Gestion d'événements complexes pour une ville « intelligente » en BCM4Java

Cahier des charges — version 1.01

1er février 2022

Résumé

Ce document définit le cahier des charges du projet de l'unité d'enseignement Composants (CPS) pour son instance 2022. Le projet vise à comprendre les principes d'une architecture à base de composants, l'introduction du parallélisme, la gestion de la concurrence et les problématiques de répartition entre plusieurs ordinateurs connectés en réseau. Des notions d'architecture et de configuration pour la gestion de la performance et le passage à l'échelle seront aussi abordées.

L'objectif du projet est plus précisément d'implanter un prototype simplifié de système de gestion d'événements complexes. Ce type d'architecture logicielle devient populaire dans l'industrie depuis plus d'une décennie, mais il ne s'agit pas ici de construire un outil réaliste, simplement de comprendre les notions définies précédemment grâce à un exemple à la fois intéressant et concret. De nombreuses simplifications seront nécessairement faites pour rendre l'objectif atteignable en temps restreint. En guise d'applications, ce système de gestion d'événement complexes va servir à implanter des fonctions dites de « ville intelligente ».

1 Généralités

Avec la généralisation des systèmes d'information en exécution permanente, 24h/24, 7j/7, une architecture logicielle qui gagne en popularité depuis des années consiste à organiser ces SI autour de la *détection/génération* et de la *corrélation d'événements* en vue de *déclencher des actions*. Cette architecture est dénommée système de traitement d'événements complexes (*i.e., Complex Events Processing – CEP – systems*). Pour illustrer ce genre de systèmes, nous allons utiliser un exemple à la fois concret et parlant : la gestion intelligente d'une ville. Il faut imaginer une ville dont les systèmes d'alarmes, les réseaux urbains (eau, électricité, transports collectifs), les véhicules de service public (pompiers, SAMU, bus) et les feux de circulation sont truffés de capteurs (détecteurs de position, détecteurs d'incendie, alarmes médicales, etc.) et de systèmes d'opération permettant d'exécuter ou déclencher des actions (déclencher une intervention, donner une priorité aux véhicules d'urgence aux feux, etc.).

Tous ces capteurs et systèmes d'opération sont connectés à un système de gestion des événements complexes qui reçoit des événements venant des capteurs, qui les analyse et les corrèle pour enfin déclencher des actions selon des règles prédéfinies. Pour obtenir une architecture modulaire, extensible et capable de monter en charge, le système est décomposé en une multitude d'entités logicielles connectées aux sources d'événements et entre elles et qui font la corrélation des événements *i.e.*, des *corrélateurs*, pour ensuite déclencher des actions mais aussi agréger les événements plus simples en événements complexes (d'où leur nom de systèmes de traitements d'événements complexes) qui sont transmis à d'autres corrélateurs qui vont pouvoir les corréler avec d'autres événements, et ainsi de suite.

La figure 1 illustre un système abstrait de traitement d'événements complexes. Les dispositifs physiques connectés peuvent avoir deux rôles :

émetteur d'événements : détecteur/générateur qui envoie des évenements au système ;

exécuteur d'actions : dispositif permettant d'exécuter une action.

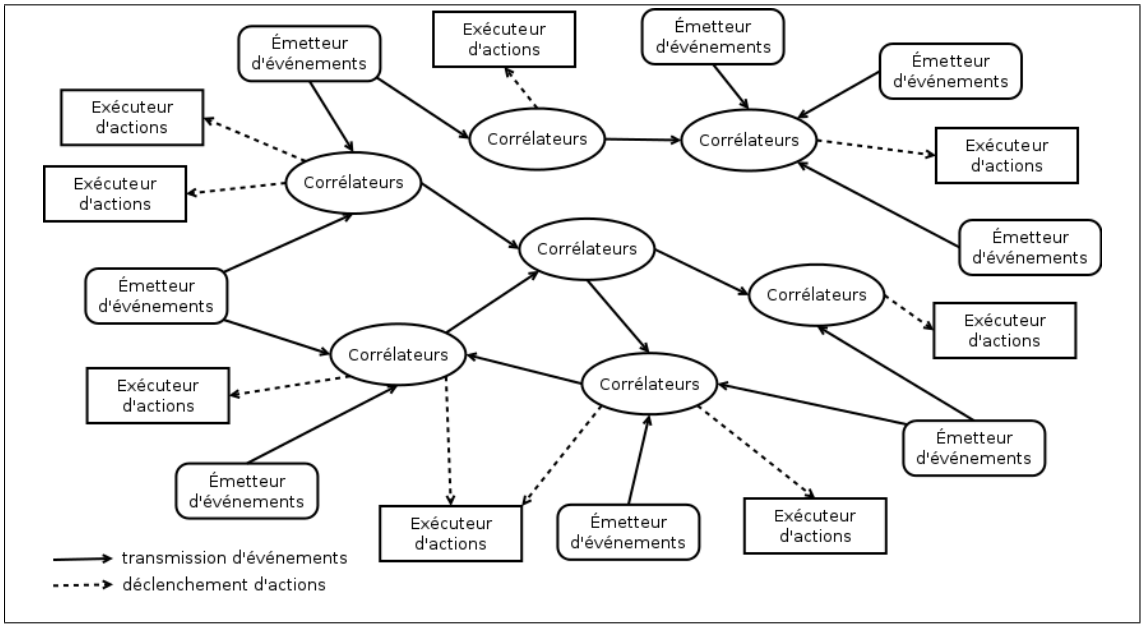

FIGURE 1 – Architecture abstraite d'un système de traitement d'événements complexes (CEP).

Dans un système réel, comme un système de gestion de ville intelligente, un même dispositif physique peut avoir les deux rôles. Par exemple, un feu de circulation peut agir selon le rôle de d'émetteur pour envoyer des événements marquant le passage d'un véhicule d'urgence, mais aussi agir selon le rôle d'exécuteur d'actions pour assurer, par exemple, le passage en mode priorité aux véhicules d'urgence afin de permettre à ce véhicule de se rendre plus vite sur son lieu d'intervention.

Le corrélateur, pour sa part, reçoit des événements et recherche dans la séquence d'événements qu'il a reçus des *patrons* qui pourront déclencher une action, émettre un événement complexe causé par ce patron d'occurrences d'événements, voire les deux.

Exemple 1 *Un corrélateur a reçu les occurrences d'événement « alerte médicale de traçage » indiquant par exemple qu'une personne âgée a fait une chute à son domicile. Il peut recevoir ensuite un autre événement « signalement manuel » émis par cette personne via son dispositif de détection connecté. Le corrélateur applique alors une règle prédéfinie qui lui fait déduire que la personne a fait une chute mais qu'elle est consciente, donc elle déclenche un événement complexe « chute de personne consciente » et l'action d'une intervention téléphonique d'un médecin du centre d'appel du SAMU pour vérifier l'état de santé de la personne et agir en conséquence.*

Un corrélateur est une entité logicielle qui possède une base d'événements, constituée des événements reçus et non encore détruits (traités ou non), ainsi qu'une base de *règles de corrélation* qui vont être appliquées sur la base d'événements de manière à détecter des patrons dans la séquence des événements reçus puis déclencher des actions. Une action peut consister à appeler un exécuteur d'actions pour que ce dernier l'exécute ou encore d'émettre un événement complexe créé à partir des événements qui auront été détectés par le patron (on dira alors que ces événements *ont causé* l'événement complexe).

Dans la suite de ce document, les éléments logiciels nécessaire à la mise en œuvre d'un prototype simplifié de système de traitement d'événements complexes vont être progressivement définis. Nous allons commencer par les éléments de base : événements, base d'événements, règles de corrélation et bases de règles. Ces éléments, qui sont manipulés par le système, seront définis comme des objets Java et programmés avec des classes. Ensuite, nous allons passer aux entités logicielles qui vont constituer les éléments actifs du système : les émetteurs d'événements, exécuteurs d'actions et corrélateurs. Ces entités seront définies comme des composants BCM, avec leurs interfaces requises et offertes, leurs ports, leurs connecteurs et leur services.

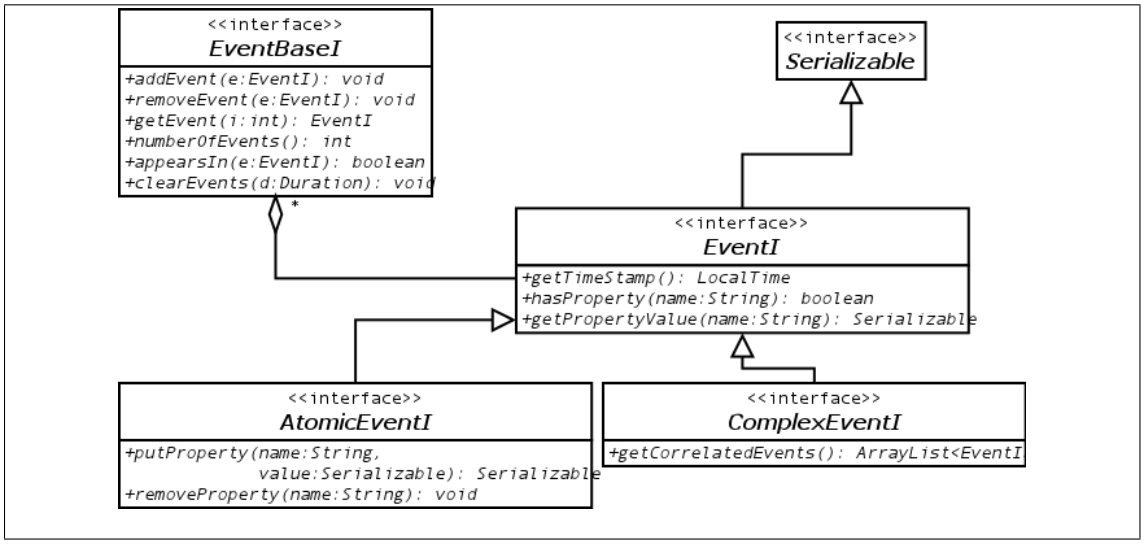

FIGURE 2 – Diagramme de classes présentant les événements.

