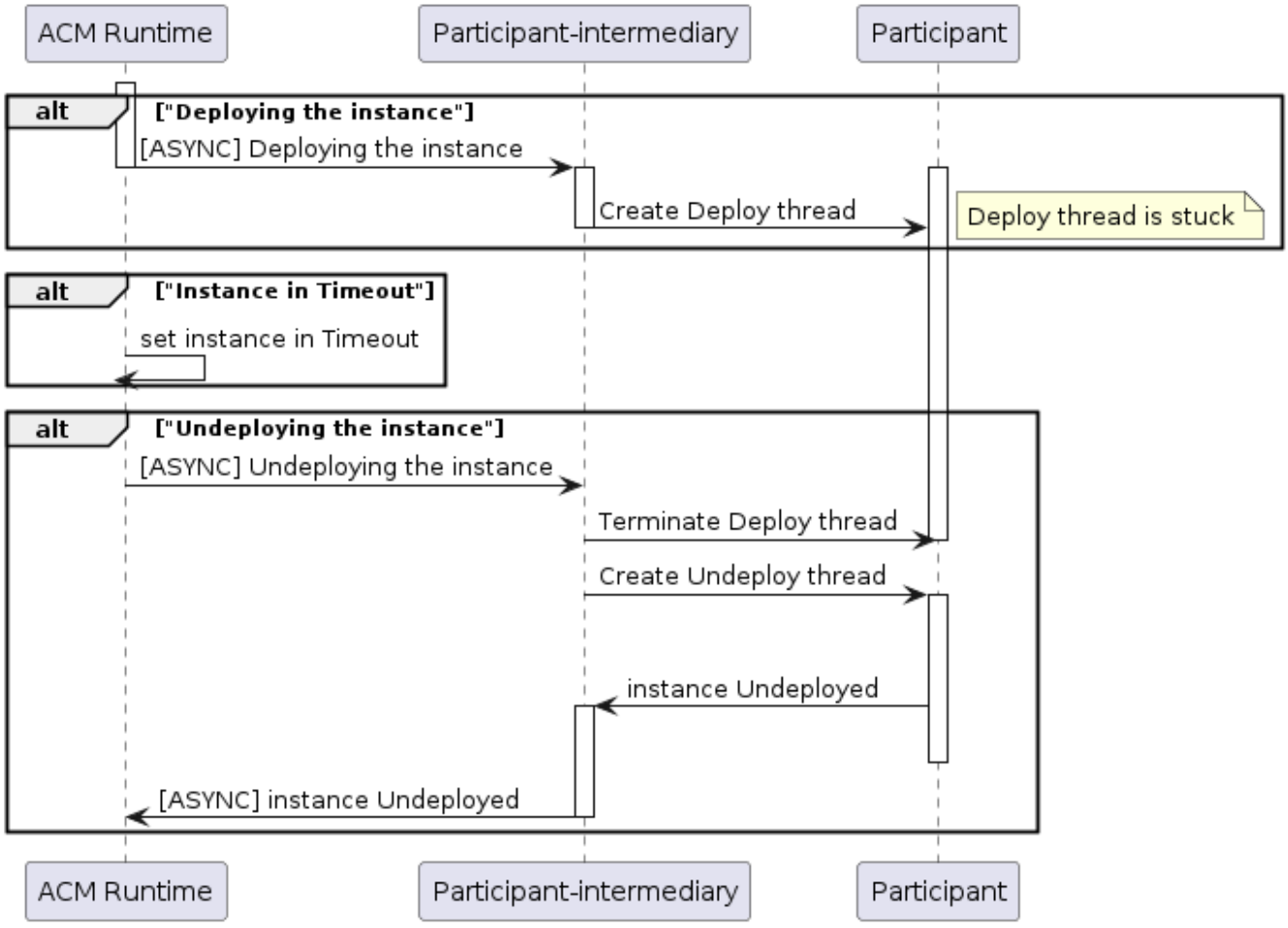
Replica support in Participant

In the current implementation, ACM supports multi-participant with same supported element Type but different participantId, so they need different properties file.

In order to support replication, we need to support replicas of participants.

