D1.4-a Rapport d'implémentation du module de gestion des m-traces

Pierre-Antoine Champin
Damien Clauzel
Yannick Prié
Karim Sehaba
{Pierre-Antoine.Champin, Damien.Clauzel, Yannick.Prie, Karim.Sehaba}@liris.cnrs.fr

janvier 2009

Table des matières

1 Introduction

Nous appelons *système à base de traces* (ou SBT) un environnement informatique mémorisant les actions de l'utilisateur sous la forme de *traces modélisées* (ou m-traces). Ces traces peuvent ensuite être exploitées par l'utilisateur lui même, un agent logiciel, ou des tierces personnes. Dans le cadre d'une activité collaborative synchrone, qui nous intéresse plus particulièrement dans le projet Ithaca, ces tierces personnes peuvent notamment être les autres participants de l'activité collaborative : nous envisageons les SBT comme un outils permettant d'améliorer la collaboration, notamment par le *partage de traces individuelles* entre les participants, et la construction de *traces de groupe* permettant de mettre en évidence, voire de découvrir, des comportements collectifs.

Après un bref rappel sur les notions relatives aux traces modélisées, telles qu'elles sont développées dans l'équipe SILEX, ce rapport décrit l'architecture logicielle d'un SBT, et spécifie les différents composants logiciels nécéssaires à sa mise en œuvre.

2 Rappels sur les fondements théoriques

Une *trace modélisée*, ou m-trace, est une collection d'*observés* temporellement situés, conforme à un *modèle de trace*. Tout observé possède un type, deux bornes temporelles (entiers), un sujet et un ensemble d'attributs. Il est également relié à d'autres observés de la même trace par des relations binaires orientées.

Un modèle de trace défini un ensemble de *types d'observés*, avec pour chacun les attributs qu'il peut posséder, et les relations auxquelles il peut participer. Le modèle de trace définit également le *domaine temporel*, qui spécifie comment les bornes temporelles des observés doivent être interprétées.

3 Architecture logicielle d'un système à base de trace

Un système à base de traces peut-être décomposé en 5 types de composants (cf. figure 3).

Le *système observé* englobe tous les éléments qui présentent un intérêt pour le SBT et qui lui sont techniquement accessibles. Ainsi, les interactions de l'utilisateur avec une ou plusieurs applications informatiques seront des éléments typiques du système observé.

Le *noyau* du SBT (ou noyau SBT) implémente toutes les fonctions liées au stockage et à la gestion des traces modélisées, mais aussi de leurs modèles, des requêtes et des transformations. C'est un composant génériques, indépendant du système observé (avec lequel il n'a d'ailleurs aucune interaction directe), et donc réutilisable.

L'interface entre le noyau SBT et le système observé est assuré par les *agents de collecte*. Ce sont des composants logiciels qui captent les informations pertinentes du système observé, les interprètent conformément à un modèle de trace, et les transmettent au noyau sous la forme d'un *flux de traçage*. Les agents de collecte sont donc spécifique à un système (ou sous-système) observé particulier.

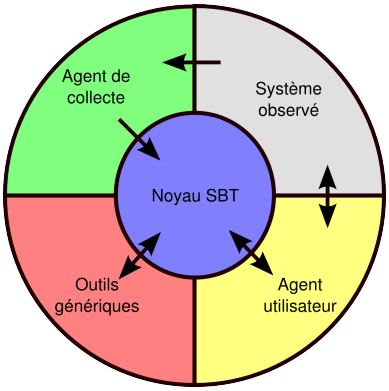

FIG. 1 – Vue d'ensemble de l'architecture d'un SBT

L'exploitation des traces est quant à elle assurée par les *agents utilisateurs*. Ces composants logiciels utilisent les traces stockées par le noyau SBT pour remplir diverses fonctions : visualisation de la trace, partage avec d'autres utilisateurs, assistance à l'utilisateur, etc. Une catégorie particulière d'agents utilisateur est celle des *agents de pilotage*, qui sont capables de rejouer (éventuellement en l'adaptant), une portion de trace dans le système observé. Comme les agents de collecte, les agents utilisateurs sont spécifiques à un système observé particulier.

Enfin, les *outils génériques* sont similaires aux agents utilisateurs, en ce sens qu'ils permettent d'interagir avec le noyau SBT. En revanche, contrairement aux précédents, ils sont indépendant du système observé, et offrent donc des fonctionnalités génériques, de plus haut niveau, et donc moins « adaptées ». En contrepartie, ces outils sont réutilisables, à l'instar du noyau lui même.

Un exemple de système observé mono-utilisateur serait le couple d'applications Navigateur + Courrier électronique. Un agent observateur serait un logiciel spécifique traçant les interactions de l'utilisateur avec ces deux applications. Un agent utilisateur pourrait être un assistant reconnaissant, par exemple, qu'une page précédemment envoyée à un contact est similaire à la page courante, et proposant d'envoyer également cette dernière.

Dans le cadre de l'activité collaborative synchrone, chaque utilisateur utilise son propre SBT, mais les systèmes observés s'intersectent (cf. figure 3). En plus de la collaboration habituellement permise par les applications constituant les systèmes observés, des agents utilisateurs spécialisés peuvent favoriser de nouvelles formes de collaboration. Par exemple, un utilisateur peut partager une partie de ses traces avec un autre, lui permettant de profiter de son expérience. Par exemple, si l'utilisateur 1 partage ses traces avec l'utilisateur 2, et envoie par ailleurs une page à l'utilisateur 3, l'utilisateur 2 pourra se voir suggérer, en visitant une page similaire, de l'envoyer également 3.

Bien sûr, cette possibilité de partager ses traces suppose un contrôle absolu de l'utilisateur quant à ce qu'ils souhaite effectivement partager avec les autres et ce qu'ils souhaitent garder privé.

