

# Iteration 3

---

L'itération 3 batit sur l'itération 2, tout votre travail vous sera grandement utile pour celle-ci. Dans un effort de vous aidez, nous avons fait certaines mise à jours aux systèmes existant.

## Docker Compose

---

Le NodeController gère les conteneurs une fois ceux-ci lancés, mais il demeure dépendant des configurations faites dans docker compose. Afin de facilité la configuration de votre architecture, certains changements ont été fait à la syntaxe spécifique au laboratoire.

### Changement de syntaxe

```
tripcomparator2:
  container_name: AUX.TripComparator
  image: tripcomparator
  restart: always
  build:
    context: ../
    dockerfile: TripComparator/Controllers/Dockerfile
  environment:
    SERVICES_ADDRESS: '${SERVICES_ADDRESS}'
    NODE_CONTROLLER_PORT: '${NODE_CONTROLLER_PORT}'
    ASPNETCORE_URLS: 'http://+:80'
    ID: TripComparator2
    MQ_SERVICE_NAME: EventStream
  ports:
    - '32775:80'
  labels:
    ARTIFACT_CATEGORY: Computation
    ARTIFACT_NAME: TripComparator
    POD_NAME: AUX
    POD_ID: Init2
    REPLICAS: 3
    DNS: Public
  depends_on:
    - nodecontroller
```

Ancienne syntaxe

```

tripcomparator:
  container_name: TripComparator
  image: tripcomparator
  restart: always
  build:
    context: ../
    dockerfile: TripComparator/Controllers/Dockerfile
  environment:
    SERVICES_ADDRESS: '${SERVICES_ADDRESS}'
    NODE_CONTROLLER_PORT: '${NODE_CONTROLLER_PORT}'
    ID: Core.TripComparator
    MQ_SERVICE_NAME: EventStream
  ports:
    - '32773:80'
  labels:
    ARTIFACT_CATEGORY: Computation
    Replicas: 1
    Pod_Links: RouteTimeProvider, stm
  depends_on:
    - nodecontroller

```

Nouvelle syntaxe

## Changements

- Pod\_Links: Il n'est plus nécessaire de spécifier le POD\_ID, POD\_NAME et DNS dans chaque services. Vous pouvez simplement écrire le nom de la variable d'environnement "ID" des services qui compose le pod. Le DNS du tripComparator serait automatiquement déclaré comme Public. De plus, de cette façon il est possible de définir plusieurs pods différents sans que chaque services qui les compose soit redéfinit plusieurs fois dans le compose (tripcomparator2).
- Replicas: Il est possible d'inscrire le nombre de replications du pod ou d'écrire les hostnames à assigner au service principal des pods (le système déduit le nombre de répliques souhaité à partir du nombre de noms). Particulièrement important avec rabbitmq puisque les clusters sont formés à partir, entre autres, des hostnames.

```

labels:
  ARTIFACT_CATEGORY: Connector
  Replicas: EventStream, EventStream2, EventStream3

```

3 services avec les hostnames décrit sont garantis par le nodecontroller

- ARTIFACT\_NAME: Ce n'est plus nécessaire! Le "ID" dans environnement est désormais utilisé.
- POD\_NAME: Si vous souhaitez définir un nom spécial pour votre pod, pour des raisons de networking ou par soucis d'organisation, dans l'image ci-bas le pod de nom "Core" est défini dans le "ID" de environnement suivi du nom du service.

```
environment:
  SERVICES_ADDRESS: '${SERVICES_ADDRESS}'
  NODE_CONTROLLER_PORT: '${NODE_CONTROLLER_PORT}'
  ID: Core.TripComparator
  MQ_SERVICE_NAME: EventStream
```

## NodeController

---

Le NodeController va subir une mise à jour significative, au moment de l'écriture de ce document, les fonctionnalités sont tous en place et les tests sont presque finis (disponible le 14 ou 15 juillet). Le but de cette mise à jour est d'alléger votre charge de travail de multiples façons pour l'itération 3.

### Nouvelles fonctionnalités

1. L4 Load Balancing / Routing: Le NodeController permettait déjà de faire du service discovery et du balancement de charge sur L7 (Application - http). Puisque cette itération ci s'attaque aux queues de messages et aux bases de données qui fonctionnent habituellement avec des TCP Sockets, le L7 en place ne peut pas gérer ces connexions de manière appropriée. Au lieu de vous forcer à déployer un load balancer à la Traefik, NGINX Plus ou autre et de le garder lui aussi en vie... Nous avons décidé d'en ajouter un au NodeController. Ce qui le rends capable autant en L7 que L4 (Transport). Pour fonctionner avec ce nouveau système si vous n'utilisez pas C#, vous devez faire un preflight au NodeController API **Routing/NegotiateSocket** avec le type de service auquel vous voulez être acheminé. La réponse est un int représentant un port. Vous vous connecteriez alors avec "[Protocol://]host.docker.internal:[Port]".
2. Stable Hostnames: Les hostnames des conteneurs (sauf sidecars) seront recyclés dans la mesure du possible quand les services sont recréés après un kill. Ce qui vous permet d'appeler un conteneur par nom plutôt qu'utiliser le NodeController ou autre pour avoir l'adresse d'un service.

```
environment:
  SERVICES_ADDRESS: '${SERVICES_ADDRESS}'
  NODE_CONTROLLER_PORT: '${NODE_CONTROLLER_PORT}'
  ID: Core.TripComparator
  MQ_SERVICE_NAME: EventStream
```

