FFI-RAPPORT

16/01189

Web Live, Virtual and Constructive (WebLVC)

–

Tuva Kristine Thoresen

Web Live, Virtual and Constructive (WebLVC)

Tuva Kristine Thoresen

Forsvarets forskningsinstitutt (FFI) 18. juli 2016

Emneord

HLA
Distribuert simulering
Modellering og simulering
Tjenesteorientert arkitektur

FFI-rapport:
FFI-RAPPORT 16/01189

Prosjektnummer
1380

ISBN:
P: 978-82-464-2778-2
E: 978-82-464-2779-9

Godkjent av
Karsten Bråthen, *forskningsleder*
Anders Eggen, a*vdelingssjef*

Sammendrag

Det er bestemt at Forsvarets systemporteføljе skal følge tjenesteorienterte prinsipper. For å oppnå kostnadseffektiv trening og for å kunne utnytte simulering til støtte i operasjoner er det viktig at simuleringssystemer også kan inngå som en del av Forsvarets Informasjonsinfrastruktur (INI). Et viktig prinsipp vil være at brukeren skal ha enkel tilgang til simuleringsapplikasjoner, og at de er tilgjengelig der brukeren befinner seg. Denne rapporten beskriver en standard som gjør at nettleserapplikasjoner kan inngå i en distribuert simulering.

Web Live, Virtual and Constructive (WebLVC) er en standard som er under utvikling i Simulation Interoperability Standards Organization (SISO), og som kan brukes til å koble nettleserapplikasjoner til et distribuert simuleringssystem som benytter enten Distributed Interactive Simulation (DIS) eller High Level Architecture (HLA). Den fremtidige standarden skal ha høy ytelse og være enkel å bruke for nettleserapplikasjoner. Standarden består av en meldingsprotokoll og en WebLVC-tjener som kobler nettleserapplikasjonene og simuleringssystemet sammen.

Det er utviklet en demonstrator som implementerer de viktigste delene av standarden, og som viser at det er mulig å inkludere en nettleserapplikasjon i en HLA-føderasjon ved bruk av WebLVC-protokollen. WebLVC er enkel å bruke for nettleserapplikasjoner, men virker mer kompleks enn nødvendig å implementere på tjener-siden. Standarden er fortsatt under utvikling, og det er i rapporten identifisert flere forenklinger som bør vurderes i det videre standardiseringsarbeidet.

Summary

The systems portfolio of the Norwegian Armed Forces should follow service-oriented principles. To achieve cost effective training and be able to use simulations for support in operations, it is important that simulation systems can be a part of the Norwegian Armed Forces' information infrastructure (INI). An important principle is to give the user easy access to simulation applications, and that applications are available at the location of the user. This report describes a standard that makes it possible for web applications to participate in a distributed simulation.

Web Live, Virtual and Constructive (WebLVC) is a new standard under development by the Simulation Interoperability Standards Organization (SISO), which can be used to link web applications with a distributed simulation system that uses either Distributed Interactive Simulation (DIS) or High Level Architecture (HLA). The future standard should be high-performance and be natural to use for web applications. The standard consists of a messaging protocol and a WebLVC-server that links web applications with the simulation system.

A demonstrator for the most important parts of the standard has been developed, which shows that is it possible for a web application to participate in a HLA federation using the WebLVC protocol. WebLVC is easy to use for web applications, but seems more complex than necessary to implement on the server side. The standard is still under development, and this report identifies several simplifications that should be considered in the future.

Innhold

1 Innledning

Ny teknologi gjør det mulig for nettleserapplikasjoner, applikasjoner som kjører i en nettleser, å håndtere tyngre grafikk og interaktiv kommunikasjon. Det blir dermed mulig for nettleserapplikasjoner å delta i interaktive distribuerte simuleringer. Nettleserbaserte simuleringer gir økt tilgjengelighet og gjenbruk av funksjonalitet og er et viktig skritt på veien til tjenesteorienterte simuleringssystemer. De siste årene har det vært mye forskning på mellomvareteknologi for nettbaserte distribuerte simuleringer [1]. Tidligere forsøk har vært basert på Web services og "request/response"-paradigmer. Dette gir ikke den ytelsen som kreves for å utveksle store mengder simuleringsdata på kort tid. Dataformatet som sendes mellom simuleringene har også vært vanskelig å bruke i nettleserapplikasjoner. Web Live, Virtual and Constructive (WebLVC) er en ny standard under utvikling som skal gjøre det enklere for nettleserapplikasjoner å delta i distribuerte simuleringer. WebLVC skal ha høy ytelse, ha et nettleservennlig format og skal kunne håndtere flere ulike mellomvarer for distribuert simulering, som Distributed Interactive Simulation (DIS) [2] og High Level Architecture (HLA) [3-6].