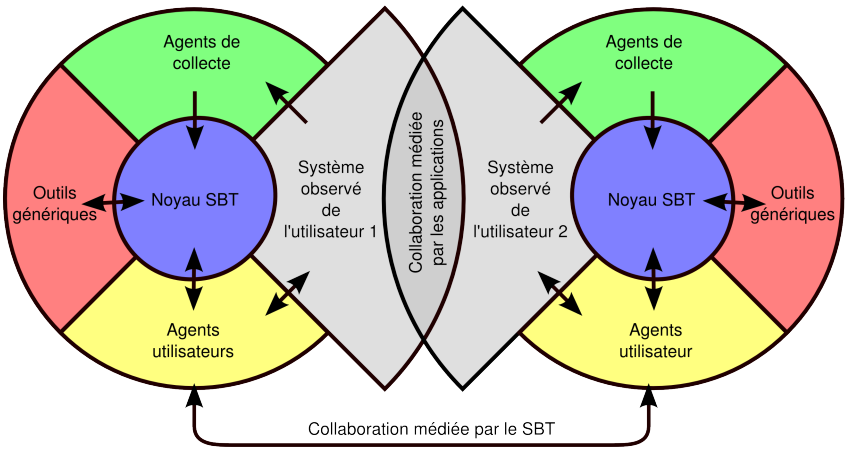

FIG. 2 – Deux SBT dans une situation de collaboration synchrone

4 Noyau SBT

4.1 Fonctionnalités du Noyau SBT

Le noyau SBT est, comme son nom l'indique, le cœur du SBT (il est parfois également appelé système de gestion de base de traces, ou SGBT). Tous les autres composants sont spécifiés *par rapport* aux interactions qu'ils ont avec le noyau SBT. (NB : les interactions du système observé avec les agents collecteurs d'une part, et avec les agents de pilotage d'autre part, sont spécifiques au système observé et ne sont donc pas l'objet de cette spécification). Le noyau SBT comprend les fonctionnalités suivantes :

- stockage et gestion
  - des traces
    - création et suppression
    - collecte (pour les traces premières)
    - calcul (pour les traces transformées)
    - amendement d'une trace par son propriétaire
    - import et export dans un format standard
  - des modèles de traces
    - création et suppression
    - modification
    - import et export dans un format standard
  - des requêtes sur les traces
    - création et suppression
    - exécution ponctuelle
    - exécution continuelle (*streaming*)
    - exécution sur plusieurs traces à la fois
    - import et export dans un format standard
  - des transformations de traces
    - création et suppression
    - exécution (lié au calcul de trace transformée)

    - import et export dans un format standard
- gestion des agents collecteurs
- gestion des agents utilisateurs
- gestion des utilisateurs et des permissions (transversal à toutes les autres fonctions)
- interrogation du noyau SBT sur les éléments qu'il gère (traces, modèles, requêtes, etc...)

4.2 Architecture générale et méthodologie

On se basera autant que possible sur des technologies existantes et matures pour définir l'architecture du noyau SBT.

Protocoles. Afin de répondre à divers besoins, le noyau SBT sera accessible *via* deux protocoles standards. Toutes les opérations seront disponibles avec le protocole HTTP [4], plus précisément à l'aide de services RESTful [**?**]. Les opérations nécessitant un mode de communication de type *push* et une faible latence pourront avoir recours au protocole XMPP [7].

Authentification et confidentialité. La nature personnelle des traces en fait des informations très sensibles. Le noyau SBT doit donc offrir un mécanisme d'authentification robuste et une possibilité de chiffrer les données pour assurer leur confidentialité, dans le cas ou elles circuleraient sur un canal non sûr (comme par exemple Internet). Les deux protocoles que nous souhaitons utiliser disposent de tels mécanismes, avec des implémentations matures et éprouvées. Nous nous appuierons donc sur les mécanismes offerts par ces protocoles pour assurer ces fonctionnalités.

Représentation des données. On utilisera le modèle de données RDF [5]. Ce modèle est en effet bien adapté à la nature dynamique et ouverte des données que le noyau SBT est appelé à manipuler. Il est muni d'un langage de requête et de transformation, SPARQL [6] que nous utiliseront pour décrire les requêtes et les transformations de trace. Il dispose de plusieurs syntaxes ; notre implémentation en supportera deux : la syntaxe standard, basée sur XML, et la syntaxe Turtle [1], plus compacte et facile à rédiger et à lire.

L'utilisation de RDF suppose la définition d'un vocabulaire spécifique. Le vocabulaire KTBS, décrit en annexe A.1, définit tous les termes (classes, propriétés et instances) nécessaires aux opérations sur le noyau SBT. Tous les termes de ce vocabulaire sont écrit en police sans serif (exemple : Trace).

Développement itératif. Afin de disposer le plus rapidement possible d'un prototype utilisable, nous proposons de le développer de manière itérative. La première itération fournira les fonctionnalités basiques du noyau SBT sur le protocole HTTP, même si certaines de ces fonctionnalités sont assurées par des outils externes génériques. Elle est spécifiée en détail dans ce document. La deuxième itération complétera les fonctionnalités du SBT conformément à la liste dressée au début de cette section, et fournira un premier support du protocole XMPP. Elle est également spécifiée dans ce document, quoique de manière moins détaillée. La troisième itération et les suivantes se focaliseront

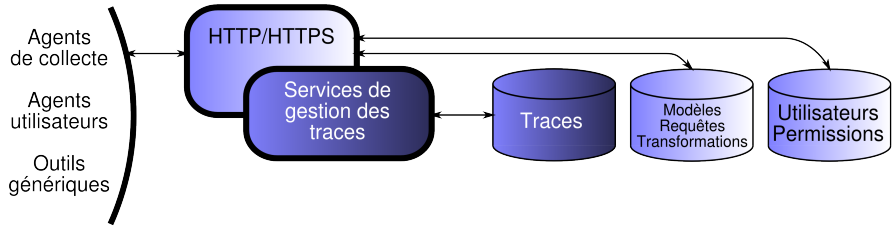

FIG. 3 – Architecture technique du noyau SBT à l'itération 1. Les composants au fond clair sont les composants standard. Les composants au fond sombre sont les composants spécifiques.

sur l'optimisation et des extensions de fonctionnalité suggérées par les retour d'expérience.

4.3 Itération 1

Cette itération se focalise sur les services RESTful d'accès au noyau, utilisant le protocole HTTP. Ce style architectural préconise que chaque ressource soit identifiée par un URI, et manipulée par les opérations HTTP standard `GET` pour la lecture, `POST` pour la création, `PUT` pour la modification et `DELETE` pour la suppression.

