```
package io.scalecube.cluster;

import io.scalecube.cluster.membership.MembershipEvent;
import io.scalecube.transport.Address;
import io.scalecube.transport.Message;
import io.scalecube.transport.NetworkEmulator;
import java.util.Arrays;
import java.util.Collection;
import java.util.Map;
import java.util.Optional;
import reactor.core.Exceptions;
import reactor.core.publisher.Flux;
import reactor.core.publisher.Mono;

/** Facade cluster interface which provides API to interact with cluster members.
*/
public interface Cluster {

  /** Init cluster instance and join cluster synchronously. */
  static Cluster joinAwait() {
    try {
      return join().block();
    } catch (Exception e) {
      throw Exceptions.propagate(e.getCause() != null ? e.getCause() : e);
    }
  }

  /** Init cluster instance with the given seed members and join cluster
synchronously. */
  static Cluster joinAwait(Address... seedMembers) {
    try {
      return join(seedMembers).block();
    } catch (Exception e) {
      throw Exceptions.propagate(e.getCause() != null ? e.getCause() : e);
    }
  }

  /**
   * Init cluster instance with the given metadata and seed members and join
cluster synchronously.
   */
  static Cluster joinAwait(Map<String, String> metadata, Address... seedMembers) {
    try {
      return join(metadata, seedMembers).block();
    } catch (Exception e) {
      throw Exceptions.propagate(e.getCause() != null ? e.getCause() : e);
    }
  }

  /** Init cluster instance with the given configuration and join cluster
synchronously. */
  static Cluster joinAwait(ClusterConfig config) {
    try {
      return join(config).block();
    } catch (Exception e) {
      throw Exceptions.propagate(e.getCause() != null ? e.getCause() : e);
    }
  }
```

```
  /** Init cluster instance and join cluster asynchronously. */
  static Mono<Cluster> join() {
    return join(ClusterConfig.defaultConfig());
  }

  /** Init cluster instance with the given seed members and join cluster
asynchronously. */
  static Mono<Cluster> join(Address... seedMembers) {
    ClusterConfig config =
ClusterConfig.builder().seedMembers(seedMembers).build();
    return join(config);
  }

  /**
   * Init cluster instance with the given metadata and seed members and join
cluster synchronously.
   *
   * @param metadata metadata
   * @param seedMembers seed members
   */
  static Mono<Cluster> join(Map<String, String> metadata, Address... seedMembers) {
    ClusterConfig config =

ClusterConfig.builder().seedMembers(Arrays.asList(seedMembers)).metadata(metadata).
build();
    return join(config);
  }

  /**
   * Init cluster instance with the given configuration and join cluster
synchronously.
   *
   * @param config cluster config
   * @return result future
   */
  static Mono<Cluster> join(final ClusterConfig config) {
    return new ClusterImpl(config).join0();
  }

  /**
   * Returns {@link Address} of this cluster instance.
   *
   * @return cluster address
   */
  Address address();

  /**
   * Send a msg from this member (src) to target member (specified in parameters).
   *
   * @param member target member
   * @param message msg
   * @return promise telling success or failure
   */
  Mono<Void> send(Member member, Message message);

  /**
   * Send a msg from this member (src) to target member (specified in parameters).
   *
   * @param address target address
```

```
   * @param message msg
   * @return promise telling success or failure
   */
  Mono<Void> send(Address address, Message message);

  /**
   * Sends message to the given address. It will issue connect in case if no
transport channel by
   * given transport {@code address} exists already. Send is an async operation and
expecting a
   * response by a provided correlationId and sender address of the caller.
   *
   * @param address address where message will be sent
   * @param request to send message must contain correlctionId and sender to handle
reply.
   * @return promise which will be completed with result of sending (message or
exception)
   * @throws IllegalArgumentException if {@code message} or {@code address} is null
   */
  Mono<Message> requestResponse(Address address, Message request);

  /**
   * Sends message to the given address. It will issue connect in case if no
transport channel by
   * given transport {@code address} exists already. Send is an async operation and
expecting a
   * response by a provided correlationId and sender address of the caller.
