context of distributed testing. Then, we discussed why they are not suited to test P2P systems. We showed that they fail when dealing with the volatility, either interrupting the testing sequence and deadlocking the tester or assigning false-negative verdicts to test cases (i.e., false fail verdict).

Some approaches for P2P testing propose to randomly stop the execution of peers [13, 69], or to insert faults in the network [55, 76]. While these approaches are useful to observe the behavior of the whole system, they are not totally adapted for testing a P2P system. Since they focus on tolerance to network perturbations, they fail in detecting software faults, specially those which occur due to peers' volatility. For instance, if one wants to test if a peer is able to rebuild its routing table when all peers it knows leave the system, then one needs to specify precisely the peers to drop since the random volatility

is not precise enough. Increasing significantly the rate of volatile peers is not desirable either since the high number of messages necessary to rebuild the routing tables from the remaining peers may interfere with the test.

Then, we showed that most of them do not scale up due to centralized test coordination. In fact, the ones that scale up demand to include additional code into the SUT source code. While this approach supports the volatility control, it is error prone.

A P2P system must also scale up to large numbers of peers. It means the testing architecture must scale up either. Indeed, a typical P2P system may have a high number of peers which makes the centralized testing architecture (i.e., central coordinator managing distributed testers) not scalable [44, 64, 33, 71]. It is also possible to use distributed testers where test cases are coordinated through messages exchanged at points of communication (PC) [101, 98]. Even with such PCs, the testing architecture perturbs the performance of P2P systems. Indeed, the algorithmic performance with such testing architecture scales up linearly with the number of peers, while a typical P2P system scales up logarithmically. To avoid testers intrusiveness to threaten the validity of test results, the testing architecture should thus have logarithmic performance in the worse case.

6.3 Contributions

In this thesis, we presented an integrated solution for the creation and deployment of a P2P test environment with the ability to create peers and make them join and leave the system. Then, the test objectives can combine the functional testing of the system with the volatility variations (and also scalability). The correctness of the system can thus be checked based on these three dimensions, i.e. functions, number of peers and peers volatility.

6.3.1 Incremental methodology

In this thesis, we recommend to fully control volatility in the definition of the test sequence. A peer thus, from a testing point of view, can have to leave or join the system at a given time in a test sequence. This action is specified in the test sequence in a deterministic way.

Since it has the objective to deal with a large number of peers, the second dimension of P2P system testing is scalability. Then, because it is accomplishing a treatment, the scalability and volatility dimensions have to be tested with the behavioral and functional correctness.

To take into account the three dimensional aspects of P2P systems, we presented a methodology that combines the functional testing of an application with the variations of the other two aspects [33]. Indeed, we incrementally scale up the SUT either simulating or not volatility.

Our incremental methodology is composed by the following steps :

1. small scale application testing without volatility ;
2. small scale application testing with volatility ;
3. large scale application testing without volatility ;
4. large scale application testing with volatility.

In terms of diagnosis, this methodology allows to determine the nature of the detected erroneous behavior. Indeed, the problem can be linked to a purely functional cause (Step 1), a volatility issue (Step 2), a scalability issue (Step 3) or a combination of these three aspects (Step 4). The most complex

errors are the last ones since their analysis is related to a combination of the three aspects. Steps 2 and 4 could also be preceded by two other steps (shrinkage and expansion), to help the diagnosis of errors due to either the unavailability of resources or arrival of new ones. Yet, several rates of volatility can be explored to verify how they affect the functionality aspect of the SUT (e.g., 10% joining, 20% leaving).

6.3.2 Testing architectures

We also presented three different architectures to coordinate the test sequence. These architectures are based on two original aspects : (i) the individual control of peers' volatility and (ii) a distributed testing architecture to cope with large numbers of peers. The capabilities of these architectures are (1) to automate the execution of each local-to-a-peer test case, (2) to build automatically the global verdict, (3) to allow the explicit control of each peer volatility.

The first architecture extends the classical centralized testing architecture (i.e., central coordinator managing distributed testers) with volatility control to demonstrate that such volatility is a key-parameter when testing a P2P system [34]. Basically, this architecture has two main components : the tester and the coordinator. The tester is composed of the test suites that are deployed on several logical nodes. The coordinator is deployed in only one node and is used to synchronize the execution of test cases. It acts as a *broker* [21] for the deployed testers.

However, the performance of such centralized architecture is linear while testing with large numbers of peers requires logarithmic. Therefore, we presented two fully distributed architectures to cope with large-scale P2P systems [36]. The second architecture organizes the testers in a B-Tree [17] manner where the synchronization is performed from the root to the leaves. Then, the testers communicate with each other across the B-tree to avoid using a centralized coordination. The third architecture uses gossiping messages among testers reducing communications among the testers responsible to execute consecutive test case actions. Since both distributed architectures do not rely on a central coordinator they scale up correctly.

The distributed architectures are composed of a distributed component, the tester. The tester is the application that executes in the same logical node as peers, and controls their execution and their volatility, making them leave and join the system at any time, according to the needs of a test. Thus, the volatility of peers can be controlled at a very precise level. These architectures do not address the issue of test cases generation but is a first element towards an automated P2P testing process. It can be considered analogous to the JUnit[1] testing framework for Java unit tests.

6.4 Validation

We used our framework to test two open-source DHTs, FreePastry and OpenChord. We conducted several experiments, testing the behavior of both DHTs on different conditions of volatility in order to validate the usability and efficiency of our testing framework. Furthermore, it was able to detect implementation problems. The experiments showed that both systems correctly updated their routing table and that only FreePastry has a reliable implementation of the two main DHT operations (insert/retrieve). As for OpenChord, it did not pass the tests of these same operations, when the size of the P2P system was greater than 64 peers.

We coupled the experiments with an analysis of code coverage, showing that the alteration of the three dimensional aspects improves code coverage, thus improving the confidence on test cases. During

[1] http ://junit.org/

the experiments, we focused on volatility testing and did not test these systems on more extreme situations such as performing massive inserts and retrieves or using very large data. Testing different aspects (concurrence, data transfer, etc.) would increase significantly the confidence on both SUT. However, these tests were out of the scope of this thesis. They could be performed through the interface of a single peer and would not need the framework presented in this thesis.

We also showed that testing a P2P system is not only a matter of correctness. Peers exchange messages asynchronously and may not answer a complete result set to a query due to timeout limitations or even volatility. Therefore, testing any P2P system is also a matter of time.