Dans la suite, nous utiliserons par convention `http://host/sbt/` comme URI de base du noyau SBT. Lorsqu'un URI contient un élément entre accolades, cet élément doit être remplacé par une valeur significative pour le système :

- {`uid`} : un identifiant d'utilisateur
- {`tid`} : un identifiant de trace
- {`mid`} : un identifiant de modèle
- {`qid`} : un identifiant de requête
- {`tfid`} : un identifiant de transformation

Les identifiants ne peuvent comporter que des caractères alphanumériques ou le caractère _ (blanc souligné).

4.3.1 Gestion des modèles de trace, requêtes et transformations

Comme on le voir dans la figure 4.3, les modèles de traces, requêtes et transformations sont gérés en utilisant uniquement les composants standard à l'itération 1. Cela signifie qu'ils sont stockés sous formes de fichiers statiques accessibles directement par le protocole HTTP, selon le modèle d'URI suivant :

Modèle de trace	`http://host/sbt/{uid}/models/{mid}`
Requête	`http://host/sbt/{uid}/queries/{qid}`
Transformations	`http://host/sbt/{uid}/transformations/{tfid}`

Le propriétaire des fichiers peut décider, en configurant le serveur HTTP de manière adéquate[1], de rendre ces fichiers visibles aux autres utilisateurs.

[1] fichier `.htaccess` pour le cas du serveur Apache

Le contenu du fichier accessible à l'URI d'un modèle de traces décrit en RDF le modèle lui-même, en utilisant le vocabulaire KTBS. Il permet de définir le domaine temporel, les types d'observés et de relations, les attributs, les domaines et co-domaines des attributs et relations, ainsi que les relations de spécialisation entre les types. La description d'un modèle peut également *importer* un autre modèle. Ceci doit être interprété comme une inclusion, dans le modèle importateur, de toutes les informations présentes ou importées dans l'autre modèle. Un exemple de description de modèle de traces est donné en annexe A.2.

Le contenu du fichier accessible à l'URI d'une requête spécifie cette requête en utilisant la syntaxe SPARQL, en faisant référence au vocabulaire KTBS ainsi qu'au modèle visé par la requête. La requête doit nécessairement être de type `SELECT`. Un exemple de telle requête est donné en annexe A.3.

Le contenu du fichier accessible à l'URI d'une transformation spécifie cette transformation en utilisant la syntaxe SPARQL, en faisant référence au vocabulaire KTBS ainsi qu'aux modèles source et destination de la transformation. La requête doit nécessairement être de type `CONSTRUCT`. Un exemple de telle transformation est donné en annexe A.4.

4.3.2 API de gestion des traces

Toutes les opérations sur les traces sont accessibles par des URIs dont la base est `http://host/sbt/{uid}/traces`. Tous ces ressources ne sont accessibles qu'à l'utilisateur identifié par `{uid}`, et donc soumis à une authentification.

Principe général pour la création de ressource. Conformément au style architectural REST, toute ressource est créée par l'application d'une requête de type `POST`, dont les données utiles décrivent la ressource à créer. Conformément à nos choix de conception, cette description sera faite en RDF.

Toutes les requêtes de création comporteront donc un graphe RDF incluant la ressource spéciale create en lieu et place de la ressource à créer (puisque par définition, celle-ci n'a pas encore d'URI). Un identifiant peut être proposé pour cette ressource avec la propriété withId, à la condition que cet identifiant ne soit pas déjà utilisé dans le même contexte[2]. À défaut, le système générera un identifiant automatiquement. Pour chaque requête de création, il sera spécifié un certain nombre de propriétés du vocabulaire KTBS requises ou optionnelles pour la ressource create, ainsi qu'un certain nombre de contraintes d'intégrités. Le graphe peut affecter à la ressource create d'autres propriétés que celles qui sont spécifiées, à condition que ces propriétés n'appartiennent pas au vocabulaire KTBS. Ceci permet aux outils utilisant le noyau SBT de l'enrichir avec leurs propres méta-données. Si les informations fournies sont conformes, la ressource sera créée avec toutes les propriétés de la ressource create dans le graphe fourni.

`http://host/sbt/{uid}/traces` (`GET`) Cette ressource correspond à l'ensemble des traces de l'utilisateur. Elle est représentée par un graphe RDF décrivant pour chaque trace son type (PrimaryTrace ou TransformedTrace), son modèle (hasModel) et son origine (hasOrigin). Pour les traces transformées, le graphe

[2] Par exemple, deux traces du même utilisateur ne peuvent pas avoir le même identifiant car elles auraient alors le même URI. En revanche, le même identifiant peut être partagé par une trace et un modèle, deux traces d'utilisateurs différents, ou deux observés de traces différentes.

doit en outre contenir l'URI de la transformation (hasTransformation) et la ou les traces sources (hasSource). Enfin, il doit contenir pour chaque trace la propriété hasRevision dont la valeur est générée par le noyau et permet de déterminer si la trace a été amendée (cf. ci-dessous). Le graphe peut également contenir d'autres méta-données, mais celles-ci devraient rester en nombre restreint afin de ne pas rendre le graphe trop volumineux.

`http://host/sbt/{uid}/traces` (`POST`) Cette opération sert à créer une nouvelle trace.

Propriétés requises
- `rdf:type` : soit PrimaryTrace ou TransformedTrace
- hasModel : URI du modèle de la trace à créer
- hasOrigin : origine temporelle de la trace à créer (identifiant opaque ou date au format ISO 8601)

Propriétés requises uniquement pour les traces transformées
- hasTransformation : URI de la transformation à appliquer
- hasSource : URI d'une ou plusieurs trace source (cf. ci dessous)

Propriétés optionnelles pour les traces transformées
- startsAt : entier limitant l'application de la transformation aux observés commençant à partir cette date
- finishesAt : entier limitant l'application de la transformation aux observés se terminant jusqu'à cette date.

Contraintes d'intégrité
- La seule transformation à accepter plusieurs sources est la transformation fusion. Dans ce cas, toutes les traces sources doivent toute avoir le même modèle, qui doit également être celui de la trace créée.
- La transformation copy impose que le type de la trace source soit le même que celui de la trace créée.

