Grand défi: Résilience et Frugalité

Florence Maraninchi
Univ. Grenoble Alpes, CNRS, Grenoble INP,
VERIMAG, 38000 Grenoble, France

12 décembre 2019

Résumé

Les systèmes numériques actuels ont atteint un très grand niveau de complexité (source de bugs et d'attaques), sont très gourmands en énergie, et très dépendants de ressources extérieures distantes (source de fragilité et d'obsolescence). Après avoir détaillé ces constats et leurs causes techniques, nous proposons des pistes de recherche en logiciel pour aller vers des systèmes numériques frugaux et résilients.

1 Les constats

L'industrie informatique fonctionne suivant la loi de Moore, prophétie autoréalisatrice, selon laquelle la densité des circuits intégrés doublerait tous les deux ans, d'où une augmentation exponentielle des densités de mémoire et, longtemps, des fréquences d'horloge. L'habitude a donc été prise de réaliser des logiciels de plus en plus gourmands tant en capacité de calcul qu'en mémoire, sans que cela ne se traduise nécessairement par un service amélioré en proportion. [1] [2] Ceci fait qu'un ordinateur un peu ancien devient lent et inutilisable. Par ailleurs, les machines un tant soit peu anciennes tendent à ne plus être supportées par certains systèmes d'exploitation, tandis que des versions modernes de ces systèmes sont exigées pour certaines applications. Tout ceci incite au renouvellement périodique du parc informatique, alors même que les matériels sont encore en parfait état de marche. C'est particulièrement vrai pour les appareils nomades (*smartphones*, ordinateurs portables, etc.), pour lesquels se pose en plus la question de la durée physique du matériel, notamment des batteries, et de la difficulté à changer les pièces endommagées.

De nombreuses applications sont actuellement en ligne, et supposent la disponibilité permanente d'une liaison Internet, avec ou sans fil, mais aussi d'un ensemble de routeurs et de serveurs, parfois très éloignés. Une panne chez un gros prestataire de *cloud computing* peut avoir des conséquences importantes, dépassant ses clients immédiats, tant certains systèmes dépendent d'autres systèmes. Outre les problèmes posés par le dépôt de données auprès de tierces

1. Par exemple, Slack, une application de dialogue et de coordination entre développeurs, peut consommer plusieurs giga-octets de mémoire vive...

2. L'étude "*Rebound effects of progress in information technology*" de Lorenz et. al dans *Poiesis und Praxis* montre même que le passage à un logiciel plus récent et à du matériel plus performant peut dans certains cas dégrader l'expérience utilisateur.

parties, que ce soit par rapport à la confidentialité de données personnelles ou de secrets industriels ou aux questions de juridiction dans un contexte international, se pose d'évidentes questions de fiabilité. Celles-ci deviennent d'autant plus prégnantes que les services publics, le commerce, mais aussi des opérateurs d'intérêt vital sont connectés.

Enfin, les systèmes informatiques consomment de l'électricité, et le poids du «numérique» dans la consommation électrique mondiale n'est plus négligeable [3]. Ceci a d'autant plus d'importance que dans la plupart des pays l'électricité est produite par des centrales thermiques au charbon, donc avec dégagement de dioxyde de carbone et aggravation de l'effet de serre.

En bref, nous avons construit une infrastructure informatique coûteuse en ressources et fragile. Nous devrions au contraire tendre vers une informatique frugale et résiliente.

2 Frugalité

De nos jours, on développe rarement une application à partir de rien, mais plutôt par réutilisation d'applications et de bibliothèques existantes. Ceci a d'immenses mérites en termes de productivité, et aussi de fiabilité et sécurité (il vaut mieux, par exemple, utiliser une bibliothèque de cryptographie déjà éprouvée que produire sa propre implantation), mais a pour conséquence l'embarquement dans le système de nombreuses fonctionnalités inutiles, chaque dépendance entraînant l'inclusion en chaîne d'autres dépendances. [4] Ceci tend bien sûr à l'augmentation des demandes en capacité mémoire ou de calcul, mais aussi tend à complexifier les systèmes et à augmenter leur «surface d'attaque», et donc leur vulnérabilité.

Les langages de programmation et la façon de les exécuter ont également évolué. Sauf pour de très petits systèmes embarqués ou certaines parties très spécifiques d'un exécutif système, on ne développe plus depuis longtemps en langage machine s'exécutant directement sur machine nue. On développe de moins en moins dans des langages classiquement compilés tels que C ou C++, mais plutôt dans des langages ou *byte-code* interprétés tels que Python, Java, Javascript, .net, etc. En sus d'un système d'exploitation classique s'interposant entre l'application et la machine, on trouve souvent des machines virtuelles, des conteneurs, un hyperviseur...Bien entendu, chaque couche rajoute ses complexités, inefficacités et fragilités. Si l'on inclut des mécanismes destinés à rendre plus efficaces la traversée des couches (compilateur *just in time* dans un interpréteur, assistance de virtualisation...), on peut affaiblir les protections normalement assurées.