The experiments also revealed a bug on both systems, which was indirectly found along testing. Indeed, the bootstrap peer, i.e., the peer that is used by new peers to join the system, acting like a *Broker* [21, 41], were unable to treat a certain number of simultaneous requests (more than approximately 30). This bug was particularly annoying when setting up tests for large systems. Hopefully, the developers were able to correct the bug and release a new version of their software.

6.5 Future work

The next challenging issue is to propose a solution to select scenarios that guarantees the functional coverage of the P2P functions in combination with the "coverage" of volatility/scalability. Such a multidimensional coverage notion should be defined properly as an extension of existing classical coverage criteria.

We also intend to test these P2P systems on more extreme situations such as performing massive inserts and retrieves or using very large data. Testing different aspects (concurrency, data transfer, etc.) would increase significantly the confidence on these systems. Furthermore, these different aspects can be useful for stress testing.

The purpose of stress testing is to verify how a system behaves under extreme but valid conditions : high number of simultaneous users, low memory, etc. In P2P Systems, this also concerns volatility. Thus, testing a P2P system under extreme situations upon volatility sounds fruitful as we indirectly found bugs along bootstrapping large amount of peers.

In another context, P2P systems generate large amount of logs along execution. Thus, an interesting path to pursuit can be log file analysis. As discussed in section 2.4.2, we can benefit from this to build a powerful oracle complementary to the assertion approach that we chose. Furthermore, we can load log files into databases to take advantage of its capabilities to build complex analysis reports (e.g., anomaly reports) and manage large amount of data.

In this context, tracing also can be an useful approach [32, 97]. Fault scenarios in the trace are identified that guide the developer to the origin of a system failure. These scenarios can be used either along system testing or model-checking in order to visualize fulfillment or violations of the requirements.

Finally, we also intend testing other DHT implementations such as Bamboo[2] or JDHT[3].

[2] http ://bamboo-dht.org/

[3] http ://dks.sics.se/jdht/

Appendixes

Framework install

We call our framework as PeerUnit and make it freely available on internet for download[1]. In this section, we describe some requirements and instructions to install PeerUnit.

A.1 Requirements

We compile and run Peerunit on different platforms with Java JDK1.5 :

- FreeBSD
- MacOSX
- Linux

In addition, it is necessary to use some external packages, such as :

- "Apache Ant" is a Java-based build tool similar to "make" on Unix, but it is platform independent. Peerunit is compiled and its jar package is built through a handy script in "Ant".
  `http://ant.apache.org/`
- "Jsh" is a java shell or a java launcher. That is, a program with a prompt that allows you to type sequence of commands. Our interest on Jsh is to reach the SSH connectivity tool from a java API.
  `http://downloads.sourceforge.net/jsch/jsch-0.1.32.jar`
- Deployment package. We developed a deployment tool based on "Jsh" to allow a distributed execution of our testing campaign :
  `http://www.sciences.univ-nantes.fr/lina/gdd/members/sunye/deploy.jar`

A.2 Install

You can get peerunit in two ways (unix-like command).

1. - Get the jar package like this :

   ```
   $ wget  -P  /your_path http://www.sciences.univ-nantes.fr/lina/gdd
   /members/almeida/peerunit.jar
   $ export  CLASSPATH=\$CLASSPATH:/your_path/peerunit.jar:.
   ```

2. - Compile the source code using our ant file (build.xml). The ant file will create the file peerunit.jar.

   ```
   $ ant  build
   ```

   The peerunit.jar will be generated in the "dist/" folder.

[1] Peerunit project, http ://peerunit.gforge.inria.fr

Executing test cases

In this section, we describe instructions to execute a test case on PeerUnit.
This test case verifies if some data was put in a freepastry DHT.

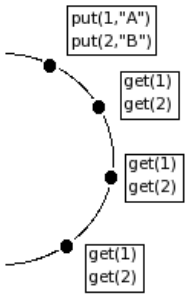


Figure B.1 – Insert test

B.1 The test case summary

On a system of "n" peers, where a peer inserts "k" values, are all peers able to retrieve all the inserted values ?

Name : DHT Test.

Objective : Test the insert/retrieve operations.

Parameters : P : the set of peers that form the SUT ; P_{init} : the initial set of peers ; P_{in} : the set of peers that join the system during the execution ; P_{out} : the set of peers that leave the system during the execution ; $Data$ the input data, corresponding to set of pairs (key, value).

Actions : (i) System creation ; (ii) Insertion of $Data$; (iii) Volatility simulation ; (iv) Data retrieval and verdict assignment.

The test sequence is as follows. In the first action, a system is created and joined by all peers in P_{init}. In the second action, a peer $p \in P_{init}$ inserts n pairs. In the third action, the volatility is simulated : peers from P_{in} join the system and/or peers from P_{out} leave the system. In the fourth action, each remaining peer ($p \in P_{init} + P_{in} - P_{out}$) tries to retrieve all the inserted data, waiting for ι seconds. When the data retrieval is finished, the retrieved data is compared to the previously inserted data and a verdict is assigned.

B.2 How to execute it (unix-like)

This example uses the centralized architecture.

1. - Download and set the CLASSPATH for Peerunit (described in the install appendix A).
2. - Download Freepastry and set the CLASSPATH. It requires three packages to run :
   - `FreePastry-2.0_02.jar`
   - `xpp3-1.1.3.4d_b2.jar`
   - `xmlpull_1_1_3_4c.jar`
3. - Configure the tester.properties that should be in the "config/" folder. This folder must be created in the same folder where you put the peerunit.jar file. Here is an example of the tester.properties file.

   To execute this test you just have to change the following properties :
   - tester.server=<IP where the coordinator will execute>
   - tester.logfile=<path and the name for the peerunit logfile>
   - tester.logfolder=<path to store the logfiles of the peers>
   - tester.peers=<number of peer that execute the test>
4. - Start the peerunit coordinator :

   ```
   $ java fr.inria.peerunit.rmi.coord.CoordinatorImpl &
   ```
5. - Starting the bootstrap peer :

   ```
   $ java freepastry.Bootstrap &
   ```
6. - Run the test(each peer will run in a single JVM) :

   ```
   $ java freepastry.test.SimpleTest &%$
   ```

   For instance, if you want to execute with 4 peers(set in tester.peers property), you can use this simple script :

   ```
   #!/bin/bash
   for ((i=0;i<=3;i+=1));
   do
           java freepastry.test.SimpleTest &
   done
   ```

Obs. Sometimes Freepastry doesn't bootstrap the peers and the verdict is "inconclusive", so you need to give another try.