Erreurs possibles
- `400 source mismatch` : Le nombre de sources et leurs types ne correspondent pas à la transformation.
- `400 unreachable model` : Le modèle n'est pas accessible.
- `400 unreachable transformation` : La transformation n'est pas accessible.
- `409 id in use` : L'identifiant est déjà utilisé par une trace du même utilisateur.

`http://host/sbt/{uid}/traces/{tid}` (`GET`) Cette ressource correspond à une trace. En l'absence de paramètre, elle est représentée par un graphe RDF décrivant toutes les observés de la trace, leurs attributs et leurs relations. Cette représentation peut-être altérée par les paramètres suivants (exclusif) :
- `?from={entier}&to={entier}` : Ne représente que les observés entièrement compris dans l'intervalle spécifié. L'une ou l'autre des bornes peut être omise, l'intervalle sera alors ouvert.

- `?after={oid}` : Ne représente que les observés collectés après celui dont l'identifiant est passé. Ceci peut-être utile pour obtenir de manière incrémentale des informations sur une trace en cours de collecte.
- `?meta` : Représente les méta-données (i.e. les propriétés et les types hors du vocabulaire KTBS) de la trace au lieu de ses observés. Ce paramètre s'applique également à une requête `PUT` (cf. ci-dessous).

Enfin, un en-tête HTTP `Expect : revision={rev}` peut être utilisé, ou `rev` est la valeur de la propriété hasRevision de la trace. Cela permet d'être averti, par un échec de la réponse HTTP (code 412), si la trace a été amendée depuis la dernière requête.

`http://host/sbt/{uid}/traces/{tid}` (`POST`) Cette opération sert à créer un nouvel observé dans la trace dans le cadre de la collecte.

Propriétés requises
- `rdf:type` : n'importe quel type d'observé du modèle de la trace
- hasSubject : chaîne de caractères représentant le sujet de l'observé
- hasBegin : entier indiquant la date de début de l'observé
- hasEnd : entier indiquant la date de début de l'observé (identifiant opaque ou date au format ISO 8601)
- *tous*[3] les attributs de l'observé et ses relations avec les observés existants.

Contraintes d'intégrité
- La trace identifiée doit être une trace première, pas une trace transformée.
- Un observé ne peut pas avoir d'autres propriétés que celles spécifiées par le modèle de trace (attributs et relations) et celles spécifiées par le vocabulaire KTBS.
- Les domaines et co-domaines des attributs et relations doivent être respectés.
- La date de début d'un observé doit être inférieure ou égale à sa date de fin.
- Un observé doit avoir une date de fin supérieure ou égale à celle de tous les observés déjà présents dans la trace (monotonie de la collecte).

Erreurs possibles
- `400 invalid trace type` : La trace n'est pas une trace première.
- `400 invalid timestamps` : Le début de l'observé est après sa fin.
- `400 invalid timestamps` : Le début de l'observé est après sa fin.
- `400 model conformity error` : Certaines propriétés de l'observé n'appartiennent pas au modèle ou ne respecte pas les contraintes (domaine et co-domaine) du modèle.
- `400 monotony error` : La contrainte de monotonie n'est pas respectée.
- `409 id in use` : L'identifiant est déjà utilisé par une trace du même utilisateur.

[3] Sauf dans le cadre d'un amendement, aucun attribut ou relation avec des observés déjà existant ne pourra être ajouté à cet observé après sa création.

`http://host/sbt/{uid}/traces/{tid}` (`PUT`) Cette opération sert à amender la trace. Les données utiles de la requête sont la version modifiée des données récupérées par un GET sur le même URI (sans paramètre). Comme pour la requête GET, il est possible (et recommandé) d'utiliser l'en-tête `Expect` afin de s'assurer qu'un autre amendement n'a pas eu lieu depuis la dernière requête. Si l'opération réussit, Le noyau SBT doit obligatoirement changer la propriété hasRevision de la trace.

Erreurs possibles

- `400 invalid timestamps` : Le début de l'observé est après sa fin.
- `400 model conformity error` : Certaines propriétés de l'observé n'appartiennent pas au modèle ou ne respecte pas les contraintes (domaine et co-domaine) du modèle.
- `409 id in use` : L'identifiant est déjà utilisé par une trace du même utilisateur.

Remarques

- Dans les futures itérations, il est envisagé de supporter, pour les données utiles de cette requête, un format compact décrivant les modifications à apporter, plutôt que le résultat souhaité. Cette remarque s'applique bien sûr à toutes les requêtes de type `PUT`.
- Il est possible d'amender une trace transformée. Dans ce cas, le noyau SBT devra mémoriser l'amendement sous une forme lui permettant de conserver les modifications même lorsque la trace est recalculée.

`http://host/sbt/{uid}/traces/{tid}?meta` (`PUT`) Cette opération sert modifier les méta-données de la trace. Les données utiles de la requête sont la version modifiée des données récupérées par un GET sur le même URI (avec le paramètre `?meta`). Cette opération n'est pas considérée comme un amendement, et ne doit donc pas modifier la propriété hasRevision de la trace.

Erreurs possibles

- `400 cannot modify KSTB metadata` : Les données utiles contiennent des propriétés ou des types du vocabulaire KTBS.
- `400 model conformity error` : Certaines propriétés de l'observé n'appartiennent pas au modèle ou ne respecte pas les contraintes (domaine et co-domaine) du modèle.
- `409 id in use` : L'identifiant est déjà utilisé par une trace du même utilisateur.

`http://host/sbt/{uid}/traces/{tid}` (`DELETE`) Cette opération sert à supprimer une trace.

Pré-conditions

- La trace identifiée ne doit pas être la source d'une trace transformée (les traces transformées doivent être supprimées au préalable).

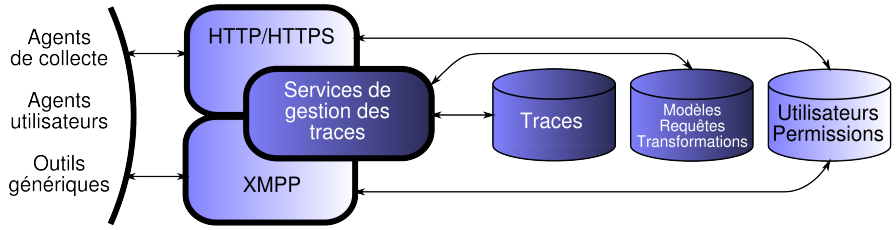

FIG. 4 – Architecture technique du noyau SBT à l'itération 2. Les composants au fond clair sont les composants standard. Les composants au fond sombre sont les composants spécifiques.

