```
package io.scalecube.cluster;

import io.scalecube.cluster.fdetector.FailureDetectorImpl;
import io.scalecube.cluster.gossip.GossipProtocolImpl;
import io.scalecube.cluster.membership.IdGenerator;
import io.scalecube.cluster.membership.MembershipEvent;
import io.scalecube.cluster.membership.MembershipProtocolImpl;
import io.scalecube.cluster.metadata.MetadataStoreImpl;
import io.scalecube.transport.Address;
import io.scalecube.transport.Message;
import io.scalecube.transport.NetworkEmulator;
import io.scalecube.transport.Transport;
import java.util.Collection;
import java.util.Collections;
import java.util.HashMap;
import java.util.Map;
import java.util.Objects;
import java.util.Optional;
import java.util.Set;
import java.util.stream.Collectors;
import java.util.stream.Stream;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import reactor.core.Disposable;
import reactor.core.Disposables;
import reactor.core.publisher.DirectProcessor;
import reactor.core.publisher.Flux;
import reactor.core.publisher.FluxSink;
import reactor.core.publisher.Mono;
import reactor.core.publisher.MonoProcessor;
import reactor.core.scheduler.Scheduler;
import reactor.core.scheduler.Schedulers;

/** Cluster implementation. */
final class ClusterImpl implements Cluster {

  private static final Logger LOGGER = LoggerFactory.getLogger(ClusterImpl.class);

  private static final Set<String> SYSTEM_MESSAGES =
      Collections.unmodifiableSet(
          Stream.of(
                  FailureDetectorImpl.PING,
                  FailureDetectorImpl.PING_REQ,
                  FailureDetectorImpl.PING_ACK,
                  MembershipProtocolImpl.SYNC,
                  MembershipProtocolImpl.SYNC_ACK,
                  GossipProtocolImpl.GOSSIP_REQ,
                  MetadataStoreImpl.GET_METADATA_REQ,
                  MetadataStoreImpl.GET_METADATA_RESP)
              .collect(Collectors.toSet()));

  private static final Set<String> SYSTEM_GOSSIPS =
      Collections.singleton(MembershipProtocolImpl.MEMBERSHIP_GOSSIP);

  private final ClusterConfig config;

  // Subject
  private final DirectProcessor<MembershipEvent> membershipEvents =
DirectProcessor.create();
```

```
  private final FluxSink<MembershipEvent> membershipSink = membershipEvents.sink();

  // Disposables
  private final Disposable.Composite actionsDisposables = Disposables.composite();
  private final MonoProcessor<Void> shutdown = MonoProcessor.create();
  private final MonoProcessor<Void> onShutdown = MonoProcessor.create();

  // Cluster components
  private Transport transport;
  private Member localMember;
  private FailureDetectorImpl failureDetector;
  private GossipProtocolImpl gossip;
  private MembershipProtocolImpl membership;
  private MetadataStoreImpl metadataStore;
  private Scheduler scheduler;

  public ClusterImpl(ClusterConfig config) {
    this.config = Objects.requireNonNull(config);
  }

  public Mono<Cluster> join0() {
    return Transport.bind(config.getTransportConfig())
        .flatMap(
            boundTransport -> {
              transport = boundTransport;
              localMember = createLocalMember(boundTransport.address().port());

              scheduler = Schedulers.newSingle("sc-cluster-" +
localMember.address().port(), true);

              // Setup shutdown
              shutdown
                  .then(doShutdown())
                  .doFinally(s -> onShutdown.onComplete())
                  .subscribeOn(scheduler)
                  .subscribe(
                      null, ex -> LOGGER.error("Exception occurred on cluster
shutdown: " + ex));

              failureDetector =
                  new FailureDetectorImpl(
                      localMember,
                      transport,
                      membershipEvents.onBackpressureBuffer(),
                      config,
                      scheduler);

              gossip =
                  new GossipProtocolImpl(
                      localMember,
                      transport,
                      membershipEvents.onBackpressureBuffer(),
                      config,
                      scheduler);

              metadataStore =
                  new MetadataStoreImpl(
                      localMember, transport, config.getMetadata(), config,
scheduler);
```

```
              membership =
                  new MembershipProtocolImpl(
                      localMember,
                      transport,
                      failureDetector,
                      gossip,
                      metadataStore,
                      config,
                      scheduler);

              actionsDisposables.add(
                  membership
                      .listen()
                      /*.publishOn(scheduler)*/
                      // TODO [AV] : make otherMembers work
                      .subscribe(
                          membershipSink::next,
                          th -> LOGGER.error("Received unexpected error: ", th)));

              failureDetector.start();
              gossip.start();
              metadataStore.start();

              return membership.start();
            })
        .thenReturn(this);
  }

  /**
   * Creates and prepares local cluster member. An address of member that's being
constructed may be
   * overriden from config variables. See {@link
io.scalecube.cluster.ClusterConfig#memberHost},
   * {@link ClusterConfig#memberPort}.