2 Événements et corrélation

2.1 Événements et bases d'événements

Le diagramme de classes de la figure 2 présente les interfaces permettant de définir les événements ainsi que les bases d'événements, auxquelles le projet doit se conformer. L'interface `EventI` définit les propriétés essentielles des événements :

- ils sont marqués d'une estampille de temps retournée par `getTimeStamp` qui donne le moment auquel l'événement a été créé (on va supposer que c'est le moment de son occurrence) en utilisant la classe Java `LocalTime` ;
- ils possèdent aussi des propriétés, un peu comme des variables qui ont un nom de type `String` et une valeur qui est sérialisable que l'on peut accéder par `getPropertyValue` ; la méthode `hasProperty` permet de tester la présence d'une propriété sur un événement.

Les événements concrets sont de deux types : atomiques, tels qu'émis par un émetteur par exemple, ou complexes, issus de la corrélation entre plusieurs événements (atomiques ou complexes). Les événements atomiques se conforment à l'interface `AtomicEventI`, étendant `EventI`, ajoutant les méthodes permettant d'ajouter et de retirer des propriétés à l'événement.

Exemple 2 *Le détecteur de chute émet un événement atomique sous la forme d'une instance d'une classe `HealthEvent` implantant l'interface `AtomicEventI` à laquelle on ajoute une propriété `"type"` dont la valeur serait `"tracking"` (pour « traçage », comme dans la détection d'une chute) :*

```
HealthEvent healthAlarm = new HealthEvent();
healthAlarm.putProperty("type", "tracking");
assert  ((String)healthAlarm.getPropertyValue("type")).equals("tracking");
```

Les événements complexes sont issus d'une corrélation entre des événements atomiques ou complexes. Ils sont des instances de classes se conformant à l'interface `ComplexEventI`. Pour un événement complexe, il est possible de récupérer l'ensemble des événements dont la corrélation les a identifiés comme sa cause par la méthode `getCorrelatedEvents`.

Exemple 3 *La corrélation des événements « alarme de santé de type traçage » et « signalement manuel » venant du même appareil de détection engendre l'événement complexe `ConsciousFall` (littéralement « chute consciente ») à partir de ces deux événements.*

Notez que les événements complexes ne possèdent pas de propriétés en propre, mais ils peuvent définir les méthodes `hasProperty` et `getPropertyValue` pour accéder à des propriétés qui sont définies par les événements corrélés. Par exemple, si les événements « alarme de santé de type

traçage » et « signalement manuel » portent une propriété `"personId"` donnant l'identifiant de la personne portant le détecteur, alors la classe `ConsciousFall` permet de récupérer cette valeur par :

```
ConsciousFall c = new ConsciousFall(...);
assert c.getCorrelatedEvents().size() == 2;
EventI fall = c.getCorrelatedEvents().get(0);
EventI signal = c.getCorrelatedEvents().get(1);
assert fall.hasProperty("personId") && signal.hasProperty("personId");
assert fall.getPropertyValue("personId").equals(
                                signal.getPropertyValue("personId"));
```

Par défaut, la méthode `getPropertyValue` doit retourner la valeur de la propriété demandée sur le premier événement parmi ses événements corrélés qui définissent cette propriété. Toutefois, rien n'empêche de redéfinir cette méthode dans la classe d'événement corrélé, comme `ConsciousFall`, pour obtenir une sémantique différente. Dans le cas de `ConsciousFall`, cette redéfinition pourrait vérifier, par exemple, que les deux événements corrélés définissent la propriété `personId` avec la même valeur.

Base d'événements

Une base d'événements contient tous les événements reçus par un corrélateur et non encore traités (ou non) puis détruits. Le corrélateur va détenir une base d'événements sur laquelle ses règles de corrélation vont opérer. Pour faciliter les corrélations utilisant des temporalités, la base d'événements maintient ces derniers dans l'ordre de leur occurrence. Dans le projet, la base est définie par une classe implantant l'interface `EventBaseI` va tenir ce rôle. Ses méthodes permettent d'ajouter ou retirer des événements, de connaître le nombre d'événements contenus dans la base, et d'accéder au i^e événement dans la base. La méthode `clear` permet de détruire tous les événements qui se sont produits à plus de `period` de temps par rapport à l'instant courant (par exemple, tous les événements qui se sont produits il y a plus d'une heure) ; une valeur `null` pour `period` est alors interprétée comme l'infini, c'est-à-dire que `clear` détruit alors tous les événements de la base.

2.2 Corrélation : règles et bases de règles

Une règle de corrélation sert à reconnaître un certain patron d'événements dans la base d'événements puis, si un tel patron a été reconnu, à émettre un événement complexe, à déclencher une action et à faire des effets de bord sur la base d'événements, comme éliminer tous les événements qui ont été reconnus par le patron ou les remplacer par un événement complexe issu de cette corrélation.

Si on reprend l'exemple 1 de chute d'une personne encore consciente, il s'agit de définir un patron reconnaissant qu'il y a eu d'abord l'événement de « santé » puis l'événement « signalement manuel » dans un délai assez court, ce qui permettra de déduire qu'une chute d'une personne encore consciente s'est produite. La règle pourra alors déclencher une action consistant à demander à un médecin du centre SAMU d'appeler la personne pour prendre de ses nouvelles.

La corrélation d'événements va utiliser des règles inspirées des règles *Events/Conditions/Actions, ou ECA*. Une règle ECA s'interprète en gros de la manière suivante :

> Si les *événements* mentionnés sont présents et que les *conditions* sont vraies alors les *actions* prévues sont exécutées.

Nous allons en fait définir des règles légèrement plus détaillées en cinq points :

1. Une partie appariement (*match*) qui définit les événements devant être présents pour activer la règle.
2. Une partie corrélation (*correlate*) qui vérifie des conditions entre les événements appariés telle que le fait qu'ils se sont produits à la même position.
3. Une partie filtrage (*filter*) qui permet de vérifier des conditions ne faisant pas partie des événements mais plutôt du contexte ou de l'environnement.

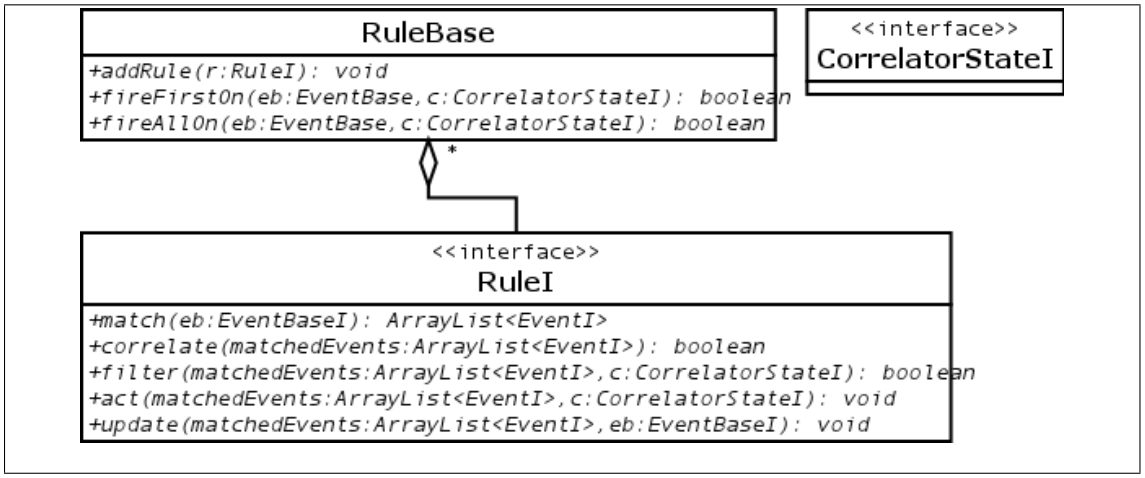

FIGURE 3 – Diagramme de classes présentant les règles de corrélation.

Si les trois parties précédentes ont été appliquées avec succès, alors on déclenche la règle en exécutant les deux parties suivantes :

4. Une partie action (ou *act*) qui permet d'exécuter des actions sur le système comme déclencher des alarmes ou de propager des événements vers d'autres corrélateurs.
5. Une partie mise à jour (ou *update*) qui permet de modifier la base d'événements du corrélateur en retirant des événements corrélés ou en ajoutant des événements complexes, par exemple.

Nous allons maintenant détailler ces fonctionnalités en suivant les éléments du diagramme de classe de la figure 3.

Appariement des événements (`match`)

L'appariement d'événements est la première étape dans le déclenchement des règles visant à vérifier si tous les événements requis avec les bonnes propriétés sont présents dans la base d'événements pour les retourner dans une liste s'ils sont bien présents (ou `null` sinon). Le principe de l'appariement est de ne s'intéresser qu'aux conditions sur chaque événement pris *individuellement*. Cela se limite le plus souvent à deux types de vérification :

type des événements : les événements étant définis par des classes implantant l'interface `EventI`, le plus simple est d'utiliser un test `instanceof` de Java pour identifier le classe d'instantiation de l'événement ;

valeurs des propriétés : la règle peut ajouter des contraintes sur les valeurs des propriétés de chaque événement, et alors il faut utiliser les méthodes `hasProperty` et `getPropertyValue` de l'interface `EventI` pour vérifier respectivement la présence et la valeur d'une propriété, voire des contraintes croisées entre les valeurs de propriétés de l'événement.