Erreurs possibles

– `400 trace in use` : La trace est la source d'une ou plusieurs trace transformée.

`http://host/sbt/{uid}/traces/{tid}/@sparql` (`GET/POST`) Cette ressource est un service SPARQL HTTP conforme à la spécification donnée dans [2], dont le graphe par défaut est la représentation par défaut de la trace (i.e. l'ensemble de ses observés). Ce service refusera de traiter une requête sur un autre graphe par défaut.

4.4 Iteration 2

Cette itération, représentée en figure 4.4, vise d'une part à ajouter un support complet pour les modèles de traces, requêtes et transformations ; d'autre part, à implémenter le protocole XMPP, afin d'améliorer les performances des agents temps-réel.

4.4.1 Support complet des modèles, requêtes et transformations

Le noyau de l'itération 1 n'a pas de connaissance des modèles de transformations, requêtes et transformations, qui sont accédés directement par la couche HTTP. Une première conséquence est que ces ressources ne peuvent pas être crées, modifiées ou supprimées en utilisant l'API RESTful. Une autre conséquence est l'impossibilité d'interroger le noyau sur ces composants (par exemple « Quelles sont toutes les transformations applicables au modèle M1 ? »).

Il convient donc de remplacer la gestion de ces éléments comme des ressources statiques par une gestion spécifique par le noyau, similaire à celle développée pour les traces à l'itération 1. On pourra alors implémenter les différentes opérations REST sur les resources suivantes, avec la sémantique habituelle :

`http://host/sbt/{uid}/models`	`GET`, `POST`
`http://host/sbt/{uid}/models/{mid}`	`GET`, `PUT`, `DELETE`
`http://host/sbt/{uid}/queries`	`GET`, `POST`
`http://host/sbt/{uid}/queries/{qid}`	`GET`, `PUT`, `DELETE`
`http://host/sbt/{uid}/transformations`	`GET`, `POST`
`http://host/sbt/{uid}/transformations/{tfid}`	`GET`, `PUT`, `DELETE`

4.4.2 Implémentation de XMPP

Initialement conçu pour la messagerie instantanée (mais tourné dès le départ vers l'extensibilité, comme son nom l'indique), le protocole XMPP permet une communication sous forme de flux. Ceci permet de palier deux inconvénients de l'API RESTful. Tout d'abord, une caractéristique importante du protocole HTTP est l'absence d'état : chaque requête doit être indépendante de celles qui l'ont précédé, ce qui suppose une renégociation des paramètres de la connexion à chaque requête, et donc un surcoût en bande passante et en temps. D'autre part, le protocole HTTP fonctionne uniquement en *pull* : le client n'obtient des informations que lorsqu'il les demande explicitement.

Les agents de collecte, agents utilisateurs et outils génériques devront donc être en mesure d'établir une « conversation » avec une trace. Pour les agents collecteurs, l'intérêt réside dans la possibilité d'ajouter des observés à cette trace avec une latence plus faible. Pour les autre, un intérêt supplémentaire consistera à pouvoir être notifiés *par le noyau* des changements dans l'état de cette trace (communication en mode *push*).

4.5 Iterations 3+

Cette section expose quelques pistes qui n'ont pas été retenues pour les deux premières itérations, mais devront être considérées pour les itérations ultérieures.

Exécution continuelle de requête À partir du moment ou le protocole XMP autorise une communication en mode *push*, on peut envisager d'exécuter une requête sur une trace en cours de collecte, et d'obtenir les résultats de cette requête *au fur et à mesure* de la collecte. Ceci requerra une adaptation du moteur SPARQL, ce qui représente un certain effort de développement, mais pourra être mis à profit pour optimiser les transformations.

Autres types de transformations Il pourrait être intéressant de définir d'autres types de transformations que ceux initialement proposés. Des *transformations composites* permettrait d'encapsuler plusieurs transformations élémentaires dont les résultats intermédiaires ne présentent pas d'intérêt pour l'utilisateur. Des *transformations externes* pourraient être réalisées par un programme (*plugin*) externe au SBT, dont il faudrait alors définir l'API.

Support d'autres formats Le protocole HTTP permet à un client de négocier avec le serveur le format des données reçues. Il pourrait être intéressant de supporter d'autres formats de sortie, notamment une version HTML pour être lisible par un utilisateur humain. De la même manière, d'autres

formats d'entrée (requêtes `POST` et `PUT`) pourraient être intéressants, notamment des formats permettant d'exprimer les *modifications* à apporter aux données plutôt que la totalité des données modifiées.

Gestion intégrée des utilisateurs et des agents de collecte Développer une gestion spécifiques (par opposition à la réutilisation de composants standards) des utilisateurs et des agents de collecte présenteraient un certain nombre d'avantages. Elle permettrait notamment aux utilisateurs de *déléguer* aux agents de collecte leurs droits sur une trace, plutôt que d'imposer que les collecteurs ne se connectent sous le nom de l'utilisateur lui même. Cette solution, préconisée dans les deux premières itérations, est satisfaisante lorsque l'utilisateur a un contrôle total sur l'agent de collecte, mais pas, par exemple, si ce dernier est un logiciel en source fermées ou s'il s'exécute sur une autre machine (application Web).

5 Agents de collecte

Les agents de collecte (parfois également appelés « agents collecteurs » ou « collecteurs ») sont chargé de

- capter les informations provenant du système observé,
- les traduire en observés et relations conforme à un modèle de trace,
- transmettre ces informations au noyau sous la forme d'un *flux de traçage*, pour leur intégration à une trace.