Enfin, la complexité des systèmes rend difficile l'analyse de leur sûreté de fonctionnement et de leur sécurité. Là où l'on avait un unique processeur, au logiciel entièrement connu, on a de multiples dispositifs reprogrammables (pro-

3. Les études récentes comme celles de *GreenIt* ou de *The Shift Project* montrent que la part du numérique dans la consommation électrique mondiale serait d'environ 5 à 10%, et surtout que les taux de croissance dans ses besoins énergétiques seraient d'environ +9% par an.

4. Par exemple, une application destinée à être déployée uniquement dans un environnement francophone pourra devoir embarquer un système de gestion des sinogrammes et les polices de caractères associées ; un système embarqué sans interface graphique pourra devoir intégrer un environnement graphique complet ; etc.

cesseur principal, processeur secondaire gérant la séquence de démarrage, etc.) au contenu non maîtrisé par l'utilisateur final.

Ces constats ont par exemple amené l'*Office of Naval Research* américain à financer des projets avec des titres évocateurs, tels que *Verified Application Debloating and Delayering* : élimination de la «mauvaise graisse», élimination des couches intermédiaires.

Peut être ces projets prennent-ils le problème une fois qu'il est trop tard. Plutôt que de chercher à éliminer a posteriori de bases de code ce qui ne sert pas en réalité, à éliminer des couches coûteuses à traverser, il faudrait plutôt produire par construction un système frugal.

3 Résilience et déconnexion

Nous avons déjà évoqué en quoi la tendance à l'obésité des systèmes est cause de complexité, de fragilité et de vulnérabilité aux attaques, et comment le recours permanent à des ressources en ligne crée également de la fragilité. L'exemple le plus criant de cette fragilité est sans doute celui d'accessoires de domotique vers lesquels les connexions passent par des serveurs tiers, de sorte que si le fournisseur fait faillite ou décide de ne plus continuer à soutenir ces matériels, ceux-ci deviennent inutilisables.

De même que les ressources matérielles et logicielles requises par une application devraient être limitées au minimum nécessaire, le recours à des ressources distantes devraient être limité, et les applications devraient être conçues pour pouvoir continuer à fonctionner hors connexion, dans la mesure du possible. Là encore, il s'agit de limiter des chaînes de dépendances. Il faudrait pouvoir garantir, lors de la construction d'un système, la maîtrise et la limitation de ses besoins extérieurs.

Les éditeurs de liens, ou, à une autre échelle, les gestionnaires de *packages* des distributions Linux, réalisent une forme rudimentaire de recherche et minimisation des dépendances. Il s'agit de systèmes à très gros grains et difficile à contrôler, qui par ailleurs ont souvent des difficultés à gérer des versions multiples de composants. Quant aux dépendances vers des systèmes en ligne, il n'y a pas d'outil standard pour les gérer.

4 Programme de recherche

De nombreuses pistes de recherche peuvent être envisagées, adoptant des points de vue plus ou moins radicaux sur la résolution des problèmes énoncés plus haut :

1. Concevoir des outils d'**analyse et "nettoyage"** de logiciels existants, pour aller vers plus de frugalité et de déconnexion. Techniquement, cette piste peut être vue comme une nouvelle instance de la validation (formelle ou pas), pour de nouveaux critères. Elle a vocation à s'appliquer à l'existant, et implique donc d'étudier les infrastructures logicielles et matérielles actuelles afin de déterminer leurs sources d'embonpoint et de de fragilité.
2. Concevoir des outils à intégrer dans des **processus de développement** de nouvelles applications (langages de programmation éventuel-

lement dédiés, compilateurs, alternatives aux couches système) pour garantir des solutions complètes plus frugales et résilientes. Cette piste peut contribuer à améliorer les nouveaux logiciels, sans nécessairement tout reprendre à zéro.

3. **Faire table rase** des structures très complexes des systèmes informatiques actuels, en partant du point de vue que la complexité est en partie un héritage du passé. Le but est donc de repenser les solutions techniques "de zéro", en intégrant les objectifs de frugalité et de résilience dans une réflexion plus large sur le rôle du numérique dans nos systèmes socio-techniques et sur les usages et les besoins. Cette piste peut paraître utopique, en particulier non économiquement viable, mais elle est scientifiquement intéressante, ne serait-ce que pour permettre de mesurer la distance entre la complexité de certains de nos systèmes actuels et une solution en quelque sorte minimale, libérée du poids de la compatibilité ascendante et de l'historique. Si la complexité vient en partie au moins de la complexité des architectures matérielles, on pourra même s'autoriser à travailler sur des architectures simples, éventuellement en utilisant des simulateurs si le matériel n'existe pas. Il s'agit donc d'appliquer au logiciel, et pour les propriétés de frugalité et déconnexion, une approche qui a déjà été appliquée au matériel pour la propriété de prédictibilité temporelle dans le projet `http://compsoc.eu/`.