B.3 The test case source code

```
package freepastry . test ;

import static fr . inria . peerunit . test . assertion . Assert . inconclusive ;

import java . io . IOException;
import java . net . InetAddress ;
import java . net . InetSocketAddress ;
import java . net . UnknownHostException;
import java . rmi . RemoteException;
import java . util . ArrayList ;
import java . util . List ;
import java . util . logging . Logger;
```

```
import rice.environment.Environment;
import rice.p2p.commonapi.Id;
import rice.p2p.past.PastContent;
import rice.p2p.past.PastContentHandle;
import rice.tutorial.past.MyPastContent;
import util.FreeLocalPort;
import fr.inria.peerunit.TestCaseImpl;
import fr.inria.peerunit.parser.AfterClass;
import fr.inria.peerunit.parser.BeforeClass;
import fr.inria.peerunit.parser.Test;
import fr.inria.peerunit.test.assertion.Assert;
import fr.inria.peerunit.util.TesterUtil;
import freepastry.Peer;
import freepastry.test.old.TestInsertLeaveB;

public class SimpleTest extends TestCaseImpl{
    // logger from jdk
    private static Logger log = Logger.getLogger(TestInsertLeaveB.class.getName());
    private static SimpleTest test;
    // Freepastry peer
    Peer peer = new Peer();

    /**
     * This method starts the test
     */
    @BeforeClass(place=-1,timeout=100)
    public void begin(){
        log.info("Starting the test ");
    }

    /**
     * This method starts the bootstrap peer
     */
    @Test(place=0,timeout=1000, name = "tc1", step = 1)
    public void startingNetwork(){
        try {

            log.info("I am "+test.getPeerName());
            // Loads pastry settings
            Environment env = new Environment();

            // the port to use locally
            FreeLocalPort port= new FreeLocalPort();
            int bindport = port.getPort();
            log.info("LocalPort:"+bindport);

            // build the bootaddress from the command line args
            InetAddress bootaddr = InetAddress.getByName(TesterUtil.getBootstrap());
            Integer bootport = new Integer(TesterUtil.getBootstrapPort());
            InetSocketAddress bootaddress;

            bootaddress = new InetSocketAddress(bootaddr,bootport.intValue());

            if (!peer.join(bindport, bootaddress, env, log)){
                inconclusive("I couldn't become a boostrapper, sorry");
```

```
        }

        // Setting the bootstrap address
        test.put(0, peer.getInetSocketAddress(bootaddr));
        log.info("Net created");

        while(!peer.isReady())
            Thread.sleep(1000);

    } catch (IOException e) {
        e.printStackTrace();
    } catch (InterruptedException e) {
        e.printStackTrace();
    } catch (Exception e) {
        e.printStackTrace();
    }
}

/**
 * This method starts the rest of the peers
 */
@Test(place=-1,timeout=1000000, name = "tc1", step = 2)
public void joinNet(){

    try {
        // Wait a while due to the bootstrapper performance
        Thread.sleep(10000);
        if(test.getPeerName()!=0){
            // Loads pastry settings
            Environment env = new Environment();

            // the port to use locally
            FreeLocalPort port= new FreeLocalPort();
            int bindport = port.getPort();
            log.info("LocalPort:"+bindport);

            // Each peer waits a while to join due to the freepastry bootstrap
            Thread.sleep(test.getPeerName()*1000);

            // Getting the bootstrap address
            InetSocketAddress bootaddress= (InetSocketAddress) test.get(0);
            log.info("Getting cached boot "+bootaddress.toString());

            if(!peer.join(bindport, bootaddress, env, log)){
                inconclusive("Couldn't boostrap, sorry");
            }
            log.info("Running on port "+peer.getPort());
            log.info("Time to bootstrap");

        }
    } catch (RemoteException e) {
        e.printStackTrace();
    } catch (InterruptedException e) {
        e.printStackTrace();
    } catch (UnknownHostException e) {
        e.printStackTrace();
```

```
    } catch (IOException e) {
        e.printStackTrace();
    } catch (Exception e) {
        e.printStackTrace();
    }
}

/**
 * Stabilize the network.
 */
@Test(place=-1,timeout=1000000, name = "tc1", step =3)
public void stabilize(){
    for (int i = 0; i < 4; i++) {
        try{
            // Force the routing table update
            peer.pingNodes();
            Thread.sleep(16000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
/**
 * Put some data and store in test variables.
 */
@Test(place=0,timeout=1000000, name = "tc1", step = 4)
public void put(){
    for(int i=0; i < 2 ; i++){
        // build the past content
        final String s = "test" + peer.env.getRandomSource().nextInt();
        final PastContent myContent = new MyPastContent(peer.localFactory.buildId(s), s);
        peer.insert(myContent);
    }

    // Wait until all the insert ends since it is asynchronous
    while((peer.getFailedContent().size()+peer.getInsertedContent().size())<2){
        log.info("Inserted so far : "+peer.getInsertedContent().size());
        log.info("Failed so far : "+peer.getFailedContent().size());
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    List<PastContent> expecteds= new ArrayList<PastContent>();
    for (PastContent content : peer.getInsertedContent()) {
        log.info("Expected so far : "+content.toString());
        expecteds.add(content);
    }

    // Use a test variable to store the expected data
    test.put(1, expecteds);
}
/**
 * Get the data and the verdict.
```

```
     */
    @Test(place=-1,timeout=1000000, name = "tc1", step = 5)
    public void get(){
        // Lookup
        List<PastContent> expectedContent=(List<PastContent>) test.get(1);
        Id contentKey;

        // Get the keys to lookup for data
        for (PastContent key : expectedContent) {
            contentKey=key.getId();
            if(contentKey!=null){
                log.info("Lookup Expected "+contentKey.toString());
                peer.lookup(contentKey);
            }
        }

        // Wait a little while for a response
        try {
            Thread.sleep(16000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        // A list to store the retrieved data
        List<String> actuals = new ArrayList<String>();
        for (Object actual : peer.getResultSet()) {
            if(actual!=null){
                if(!actuals.contains(actual.toString())){
                    log.info("[Local verdict] Actual "+actual.toString());
                    actuals.add(actual.toString());
                }
            }
        }

        // Generating the expecteds list
        List<String> expecteds=new ArrayList<String>();
        for(PastContent expected : expectedContent){
            expecteds.add(expected.toString());
        }

        // Assigning a verdict
        log.info("[Local verdict] Waiting a Verdict. Found "+actuals.size()+" of "+expecteds.size());
        Assert.assertCollectionEquals("[Local verdict] ",expecteds, actuals);
    }

    /**
     * This method finishes the test
     *
     */
    @AfterClass(timeout=100,place=-1)
    public void end() {
        log.info("Peer bye bye");
    }
}
```

Bibliography

[1] The Free Network Project, http://freenetproject.org/.
Disponible à l'adresse
http://freenetproject.org/.