La manière dont les agents de collecte obtiennent les informations sur le système observé dépend grandement de ce dernier. La solution offrant la plus grande souplesse consiste évidemment à modifier les logiciels afin qu'ils se comportent eux même comme des agents de collecte, mais ceci est souvent impossibles (sources non disponibles) ou trop coûteux (logiciels complexe). Certains logiciels proposent un mécanisme d'extension ou de greffon (*plugin*) permet de pallier ces problèmes. D'autres utilisent déjà des mécanismes de notification standard (Apple Events, D-Bus, flux RSS) ou des messages structures spécifiques au domaine [3]. À défaut, il est toujours possible d'exploiter des données que les logiciels produisent (fichiers logs) ou échangent avec leur environnement (communication réseau, interception clavier/souris), mais ces dernières méthodes n'autorisent pas toujours un niveau d'abstraction suffisant pour produire une trace pertinente.

Les informations collectées doivent ensuite être exprimées conformément à un modèle de trace. Cette étape peut être plus ou moins simple selon la richesse des informations effectivement disponible (cf. paragraphe ci-dessus), mais également selon que le modèle de trace est spécifique à l'application. Il est en effet préférable, dans l'optique du partage de traces, d'avoir des modèles génériques, applicable à plusieurs applications similaires (comme ceux proposés dans le librable D1.2 du projet Ithaca), mais ceci peut créer un « écart d'impédance » entre l'application et la trace, que l'agent de collecte doit combler.

La structure du flux de traçage est entièrement déterminée par l'API du noyau. Il peut prendre la forme d'une succession de requêtes HTTP de type `POST`, ou d'une communication continue sous forme de flux XMPP. Cette dernière option est à préférer si le délai entre deux observés est court. La de monotonie, imposée par le noyau aux collecteurs, force ces derniers à fournir les observés

par ordre chronologique. Cette règle contraint moins les agents de collecte eux mêmes que les concepteurs de modèle de trace. En effet, un agent de collecte n'a *a priori* aucune raison de fournir au noyau les observés dans un autre ordre que celui dans lequel il les observe effectivement. La seule raison qu'il pourrait avoir serait d'attendre d'autres observés pour renseigner ses *attributs*. Les concepteurs de modèle devraient donc être attentifs à ne requérir, pour un observé, que des attributs disponibles au moment ou cet observé est collecté. Notons cependant que cette disponibilité dépend de l'application elle même et du mécanisme de collecte, et ne peut donc pas toujours être garantie pour un modèle donné.

6 Agents utilisateurs

Les agents utilisateurs interrogent le noyau SBT pour exploiter les traces qu'il contient : la présenter à l'utilisateur sous une forme intelligible, en extraire des informations pertinentes au regard de la sémantique particulière du modèle de trace, éventuellement suggérer des actions à l'utilisateur ou les exécuter automatiquement (agents de pilotage, cf. ci-dessous). On distinguera deux grandes classes d'agents utilisateurs : ceux travaillant sur les traces *a posteriori*, une fois la trace entièrement collectée, et ceux travaillant sur les traces en cours de collecte, parfois appelés « agents temps-réel ». Ces derniers posent des problèmes techniques supplémentaires, car ils doivent avoir une faible latence pour être utilisable.

Les agents utilisateurs, notamment les agents temps-réel, sont très dépendants des fonctionnalités du noyau concernant l'interrogation des traces. La première itération du noyau (en cours de développement) n'offre que des fonctionnalités d'interrogation minimale, mais permet de recevoir les informations sur une trace de manière itérative (en scrutation). Ceci ne garantit pas une faible latence si la fréquence de collecte est trop élevée, mais permettra au moins de développer de premiers agents temps-réel comme preuve de concept. La deuxième itération, avec le support de XMPP, offrira un meilleur support pour les agents temps-réel.

Agents de pilotage. Les agents de pilotage sont une catégorie particulière d'agents utilisateurs permettant de *rejouer* une portion de trace dans l'application tracée — voire même dans une autre application offrant des fonctionnalités similaires, ou en adaptant la trace à un contexte légèrement différent. Ces agents supposent d'une part un modèle de trace ayant un niveau de détail suffisant, et d'autre part que l'application soit instrumentée pour permettre ce « rejouage ». Cependant, du point de vue du noyau SBT, ils ne diffèrent pas significativement des autres agents utilisateurs.

7 Outils génériques

Comme il a été dit plus haut, les outils génériques sont similaires aux agents utilisateurs, mais ne dépendent pas spécifiquement d'un modèle de trace ou d'un système observé. Rentreront donc dans cette catégorie des outils standards, ayant vocation à être distribués en même temps que le noyau SBT afin de faciliter son déploiement et son utilisation, parmi lesquels :

Outils génériques de visualisation et d'édition de trace Ces outils doivent permettre de visualiser et de modifier n'importe quelle trace (voir aussi le livrable D1.3). Ils seront probablement moins intelligibles et simples d'utilisation que des agents utilisateurs spécialisés, mais seront en contrepartie utilisables même en l'absence de tels agents utilisateurs. Ils peuvent également servir de base à leur développement.

Outils d'administration du noyau SBT Ces outils doivent offrir une interface utilisateur au noyau SBT, permettant la gestion manuelle des traces, modèles, requêtes et transformations.

Outils de gestion des droits Ces outils doivent permettre à l'utilisateur de définir quelles traces ou parties de trace il souhaite partager, et avec qui. Ces outils sont un élément critique du système à base de trace, étant garant du respect de la vie privé des utilisateurs.

Références

[1] David Beckett. Turtle - terse RDF triple language. Technical report, November 2007.

[2] Kendall Grant Clark, Lee Feigenbaum, and Elias Torres. SPARQL protocol for RDF. W3C recommendation, W3C, 2008.

[3] Damien Clauzel, Claudia Roda, Marco Raglianti, Georgi Stojanov, and Claudia Roda. AtGentive conceptual framework and application scenarios. Deliverable 1.3, AtGentive project, September 2006. IST-4-027529-STP.

[4] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol – HTTP/1.1. RFC 2616, IETF, June 1999.

[5] Frank Manola, Eric Miller, and Brian McBride. RDF primer. W3C recommendation, W3C, February 2004.

[6] Eric Prud'hommeaux and Andy Seaborne. SPARQL query language for RDF. W3C recommendation, W3C, 2008.

[7] Peter Saint-Andre. Extensible messaging and presence protocol (XMPP) : core. RFC 3920, IETF, October 2004.

A Annexe

A.1 Vocabulaire RDF

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix owl:  <http://www.w3.org/2002/07/owl#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix : <http://liris.cnrs.fr/~ithaca/ns/ktbs/0.1/> .