Exemple 4 *Reprenons l'exemple de l'événement de type* ***`ConsciousFall`****. On veut alors apparier un événement* ***`HealthEvent`*** *dont la propriété* ***`"type"`*** *est égale à* ***`"tracking"`*** *et un événement de type* ***`SignalOK`****, ce qui peut donner un code comme suit :*

```
public ArrayList<EventI> match(EventBaseI eb) {
  EventI he = null; EventI s = null;
  for (int i = 0 ; i < eb.numberOfEvents() && (he == null || s == null) ; i++) {
    EventI e = eb.get(i);
    if (e instanceof HealthEvent && e.hasProperty("type")
          && ((String)e.getPropertyValue("type")).equals("tracking")) {
      healthEvent = e;
    }
    if (e instanceof SignalOK) { s = e; }
  }
```

```
    if (he != null && s != null) {
      ArrayList<EventI> matchedEvents = new ArrayList<>();
      matchedEvents.add(he); matchedEvents.add(s);
      return matchedEvents;
    } else {
      return null;
    }
  }
```

Corrélation des événements (`correlate`)

La corrélation est l'étape suivante où, ayant en main tous les événements permettant de déclencher la règle, il faut vérifier les *contraintes croisées* entre ces événements. La méthode `correlate` prend en paramètre une liste des événements appariés par la méthode `match` précédente puis applique les contraintes définies par la règle.

Exemple 5 *Toujours dans notre exemple de la chute avec personne consciente, le résultat de* ***`match`*** *retourne deux événements, le premier instance de la classe* ***`HealthEvent`*** *et le second instance d'une classe, par exemple,* ***`SignalOK`****. La règle dit alors que les deux événements doivent émaner de la même personne. Elle dit également que le délai maximal entre les deux événements ne doit pas dépasser 10 minutes (dix minutes après la réception de l'événement* ***`HealthEvent`****, le système réagira en envoyant une équipe sur place car il déduira que la personne est inconsciente). On peut alors écrire le code suivant :*

```
  public boolean correlate(ArrayList<EventI> matchedEvents) {
    return matchedEvents.get(0).hasProperty("personId") &&
           matchedEvents.get(1).hasProperty("personId") &&
           matchedEvents.get(0).getProperty("personId").equals(
                       matchedEvents.get(1).getProperty("personId")) &&
           matchedEvents.get(0).getTimeStamp().isBefore(
                                       matchedEvents.get(1).getTimeStamp()) &&
           matchedEvents.get(0).getTimeStamp().plus(
                       Duration.of(10, ChronoUnit.MINUTES)).isAfter(
                                       matchedEvents.get(1).getTimeStamp());

  }
```

Filtrage des événements (`filter`)

La troisième étape concerne des conditions de déclenchement de la règle qui dépendent du contexte et non des événements. Par exemple, un corrélateur d'événements peut être associé à un centre SAMU pour recevoir les événements autour de la gestion de la santé. Ce corrélateur peut maintenir à jour dans son état le fait que des médecins sont disponibles au centre (tous ne sont pas en intervention), ce qui peut devenir une condition d'application d'une règle.

Une difficulté se pose toutefois, puisque si cet état est maintenu par le corrélateur, l'objet représentant la règle de corrélation n'a pas accès directement à cette information. La méthode `filter` prend ainsi deux paramètres : la liste des événements appariés et une référence permettant d'appeler des méthodes sur le corrélateur. Ces méthodes sont de deux types :

- des méthodes permettant de vérifier des prédicats ou de récupérer des informations sur l'état du corrélateur ;
- des méthodes permettant de déclencher des actions sur le système qui vont servir à l'exécution d'actions (voir ci-après).

Exemple 6 *Toujours en poursuivant notre exemple de la règle de chute, une condition nécessaire pour déclencher la règle est qu'un médecin soit disponible pour téléphoner à la personne pour en prendre des nouvelles. Le corrélateur va fournir une référence implantant l'interface définie à la figure 4, ce qui permet d'écrire le code suivant :*

```
public interface HealthCorrelatorStateI extends CorrelatorStateI
{
    public boolean isMedicAvailable();
    public void    triggerMedicCall(String personId);
}
```

FIGURE 4 – Interface d'un corrélateur d'événements de santé.

```
public boolean filter(ArrayList<EventI> matchedEvents, CorrelatorStateI cs) {
    HealthCorrelatorStateI samuState = (HealthCorrelatorStateI)cs;
    return samuState.isMedicAvailable();
}
```

Exécutions des actions (`act`)

Lorsque les trois premières étapes de la règle auront réussi, la première en retournant une liste d'événements appariés et les deux suivantes en retournant vrai, alors il faut passer à l'exécution des actions et des mises à jour prescrites par cette règle. L'action est donc la quatrième étape implantée par la méthode `act` prend ainsi deux paramètres : les événements appariés et une référence permettant d'appeler des méthodes du corrélateur pour réaliser les actions concernées. Pour des raisons de simplicité, on utilise la même interface `CorrelatorStateI` pour marquer la référence qui permet d'appeler ces méthodes.

Exemple 7 *Toujours en poursuivant notre exemple de la règle de chute, la règle demande à ce qu'un médecin soit charger de téléphoner à la personne pour en prendre des nouvelles. Le corrélateur va fournir une référence implantant l'interface définie à la figure 4, ce qui permet d'écrire le code suivant :*

```
public void act(ArrayList<EventI> matchedEvents, CorrelatorStateI cs) {
    HealthCorrelatorStateI samuState = (HealthCorrelatorStateI)cs;
    samuState.triggerMedicCall(matchedEvents.get(0).getProperty("personId"));
}
```

Mise à jour de la base d'événements (`update`)

La dernière étape vise à mettre à jour la base d'événements, le plus souvent pour en retirer les événements traités par la règle, mais il est possible d'ajouter des événements à cette base pour déclencher d'autres règles. La méthode `update` prend elle aussi deux paramètres : la liste des événements appariés et la base d'événements. Elle peut ainsi faire les effets de bord souhaités sur cette base.

Exemple 8 *Pour compléter notre exemple de la règle de chute, la règle demande à ce que les deux événements appariés soient retirés de la base. On peut alors écrire le code suivant :*

```
public void update(ArrayList<EventI> matchedEvents, EventBaseI eb) {
    eb.remove(matchedEvents.get(0));
    eb.remove(matchedEvents.get(1));
}
```

Bases de règles

Un corrélateur peut utiliser plusieurs règles. Pour organiser ces dernières, on utilise une base de règles. Dans le projet, une base de règles est définie par la classe `RuleBase` à laquelle on peut simplement ajouter des règles qui vont être conservées dans l'ordre des ajouts à cette base. Pour exécuter les règles d'une base, deux méthodes peuvent être utilisées :

`fireFirstOn :` cette méthode passe les règles les unes après les autres pour appeler la méthodes `match`, puis si le résultat n'est pas null, elle appelle `correlate` et si le résultat est vrai elle appelle la méthode `filter`. Si le résultat est encore vrai, elle va alors déclencher cette règle en appelant `act` puis `update`. Elle s'arrête dès qu'une règle s'est déclenchée et alors elle retourne vrai. Si aucune des règles ne s'est déclenchée, elle retourne faux.

`fireAllOn :` cette méthode appelle la méthode `fireFirstOn` tant qu'une règle se déclenche et ne s'arrête que lorsque plus aucune règle ne se déclenche ; elle retourne vrai si au moins une règle s'est déclenchée et faux sinon.

La seconde méthode permet de vérifier plusieurs corrélations successivement, avec des effets de bord sur la base des événements au fur et à mesure des déclenchements. Sa terminaison n'est pas intrinsèquement garantie. Il est donc important quand on constitue une base de règles de s'assurer que l'exécution de chaque règle modifie la base d'événements (par les méthodes `update`) de telle manière à s'assurer que ce processus s'arrête, principalement en retirant des événements. Un corrélateur peut aussi avoir plusieurs bases de règles distinctes pour faire des corrélations tout en évitant de déclencher indéfiniment les mêmes règles.

L'ordre dans lesquelles les règles sont déclenchées étant dicté par l'ordre dans lequel elles sont ajoutées à la base de règles, il est important de s'assurer que les ajouts soient faits dans le bon ordre. L'utilisation de plusieurs bases de règles peut également aider à résoudre les conflits dans l'ordre d'application des règles.

3 Architecture du CEP à base de composants

Le CEP comporte trois principaux types d'entités :
— des émetteurs d'événements, qui émettent les événements atomiques issus de capteurs et d'autres dispositifs physiques ;
— des exécuteurs d'actions, qui permettent d'agir sur les entités du monde physique ; et
— des corrélateurs, qui reçoivent les événements, appliquent des règles de corrélation et appellent des actions sur les exécuteurs d'actions.