   *
   * @param listenPort transport listen port
   * @return local cluster member with cluster address and cluster member id
   */
  private Member createLocalMember(int listenPort) {
    String localAddress = Address.getLocalIpAddress().getHostAddress();
    Integer port = Optional.ofNullable(config.getMemberPort()).orElse(listenPort);

    // calculate local member cluster address
    Address memberAddress =
        Optional.ofNullable(config.getMemberHost())
            .map(memberHost -> Address.create(memberHost, port))
            .orElseGet(() -> Address.create(localAddress, listenPort));
    return new Member(IdGenerator.generateId(), memberAddress);
  }

  @Override
  public Address address() {
    return member().address();
  }

  @Override
  public Mono<Void> send(Member member, Message message) {
    return send(member.address(), message);
  }

  @Override
  public Mono<Void> send(Address address, Message message) {
    return transport.send(address, message);
  }

  @Override
  public Mono<Message> requestResponse(Address address, Message request) {
    return transport.requestResponse(request, address);
  }

  @Override
  public Mono<Message> requestResponse(Member member, Message request) {
    return transport.requestResponse(request, member.address());
  }

  @Override
  public Flux<Message> listen() {
    // filter out system messages
    return transport.listen().filter(msg -> !
SYSTEM_MESSAGES.contains(msg.qualifier()));
  }

  @Override
  public Mono<String> spreadGossip(Message message) {
    return gossip.spread(message);
  }

  @Override
  public Flux<Message> listenGossips() {
    // filter out system gossips
    return gossip.listen().filter(msg -> !
SYSTEM_GOSSIPS.contains(msg.qualifier()));
  }

  @Override
  public Collection<Member> members() {
    return membership.members();
  }

  @Override
  public Collection<Member> otherMembers() {
    return membership.otherMembers();
  }

  @Override
  public Map<String, String> metadata() {
    return metadataStore.metadata();
  }

  @Override
  public Map<String, String> metadata(Member member) {
    return metadataStore.metadata(member);
  }

  @Override
  public Member member() {
    return localMember;
  }

  @Override
  public Optional<Member> member(String id) {
    return membership.member(id);
  }

  @Override
  public Optional<Member> member(Address address) {
    return membership.member(address);
  }

  @Override
  public Mono<Void> updateMetadata(Map<String, String> metadata) {
    return Mono.fromRunnable(() -> metadataStore.updateMetadata(metadata))
        .then(membership.updateIncarnation())
        .subscribeOn(scheduler);
  }

  @Override
  public Mono<Void> updateMetadataProperty(String key, String value) {
    return Mono.fromCallable(() -> updateMetadataProperty0(key, value))
        .flatMap(this::updateMetadata)
        .subscribeOn(scheduler);
  }

  private Map<String, String> updateMetadataProperty0(String key, String value) {
    Map<String, String> metadata = new HashMap<>(metadataStore.metadata());
    metadata.put(key, value);
    return metadata;
  }

  public Mono<Void> removeMetadataProperty(String key) {
    return Mono.fromCallable(() -> removeMetadataProperty0(key))
        .flatMap(this::updateMetadata)
        .subscribeOn(scheduler)
        .then();
  }

  private Map<String, String> removeMetadataProperty0(String key) {
    Map<String, String> metadata = new HashMap<>(metadataStore.metadata());
    metadata.remove(key);
    return metadata;
  }

  @Override
  public Flux<MembershipEvent> listenMembership() {
    return Flux.defer(
        () ->
            Flux.fromIterable(otherMembers())
                .map(member -> MembershipEvent.createAdded(member,
metadata(member)))
                .concatWith(membershipEvents)
                .onBackpressureBuffer());
  }

  @Override
  public Mono<Void> shutdown() {
    return Mono.defer(
        () -> {
          shutdown.onComplete();
          return onShutdown;
        });
  }

  private Mono<Void> doShutdown() {
    return Mono.defer(
        () -> {
          LOGGER.info("Cluster member {} is shutting down", localMember);
          return Flux.concatDelayError(leaveCluster(localMember), dispose(),
transport.stop())
              .then()
              .doOnSuccess(avoid -> LOGGER.info("Cluster member {} has shut down",
localMember));
        });
  }

  private Mono<Void> leaveCluster(Member member) {
    return membership
        .leaveCluster()
        .doOnSuccess(
            s ->
                LOGGER.info(
                    "Cluster member {} notified about his leaving and shutting
down", member))
        .doOnError(
            e ->
                LOGGER.warn(
                    "Cluster member {} failed to spread leave notification "
                        + "to other cluster members: {}",
                    member,
                    e))
        .then();
  }

  private Mono<Void> dispose() {
    return Mono.fromRunnable(
        () -> {
          // Stop accepting requests
          actionsDisposables.dispose();

          // stop algorithms
          metadataStore.stop();
          membership.stop();
          gossip.stop();
          failureDetector.stop();

          // stop scheduler
          scheduler.dispose();
        });
  }

  @Override
  public NetworkEmulator networkEmulator() {
    return transport.networkEmulator();
  }

  @Override
  public boolean isShutdown() {
    return onShutdown.isDisposed();
  }
}
```