# -------- trace model description --------

:TraceModel a rdfs:Class ;
  rdfs:label "TraceModel" ;
```

```
  rdfs:comment """La classe de tous les modèles de trace."""@fr ;
  rdfs:subClassOf owl:Ontology .

:imports a rdf:Property ;
  rdfs:label "imports" ;
  rdfs:comment """Indique qu'un modèle de trace en importe un autre."""@fr ;
  rdfs:subPropertyOf owl:imports ;
  rdfs:domain :TraceModel ;
  rdfs:range :TraceModel .

:TemporalDomain a rdfs:Class ;
  rdfs:label "TemporalDomain" ;
  rdfs:comment """La classe de tous les domaines temporels."""@fr ;
  rdfs:subClassOf owl:Class .

:hasTemporalDomain a rdf:Property ;
  rdfs:label "hasTemporalDomain" ;
  rdfs:comment """Indique le domaine temporel d'un modèle de trace."""@fr ;
  rdfs:domain :TraceModel ;
  rdfs:range :TemporalDomain .

# standard temporal domains
:sequence a :TemporalDomain ;
  rdfs:label "sequence" ;
  rdfs:comment """Domaine temporel séquentiel."""@fr .

:milliseconds a :TemporalDomain ;
  rdfs:label "milliseconds" ;
  rdfs:comment """Domaine temporel chronométrique en millisecondes."""@fr .

:seconds a :TemporalDomain ;
  rdfs:label "seconds" ;
  rdfs:comment """Domaine temporel chronométrique en secondes."""@fr .

:minutes a :TemporalDomain ;
  rdfs:label "minutes" ;
  rdfs:comment """Domaine temporel chronométrique en minutes."""@fr .

:hours a :TemporalDomain ;
  rdfs:label "hours" ;
  rdfs:comment """Domaine temporel chronométrique en heures."""@fr .

:days a :TemporalDomain ;
  rdfs:label "days" ;
  rdfs:comment """Domaine temporel chronométrique en jours."""@fr .

:ObselType a rdfs:Class ;
  rdfs:label "ObselType" ;
  rdfs:comment """La classe de tous les types d'observé."""@fr ;
  rdfs:subClassOf owl:Class .
```

```
:hasSuperObselType a rdf:Property ;
  rdfs:label "hasSuperObselType" ;
  rdfs:comment """Indique un type d'observé plus générique."""@fr ;
  rdfs:subPropertyOf rdfs:subClassOf ;
  rdfs:domain :ObselType ;
  rdfs:range :ObselType .

:RelationType a rdfs:Class ;
  rdfs:label "RelationType" ;
  rdfs:comment """La classe de tous les types de relation."""@fr ;
  rdfs:subClassOf owl:ObjectProperty .

:hasSuperRelationType a rdf:Property ;
  rdfs:label "hasSuperRelationType" ;
  rdfs:comment """Indique un type de relation plus générique."""@fr ;
  rdfs:subPropertyOf rdfs:subPropertyOf ;
  rdfs:domain :RelationType ;
  rdfs:range :RelationType .

:rDomain a rdf:Property ;
  rdfs:label "rDomain" ;
  rdfs:comment """Indique le domaine d'un type de relation."""@fr ;
  rdfs:subPropertyOf rdfs:domain ;
  rdfs:domain :RelationType ;
  rdfs:range :ObselType .

:rRange a rdf:Property ;
  rdfs:label "rRange" ;
  rdfs:comment """Indique le co-domaine d'un type de relation."""@fr ;
  rdfs:subPropertyOf rdfs:range ;
  rdfs:domain :RelationType ;
  rdfs:range :ObselType .

:Attribute a rdfs:Class ;
  rdfs:label "Attribute" ;
  rdfs:comment """La classe de tous les attributs."""@fr ;
  rdfs:subClassOf owl:DatatypeProperty .

:aDomain a rdf:Property ;
  rdfs:label "aDomain" ;
  rdfs:comment """Indique le domaine d'un attribut."""@fr ;
  rdfs:subPropertyOf rdfs:domain ;
  rdfs:domain :Attribute ;
  rdfs:range :ObselType .

:aRange a rdf:Property ;
  rdfs:label "aRange" ;
  rdfs:comment """Indique le co-domaine d'un attribut."""@fr ;
  rdfs:subPropertyOf rdfs:range ;
```

```
  rdfs:domain :Attribute ;
  rdfs:range rdfs:Datatype .

# ---------- transformation description ----------

:Transformation a rdfs:Class ;
  rdfs:label "Transformation" ;
  rdfs:comment """La classe de toutes les transformations."""@fr .

:fusion a :Transformation ;
  rdfs:label "fusion" ;
  rdfs:comment """Une transformation standard fusionnant n traces ayant le même modèle."""

:copy a :Transformation ;
  rdfs:label "copy" ;
  rdfs:comment """Une transformation standard copiant une trace à l'identique (utile pour

# ---------- trace description ----------

:Trace a rdfs:Class ;
  rdfs:label "Trace" ;
  rdfs:comment """La classe de toutes les traces."""@fr .

:hasModel a rdf:Property ;
  rdfs:label "hasModel" ;
  rdfs:comment """Indique le modèle auquel se conforme une trace."""@fr ;
  rdfs:domain :Trace ;
  rdfs:range :TraceModel .

:hasOrigin a rdf:Property ;
  rdfs:label "hasOrigin" ;
  rdfs:comment """Indique l'origine temporelle de la trace."""@fr ;
  rdfs:domain :Trace ;
  rdfs:range :TemporalDomain .

:hasRevision a rdf:Property ;
  rdfs:label "hasRevision" ;
  rdfs:comment """Contient un jeton opaque généré par le noyau, et changeant à chaque amen
  rdfs:domain :Trace ;
  rdfs:range rdf:Literal .

:PrimaryTrace a rdfs:Class ;
  rdfs:subClassOf :Trace ;
  rdfs:label "PrimaryTrace" ;
  rdfs:comment """La classe de toutes les traces premières."""@fr .