L'architecture à base de composants définit les interfaces de composants associées à ces entités, sachant que l'émission d'événements et l'exécution d'actions sont en fait des rôles pouvant être implantés sous forme de greffons (« *plug-ins* ») et être réalisés par des composants distincts mais aussi par un même composant pouvant à la fois émettre des événements et exécuter des actions. [1]

3.1 Bus d'événements

La figure 1 présente une architecture logique (abstraite) pour un système de traitement des événements complexes. Dans cette architecture logique, chaque émetteur d'événements, comme chaque exécuteur d'actions, est directement connecté au corrélateur devant traiter ses événements. Si une implantation directe de cette architecture se justifirait peut-être pour les éxécuteurs d'actions, pour la transmission des événements, en pratique, ce n'est pas nécessairement la meilleure solution. Les événements émis peuvent concerner plusieurs corrélateurs, exigeant alors des connexions multiples pour l'émetteur d'événements. Par ailleurs, dans un système réparti, les événements peuvent devoir être propagés par réseau, et alors il faudrait éviter de transmettre plusieurs fois le même événement vers un hôte quand cet hôte contient plusieurs corrélateurs destinataires de cet événement. Ainsi, la transmission des événements requiert une implantation qui assure un bon niveau de performance tout en autorisant un passage à l'échelle. Pour explorer ces possibilités, l'architecture à base de composants demandée dans ce projet impose l'utilisation d'un *bus d'événements* pour la communication.

Le bus d'événements va servir à la transmission des événements depuis les émetteurs jusqu'aux corrélateurs et entre corrélateurs. La figure 5 présente cette architecture abstraite. Au lieu de connecter les émetteurs directement aux corrélateurs et les corrélateurs entre eux, tous sont connectés au bus d'événements. Ainsi, lorsqu'ils émettent un événement, au lieu de le transmettre

1. Par exemple, un centre SAMU peut émettre des événements indiquant la disponnibilité des ses ambulances et de ses médecins mais aussi exécuter des actions comme faire appeler une personne par un médecin pour vérifier son état de santé suite à une chute.

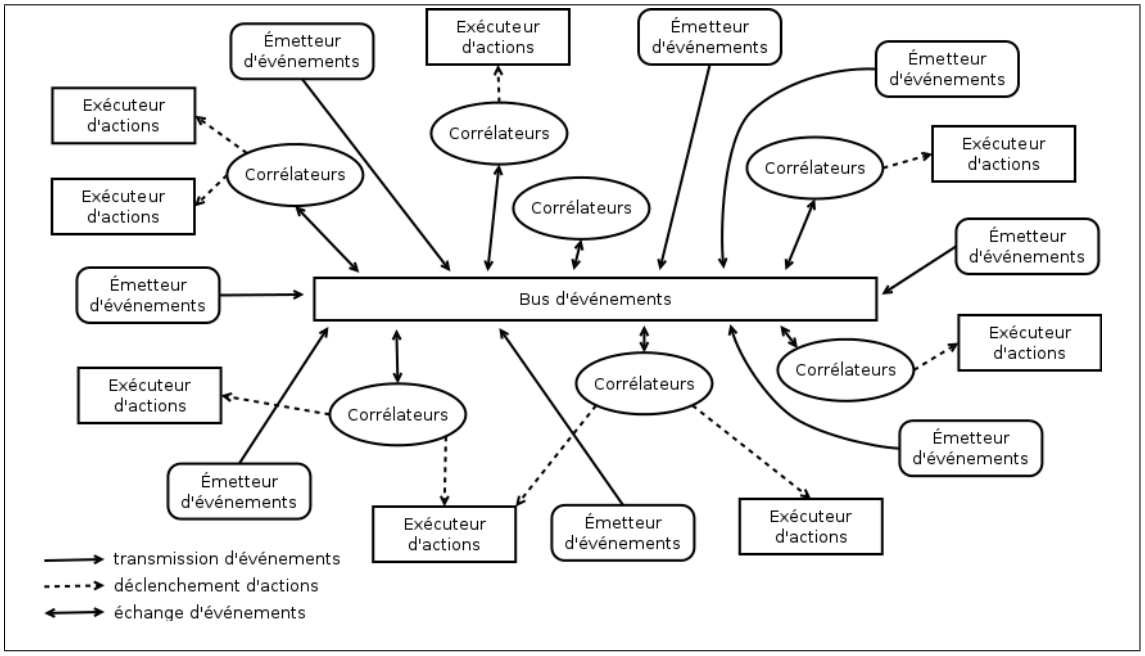

FIGURE 5 – CEP utilisant un bus pour les transmissions d'événements.

aux corrélateurs un par un, ils ne l'enverront qu'une fois au bus et c'est ce dernier qui va se charger de les transmettre aux corrélateurs. Les corrélateurs vont aussi se connecter au bus à la fois pour recevoir les événements mais aussi pour transmettre les événements que leurs actions émettent.

Le bus d'événements présente donc plusieurs avantages par rapport à la connexion directe. Par exemple, l'émission unique peu importe le nombre de corrélateurs devant recevoir l'événement, comme expliqué précédemment. Également, cela simplifie les interconnexions. Plutôt que d'émettre des événements en donnant explicitement les destinataires, on peut passer à un mode de réception plus flexible par abonnement. Les émetteurs s'enregistrent puis envoient les événements avec leur URI ; les corrélateurs s'enregistrent auprès du bus puis s'abonnent à la recéption d'événements d'un émetteur d'une certaine URI pour recevoir les événements qui leurs sont destinés. En réparti, le bus va aussi permettre d'éviter les multiples transmissions d'événements entre JVM (ou ordinateurs), mais nous y reviendrons plus loin.

Dans l'architecture, le bus d'événements devra être implanté dans un comme un composant BCM défini par la classe `CEPBus`. Pour désigner les entités du système dans le bus et lui permettre d'acheminer les événements, toutes les entités vont avoir une URI. Le composant `CEPBus` offre une interface de composant de gestion `CEPBusManagementCI` proposant les opérations suivantes :

`registerEmitter :` enregistre l'émetteur d'événements dont l'URI est donnée par `uri` et retourne l'URI de son port entrant offrant l'interface `EventEmissionCI` pour permettre à l'émetteur de se connecter au bus.

`unregisterEmitter :` désenregistre l'émetteur d'événements dont l'URI est donnée par `uri`.

`registerCorrelator :` enregistre le corrélateur dont l'URI est donnée par `uri` avec l'URI de son port entrant offrant l'interface `EventReceptionCI` (ce qui permet au bus de s'y connecter) puis retourne l'URI de son port entrant offrant l'interface `EventEmissionCI` pour permettre au corrélateur de se connecter à lui pour émettre des événements le cas échéant.

`unregisterCorrelator :` désenregistre le corrélateur dont l'URI est donnée par `uri`.

`registerExecutor :` enregistre l'exécuteur d'actions dont l'URI est donnée par `uri` avec l'URI de son port entrant offrant l'interface `ActionExecutionCI`.

`getExecutorInboundPortURI :` retourne l'URI du port entrant de l'exécuteur d'actions dont l'URI est donnée par `uri` ; cette opération permet aux corrélateurs de récupérer l'URI des exécuteurs d'actions auxquels ils souhaitent se connecter.

`unregisterExecutor :` désenregistre l'exécuteur d'actions dont l'URI est donnée par `uri`.

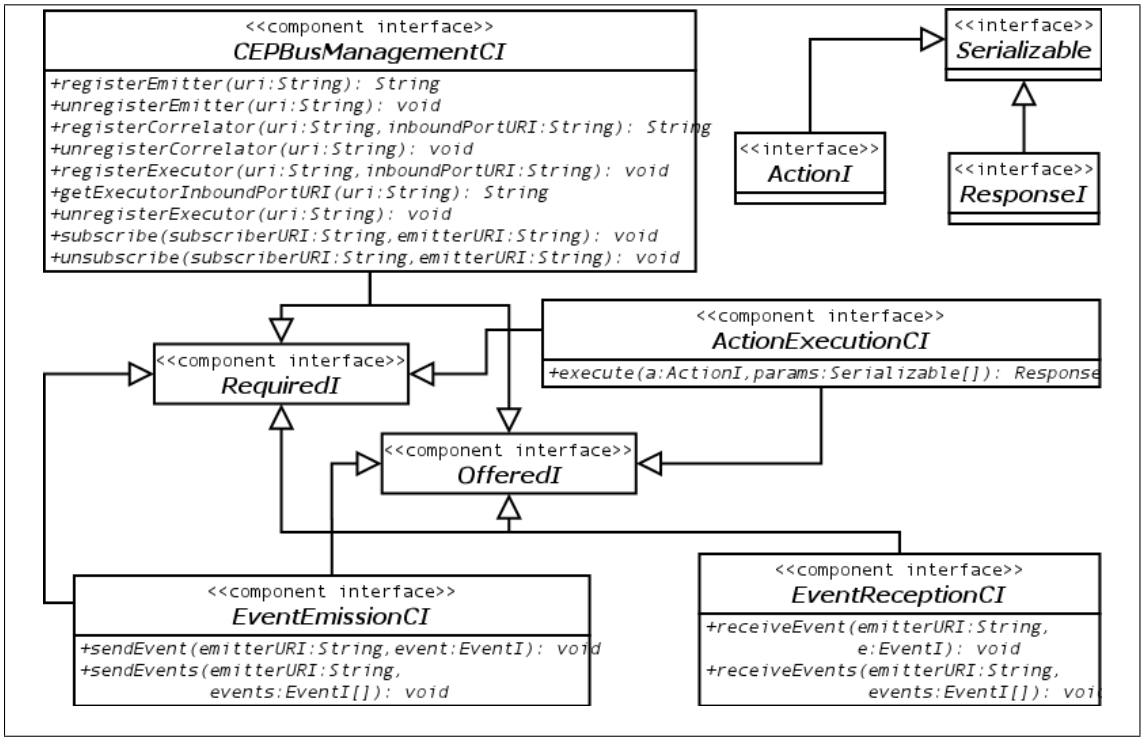

FIGURE 6 – Les interfaces de composants du CEP.