### Changements de fonctionnalités

1. Le système de lecture du nouveau format Docker Compose n'est pas compatible avec l'ancienne syntaxe (bien qu'il ne va pas crasher). Si vous voulez absolument qu'il soit compatible avec les anciennes configurations, faites-en part à votre chargé de lab avec une explication de votre cas spécifique.
2. L'algorithme de kills est modifié pour suivre une forme logarithmique plutôt que linéaire. Ce qui nous permet de mettre le nombre de Kills/min plus bas mais d'avoir un nombre total plus élevé sur la durée du test.
3. L'API Routing/... à été changé, vous pouvez avoir l'ancien en spécifiant dans vos requêtes la version de l'API V1. Le type retourné dans tous les cas est tout de même modifié un peu parce que 2 propriétés de "RoutingData" étaient largement non utilisées et ils ont été remplacés par 2 nouvelles propriétés Host et Port (principalement pour l'intégration dans d'autres langages que C#).

## ServiceMeshHelper

---

Le ServiceMeshHelper à subit une mise à jour lui aussi pour deux raisons. Un tutoriel de comment mettre la version a jour sera déposé en même temps que la mise a jour du NodeController.

1. Incompatibilité avec la STM: Dû à un conflit de version de Newtonsoft entre la stm et le ServiceMeshHelper celui ci rendait le conteneur inopérable a moins de résoudre la dépendance par vous même. La nouvelle version règle ce problème en éliminant la dépendance entièrement.
2. Ajout d'un TcpController: Le ServiceMeshHelper contient un RestController pour faire des requêtes http Get et Post, mais rien pour les TCP Sockets. Ce nouveau controleur gère l'entièreté du processus pour vous en incluant le Preflight au Nodecontrolleur. Il suffit de lui fournir le **Type** de service cible pour la connection. Le host de rabbitmq devrait utilisé cette méthode.

```
private static async Task ConfigureMassTransit(IServiceCollection services)
{
    await Task.Delay(5_000);

    var host:string = await TcpController.GetTcpSocketForSericeType(HostInfo.MqServiceName);

    const string baseQueueName = "time_comparison.node_controller-to-any.query";

    var uniqueQueueName = $"{baseQueueName}-{Guid.NewGuid()}";

    services.AddMassTransit(x:IBusRegistrationConfigurator =>
    {
        x.AddConsumer<TripComparatorMqController>();

        x.UsingRabbitMq(configure:(context, cfg) =>
        {
            cfg.Host(host);
        });
    });
}
```

## Conseils généraux

---

- Concentrez vous sur vos algorithms de synchronisation de l'itération 2 si ceux-ci n'était pas à niveau, ils sont tout aussi important pour cette itération.
- Utilisez des Quorum queues pour RabbitMq, l'option exist dans MassTransit.
- Pas de volumes, pas de hardware failures...
- Pour les cluster de rabbitmq, n'oubliez pas le hostname du service créé par docker compose, sinon le cluster ne pourra pas le trouvé si vous fonctionnez par DNS. Vous devez aussi vous assurer que chaque "Node" du cluster partage le même cookie, il peut être placé dans un fichier.

```

event-store:
  container_name: EventStream
  image: cluster_mq
  restart: always
  hostname: EventStream
  build:
    context: .
    dockerfile: Dockerfile
  ports:
    - '32771:5672'
    - '30001:15672'
    - '25673:25672'
  environment:
    ID: EventStream
    RABBITMQ_ERLANG_COOKIE_FILE: /var/lib/rabbitmq/.erlang.cookie
  labels:
    ARTIFACT_CATEGORY: Connector
    Replicas: EventStream, EventStream2, EventStream3

```

- Vous pouvez batir des images docker sur des images docker...

```

Dockerfile > ...
1 FROM rabbitmq:3.8-management
2
3 COPY .erlang.cookie /var/lib/rabbitmq/.erlang.cookie
4 RUN chown rabbitmq:rabbitmq /var/lib/rabbitmq/.erlang.cookie && \
5     chmod 600 /var/lib/rabbitmq/.erlang.cookie
6
7 COPY rabbitmq.conf /etc/rabbitmq/
8

```

Une nouvelle image est créé en batissant à partir de rabbitmq:3.8-management

- Voici une config de base pour résolution de DNS dans RabbitMq

```

rabbitmq.conf
1 cluster_formation.peer_discovery_backend = rabbit_peer_discovery_classic_config
2
3 cluster_formation.classic_config.nodes.1 = rabbit@EventStream
4 cluster_formation.classic_config.nodes.2 = rabbit@EventStream2
5 cluster_formation.classic_config.nodes.3 = rabbit@EventStream3
6
7 loopback_users = none
8

```

## Parametres du test

Attribut	Computation	Connecteurs	Database
cpu	0.25	-	-
memory	1000	-	-

kills/min	5	2	2
hardware failures	2	1	1
max number of pods	10	6	6