Denne rapporten gir en introduksjon til WebLVC og beskriver erfaringene med bruk av den fremtidige standarden. Det er implementert en WebLVC-tjener som kan delta i en HLA-føderasjon og en tilhørende testføderasjon. Formålet var å undersøke hvor godt WebLVC fungerer målt opp mot kravene det er satt for standarden. Hvor enkel er standarden å bruke på klientsiden? Er ytelsen god nok til å bruke i en simuleringsføderasjon hvor mye data skal utveksles på kort tid? Det er også sett på hvor enkel standarden er å implementere på tjener-siden og noen forbedringer/forenklinger som kunne vært gjort er vurdert.

Denne rapporten er tiltenkt alle som har interesse for WebLVC, både beslutningstakere som lurer på om WebLVC er noe for dem, og utviklere som skal bruke den fremtidige standarden. De ulike kapitlene har derfor noe ulikt teknisk nivå. Kapittel 2 gir en kort bakgrunn for standarden og beskriver eksisterende teknologi. Kapittel 3 tar for seg detaljene i standarden, både arkitekturen og meldingsformatet. Kapittel 4 beskriver implementasjonen av tjeneren, og er skrevet med tanke på utviklere som skal videreutvikle eller utvikle egne WebLVC-tjenere. Kapittel 5 tar for seg en testføderasjon som ble utviklet for å teste standarden og beskriver implementasjonen av denne. Kapittel 6 beskriver testingen som ble utført og resultatene, mens kapittel 7 gir noen anbefalinger for videre utvikling av standarden. Til slutt gir kapittel 8 en konklusjon.

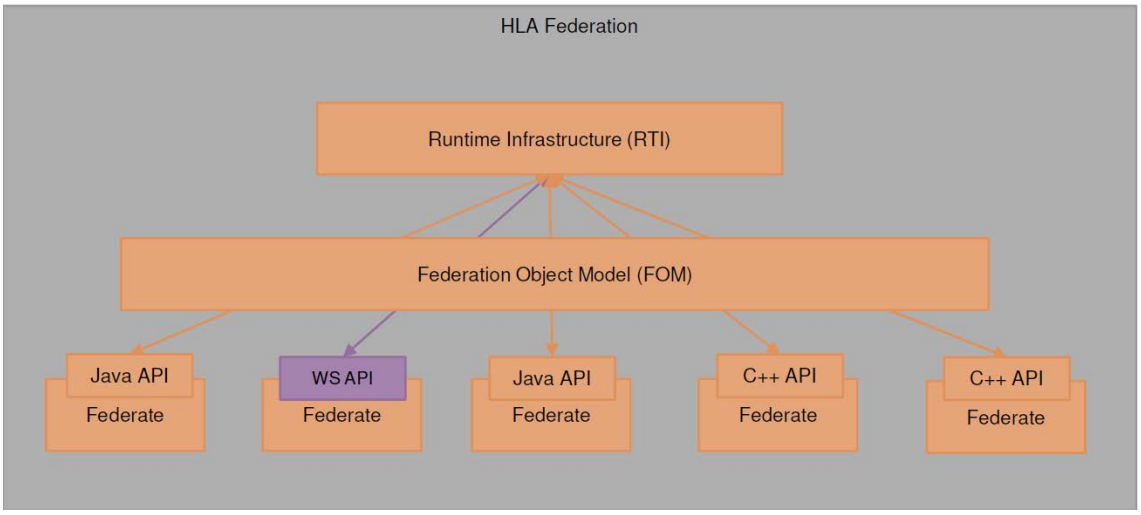
2 Bakgrunn

Dette kapittelet gir en kort introduksjon til eksisterende mellomvareteknologi for distribuerte simuleringer, tjenesteorientering og implementasjon av tjenesteorienterte systemer. Eksisterende teknologi for nettbaserte simuleringer blir beskrevet, før WebLVC introduseres.

2.1 High Level Architecture

High Level Architecture (HLA) [3-6] er en arkitektur for distribuert simulering, der målet er interoperabilitet og løs kobling mellom simuleringskomponentene. Arkitekturen er tatt i bruk i et stort antall miljøer, og NATO STANAG 4603 [7], ratifisert av Norge, sier at HLA skal brukes i Nato-sammenheng. En distribuert simulering basert på HLA kalles en føderasjon og simuleringskomponentene som deltar kalles føderater. Figur 2.1 viser en illustrasjon av en HLA-føderasjon. HLA-standarden består av tre deler; et sett med regler som beskriver de underliggende designprinsippene, en mal for objektmodellen for simuleringsdataene som utveksles mellom føderatene, og grensesnittet til en kjøretidskomponent, the Run-Time Infrastructure (RTI), som styrer interaksjonen mellom føderatene [8].

HLA-føderatene utveksler to ulike typer data; objekter og interaksjoner. Objekter er persistente entiteter som er synlig for flere føderater, og består av attributter som utveksles mellom føderatene. Interaksjoner modellerer hendelser som ikke er direkte assosiert med en objekt-attributt, for eksempel et skudd med et våpen. Alle objekter og interaksjoner som skal deles mellom føderatene, føderasjonens delte tilstand, må spesifiseres i en Federation Object Model (FOM).