Note:

- In a scenario of high number of compositions, if participant is restarting it will be slow-down the restarting action: AC-runtime will send a message for each composition primed and instance deployed to the participant.
  To avoid the restarting action, we need proper participant replication support;
- In a scenario where a participant is stuck in deploying, the instance will be in TIMEOUT and the user can take action like deploy again or undeploy. In that scenario the intermediary-participant has to receive the next message, kill the thread that is stuck in deploying and create a new thread.
- In a scenario where we are increasing the number or participants, could be useful to have different topic name for source and sink. This solution will eliminate the number of useless messages in kafka.
  Example:
  - for ACM-runtime:
    - sink: POLICY-ACM-PARTICIPANT
    - source: POLICY-ACM-RUNTIME
  - for participant:
    - sink: POLICY-ACM-RUNTIME
    - source: POLICY-ACM-PARTICIPANT



Solutions

Solution 1: Replicas and Dynamic participantId - still using cache

Changes in Participant intermediary:

- UUID participantId will be generated in memory instead to fetch it in properties file.

- consumerGroup will be generated in memory instead to fetch it in properties file.

Changes in ACM-runtime:

- When participant go OFF_LINE:
  - if there are compositions connected to that participant, ACM-runtime will find other ON_LINE participant with same supported element type;
  - if other ON_LINE participant is present it will change the connection with all compositions and instance;
  - after that, it will execute restart for all compositions and instances to the ON_LINE participant.
- When receive a participant REGISTER:
  - it will check if there are compositions connected to a OFF_LINE participant with same supported element type;
  - if there are, it will change the connection with all compositions and instances to that new registered participant;
  - after that it will execute restart for all compositions and instances changed.
  - Refactor restarting scenario to apply the restarting only for compositions and instances in transition

Issues:

- Participants create randomly participantId and Kafka consumerGroup. This solution has been tested and has the issue to create a new Kafka queue in restarting scenario.
  During restart scenario, a new consumerGroup is created, that cause some missing initial messages due the creation of new Kafka queue . The result is that to fail to receive messages from ACM to restore compositions and instances.

Solution 2: StatefulSets - still uses cache

Participant replicas can be a kubernetes StatefulSets that consume different properties file with unique consumer groups and unique UUIDs/replica (participants with same UUIDs have different replica number).

The StatefulSet uses the SPRING_CONFIG_NAME environment variable pointing to the spring application properties file unique to each of the participant replica.

Each of the properties file with the names pod-0.yaml, pod-1.yaml is mounted to the volumes. And the SPRING_CONFIG_NAME variable can be set to /path/to/$HOSTNAME.yaml to use the corresponding

properties file.

By this approach the participant can have multiple replicas to work with a shared data.

```
env:
- name: HOSTNAME
  valueFrom:
    fieldRef:
      fieldPath: metadata.name


- name: SPRING_CONFIG_NAME
  value: /path/to/${HOSTNAME}.yaml
```

```
For example considering the http participant replica, ${HOSTNAME} will be "policy-http-ppnt-0" and "policy-http-ppnt-1" and their
corresponding properties files with the names "http-ppnt-0.yaml" and "http-ppnt-1.yaml" is volume mounted.
```

Note: In a scenario of two participants in replicas (we are calling "policy-http-ppnt-0" and "policy-http-ppnt-1"), ACM-Runtime will assignee as before any composition definition in prime time to specific participant based of supported element definition type. All participants will receive the same messages and store same data, so all participants are synchronized with ACM-R. Into all messages from ACM-R to participants will be present the replica number to indicate what participant will do the job (like prime or deploy). Example ACM-R send deploy message with replica 0, so "policy-http-ppnt-0" save the new instance and deploy and "policy-http-ppnt-1" just save that instance. When "policy-http-ppnt-0" send outProperties, then ACM-R and "policy-http-ppnt-1" receive the message and save that properties. When "policy-http-ppnt-0" has completed the deploy, the send the message and then then ACM-R and "policy-http-ppnt-1" receive the message and save the result. No issue if ACM-R send an undeploy message with replica 1, because all applications are synchronized.

Changes in Participant:

- Register, Status and Unregister message have to contain the replica number
- store data from messages from ACM-R if the participantId is matching and enable the actions only if the participantId and replica number are matching with the message
- store data (outProperties, and action completed) from messages from participants with same participantId and different replica
- implement time-out to stop the process if is running out of time

Changes in ACM-R:

- Save the replica numbers available
- Random replica into any message to participants

Changes in docker/Kubernetes environment

- implement StatefulSets

Solution 3: Replicas and Database support - no cache

Changes in Participant intermediary:

- Redesign TimeOut scenario: Participant has the responsibility to stop the thread in execution after a specific time.
- Add client support for database (MariaDB or PostgreSQL).
- Add mock database for Unit Tests.
- Refactor CacheProvider to ParticipantProvider to support insert/update, intermediary-participant with transactions.
- Refactor Intermediary to use insert/update of ParticipantProvider.
- Refactor Participants that are using own HashMap in memory (Policy Participant saves policy and policy type in memory)