[2] JXTA, https://jxta.dev.java.net/.
Disponible à l'adresse
https://jxta.dev.java.net/.

[3] Napster, http://free.napster.com/.
Disponible à l'adresse
http://free.napster.com/.

[4] SysUnit project, http://docs.codehaus.org/display/SYSUNIT/Home.
Disponible à l'adresse
http://docs.codehaus.org/display/SYSUNIT/Home.

[5] ISO International Standard 9646.
Open systems interconnection conformance testing methodology and framework, 1991.

[6] Karl ABERER, Philippe CUDRÉ-MAUROUX, Anwitaman DATTA, Zoran DESPOTOVIC, Manfred HAUSWIRTH, Magdalena PUNCEVA et Roman SCHMIDT.
P-Grid: A self-organizing structured P2P system.
ACM SIGMOD, 2003.

[7] Reza AKBARINIA.
Data Acess Techniques in P2P Systems.
Thèse de Doctorat, Université de Nantes, 2007.

[8] Sihem AMER-YAHIA et Sophie CLUET.
A declarative approach to optimize bulk loading into databases.
ACM Trans. Database Syst., 29(2):233–281, 2004.

[9] James H. ANDREWS.
Testing using log file analysis: Tools, methods, and issues.
Dans *ASE '98: Proceedings of the 13th IEEE international conference on Automated software engineering*, page 157, Washington, DC, USA, 1998. IEEE Computer Society.

[10] James H. ANDREWS.
Theory and practice of log file analysis.
Rapport technique, Department of Computer Science, University of Western Ontario, May 1998.
Technical Report 524.

[11] James H. ANDREWS et Yingjun ZHANG.
Broad-spectrum studies of log file analysis.
Dans *ICSE '00: Proceedings of the 22nd international conference on Software engineering*, pages 105–114, New York, NY, USA, 2000. ACM.

[12] Stephanos ANDROUTSELLIS-THEOTOKIS et Diomidis SPINELLIS.
A survey of peer-to-peer content distribution technologies.
ACM Comput. Surv., 36(4):335–371, December 2004.

Disponible à l'adresse
http://portal.acm.org/citation.cfm?id=1041681.

[13] Gabriel ANTONIU, Luc BOUGÉ, Mathieu JAN et Sébastien MONNETT.
Large-scale deployment in P2P experiments using the JXTA distributed framework.
Rapport technique, JXTA - jdf.jxta.org, 2004.

[14] Luciano BARESI, Carlo GHEZZI et Luca MOTTOLA.
On accurate automatic verification of publish-subscribe architectures.
Dans *29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May 20-26, 2007*, pages 199–208. IEEE Computer Society, 2007.

[15] Luciano BARESI et Michal YOUNG.
Test oracles.
Technical Report CIS-TR-01-02, University of Oregon, Dept. of Computer and Information Science, Eugene, Oregon, U.S.A., August 2001.
http://www.cs.uoregon.edu/~michal/pubs/oracles.html.

[16] David BARKAI.
Peer-to-Peer Computing: Technologies for Sharing and Collaborating on the Net.
Intel Press, 2001.

[17] Rudolf BAYER et Edward M. MCCREIGHT.
Organization and maintenance of large ordered indices.
Acta Inf., 1:173–189, 1972.

[18] Axel BELINFANTE, Jan FEENSTRA, René G. de VRIES, Jan TRETMANS, Nicolae GOGA, Loe FEIJS, Sjouke MAUW et Lex HEERINK.
Formal test automation: A simple experiment.
Dans G. CSOPAKI, S. DIBUZ et K. TARNAY, réds., 12^{th} *Int. Workshop on Testing of Communicating Systems*, pages 179–196. Kluwer Academic Publishers, 1999.

[19] William BERG, Marshall CLINE et Mike GIROU.
Lessons learned from the os/400 oo project.
Commun. ACM, 38(10):54–64, 1995.

[20] Robert V. BINDER.
Testing object-oriented systems: models, patterns, and tools.
Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[21] Frank BUSCHMANN, Regine MEUNIER, Hans ROHNERT, Peter SOMMERLAD et Michael STAL.
Pattern-oriented software architecture: a system of patterns.
John Wiley & Sons, Inc., *New York‹ NY‹* 1996.

[22] Antonio CARZANIGA, David S. ROSENBLUM et Alexander L. WOLF.
Achieving scalability and expressiveness in an internet-scale event notification service.
Dans *PODC '00: Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*, pages 219–227, New York, NY, USA, 2000. ACM.

[23] Yatin CHAWATHE, Sylvia RATNASAMY, Lee BRESLAU, Nick LANHAM et Scott SHENKER.
Making gnutella-like p2p systems scalable.
Dans Anja FELDMANN, Martina ZITTERBART, Jon CROWCROFT et David WETHERALL, réds., *Proceedings of the ACM SIGCOMM 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication, August 25-29, 2003, Karlsruhe, Germany*, pages 407–418. ACM, 2003.

[24] Kai CHEN, Fan JIANG et Chuan dong HUANG.
A new method of generating synchronizable test sequences that detect output-shifting faults based on multiple uio sequences.
Dans *SAC*, pages 1791–1797, 2006.

[25] Wen-Huei CHEN et Hasan URAL.
Synchronizable test sequences based on multiple uio sequences.
IEEE/ACM Trans. Netw., 3(2):152–157, 1995.

[26] Walfredo CIRNE, Francisco BRASILEIRO, Nazareno ANDRADE, Lauro COSTA, Alisson ANDRADE, Reynaldo NOVAES et Miranda MOWBRAY.
Labs of the world, unite!!!
Journal of Grid Computing, 4(3):225–246, 2006.