`subscribe :` abonne le corrélateur dont l'URI est donnée par `subscriberURI` aux événements émis par l'émetteur d'événements ou le corrélateur dont l'URI est donnée par `emitterURI`.

`unsubscribe :` désabonne le corrélateur dont l'URI est donnée par `subscriberURI` aux événements émis par l'émetteur d'événements ou le corrélateur dont l'URI est donnée par `emitterURI`.

Le composant `CEPBus` offre l'interface de composant `EventEmissionCI` par laquelle il recevra les événements émis par les émetteurs d'événements et qui sera décrite un peu plus loin. Il requiert également l'interface de composant `EventReceptionCI` par laquelle il va transmettre les événements aux corrélateurs, et qui sera aussi décrite un peu plus loin.

3.2 Composants en contact avec le monde physique

Pour notre application, les composants représentant les entités physiques (détecteur d'incendie, centre SAMU, etc., jouent le rôle des logiciels qui seraient déployés dans les entités dans la réalité. Ces logiciels pourraient être embarqués sur des objets connectés (détecteurs) ou tourneraient sur les ordinateurs d'une entité (centre SAMU). Ils peuvent assurer l'un ou l'autre de deux rôles suivants, voire les deux à la fois :

Émetteur d'évenement : les composants disposants de détecteurs ou d'un état qui vont envoyer des événements au système comme un détecteur d'incendie qui émet des événements au besoin.

Exécuteur d'actions : les composants disposants de moyens d'agir sur leur environnement ou de changer d'état et qui vont donc être capables d'exécuter des actions comme un feu de circulation dont on peut changer le mode pour donner la priorité à un certain sens de circulation.

Émetteurs d'événements

Les émetteurs d'événements s'inscrivent auprès du bus, et reçoivent en retour l'URI du port entrant du bus offrant l'interface de composant `EventEmissionCI`. Cette interface est donc offerte par le bus d'événements `CEPBus` et requise par les composants émetteurs d'événements, et c'est

alors au composant émetteur d'événements qu'il revient de se connecter au bus pour émettre leurs événements. Les signatures définies dans cette interface sont les suivantes :

`sendEvent :` émet un événement `event` à propager vers les destinataires abonnés aux événements en provenance de l'émetteur identifié par `emitterURI`.

`sendEvents :` envoie un tableau d'événements `events` à propager vers les destinataires abonnés aux événements en provenance de l'émetteur identifié par `emitterURI`.

Exécuteurs d'actions

Les exécuteurs d'actions s'inscrivent aussi auprès du bus en fournissant l'URI de leur port entrant offrant l'interface de composant `ActionExecutionCI`. En fait, pour les exécuteurs d'actions, les actions possibles sont très variées. Afin de simplifier les interfaces en proposant une seule interface de composant pour tous les exécuteurs d'actions, on adopte une approche dans l'esprit du patron de conception Commande où une unique signature d'opération est fournie pour faire exécuter toutes les actions possibles. Cette opération prend en paramètre l'identification de l'action à exécuter et un tableau de paramètres à utiliser si nécessaire. L'implantation de cette opération par une méthode dans les composants exécuteurs d'actions se chargera de déclencher l'action correspondante en fonction des paramètres réels reçus.

L'interface de composant `ActionExecutionCI` propose l'opération suivante :

`ResponseI execute(ActionI a, Serializable[] params :` exécute l'action identifiée par `a` avec les paramètres `params` puis retourne un résultat sérialisable.

Exemple 9 *Supposons l'exécuteur d'actions correspondant à un poste de pompiers qui propose trois actions : envoyer un camion simple, un camion à échelle ou une grande échelle sur le lieu d'un feu. Les actions possibles peuvent être définies par une énumération de la manière suivantes :*

```
public enum FireStationActions implements ActionI {
  SendStandardTruck, SendLadderTruck, SendHighLadderTruck
}
```

Puis, une implantation possible du service `execute` *dans le composant exécuteur d'actions pourrait prendre la forme suivante (avec la classe* `Address` *préalablement définie) :*

```
public ResponseI execute(ActionI a, Serializable[] params) throws Exception {
  assert a instanceof FireStationActions;
  assert params != null && params.length == 1 && params[0] instanceof Address;
  Address address = (Address) params[0];
  switch ((FireStationActions)a) {
  case SendStandardTruck: sendStandardTruck(address); break;
  case SendLadderTruck: sendLadderTruck(address); break;
  case SendHighLadderTruck: sendHighLadderTruck(address);
  }
  return true;
}
```

Rôles et greffons

Émission d'événements et exécution d'actions sont en réalité des rôles qui sont tenus par des composants. Ces derniers peuvent tenir l'un ou l'autre voire les deux rôles à la fois. Ils se prêtent donc bien à la définition de greffons (*plug-ins*) dans BCM, ce qui permettra de factoriser et réutiliser plus facilement le code nécessaire pour les mettre en œuvre entre différents composants et donc d'accélérer le développement des différents types de composants.

3.3 Corrélateurs

Les composants corrélateurs reçoivent des événements, les ajoutent dans leur base d'événements, appliquent des règles de corrélation de leur base de règles à chaque ajout d'événements. Si

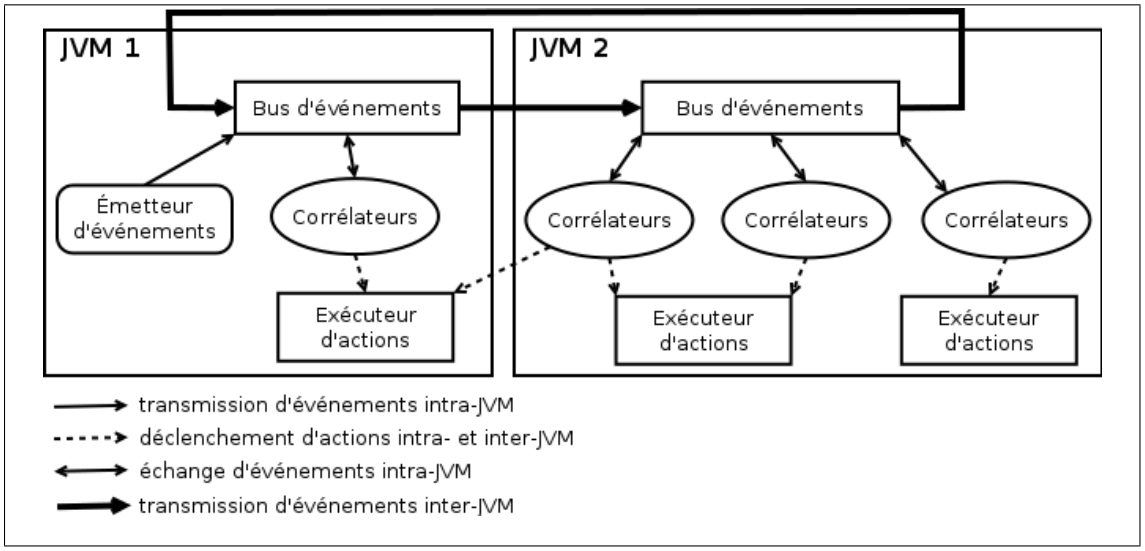

FIGURE 7 – Déploiement réparti du CEP.

au moins une règle se déclenche après réception d'un événement, ces composants appellent les composants exécuteurs d'actions en leur passant les informations nécessaires, émettent des événements vers d'autres corrélateurs et, enfin, ils exécutent les effets de bord sur leur base d'événements.

Pour recevoir les événements, les corrélateurs s'enregistrent auprès du bus en donnant leur URI et l'URI de leur port entrant offrant l'interface `EventReceptionCI`. S'ils ont besoin d'émettre des événements, ils doivent requérir l'interface de composant `EventEmissionCI` et se connecter au bus de la même manière qu'un composant représentant un dispositif physique émettant aussi des événements. À cette fin, l'opération d'enregistrement des corrélateurs auprès du bus retourne l'URI du port entrant du bus offrant l'interface `EventEmissionCI`, comme pour l'enregistrement des émetteurs d'événements.

Pour appeler un composant représentant un dispositif physique exécutant des actions, le composant corrélateur appelle l'opération `getExecutorInboundPortURI` sur le bus en passant l'URI du composant auquel il cherche à se connecter. Le bus retourne alors l'URI du port entrant de ce composant qui offre l'interface `ActionExecutionCI` . Après avoir récupéré cette URI de port entrant, le composant corrélateur se connecte directement au composant exécuteur de commandes via un port sortant requérant l'interface `ActionExecutionCI`.

3.4 Passage à un CEP réparti

Grâce à BCM, il devient possible de passer à un CEP réparti sur plusieurs JVM et potentiellement sur plusieurs ordinateurs. Si tous les préceptes de programmation de BCM ont été observés, la première étape consistera simplement à définir un déploiement réparti du système sur plusieurs JVM. L'objectif sera d'avoir quatre JVM pour déployer un corrélateur par JVM, les autres composants étant déployés entre ces quatre JVM. Ce premier pas vous permettra de vérifier que votre code passe bien au réparti.

Toutefois, un tel déploiement demande de décider dans quelle JVM sera déployé le composant `CEPBus`. Ce choix soulève un problème de performance propre au déploiement réparti. Lorsqu'un événement est transmis entre deux composants et que ces deux composants sont dans la même JVM , l'appel va se faire avec la performance d'un appel Java (un peu moins en réalité car un appel entre composants dans la même JVM demande plusieurs appels Java). Par contre, si les deux compsants sont déployés dans deux JVM distinctes, cet appel va se faire avec la performance d'un appel RMI qui peut être entre 2 ou 3 ordres de grandeurs (100 à 1000 fois) plus lent qu'un appel Java. On comprend tout de suite que si un événement est émis dans une JVM et reçu plusieurs fois dans chacune des JVM, on va se retrouver avec un grand nombre d'appels RMI coûteux.