Changes in Participant:

- Add @EnableJpaRepositories and @EntityScan in Application:
  - **Application**

```
@SpringBootApplication
@EnableJpaRepositories({
    "org.onap.policy.clamp.acm.participant.intermediary.persistence.repository"
})
@ComponentScan({
    "org.onap.policy.clamp.acm.participant.sim",
    "org.onap.policy.clamp.acm.participant.intermediary"
})
@EntityScan({
    "org.onap.policy.clamp.acm.participant.intermediary.persistence.concepts"
})
@ConfigurationPropertiesScan("org.onap.policy.clamp.acm.participant.sim.parameters")
public class Application {

    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

- Add db connection in properties file and properties file for tests:

properties.yaml

```
spring:
  security:
    user:
      name: participantUser
      password: zb!XztG34
  mvc:
    converters:
      preferred-json-mapper: gson
  datasource:
    url: jdbc:mariadb://${mariadb.host:localhost}:${mariadb.port:3306}/participantsim
    driverClassName: org.mariadb.jdbc.Driver
    username: policy
    password: P01icY
    hikari:
      connectionTimeout: 30000
      idleTimeout: 600000
      maxLifetime: 1800000
      maximumPoolSize: 10
  jpa:
    hibernate:
      ddl-auto: update
      naming:
        physical-strategy: org.hibernate.boot.model.naming.PhysicalNamingStrategyStandardImpl
        implicit-strategy: org.onap.policy.common.spring.utils.CustomImplicitNamingStrategy
    properties:
      hibernate:
        format_sql: true
```

properties-test.yaml