[27] Ian CLARKE, Oskar SANDBERG, Brandon WILEY et Theodore W. HONG.
Freenet: A distributed anonymous information storage and retrieval system.
Lecture Notes in Computer Science, 2009, 2001.
Disponible à l'adresse
`citeseer.ist.psu.edu/clarke00freenet.html`.

[28] TPC-Transaction Processing Performance COUNCIL.
Disponible à l'adresse
`http://www.tpc.org/`.

[29] Arturo CRESPO et Hector GARCIA-MOLINA.
Routing indices for peer-to-peer systems.
Dans *Proceedings of the International Conference on Distributed Computing Systems (ICDCS). July 2002*, pages 23–, 2002.

[30] Daniel Paranhos da SILVA, Walfredo CIRNE et Francisco Vilar BRASILEIRO.
Trading cycles for information: Using replication to schedule bag-of-tasks applications on computational grids.
Dans Harald KOSCH, László BÖSZÖRMÉNYI et Hermann HELLWAGNER, réds., *Euro-Par*, volume 2790 de *Lecture Notes in Computer Science*, pages 169–180. Springer, 2003.

[31] Frank DABEK, Emma BRUNSKILL, M. Frans KAASHOEK, David KARGER, Robert MORRIS, Ion STOICA et Hari BALAKRISHNAN.
Building peer-to-peer systems with chord, a distributed lookup service.
8th Workshop on Hot Topics in Operating Systems, 2001.

[32] P. DAUPHIN.
Knowledge bases in debugging parallel and distributed systems based on event traces.
Dans *Proceedings of the 3rd ACM/ONR Workshop on Parallel and Distributed Debugging*, 1993.

[33] Eduardo Cunha de ALMEIDA, Gerson SUNYÉ, Yves Le TRAON et Patrick VALDURIEZ.
A framework for testing peer-to-peer systems.
Dans *19th International Symposium on Software Reliability Engineering (ISSRE 2008), 11-14 November 2008, Redmond, Seattle, USA*. IEEE Computer Society, 2008.

[34] Eduardo Cunha de ALMEIDA, Gerson SUNYÉ, Yves Le TRAON et Patrick VALDURIEZ.
Testing peers' volatility.
Dans *23nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008), September 15-19, 2008, L'Aquila, Italy*, 2008.

[35] Eduardo Cunha de ALMEIDA, Gerson SUNYÉ et Patrick VALDURIEZ.
Action synchronization in p2p system testing.

Dans Anne DOUCET, Stéphane GANÇARSKI et Esther PACITTI, réds., *Proceedings of the 2008 International Workshop on Data Management in Peer-to-Peer Systems, DaMaP 2008, Nantes, France, March 25, 2008*, ACM International Conference Proceeding Series, pages 43–49. ACM, 2008.

[36] Eduardo Cunha de ALMEIDA, Gerson SUNYÉ et Patrick VALDURIEZ.
Testing architectures for large scale grids.
Dans *International Workshop on High-Performance Data Management in Grid Environments, HPDGrid, Toulouse, France, June 24, 2008*, 2008.

[37] René G. de VRIES et Jan TRETMANS.
On-the-fly conformance testing using spin.
International Journal on Software Tools for Technology Transfer (STTT), 2(4):382–393, 2000.

[38] René G. de VRIES DE et Jan TRETMANS.
On-the-fly conformance testing using SPIN.
Dans G. HOLZMANN, E. NAJM et A. SERHROUCHNI, réds., *Fourth Workshop on Automata Theoretic Verification*, ENST 98 S 002, pages 115–128. Ecole Nationale Supérieure des Télécommunications, November 2, 1998.

[39] Mayur DESHPANDE et Nalini VENKATASUBRAMANIAN.
The different dimensions of dynamicity.
Dans *P2P '04: Proceedings of the Fourth International Conference on Peer-to-Peer Computing*, pages 244–251, Washington, DC, USA, 2004. IEEE Computer Society.

[40] Roger DINGLEDINE, Michael J. FREEDMAN et David MOLNAR.
The Free Haven Project: Distributed anonymous storage service.
Dans Hannes FEDERRATH, réd., *Designing Privacy Enhancing Technologies, International Workshop on Design Issues in Anonymity and Unobservability, Berkeley, CA, USA, July 25-26, 2000, Proceedings*, volume 2009 de *Lecture Notes in Computer Science*, pages 67–95. Springer, 2000.

[41] Bruce Powel DOUGLASS.
Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems.
Addison Wesley Professional, 2002.

[42] Florin DRAGAN, Bogdan BUTNARU, Ioana MANOLESCU, Georges GARDARIN, Nicoleta PREDA, Benjamin NGUYEN, Radu POP et Laurent YEH.
P2PTester: a tool for measuring P2P platform performance.
Dans *A demonstration in BDA conference*, 2006.

[43] Peter DRUSCHEL et Antony I. T. ROWSTRON.
PAST: A large-scale, persistent peer-to-peer storage utility.
Dans *Proceedings of HotOS-VIII: 8th Workshop on Hot Topics in Operating Systems, May 20-23, 2001, Elmau/Oberbayern, Germany*, pages 75–80. IEEE Computer Society, 2001.

[44] Alexandre DUARTE, Walfredo CIRNE, Francisco BRASILEIRO et Patricia MACHADO.
Using the computational grid to speed up software testing.
Dans *Proceedings of the 19th Brazilian Symposium on Software Engineer.*, 2005.

[45] Alexandre DUARTE, Walfredo CIRNE, Francisco BRASILEIRO et Patricia MACHADO.
Gridunit: software testing on the grid.
Dans *ICSE '06: Proceeding of the 28th international conference on Software engineering*, pages 779–782, New York, NY, USA, 2006. ACM Press.

[46] Martin S. FEATHER.

Rapid application of lightweight formal methods for consistency analyses.
IEEE Trans. Softw. Eng., 24(11):948–959, 1998.

[47] Martin S. FEATHER et Ben SMITH.
Automatic generation of test oracles-from pilot studies to application.
Dans *ASE '99: Proceedings of the 14th IEEE international conference on Automated software engineering*, page 63, Washington, DC, USA, 1999. IEEE Computer Society.

[48] Presanna GANESAN et Hector GARCIA-MOLINA.
Efficient queries in peer-to-peer systems.
IEEE - Bulletin of the Technical Committee on Data Engineering, 2005.

