ContextChainImpl.java

```
@Override
public void instantiateServiceInstance() {

        try {
            contexts.forEach(this::initializeContextService);
            LOG.info("Started clustering services for node {}", deviceInfo);
        } catch (final Exception ex) {
            LOG.warn("Not able to start clustering services for node {}", deviceInfo);
            executorService.submit(() -> mastershipChangeListener
                    .onNotAbleToStartMastershipMandatory(deviceInfo, ex.getMessage()));
        }
    }
```

ContextChainHolderImpl.java

```
@Override
public void onNotAbleToStartMastership(final DeviceInfo deviceInfo, @Nonnull final String
reason, final boolean mandatory) {
        LOG.warn("Not able to set MASTER role on device {}, reason: {}", deviceInfo, reason);

        if (!mandatory) {
            return;
        }

        Optional.ofNullable(contextChainMap.get(deviceInfo)).ifPresent(contextChain -> {
            LOG.warn("This mastering is mandatory, destroying context chain and closing
connection for device {}.", deviceInfo);
            destroyContextChain(deviceInfo);
        });
    }
```

```
private synchronized void destroyContextChain(final DeviceInfo deviceInfo) {
        Optional.ofNullable(contextChainMap.get(deviceInfo)).ifPresent(contextChain -> {

deviceManager.sendNodeRemovedNotification(deviceInfo.getNodeInstanceIdentifier());
            contextChain.close();
        });
    }
```

ContextChainImpl.java

```
    @Override
    public void close() {
```

```
        if (CONTEXT_STATE.TERMINATION.equals(state)) {
            LOG.debug("ContextChain for node {} is already in TERMINATION state.",
deviceInfo);
            return;
        }

        state = CONTEXT_STATE.TERMINATION;
        unMasterMe();

        // Close all connections to devices
        auxiliaryConnections.forEach(connectionContext ->
connectionContext.closeConnection(false));
        auxiliaryConnections.clear();

        // If we are still registered and we are not already closing, then close the registration
        if (Objects.nonNull(registration)) {
            try {
                registration.close();
                registration = null;
                LOG.info("Closed clustering services registration for node {}", deviceInfo);
            } catch (final Exception e) {
                LOG.warn("Failed to close clustering services registration for node {} with
exception: ",
                        deviceInfo, e);
            }
        }

        LOG.info("Going to close all contexts for {}", deviceInfo.getNodeId());
        // Close all contexts (device, statistics, rpc)
        contexts.forEach(OFPContext::close);
        contexts.clear();
        LOG.info("Closed all contexts for {}", deviceInfo.getNodeId());
        // We are closing, so cleanup all managers now
        deviceRemovedHandlers.forEach(h -> h.onDeviceRemoved(deviceInfo));
        deviceRemovedHandlers.clear();
        LOG.info("Closed all managers for {}", deviceInfo.getNodeId());
        primaryConnection.closeConnection(false);

    }
```

DeviceContextImpl.java

```
    @Override
    public void close() {
        LOG.info("Closing DeviceContextImpl");
        if (CONTEXT_STATE.TERMINATION.equals(state.get())) {
            if (LOG.isDebugEnabled()) {
```

```
            LOG.debug("DeviceContext for node {} is already in TERMINATION state.",
getDeviceInfo());
        }

        return;
    }

    state.set(CONTEXT_STATE.TERMINATION);
    slaveTask.cancel();
    changeLastRoleFuture(null);

    // Close all datastore registries and transactions
    if (initialized.get()) {
        initialized.set(false);
        deviceGroupRegistry.close();
        deviceFlowRegistry.close();
        deviceMeterRegistry.close();

        final ListenableFuture<Void> txChainShuttingDown =
transactionChainManager.shuttingDown();

        try {
            txChainShuttingDown.get(TX_CHAIN_CLOSE_TIMEOUT,
TimeUnit.MILLISECONDS);
        } catch (InterruptedException | ExecutionException | TimeoutException e) {
            txChainShuttingDown.cancel(true);
            LOG.warn("Failed to shut down transaction chain for device {}: {}", deviceInfo, e);
        }

        transactionChainManager.close();
        transactionChainManager = null;
    }

    requestContexts.forEach(requestContext -> RequestContextUtil
            .closeRequestContextWithRpcError(requestContext, "Connection closed."));
    requestContexts.clear();
    LOG.info("Finished Closing DeviceContextImpl");
  }
```

RequestContextUtil.java

```
  public static <T> ListenableFuture<RpcResult<T>>
closeRequestContextWithRpcError(final RequestContext<T> requestContext, final String
errorMessage) {
    RpcResultBuilder<T> rpcResultBuilder =
RpcResultBuilder.<T>failed().withRpcError(RpcResultBuilder.newError(RpcError.ErrorType.
APPLICATION, "", errorMessage));
```

```
        requestContext.setResult(rpcResultBuilder.build());
        closeRequestContext(requestContext);
        return requestContext.getFuture();
    }
```

```
    public static void closeRequestContext(final RequestContext<?> requestContext) {
        try {
            requestContext.close();
        } catch (Exception e) {
            LOG.error("Request context failed to close", e);
        }
    }
```

DeviceContextImpl.java

```
    @Nullable
    @Override
    public <T> RequestContext<T> createRequestContext() {
        final Long xid = deviceInfo.reserveXidForDeviceMessage();
        LOG.info("Create RequestContext");
        final AbstractRequestContext<T> abstractRequestContext = new
AbstractRequestContext<T>(xid) {
            @Override
            public void close() {
                LOG.info("Remove RequestContext");
                requestContexts.remove(this); <-- ConcurrentModificationException
            }
        };

        requestContexts.add(abstractRequestContext);
        return abstractRequestContext;
    }
```