```
spring:
  datasource:
    url: jdbc:h2:mem:testdb
    driverClassName: org.h2.Driver
    hikari:
      maxLifetime: 1800000
      maximumPoolSize: 3
  jpa:
    hibernate:
      ddl-auto: create
    open-in-view: false
```

- Unit Tests may need some changes

Changes in docker/Kubernetes environment

- Refactor CSIT to support database configuration for participants
- Refactor OOM to support database configuration for participants
- DB Migrator must be added to the helm chart and docker environments. The database schema of the older and newer versions will be different. Cache data added to the db.

Addition of DB Migrator

- Db migrator will alter old version of the db to add new parts of the schema required by this participant change
- Liquibase used for script generation
- Separate image needed for DB Migrator - this will have to be released as a new dependency
- New Job in kubernetes and new service in docker should be added for this migration

Advantages of DB use

- Multiple participant replicas possible - it can deal with messages across many participants
- All participants should have same group-id in kafka
- All should have the same participant-id.

Disadvantages of DB use

1. The Participant as a concept envisages an extremely lightweight mechanism, which can be deployed as a very lightweight plugin into the runtime entity implementing a participant
2. Introducing a database into the participant means that participant becomes a heavyweight plugin with cross participant and participant/acm-r configuration required
3. There are multiple sources of truth for participant and element data, the ACM-R DB and the this new participant DB have duplicated data
4. Participant configuration is no longer transparent to the participant implementing component, having a DB for each participant forces orthogonal configuration across participant implementing components
5. Configuration of components implementing participants becomes much more complex, the components must manage the participant replication
6. The replication mechanism enforces the use of an additional 3pp, the database

Solution 4: Distributed Cache

Issues:

- Not persistent - if the application that handles cache server restarts - data is lost.
- Approval issues - with Redis, Etcd, Search Engine.

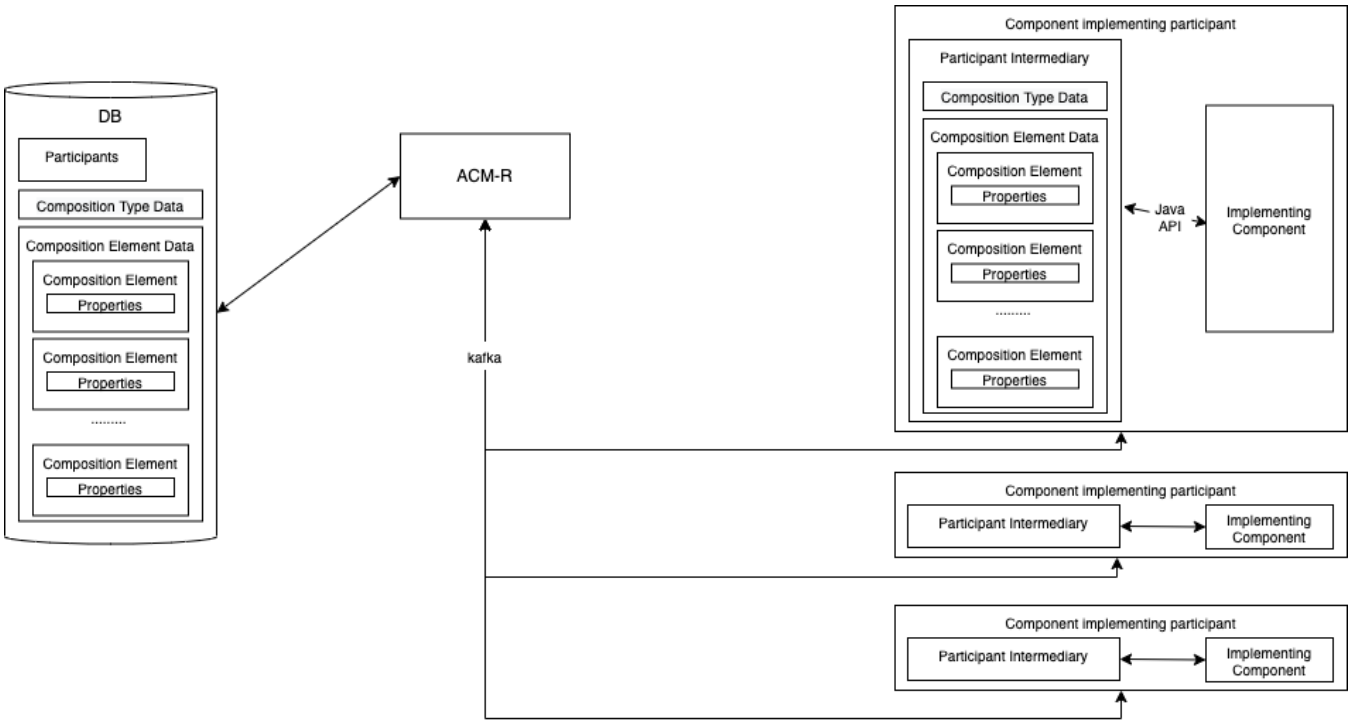
Solution 5: True Participant Replicas

The requirements are:

1. Participants can be replicated, each participant can have an arbitrary number of replicas
2. Composition definitions, instances, element instances and all their data including properties is identical in all participant replicas
3. When anything is changed in one replica, the change is propagated to all the replicas of a participant
4. An operation on a composition element can be sent to any replica of a participant, which means that for a given element, the deploy could be on replica 1, the update could be on replica 2 and the delete could be on replica 3, as one would expect in any HA solution
5. A single REST operation called on ACM-R will select a participant replica (probably using round robin initially but we could support other algorithms in the future), and use that replica for that operation.
6. The ACM runtime will be made HA (more than one replica of ACM-R will be supported), it will run on a HA postgres.
7. The replication mechanism used between ACM-R and participants is transparent to participant API users
8. Replicas are "eventually consistent", with consistency typically occurring in 100s of milliseconds

This solution uses a similar approach to that used by Kubernetes when using etcd to implement CRDs and CRs. It implements replication in participants by introducing

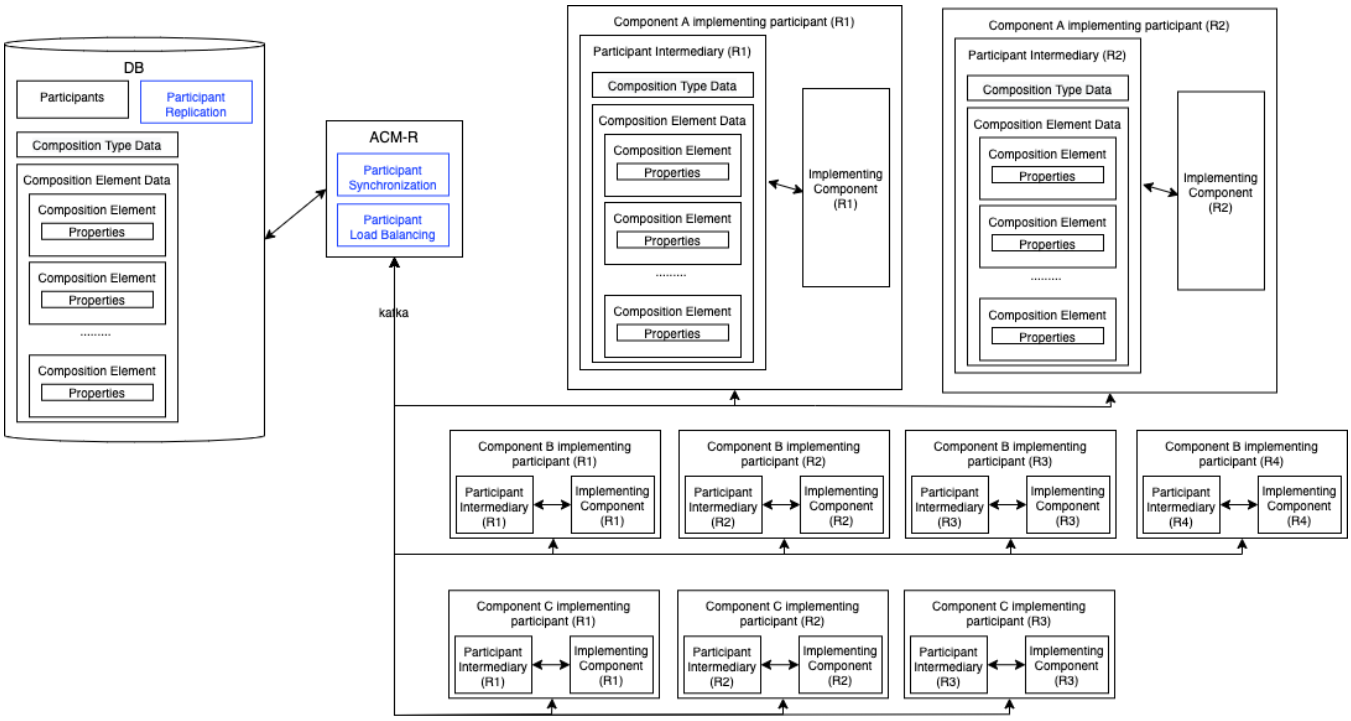
1. The concept of participant replicas into ACM-R and the Participant Intermediary