[49] David GARLAN, Serge KHERSONSKY et Jung Soo KIM.
Model checking publish-subscribe systems.
Dans Thomas BALL et Sriram K. RAJAMANI, réds., *Model Checking Software, 10th International SPIN Workshop. Portland, OR, USA, May 9-10, 2003, Proceedings*, volume 2648 de *Lecture Notes in Computer Science*, pages 166–180. Springer, 2003.

[50] Júlio GERCHMAN, Gabriela JACQUES-SILVA, Roberto Jung DREBES et Taisy Silva WEBER.
Ambiente distribuido de injeçao de falhas de comunicaçao para teste de aplicaçoes java de rede.
SBES, 2005.

[51] GNU NANA.
Disponible à l'adresse
http://www.gnu.org/software/nana/nana.html.

[52] Todd L. GRAVES, Mary Jean HARROLD, Jung-Min KIM, Adam PORTER et Gregg ROTHERMEL.
An empirical study of regression test selection techniques.
ACM Trans. Softw. Eng. Methodol., 10(2):184–208, 2001.

[53] Ceki GULCU.
The Complete Log4j Manual.
QOS.ch, 2003.

[54] Robert M. HIERONS.
Testing a distributed system: generating minimal synchronised test sequences that detect output-shifting faults.
Information and Software Technology, 43(9):551–560, 2001.

[55] William HOARAU, Sébastien TIXEUIL et Fabien VAUCHELLES.
Fault injection in distributed java applications.
Dans *20th International Parallel and Distributed Processing Symposium (IPDPS 2006), Proceedings, 25-29 April 2006, Rhodes Island, Greece*, 2006.

[56] Ryan HUEBSCH, Joseph M. HELLERSTEIN, Nick LANHAM, Boon Thau LOO, Scott SHENKER et Ion STOICA.
Querying the internet with PIER.
Dans *VLDB*, pages 321–332, 2003.

[57] Daniel HUGHES, Phil GREENWOOD et Geoff COULSON.
A framework for testing distributed systems.
Dans *P2P '04: Proceedings of the Fourth International Conference on Peer-to-Peer Computing*, pages 262–263, Washington, DC, USA, 2004. IEEE Computer Society.

[58] Claude JARD.
Principles of distribute test synthesis based on true-concurrency models.
Rapport technique, IRISA/CNRS, 2001.

[59] Claude JARD et Thierry JÉRON.
TGV: theory, principles and algorithms: A tool for the automatic synthesis of conformance test cases for non-deterministic reactive systems.
Int. J. Softw. Tools Technol. Transf., 2005.

[60] Christophe JOUBERT et Radu MATEESCU.
Distributed on-the-fly model checking and test case generation.
Dans Antti VALMARI, réd., *Model Checking Software, 13th International SPIN Workshop, Vienna, Austria, March 30 - April 1, 2006, Proceedings*, volume 3925 de *Lecture Notes in Computer Science*, pages 126–145. Springer, 2006.

[61] JTIGER.
Disponible à l'adresse
http://jtiger.org/.

[62] JUNIT.
Junit website. http://www.junit.org, 2006.

[63] Vana KALOGERAKI, Dimitrios GUNOPULOS et Demetrios ZEINALIPOUR-YAZTI.
A local search mechanism for peer-to-peer networks.
Dans *Proceedings of the 2002 ACM CIKM International Conference on Information and Knowledge Management, McLean, VA, USA, November 4-9, 2002*, pages 300–307. ACM, 2002.

[64] Gregory M. KAPFHAMMER.
Automatically and transparently distributing the execution of regression test suites.
Dans *Proceedings of the 18th International Conference on Testing Computer Software*, Washington, D.C., June 2001.

[65] KAZAA.
Kazaa website. http://www.kazaa.com., 2008.

[66] Jung-Min KIM et Adam PORTER.
A history-based test prioritization technique for regression testing in resource constrained environments.
Dans *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 119–129, New York, NY, USA, 2002. ACM.

[67] Wilburt Juan LABIO, Janet L. WIENER, Hector GARCIA-MOLINA et Vlad GORELIK.
Efficient resumption of interrupted warehouse loads.
SIGMOD Rec., 29(2):46–57, 2000.

[68] Qinghu LI, Jianmin WANG et Jia-Guang SUN.
Gfs-btree: A scalable peer-to-peer overlay network for lookup service.
Dans *GCC (1)*, pages 340–347, 2003.

[69] Yunhao LIU, Xiaomei LIU, Li XIAO, Lionel M. NI et Xiaodong ZHANG.
Location-aware topology matching in p2p systems.
Dans *INFOCOM*, 2004.

[70] Sun Microsystems Java Logging API.
Disponible à l'adresse
http://java.sun.com/j2se/1.4.2/docs/guide/util/logging/.

[71] Brad LONG et Paul A. STROOPER.
A case study in testing distributed systems.
Dans *Proceedings 3rd International Symposium on Distributed Objects and Applications (DOA'01)*, pages 20–30, 2001.

[72] Qin LV, Pei CAO, Edith COHEN, Kai LI et Scott SHENKER.
Search and replication in unstructured peer-to-peer networks.
Dans *ICS '02: Proceedings of the 16th international conference on Supercomputing*, pages 84–95, New York, NY, USA, 2002. ACM.

[73] Zohar MANNA et Richard J. WALDINGER.
The logic of computer programming.
IEEE Trans. Software Eng., 4(3):199–229, 1978.

[74] Aviel D. Rubin MARC WALDMAN et Lorrie Faith CRANOR.
Publius: A robust, tamper-evident, censorship-resistant, web publishing system.
Dans *Proc. 9th USENIX Security Symposium*, pages 59–72, August 2000.
Disponible à l'adresse
citeseer.ist.psu.edu/waldman00publius.html.

[75] Eliane MARTINS, Ana Maria AMBROSIO et Maria de FÁTIMA MATTIELLO-FRANCISCO.
Atifs: a testing toolset with software fault injection.
Proceedings of York Computer Science Yellow Report 2003 - Workshop SofTest: UK Testing Research II - Department of Computer Science at York, 2003.

[76] Eliane MARTINS et Maria de FÁTIMA MATTIELLO-FRANCISCO.
A tool for fault injection and conformance testing of distributed systems.
Dans Rogério de LEMOS, Taisy Silva WEBER et João Batista Camargo JR., réds., *LADC*, volume 2847 de *Lecture Notes in Computer Science*, pages 282–302. Springer, 2003.