   *
   * @param member where message will be sent
   * @param request to send message must contain correlctionId and sender to handle
reply.
   * @return promise which will be completed with result of sending (message or
exception)
   * @throws IllegalArgumentException if {@code message} or {@code address} is null
   */
  Mono<Message> requestResponse(Member member, Message request);

  /**
   * Subscription point for listening incoming messages.
   *
   * @return stream of incoming messages
   */
  Flux<Message> listen();

  /**
   * Spreads given message between cluster members using gossiping protocol.
   *
   * @param message message to disseminate.
   * @return result future
   */
  Mono<String> spreadGossip(Message message);

  /**
   * Listens for gossips from other cluster members.
   *
   * @return gossip publisher
   */
  Flux<Message> listenGossips();
```

```
  /**
   * Returns local cluster member metadata.
   *
   * @return local member metadata
   */
  Map<String, String> metadata();

  /**
   * Returns cluster member metadata by given member reference.
   *
   * @param member cluster member
   * @return cluster member metadata
   */
  Map<String, String> metadata(Member member);

  /**
   * Returns local cluster member which corresponds to this cluster instance.
   *
   * @return local member
   */
  Member member();

  /**
   * Returns cluster member with given id or null if no member with such id exists
at joined
   * cluster.
   *
   * @return member by id
   */
  Optional<Member> member(String id);

  /**
   * Returns cluster member by given address or null if no member with such address
exists at joined
   * cluster.
   *
   * @return member by address
   */
  Optional<Member> member(Address address);

  /**
   * Returns list of all members of the joined cluster. This will include all
cluster members
   * including local member.
   *
   * @return all members in the cluster (including local one)
   */
  Collection<Member> members();

  /**
   * Returns list of all cluster members of the joined cluster excluding local
member.
   *
   * @return all members in the cluster (excluding local one)
   */
  Collection<Member> otherMembers();

  /**
   * Updates local member metadata with the given metadata map. Metadata is updated
```

```
asynchronously
   * and results in a membership update event for local member once it is updated
locally.
   * Information about new metadata is disseminated to other nodes of the cluster
with a
   * weekly-consistent guarantees.
   *
   * @param metadata new metadata
   */
  Mono<Void> updateMetadata(Map<String, String> metadata);

  /**
   * Updates single key-value pair of local member's metadata. This is a shortcut
method and anyway
   * update will result in a full metadata update. In case if you need to update
several metadata
   * property together it is recommended to use {@link #updateMetadata(Map)}.
   *
   * @param key metadata key to update
   * @param value metadata value to update
   */
  Mono<Void> updateMetadataProperty(String key, String value);

  /**
   * Remove single key-value pair of local member's metadata. This is a shortcut
method and anyway
   * update will result in a full metadata update. In case if you need to update
several metadata
   * property together it is recommended to use {@link #updateMetadata(Map)}.
   *
   * @param key metadata key to remove.
   */
  Mono<Void> removeMetadataProperty(String key);

  /**
   * Listen changes in cluster membership.
   *
   * @return membershiop publisher
   */
  Flux<MembershipEvent> listenMembership();

  /**
   * Member notifies other members of the cluster about leaving and gracefully
shutdown and free
   * occupied resources.
   *
   * @return Listenable future which is completed once graceful shutdown is
finished.
   */
  Mono<Void> shutdown();

  /**
   * Check if cluster instance has been shut down.
   *
   * @return Returns true if cluster instance has been shut down; false otherwise.
   */
  boolean isShutdown();

  /**
```

```
   * Returns network emulator associated with this instance of cluster. It always
returns non null
   * instance even if network emulator is disabled by transport config. In case
when network
   * emulator is disable all calls to network emulator instance will result in no
operation.
   *
   * @return network emulator object
   */
  NetworkEmulator networkEmulator();
}
```