:TransformedTrace a rdfs:Class ;
  rdfs:subClassOf :Trace ;
  rdfs:label "TransformedTrace" ;
  rdfs:comment """La classe de toutes les traces transformées."""@fr .
```

```
:hasTransformation a rdf:Property ;
  rdfs:label "hasTransformation" ;
  rdfs:comment """Indique la transformation qui a permis de calculer cette trace."""@fr ;
  rdfs:domain :TransformedTrace ;
  rdfs:range :Transformation .

:hasSource a rdf:Property ;
  rdfs:label "hasSource" ;
  rdfs:comment """Indique une trace à partir de laquelle cette trace a été calculée."""@fr
  rdfs:domain :TransformedTrace ;
  rdfs:range :Trace .

:startsAt a rdf:Property ;
  rdfs:label "startsAt" ;
  rdfs:comment """Limite l'application de la transformation aux observés commençant à part
  rdfs:domain :TransformedTrace ;
  rdfs:range xsd:int .

:finishesAt a rdf:Property ;
  rdfs:label "finishesAt" ;
  rdfs:comment """Limite l'application de la transformation aux observés se terminant jusq
  rdfs:domain :TransformedTrace ;
  rdfs:range xsd:int .

:Obsel a rdfs:Class ;
  rdfs:label "Obsel" ;
  rdfs:comment """La classe de tous les observés. C'est par définition la superclasse comm

:hasTrace a rdf:Property ;
  rdfs:label "hasTrace" ;
  rdfs:comment """Indique la trace à laquelle appartient un observé."""@fr ;
  rdfs:domain :Obsel ;
  rdfs:range :Trace .

:hasSubject a rdf:Property ;
  rdfs:label "hasSubject" ;
  rdfs:comment """Indique le sujet de la trace."""@fr ;
  rdfs:domain :Obsel ;
  rdfs:range rdf:Literal .

:hasBegin a rdf:Property ;
  rdfs:label "hasBegin" ;
  rdfs:comment """Indique la date de début de l'observé."""@fr ;
  rdfs:domain :Obsel ;
  rdfs:range xsd:integer .

:hasEnd a rdf:Property ;
  rdfs:label "hasEnd" ;
  rdfs:comment """Indique la date de fin de l'observé."""@fr ;
```

```
  rdfs:domain :Obsel ;
  rdfs:range xsd:integer .

# ---------- création description ----------

:create a rdf:Resource ;
  rdfs:label "toCreate" ;
  rdfs:comment """URI standard n'ayant de sens que dans les données utiles d'une requête R

:withId a rdf:Property ;
  rdfs:label "withId" ;
  rdfs:comment """Indique l'identifiant souhaité pour l'objet à créer."""@fr ;
  rdfs:range [ a owl:Class ; owl:oneOf ( :create ) ] ;
  rdfs:domain rdf:Literal .
```

A.2 Exemple de modèle de trace en RDF

```
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix ktbs: <http://liris.cnrs.fr/~ithaca/ns/ktbs/0.1/>.
@prefix : <> . # e.g. http://host/sbt/jdoe/models/mon_modele1

<> a ktbs:TraceModel ;
  ktbs:hasTemporalModel ktbs:seconds ;
  ktbs:imports <http://example.com/modele_de_trace_1> .

:WriteMessage a ktbs:ObselType .
:SendMessage a ktbs:ObselType .
:Contact a ktbs:ObselType .
:FavouriteContact a ktbs:ObselType ;
  ktbs:hasSuperObselType :Contact.

:body a ktbs:Attribute ;
  ktbs:aDomain :SendMessage ;
  ktbs:aRange xsd:string .

:login a ktbs:Attribute ;
  ktbs:aDomain :Contact ;
  ktbs:aRange xsd:string .

:sent a ktbs:RelationType ;
  ktbs:rDomain :WriteMessage ;
  ktbs:rRange :SendMessage .

:to a ktbs:RelationType ;
  ktbs:rDomain :SendMessage ;
  ktbs:rRange :Contact .
```

A.3 Exemple de requête en SPARQL

```
PREFIX rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX :     <http://host/sbt/jdoe/models/mon_modele1>

SELECT  ?w, ?c
WHERE   {
    ?w rdf:type :WriteMessage ;
       :sent ?s .
    ?s :to ?c .
    ?c rdf:type :FavouriteContact .
}

# Cette requête sélectionne tous les observés WriteMessage ayant été envoyés
# à un contact préféré (FavouriteContact).
```

A.4 Exemple de transformation en SPARQL

```
PREFIX rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ktbs: <http://liris.cnrs.fr/~ithaca/ns/ktbs/0.1/>
PREFIX s:    <http://host/sbt/jdoe/models/mon_modele1>
PREFIX d:    <http://host/sbt/jdoe/models/mon_modele2>

CONSTRUCT  {
    [ rdf:type d:SentMessage
      ktbs:hasId ?id ;
      ktbs:hasSubject ?subject ;
      ktbs:hasBegin ?begin ;
      ktbs:hasEnd ?end ;
      d:to_contact ?login .
    ]
}
WHERE   {
    ?w rdf:type s:WriteMessage ;
       ktbs:hasId ?id ;
       ktbs:hasSubject ?subject ;
       ktbs:hasBegin ?begin ;
       s:sent ?s .
    ?s s:to ?c ;
       ktbs:hasEnd ?end ;
    ?c s:login ?login .
}

# Cette transformation ne retient que les messages effectivement envoyés (par
# opposition à ceux qui ont été écrits puis abandonnés), et insère le login
# contact comme attribut du nouvel observé.
```

A.5 Exemple de création de trace

```
@prefix : <http://liris.cnrs.fr/~ithaca/ns/ktbs/0.1/> .
```

```
@prefix mymodels: <http://host/sbt/jdoe/models/> .
@prefix mytrans: <http://host/sbt/jdoe/transformations/> .
@prefix mytraces: <http://host/sbt/jdoe/traces/> .
@prefix dc: <http://purl.org/dc/terms/> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .

:create a :TransformedTrace ;
    :withId "nouvelle_trace" ;
    :hasModel mymodels:modele_1 ;
    :hasOrigin "2009-01-16T12:34Z" ;
    :hasTransformation mytrans:transformation_1 ;
    :hasSource mytraces:trace_1 ;
# méta-données supplémentaires
    dc:creator "John Doe" ;
    rdfs:comment "Cette trace est plus lisible que la trace première." .
```