[77] Vidal MARTINS.
Data Replication in P2P Systems.
Thèse de Doctorat, Université de Nantes, May 2007.

[78] Petar MAYMOUNKOV et David MAZIÈRES.
Kademlia: A peer-to-peer information system based on the xor metric.
Dans Peter DRUSCHEL, M. Frans KAASHOEK et Antony I. T. ROWSTRON, réds., *Peer-to-Peer Systems, First International Workshop, IPTPS 2002, Cambridge, MA, USA, March 7-8, 2002, Revised Papers*, volume 2429 de *Lecture Notes in Computer Science*, pages 53–65. Springer, 2002.

[79] Bertrand MEYER.
Applying "design by contract".
IEEE Computer, 25(10):40–51, 1992.

[80] Elke MICHLMAYR.
Ant Algorithms for Self-Organization in Social Networks.
Thèse de Doctorat, Vienna University of Technology, May 2007.

[81] Wolfgang NEJDL, Wolf SIBERSKI et Michael SINTEK.
Design issues and challenges for rdf- and schema-based peer-to-peer systems.
SIGMOD Record, 32(3):41–46, 2003.

[82] Wolfgang NEJDL, Boris WOLF, Changtao QU, Stefan DECKER, Michael SINTEK, Ambjörn NAEVE, Mikael NILSSON, Matthias PALMÉR et Tore RISCH.
Edutella: a p2p networking infrastructure based on rdf.
Dans *WWW*, pages 604–615, 2002.

[83] Alexandre PETRENKO et Andreas ULRICH.
Verification and testing of concurrent systems with action races.

Dans Hasan URAL, Robert L. PROBERT et Gregor von BOCHMANN, réds., *Testing of Communicating Systems: Tools and Techniques, IFIP TC6/WG6.1 13th International Conference on Testing Communicating Systems (TestCom 2000), August 29 - September 1, 2000, Ottawa, Canada*, volume 176 de *IFIP Conference Proceedings*, pages 261–280. Kluwer, 2000.

[84] Simon PICKIN, Claude JARD, Thierry HEUILLARD, Jean-Marc JÉZÉQUEL et Philippe DESFRAY..
A uml-integrated test description language for component testing.
Dans *UML2001 workshop: Practical UML-Based Rigorous Development Methods*, Lecture Notes in Informatics (LNI), pages 208–223. Bonner Köllen Verlag, October 2001.

[85] Simon PICKIN, Claude JARD, Yves LE TRAON, Thierry JÉRON, Jean-Marc JÉZÉQUEL et Alain LE GUENNEC.
System test synthesis from UML models of distributed software.
2529:97–113, 2002.

[86] Sylvia RATNASAMY, Paul FRANCIS, Mark HANDLEY, Richard KARP et Scott SCHENKER.
A scalable content-addressable network.
Dans *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 161–172, New York, NY, USA, 2001. ACM.

[87] David S. ROSENBLUM.
A practical approach to programming with assertions.
IEEE Trans. Software Eng., 21(1):19–31, 1995.

[88] Antony ROWSTRON et Peter DRUSCHEL.
Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems.
Dans *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, nov 2001.

[89] Ina SCHIEFERDECKER et Jens GRABOWSKI.
Conformance testing with TTCN.
Telektronikk ISSN 0085-7130, 96(4):85–95, 2000.

[90] Ina SCHIEFERDECKER, Mang LI et Andreas HOFFMANN.
Conformance testing of tina service components - the ttcn/ corba gateway.
Dans *IS&N*, pages 393–408, 1998.

[91] Subhabrata SEN et Jia WANG.
Analyzing peer-to-peer traffic across large networks.
IEEE/ACM Trans. Netw., 12(2):219–232, 2004.

[92] Ion STOICA, Robert MORRIS, David KARGER, M. Frans KAASHOEK et Hari BALAKRISHNAN.
Chord: A scalable peer-to-peer lookup service for internet applications.
Dans *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 149–160, New York, NY, USA, 2001. ACM.

[93] Michael STONEBRAKER, Eric N. HANSON et Chin-Heng HONG.
The design of the postgres rules system.
Dans *Proceedings of the Third International Conference on Data Engineering*, pages 365–374, Washington, DC, USA, 1987. IEEE Computer Society.

[94] TESTNG.
Disponible à l'adresse
http://www.testng.org/.

[95] Yves Le TRAON, Benoit BAUDRY et Jean-Marc JÉZÉQUEL.
Design by contract to improve software vigilance.
IEEE Trans. Software Eng., 32(8):571–586, 2006.

[96] Jan TRETMANS et Axel BELINFANTE.
Automatic testing with formal methods.
Dans *EuroSTAR'99: 7^{th} European Int. Conference on Software Testing, Analysis & Review*, Barcelona, Spain, November 8–12, 1999. EuroStar Conferences, Galway, Ireland.

[97] A. ULRICH, H. HALLAL, A. PETRENKO et S. BORODAY.
Verifying trustworthiness requirements in distributed systems with formal log-file analysis.
Dans *HICSS '03: Proceedings of the 36th Annual Hawaii International Conference on System Sciences (HICSS'03) - Track 9*, page 337.2, Washington, DC, USA, 2003. IEEE Computer Society.

[98] Andreas ULRICH et Hartmut KONIG.
Architectures for testing distributed systems.
Dans *Proceedings of the IFIP TC6 12th International Workshop on Testing Communicating Systems*, pages 93–108, Deventer, The Netherlands, 1999. Kluwer, B.V.

[99] Erik van VEENENDAAL.
Standard glossary of terms used in Software Testing, September 2005.

[100] Kristen R. WALCOTT, Mary Lou SOFFA, Gregory M. KAPFHAMMER et Robert S. ROOS.
Timeaware test suite prioritization.
Dans Lori L. POLLOCK et Mauro PEZZÈ, réds., *ISSTA*, pages 1–12. ACM, 2006.

[101] Thomas WALTER, Ina SCHIEFERDECKER et Jens GRABOWSKI.
Test architectures for distributed systems: State of the art and beyond.
Dans Alexandre PETRENKO et Nina YEVTUSHENKO, réds., *IWTCS*, volume 131 de *IFIP Conference Proceedings*, pages 149–174. Kluwer, 1998.

[102] Jennifer WIDOM.
The starburst active database rule system.
IEEE Transactions on Knowledge and Data Engineering, 08(4):583–595, 1996.