2. A lightweight mechanism for replica update between replicas and ACM-R. Every time a replica changes its data, the change is reported to ACM-R and ACM-R updates all other replicas for that participant



The diagram above depicts data management as implemented today in ACM-R. ACM-R maintains the "source of truth" for all compositions and their elements in the ACM-R database.

1. Composition type data is pushed from ACM-R to the participants and is read-only in the participants
2. Composition element data (state information mainly) is built up by interactions between ACM-R and participants for the various ACM operations (Deploy/Update/Undeploy etc) and ACM-R always maintains the current composition element data in the ACM-R database
3. Composition element properties are pushed by ACM-R to the participant and can be updated in the participant by the participant implementation. When the properties are updated on the participant side, those changes are propagated to the ACM-R database

Therefore today, for the three types of data above, the ACM-R database has the state of the participant. This means that creation of participant replicas is rather trivial. We leave the current data handling mechanism in place, introduce the concept of participant replicas in ACM-R, and introduce data synchronization across participant replicas.



We introduce a participant replication table in the ACM-R database, which is an index used to record the replicas that exist of each participant. When the replica of a component implementing a participant is created (by Kubernetes or otherwise), the participant intermediary in the component registers with ACM-R as usual and as it does today. The only difference is that the participant intermediary will send the participant ID and the replica number. ACM-R is updated to accept registrations from participants with the same Participant ID and different replica numbers. When the first replica for a certain participant registers, ACM-R will handle this registration exactly as it does today and will add the replica as the single replica that exists for this participant. When the next replica registers, ACM-R will recongnise that this is a second replica for a participant that already exists and will record this as a replica. Rather than priming this replica, ACM-R will copy all the data from the first replica to this replica. The registration of further replicas will continue to follow this pattern.

During normal operation where ACM-R receives and executes requests towards participants, ACM-R will use Participant Load Balancing to select a replica using a round-robin algorithm and execute the operation on that replica. When the operation finishes, ACM-R will synchronize the data from the replica that executed the operation to all the other replicas using Participant Synchronization.

If ACM-R is informed by a replica that an Implementing Component changed composition element properties, Participant Synchronization synchronizes these changes to all other Participant Intermediary replicas.

In this solution:

1. We will preserve Participant Design backward compatibility, there is no change to the participant intermediary interface for participant implementations
2. Participant version backward compatibility will not be preserved because we need to pass replica information in the registration and operational messages, all participants will have to be upgraded to the new version.
3. The REST API that returns participant information will be updated to include replica information
4. ACM-R is made HA so that it itself can scale
5. We can use Kafka load balancing on the participants and get the load balancing functionality for nothing
6. A new Kafka topic is used for synchronization

Optimal Solution:

After analysis, it is clear that the best solution to use is number 5.