Pour réduire le nombre d'appels RMI, on va repliquer le composant `CEPBus` à raison d'une copie par JVM. Cette architecture abstraite est illustrée à la figure 7. Supposons un événement e

émis dans une JVM 1, reçu par un corrélateur dans la JVM 1 et par trois autres dans la JVM 2. Sans réplication, un composant `CEPBus` déployé dans la JVM 1 devrait faire un appel Java pour transmettre au corrélateur de la JVM 1 puis trois appels RMI pour transmettre e aux corrélateurs de la JVM 2. Avec réplication, la copie de `CEPBus` de la JVM 1 va recevoir e, le transmettre avec un appel Java au corrélateur de la JVM 1 puis elle va transmettre e à la copie de `CEPBus` de la JVM 2 avec un appel RMI, puis cette dernière va le retransmettre aux trois corrélateurs locaux avec trois appels Java. On passe d'un appel Java et 3 appels RMI à 4 appels Java et un appel RMI.

Pour aller au bout de cette idée, il faut définir un protocole qui va permettre aux copies de `CEPBus` de se transmettre les événements pour en faire la transmissions locale aux récepteurs. Par exemple, il est possible d'organiser les copies de `CEPBus` en anneau circulaire pour être sûr que toutes les copies pourront se transmettre tous les événements pour leur transmission locale. Les composants `CEPBus` vont alors devoir requérir et offrir l'interface de composant `EventReceptionCI` pour se passer les événements.

4 Application « ville intelligente »

Pour tester l'implantation du CEP, vous allez développer une application reprenant les concepts de « ville intelligente » populaire aujourd'hui et qui commencent à être mise en œuvre dans le monde. L'idée est de mettre en place des capteurs et des dispositifs de pilotage à distance des services et équipements de nos villes pour améliorer le vie en ville en optimisant les services existants et en offrant de nouveaux services.

4.1 Description de la « ville »

Notre ville intelligente va être une version largement simplifiée de ce que ce concept vise en réalité, mais suffisamment riche pour tester différents cas d'utilisation et permettre une montée en charge pour tester la pertinence de vos solutions de gestion du parallélisme et de la concurrence dans votre implantation. Les éléments qui vont être représentés dans cette ville intelligente simplifiée sont les suivants :

Centres SAMU : un centre SAMU dispose d'ambulances avec leurs équipes et de médecins en résidence qui peuvent soit faire des téléconsulations téléphoniques ou en visoconférence quand ils sont sur place ou encore intervenir sur le terrain avec leur véhicule pour répondre à des urgences qui ne nécessitent pas de transporter les patients. Les événements ainsi que les règles servant de cas d'utilisation pour les centres SAMU sont décrits de manière informelle mais structurée dans l'annexe A.1.

Casernes de pompiers : une caserne de pompiers dispose d'équipes de pompiers et de véhicules de trois types : camion standard sans échelle, camion avec échelle courte capable d'intervenir sur des feux dans des immeubles de trois étages maximum et des camions à grande échelle capables d'intervenir sur des feux dans des immeubles de huit étages au maximum. Les événements ainsi que les règles servant de cas d'utilisation pour les casernes de pompiers sont décrits de manière informelle mais structurée dans l'annexe A.2.

Réseau de circulation : le réseau de circulation prend une forme régulière typique des villes nord-américaines avec des rues parallèles numérotées du nord au sud croisant des avenues perpendiculaires elles-mêmes numérotées de l'ouest vers l'est. À chaque croisement se trouve un feu de circulation. Les positions sur cette grille sont fournies par des coordonnées en deux dimensions : selon les rues et selon les avenues. Les croisements correspondent à des coordonnées entières puisqu'ils se situent sur les rues et les avenues. Par exemple, s'il y a 10 rues et 10 avenues, le feu (4,5) se trouve au croisement de la 4^{e}ruen et de la 5^{e}avenue. Les adresses de bâtiments ou de maisons se caractérisent par le fait que la coordonnée de l'une ou l'autre des deux dimensions (mais pas les deux) est un réel entre deux croisements. Par exemple, une caserne de pompier peut se trouver à la position (2.5, 3), c'est à dire sur la 3^{e}avenue, entre les 2^{e}et 3^{e}rues. Les feux de circulation sont pilotables pour provoquer le changement à vert, pour les bloquer en position vert dans un certain sens de circulation ou encore les mettre dans un mode où seuls les véhicules prioritaires peuvent passer. *Cas d'utilisation à venir.*

Véhicules : des véhicules circulent sur le réseau de circulation, passant les feux, depuis un point de départ jusqu'à leur destination donnés par des positions. Les ambulances et les camions de pompiers sont considérés comme des véhicules d'urgence qui circulent depuis le centre SAMU ou la caserne de pompiers jusqu'à une adresse d'intervention puis rentre à leur point de départ. Les véhicules peuvent émettre leur position. *Cas d'utilisation à venir.*

L'objectif du projet est de mettre en œuvre les cas d'utilisations énoncés sous forme de règles dans l'annexe de ce document. Ces règles font apparaître dans leur définition les événements et les actions impliquées, indiquant ainsi aussi ces éléments qui devront être implantés.

4.2 Simulateur

Pour tester votre code, il vous sera fourni des simulateurs de la ville intelligente qui se distingueront par leur degré de réalisme, leur taille et la quantité de notifications émises.

Simulateur de base : ce simulateur ne fait qu'émettre des données correspondants à tous les événements possibles dans l'application et selon les patrons définis dans les règles décrites dans les annexes de ce document. Ce simulateur vise à fournir une base simple permettant de tester si votre code reconnaît bien tous les événements et les patrons d'événements. Il offre aussi des méthodes correspondants à toutes les actions mais qui se contentent simplement de fournir une trace des appels pour vous permettre de tester si le déclenchement de vos règles vont bien jusqu'à exécuter les actions souhaitées sur le système.

Simulateur de très petite ville : ce simulateur implante une version réaliste de la ville mais d'une taille très petite pour faciliter les premiers tests : un seul centre SAMU, une seule caserne de pompiers, réseau de circulation à trois rues et trois avenues, juste suffisant pour tester votre code dans un contexte réaliste. Il émet juste assez d'événemets selon des patrons correspondants à toutes les règles définis dans le document. Il simule également les actions, ce qui permet de vérifier toute la chaîne depuis l'émission des événements jusqu'à l'exécution des actions en constant leurs effets.

Simulateur complet : ce simulateur reprend le simulateur simple mais il propose plusieurs centres SAMU, plusieurs casernes de pompiers, un réseau de circulation de plus grande taille et il injecte des véhicules non prioritaires dans ce réseau pour constater les effets des décision sur les feux (passage au vert, sens prioritaire, etc.).

Ces simulateurs sont programmés de telle façon à émettre des notifications et pour les deux derniers à fournir des services sur des composants permettant d'exécuter les actions prévues dans les règles. Les notifications vont fournir les données nécessaires pour créer vos événements alors que les services définis vont indiquer comment les actions des règles vont devoir les appeler pour agir sur le simulateur. En quelque sorte, ces définissent des scénarios de test pour votre code. Le simulateur de base vous servira essentiellement à mettre au point votre code. Le simulateur de très petite ville servira de scénario de test d'intégration pour la soutenance de mi-semestre. Le simulateur complet servira de test d'intégration lors de la soutenance finale.

Ces simulateurs vont présenter des interfaces et des points de connexion sous la forme de ports sur des composants qui vont vous permettre de connecteur vos émetteurs d'événements et vos exécuteurs d'actions de manière à créer et émettre vos événements puis à exécuter vos actions pour piloter la ville intelligente. Ces interfaces et points d'interconnexion vont être définis très bientôt et ce document sera complété pour les décrire. Les simulateurs vous seront fournis sous la forme de jars avec des exemples de composants connectés aux sources de données des simulateurs et de la documentation javadoc vous expliquant comment les utiliser pour implanter vos propres émetteurs d'événements et exécuteurs d'actions.

5 Déroulement du projet

5.1 Étape 1 : événements et règles

Pour la première étape du projet, qui va faire l'objet du premier audit puis indirectement de la soutenance à mi-semestre (voir plus loin les modalités d'évaluation), il faudra mettre en place

tout ce qui concerne les événements, les bases d'événements, les règles et les bases de règles. Pour tester votre code, vous pourrez commencer à implanter les événements et les règles de l'application ville intelligente dont les descriptions sont données en annexe.

Résultats attendus pour l'étape 1

— Tous les éléments concernant les événements, les bases d'événements, les règles de corrélation et les bases de règles.
— Les tests unitaires JUnit pour les éléments précédents testant l'ensemble des événements et des règles de l'application ville intelligente. Pour les règles, le filtrage et l'exécution d'actions vont nécessiter la programmation de bouchons logiciels qui vont simplement tracer qu'ils ont bien été appelés et qui, pour les conditions, vont fournir les réponses attendues par le test pour qu'il se déroule comme attendu.

À cette étape, le principal critère d'évaluation sera d'avoir complété le développement des éléments demandés ainsi que le taux de couverture des cas d'utilisation par vos cas de test JUnit et leur exécution correcte.