[103] W. Eric WONG, Joseph R. HORGAN, Saul LONDON et Hira Agrawal BELLCORE.
A study of effective regression testing in practice.
Dans *ISSRE '97: Proceedings of the Eighth International Symposium on Software Reliability Engineering (ISSRE '97)*, page 264, Washington, DC, USA, 1997. IEEE Computer Society.

[104] Tao XIE.
Augmenting automatically generated unit-test suites with regression oracle checking.
Dans Dave THOMAS, réd., *ECOOP 2006 - Object-Oriented Programming, 20th European Conference, Nantes, France, July 3-7, 2006, Proceedings*, volume 4067 de *Lecture Notes in Computer Science*, pages 380–403. Springer, 2006.

[105] Gang XING et Michael R. LYU.
Testing, reliability, and interoperability issues in the corba programming paradigm.
Dans *6th Asia-Pacific Software Engineering Conference (APSEC '99), 7-10 December 1999, Takamatsu, Japan*, pages 530–537. IEEE Computer Society, 1999.

[106] Beverly YANG et Hector GARCIA-MOLINA.
Comparing hybrid peer-to-peer systems.
Dans *VLDB 2001, Proceedings of 27th International Conference on Very Large Data Bases, September 11-14, 2001, Roma, Italy*, pages 561–570. Morgan Kaufmann, 2001.

[107] Beverly YANG et Hector GARCIA-MOLINA.
Improving search in peer-to-peer networks.
Dans *Proceedings of the International Conference on Distributed Computing Systems (ICDCS).*, pages 5–14, 2002.

[108] Luca ZANOLIN, Carlo GHEZZI et Luciano BARESI.
An approach to model and validate publish/subscribe architectures.
Dans *SAVCBS'03 Workshop: Proceedings of the Specification and Verification of Component-Based Systems Workshop, Helsinki, Finland, Sept. 2003.*, 2003.
Disponible à l'adresse
citeseer.ist.psu.edu/zanolin03approach.html.

[109] Ben Y. ZHAO, Ling HUANG, Jeremy STRIBLING, Sean C. RHEA, Anthony D. JOSEPH et John KUBIATOWICZ.
Tapestry: a resilient global-scale overlay for service deployment.
IEEE Journal on Selected Areas in Communications, 22(1):41–53, 2004.

[110] Zhizhi ZHOU, Hao WANG, Jin ZHOU, Li TANG, Kai LI, Weibo ZHENG et Meiqi FANG.
Pigeon: a framework for testing peer-to-peer massively multiplayer online games over heterogeneous network.
Dans *Consumer Communications and Networking Conference, 2006. CCNC 2006. 3rd IEEE Volume 2, Issue , 8-10 Jan.*, pages 1028 – 1032, 2006.

Test et Validation des Systèmes Pair-à-pair

Eduardo Cunha DE ALMEIDA

Résumé

Le pair-à-pair (P2P) offre de bonnes solutions pour de nombreuses applications distribuées, comme le partage de grandes quantités de données et/ou le support de collaboration dans les réseaux sociaux. Il apparaît donc comme un puissant paradigme pour développer des applications distribuées évolutives, comme le montre le nombre croissant de nouveaux projets basés sur cette technologie.
Construire des applications P2P fiables est difficile, car elles doivent être déployées sur un grand nombre de noeuds, qui peuvent être autonomes, refuser de répondre à certaines demandes, et même quitter le système de manière inattendue. Cette volatilité des noeuds est un comportement commun dans les systèmes P2P et peut être interprétée comme une faute lors des tests.
Dans cette thèse, nous proposons un cadre et une méthodologie pour tester et valider des applications P2P. Ce cadre s'appuie sur le contrôle individuel des noeuds, permettant de contrôler précisément la volatilité des noeuds au cours de leur exécution.
Nous proposons également trois différentes approches de contrôle d'exécution de scénarios de test dans les systèmes distribués. La première approche étend le coordonnateur centralisé classique pour gérer la volatilité des pairs. Les deux autres approches permettent d'éviter le coordinateur central afin de faire passer à l'échelle l'exécution des cas de tests.
Nous avons validé le cadre et la méthodologie à travers la mise en oeuvre et l'expérimentation sur des applications P2P open-source bien connues (FreePastry et OpenChord). Les expérimentations ont permis de tester le comportement des systèmes sur différentes conditions de volatilité, et de détecter des problèmes d'implémentation complexes.

Mots-clés: testing, pair-à-pair

Testing and Validation of Peer-to-peer Systems

Abstract

Peer-to-peer (P2P) offers good solutions for many applications such as large data sharing and collaboration in social networks. Thus, it appears as a powerful paradigm to develop scalable distributed applications, as reflected by the increasing number of emerging projects based on this technology.
However, building trustworthy P2P applications is difficult because they must be deployed on a large number of autonomous nodes, which may refuse to answer to some requests and even leave the system unexpectedly. This volatility of nodes is a common behavior in P2P systems and can be interpreted as a fault during tests.
In this thesis, we propose a framework and a methodology for testing and validating P2P applications. The framework is based on the individual control of nodes, allowing test cases to precisely control the volatility of nodes during their execution.
We also propose three different architectures to control the execution of test cases in distributed systems. The first approach extends the classical centralized test coordinator in order to handle the volatility of peers. The other two approaches avoids the central coordinator in order to scale up the test cases.
We validated the framework and the methodology through implementation and experimentation on two popular open-source P2P applications (i.e. FreePastry and OpenChord). The experimentation tests the behavior of the system on different conditions of volatility and shows how the tests were able to detect complex implementation problems.

Keywords: testing, peer-to-peer

ACM Classification

Categories and Subject Descriptors : C. [**Computer Systems Organization**]: C.2 COMPUTER-COMMUNICATION NETWORKS; C.2.4 [**Distributed Systems**]: Distributed applications,Distributed databases; D. [**Software**]: D.2 SOFTWARE ENGINEERING; D.2.5 [**Testing and Debugging**]: Testing tools (e.g., data generators, coverage testing).

General Terms: Experimentation, Measurement, Performance, Reliability, Verification.

Discipline : INFORMATIQUE
Spécialité : INFORMATIQUE

Laboratoire : **L**ABORATOIRE D'**I**NFORMATIQUE DE **N**ANTES **A**TLANTIQUE.
UMR CNRS 6241. 2, rue de la Houssinière, BP 92 208 – 44 322 Nantes, CEDEX 3.