5.2 Étape 2 : implantation séquentielle du CEP

Pour la seconde étape, une première version séquentialisée du CEP devra être développée, ce qui permettrea de tester l'ensemble des fonctionnalités du système. Cette version utilisera un minimum de *threads* puisqu'essentiellement séquentialisée.

Résultats attendus pour l'étape 2

Pour l'intégration dans le modèle à composants BCM4Java, à partir du diagramme de la figure 6 et les interfaces que ce dernier déclare, le système doit définir :

— toutes les ports entrants et sortants ainsi que les connecteurs correspondant aux interfaces de composants de la figure 6 ;
— tous les types de composants et les composants demandés : bus d'événements, corrélateurs, émetteurs d'événements et des exécuteurs d'actions de l'application ville intelligente ;
— les greffons (*plug-ins*) pour chacun des rôles d'émetteur d'événements et d'exécuteur de commandes et leur utiliser par les composants ;
— des scénarios de tests sous la forme de déploiements en mono-JVM avec enregistrement des composants auprès du bus d'événements, émissions d'événements des corrélations et des exécutions d'actions dûment tracées ; ces exécutions serviront de test d'intégration et vont s'appuyer sur le simulateur de très petite ville fourni.

Vous devrez rendre votre code le dimanche soir précédant la soutenance (voir les modalités décrites ci-après). Pour cette deuxième étape, en plus du taux de couverture des réalisations demandées et de leur bon fonctionnement en exécution avec le simulateur, la qualité de l'ensemble de votre code (y compris les éléments développés à l'étape 1) sera également parmi les critères d'évaluation.

5.3 Étape 3 : greffons, gestion du parallélisme et de la concurrence

Parallélisme et concurrence internes au bus d'événements

La capacité de montée en charge du bus d'événements exige d'introduire du parallélisme, donc de gérer explicitement les *threads*, leur nombre et leur répartition entre les fonctions des composants. Ce parallélisme induit à son tour la nécessité de définir des structures de données appropriées dont la gestion de concurrence assurera la cohérence malgré le parallélisme.

La réception des événements par le bus devra pouvoir utiliser un *pool* de *threads* spécifique de manière à pouvoir configurer par la suite la répartition des *threads* entre les différentes fonctions du bus d'événements. Ainsi, le routage interne (utilisant les abonnements), la livraison des événements aux récepteurs ainsi que la gestion des enregistrements et des abonnements vont aussi confiés à des *pools* de *threads* distincts à tailles configurables. L'ensemble de ces *threads* vont devoir accéder

aux structures de données partagées que vous aurez définies à l'étape 2 ; il faudra donc définir des sections critiques dans les traitements et utiliser des moyens de synchronisation pour imposer l'exclusion mutuelle nécessaire entre ces sections critiques.

Les corrélateurs pourront aussi utiliser plusieurs *pools* de *threads* pour gérer efficacement le déclenchement des règles par rapport aux exécutions d'actions. Toutefois, le problème peut devenir plus complexe que pour le bus d'événements car il y a plus de dépendances d'ordre d'exécution entre les règles. L'introduction d'une gestion du parallélisme et de la concurrence dans les corrélateurs est donc une option qui donnera lieu à un bonus lors de l'évaluation si elle a été réalisée.

Résultats attendus pour l'étape 3

— la nouvelle version du composant `CEPBus` avec gestion du parallélisme et de la concurrence ;
— en option, la nouvelle version des corrélateurs avec gestion du parallélisme et de la concurrence ;
— des scénarios de tests sous la forme de déploiements en mono-JVM avec enregistrement des composants auprès du bus d'événements, émissions d'événements des corrélations et des exécutions d'actions dûment tracées ; ces exécutions serviront de test d'intégration et elles pourront utiliser soit le simulateur de très petite ville, soit le simulateur complet.

Les principaux critères d'évaluation à cette étape seront la pertinence des choix de gestion de parallélisme et de la concurrence, leur mise en œuvre et la bonne exécution du test d'intégration.

5.4 Étape 4 : répartition et évaluation de performance

La dernière étape va se concentrer sur une version répartie du système de traitement des événements complexes puis sur une évaluation de la performance du bus d'événements dans la transmission des événements.

Répartition du bus d'événements

Il s'agit de mettre en œuvre la répartition du bus d'événements tel que décrite dans la sous-section 3.4 et de réussir à en faire une exécution sur plusieurs JVM localisées sur un seul ordinateur.

Performance du bus d'événements

Après avoir introduit la gestion du parallélisme et de la concurrence à l'étape 3, il s'agit ici d'utiliser ces possibilités pour gérer la performance du bus. Dans le contexte d'applications de grande taille exécutés dans des centres de calcul, la gestion des ressources devient une compétence de plus en plus appréciée par l'industrie chez les ingénieurs loigiciels. Cette partie du projet vise donc à vous introduire à ce genre de questions. L'idée générale va être de se placer dans la situation où chaque instance du bus d'événements dans la version répartie du CEP dispose d'un nombre limité de *threads* et il va falloir les répartir entre les différentes fonctions du composant (voir étape 3). Pour prendre cette décision d'affectation des ressources, vous devrez mener une série d'expérimentations en modifiant la répartition et en mesurant la performance de bout en bout dans la gestion des événements (délai entre l'émission d'événements et exécution de l'action), ce qui va vous demander aussi d'instrumenter votre code pour réaliser ces mesures.

Résultats attendus pour l'étape 4

Les résultats attendus pour l'étape 4 comportent :
— l'implantation de la répartition du bus d'événements en `CEPBus` déployables sur plusieurs JVM et interconnectables ;
— des scénarios de tests sous la forme de déploiements en multi-JVM avec des transmissions d'événements intra- et inter-JVM ainsi que des corrélations et des exécutions de commandes dûment tracées pour que les exécutions puissent servir de test d'intégration ;

— un plan d'expérimentation pour la configuration d'allocation des ressources (*threads*) avec des tableaux présentant les mesures obtenues (à utiliser dans votre soutenance finale).

6 Modalités générales de réalisation et calendrier des évaluations

— Le projet se fait **obligatoirement** en **équipe de deux étudiant·e·s**. Tous les fichiers sources du projet doivent comporter les noms (balise `authors`) de tous les auteurs en Javadoc. Lors de sa formation, chaque équipe devra se donner un nom et me le transmettre avec les noms des étudiant·e·s la formant au plus tard le **4 février 2022** à minuit.
— Le projet doit être réalisé avec **Java SE 8**. Attention, peu importe le système d'exploitation sur lequel vous travaillez, il faudra que votre projet s'exécute correctement sous Eclipse et sous **Mac Os X/Unix** (que j'utilise et sur lequel je devrai pouvoir refaire s'exécuter vos tests).
— L'évaluation comportera quatre épreuves : deux audits intermédiaires, une soutenance à mi-semestre et une finale, ces dernières accompagnées d'un rendu de code et de documentation. Ces épreuves se dérouleront selon les modalités suivantes :

1. Les deux audits intermédiaires dureront 15 minutes au maximum (par équipe) et se dérouleront lors des séances de TME. Le premier audit se tiendra pendant la séance 4 (21 février 2022) et il portera sur votre avancement de l'étape 1. Le second audit se déroulera lors de la séance 9 (11 avril 2022) et il portera sur votre avancement de l'étape 3. Ils compteront chacun pour 5% de la note finale de l'UE.
2. La **soutenance à mi-parcours** d'une durée de 20 minutes portera sur l'atteinte des objectifs des *deux premières étapes* du projet. Elle se tiendra pendant la semaine des premiers examens répartis du **14 au 18 mars 2022** selon un ordre de passage et des créneaux qui seront annoncés sur le site de l'UE. Elle comptera pour 35% de la note finale de l'UE. Elle comportera une discussion des réalisations pendant une quinzaine de minutes (devant l'écran sous Eclipse) et une courte démonstration de cinq minutes. Les rendus à mi-parcours se feront le **dimanche 13 mars 2022 à minuit** au plus tard (des pénalités de retard seront appliquées).
3. La **soutenance finale** d'une durée de 30 minutes portera sur l'ensemble du projet mais avec un accent sur les *troisième et quatrième étapes*. Elle aura lieu dans la semaine des seconds examens répartis du **16 au 20 mai 2022** selon un ordre de passage et des créneaux qui seront annoncés sur le site de l'UE. Elle comptera pour 55% de la note finale de l'UE. Elle comportera une présentation d'une douzaine de minutes (en utilisant des transparents), une discussion d'une dizaine de minutes également devant écran sur les réalisations suivie d'une démonstration. Les rendus finaux se feront le **dimanche 15 mai 2022 à minuit** au plus tard (des pénalités de retard seront appliquées).

— Lors des soutenances, les points suivants seront évalués :
  — le respect du cahier des charges et la qualité de votre programmation ;
  — l'exécution correcte de tests unitaires (JUnit pour les classes et les objets Java, scénario de tests pour les composants) ;
  — l'exécution correcte de tests d'intégration mettant en œuvre des composants de tous les types ;
  — la qualité et l'exécution correcte des scénarios de tests de performance, conçus pour mettre à l'épreuve les choix d'implantation versus la montée en charge ;
  — la qualité de votre code (votre code doit être commenté, être lisible - choix pertinents des identifiants, . . .– et correctement présenté – indentation, . . .– etc.) ;
  — la qualité de votre plan d'expérimentation et tests de performance (couverture de différents cas, isolation des effets pour identifier leurs impacts sur la performance globale, etc.) ;
  — la qualité de la documentation (vos rendus pour la soutenance finale devront inclure une *documentation Javadoc* des différents paquetages et classes de votre projet générée et incluse dans votre livraison dans un répertoire `doc` au même niveau que votre répertoire `src`).
— Bien que les audits et les soutenances se fassent par équipe, l'évaluation reste à chaque fois **individuelle**. Lors des audits et des soutenances, *chaque étudiant·e* devra se montrer

capable d'expliquer différentes parties du projet, et selon la qualité de ses explications et de ses réponses, sa note peut être supérieure, égale ou inférieure à celle de l'autre membre de son équipe.

— Lors des soutenances, **tout retard** non justifié d'un·e ou des membres de l'équipe de plus d'un tiers de la durée de la soutenance (7 minutes à la soutenance à mi-semestre, 10 minutes à la soutenance finale) entraînera une **absence** et une note de 0 attribuée au(x) membre(s) retardataire(s) pour l'épreuve concernée. Si un·e des membres d'une équipe arrive à l'heure ou avec un retard de moins d'un tiers de la durée de la soutenance, il·elle passera l'épreuve seul·e.

— Le rendu à mi-parcours et le rendu final se font sous la forme d'une archive `tgz` si vous travaillez sous Unix ou `zip` si vous travaillez sous Windows que vous m'enverrez à `Jacques.Malenfant@lip6.fr` comme attachement fait proprement avec votre programme de gestion de courrier préféré ou encore par téléchargement avec un lien envoyé par courrier électronique (en lieu et place du fichier). Donnez pour nom au répertoire de projet et à votre archive celui de votre équipe (ex. : équipe LionDeBelfort, répertoire de projet `LionDeBelfort` et archive `LionDeBelfort.tgz`).

— **Tout manquement à ces règles élémentaires entraînera une pénalité dans la note des épreuves concernées !**

— Pour la deuxième session, si elle s'avérait nécessaire, elle consiste à poursuivre le développement du projet pour résoudre ses insuffisances constatées à la première session et donnera lieu à un rendu du code et de documentation puis à une soutenance dont les dates seront déterminées en fonction du calendrier du master.

A Règles de gestion et de corrélation des événements

A.1 SAMU

S1	événements	alarme santé en p avec type = urgence
	corrélations	aucun
	filtres	p est dans la zone du centre SAMU S ambulance disponible en S
	actions	déclencher une intervention d'ambulance par S
	effets	aucun
S2	événements	alarme santé en p avec type = urgence
	corrélations	aucun
	filtres	p est dans la zone du centre SAMU S ambulance non disponible en S centre SAMU S' est le suivant le plus près
	actions	propager l'alarme santé à S'
	effets	retirer l'événement alarme santé en p
S3	événements	alarme santé en p avec type = médicale
	corrélations	aucun
	filtres	p est dans la zone du centre SAMU S médecin disponible en S
	actions	déclencher intervention de médecin par S
	effets	aucun
S4	événements	alarme santé en p avec type = médicale
	corrélations	aucun
	filtres	p est dans la zone du centre SAMU S médecin non disponible en S centre SAMU S' est le suivant le plus près
	actions	propager l'alarme santé à S'
	effets	retirer l'événement alarme santé en p
S5	événements	alarme santé en p avec type = traçage
	corrélations	aucun
	filtres	délai depuis l'événement > 10 minutes médecin disponible en S
	actions	déclencher intervention de médecin par S mémoriser intervention traçage en p
	effets	retirer l'événement alarme santé en p
S6	événements	alarme santé en p avec type = traçage
	corrélations	aucun
	filtres	délai depuis l'événement > 10 minutes médecin non disponible en S centre SAMU S' est le suivant le plus près
	actions	propager l'alarme santé à S'
	effets	retirer l'événement alarme santé en p
S7	événements	alarme santé pour personne1 avec type = traçage signalement manuel pour personne2
	corrélations	personne1 est égale à personne2 délai entre les événements <= 10 minutes
	filtres	médecin disponible en S
	actions	déclencher un appel du médecin à la personne
	effets	retirer les deux événements

S8	événements	alarme santé pour personne1 avec type = traçage signalement manuel pour personne2
	corrélations	personne1 est égale à personne2 délai entre les événements <= 10 minutes
	filtres	médecin non disponible en S centre SAMU S' est le suivant le plus près
	actions	propager l'événement complexe chute de personne consciente à S'
	effets	retirer les deux événements
S9	événements	chute de personne consciente
	corrélations	aucun
	filtres	médecin non disponible en S pas d'autre centre SAMU accessible
	actions	aucune
	effets	retirer les deux événements
S10	événements	toutes ambulances de S en intervention
	corrélations	aucun
	filtres	aucune
	actions	passer ambulances en S en non disponibles
	effets	retirer l'événement toutes ambulances de S en intervention
S11	événements	tous médecins de S en intervention
	corrélations	aucun
	filtres	aucune
	actions	passer médecins en S en non disponibles
	effets	retirer l'événement tous médecins de S en intervention
S12	événements	ambulances disponibles en S
	corrélations	aucun
	filtres	aucune
	actions	passer ambulances en S en disponibles
	effets	retirer l'événement ambulances disponibles en S
S13	événements	médecins disponibles en S
	corrélations	aucun
	filtres	aucune
	actions	passer médecins en S en disponibles
	effets	retirer l'événement médecins disponibles en S

A.2 Pompiers

F1	événements	alarme feu en position p avec type = immeuble
	corrélations	aucun
	filtres	p est dans la zone de la caserne C grande échelle disponible en C
	actions	déclencher une intervention sur immeuble par la la caserne C
	effets	remplacer l'événement alarme par un évenement première alarme dans un immeuble en p
F2	événements	alarme feu en position p avec type = maison
	corrélations	aucun
	filtres	p est dans la zone de la caserne C camion sans échelle disponible en C
	actions	déclencher une intervention sur maison par la caserne C
	effets	remplacer l'événement alarme par un évenement première alarme dans une maison en p
F3	événements	alarme feu en position p avec type = immeuble
	corrélations	aucun
	filtres	grande échelle non disponible en C caserne C' est la suivante la plus près de p
	actions	propager l'événement alarme dans un immeuble en position p à C'
	effets	aucune

F4	événements	alarme feu en position p avec type = maison
	corrélations	aucun
	filtres	camion sans échelle non disponible en C caserne C' est la suivante la plus près de p
	actions	propager l'événement alarme dans une maison en position p à C'
	effets	aucune
F5	événements	première alarme en p1 avec type = immeuble alarme feu en position p2 avec type = immeuble
	corrélations	p1 est égale à p2 délai entre les deux événements <= 15 minutes
	filtres	p est dans la zone de la caserne C
	actions	déclencher une intervention générale en p par la caserne C propager un événement alarme générale en p à toutes les autres casernes
	effets	remplacer l'événement première alarme dans un immeuble en p par un événement alarme générale en p
F6	événements	alarme générale en p
	corrélations	aucun
	filtres	grande échelle non disponible en C
	actions	déclencher une intervention générale en p par la caserne C
	effets	aucun
F7	événements	première alarme en p1 avec type = maison alarme en position p2 avec type = maison
	corrélations	p1 est égale à p2 délai entre les deux événements <= 15 minutes
	filtres	camion sans échelle disponible en C
	actions	envoyer un camion sans échelle en renfort en p
	effets	remplacer l'événement première alarme en p par un événement seconde alarme en p
F8	événements	première alarme en p1 avec type = maison alarme feu en position p2 avec type = maison
	corrélations	p1 est égale à p2 délai entre les deux événements <= 15 minutes
	filtres	camion sans échelle non disponible en C caserne C' est la suivante la plus près de p
	actions	propager un événement seconde alarme en p à C'
	effets	remplacer l'événement première alarme en p par un événement seconde alarme en p
F9	événements	seconde alarme en p
	corrélations	aucun
	filtres	camion sans échelle disponible en C
	actions	envoyer un camion sans échelle en renfort en p
	effets	aucun
F10	événements	seconde alarme en p
	corrélations	aucun
	filtres	camion sans échelle non disponible en C caserne C' est la suivante la plus près de p
	actions	propager l'événement seconde alarme en p à C'
	effets	aucun
F11	événements	toutes grandes échelles de C en intervention
	corrélations	aucun
	filtres	aucune
	actions	passer grande échelle en C non disponible
	effets	retirer l'événement toutes grandes échelles de C en intervention
F12	événements	tous camions sans échelle de C en intervention
	corrélations	aucun
	filtres	aucune
	actions	passer camion sans échelle en C non disponible
	effets	retirer l'événement tous camions sans échelle de C en intervention

F13	événements	grandes échelles disponibles en C
	corrélations	aucun
	filtres	aucune
	actions	passer grande échelle en C disponible
	effets	retirer l'événement grandes échelles disponibles en C
F14	événements	camions sans échelle disponibles en C
	corrélations	aucun
	filtres	aucune
	actions	passer camion sans échelle en C disponible
	effets	retirer l'événement camions sans échelle disponibles en C
F15	événements	fin de feu en p1 première alarme en p2
	corrélations	p1 est égale à p2
	filtres	aucune
	actions	propager l'évenement fin de feu en p à toutes les casernes
	effets	retirer les événements fin de feu en p et première alarme en p
F16	événements	fin de feu en p1 seconde alarme en p2
	corrélations	p1 est égale à p2
	filtres	aucune
	actions	propager l'évenement fin de feu en p à toutes les casernes
	effets	retirer les événements fin de feu en p et seconde alarme en p
F17	événements	fin de feu en p1 alarme générale en p2
	corrélations	p1 est égale à p2
	filtres	aucune
	actions	propager l'évenement fin de feu en p à toutes les casernes
	effets	retirer les événements fin de feu en p et alarme générale en p

A.3 Réseau de circulation

À venir.