Figur 2.1 Illustrasjon av en High Level Architecture (HLA)-føderasjon. Figuren er tatt fra [9].

All kommunikasjon mellom HLA-føderatene skjer gjennom kjøretidskomponenten, en RTI. HLA er basert på et "publish/subscribe"-paradigme, der føderater publiserer data til RTI-en og abonnerer på data. Hvert føderat må spesifisere hva slags data det kan sende og motta i en Simulation Object Model (SOM). RTI-en tilrettelegger for at føderatene kan være autonome og løst koblet, men til gjengjeld er de svært avhengige av RTI-en [9]. RTI-en administrerer føderasjonens delte tilstand og tilbyr flere tjenester for føderasjonen, blant annet datadistribusjon, tidsstyring og synkronisering av føderasjonen. RTI-en lagrer ikke tilstandsvariabler for føderasjonen og er uavhengig av føderasjonens objektmodell [8]. Dermed kan samme RTI brukes i flere ulike føderasjoner. Grensesnittet mellom føderatene og RTI-en er definert i en grensesnittspesifikasjon, og det finnes API-er for flere programmeringsspråk, blant annet Java og C++. Den nyeste versjonen av HLA, HLA Evolved, tilbyr også et Web service (WS)-grensesnitt.

Det største hinderet for interoperabilitet med HLA-føderasjoner er kravet om en delt FOM [9]. Det finnes flere referanse-FOM-er og noen av dem er mye brukt, som for eksempel Real-time Platform Reference Federation Object Model (RPR FOM) [10], som ble utviklet basert på Distributed Interactive Simulation (DIS) [2] Protocol Data Units (PDUs). DIS er en protokoll for distribuert simulering som ikke har mekanismer for datadistribusjon eller tidsstyring.

2.1.1 Dødregning

For å begrense mengden data som blir sendt ut på simuleringsinfrastrukturen, krever DIS og RPR FOM at alle føderater skal dødregne posisjonen til entiteter de er interessert i [8]. Dødregning innebærer at hvert føderat estimerer tilstanden til eksterne entiteter fra siste rapporterte posisjon og hastighet. For å sørge for at oppdateringer sendes ut når det er behov for det, har hvert føderat et ansvar for å simulere to modeller av entitetene sine, en intern modell og en dødregningsmodell. Hvis entitetens dødregnede tilstand skiller seg for mye fra entitetens sanne tilstand, sendes det ut en tilstandsoppdatering. RPR FOM definerer flere dødregningsmodeller som tilsvarer dødregningsmodellene definert i DIS. Disse kan sees i Tabell 2.1.

2.2 Tjenesteorienterte simuleringssystemer

Tjenesteorientering er et paradigme som har fokus på løs kobling, sen binding, gjenbruk av komponenter og å skille grensesnittet fra implementasjonen [11]. I en tjenesteorientert arkitektur (Service Oriented Architecture, SOA) deles ulike kapabiliteter inn i separate tjenester som skal kunne brukes av flere applikasjoner. Det er bestemt at Natos og Forsvarets systemporteføljе skal følge tjenesteorienterte prinsipper [9, 12]. For å oppnå kostnadseffektiv trening og for å kunne utnytte simulering til støtte i operasjoner er det viktig at simuleringssystemer også kan inngå som en del av Forsvarets Informasjonsinfrastruktur (INI). Det er derfor ønskelig at også modellerings- og simuleringssystemer blir tjenesteorienterte. Dette vil gi mulighet for løst koblede simuleringskomponenter, som enkelt og raskt kan settes sammen og byttes ut etter behov. Et mål er å kunne tilby felles simuleringsfunksjonalitet som tilgjengelige tjenester. Dette vil kunne tilrettelegge for økt tilgjengelighet og gjenbruk av simuleringskomponenter. Et viktig

Field	Model	Formula	Examples
1	STATIC	N/A	Static entities
2	DRM (FPW)	$P = P_0 + V_0\Delta t$	Constant velocity (or low acceleration) linear motion
3	DRM (RPW)	1) $P = P_0 + V_0\Delta t$ 2) $[R]_{w\to b} = [DR][R_0]_{w\to b}$	Similar to DRM 2 but where orientation is required (e.g., visual simulation)
4	DRM (RVW)	1) $P = P_0 + V_0\Delta t + \frac{1}{2}A_0\Delta t^2$ 2) $[R]_{w\to b} = [DR][R_0]_{w\to b}$	Similar to DRM 5 but where orientation is required (e.g., visual simulation)
5	DRM (FVW)	1) $P = P_0 + V_0\Delta t + \frac{1}{2}A_0\Delta t^2$	High speed (e.g., missile) or maneuvering at any speed
6	DRM (FPB)	1) $P = P_0 + [R_0]^{-1}_{w\to b}([R1]V_b)$	Similar to DRM 2 but when body-centered calculation is preferred
7	DRM (RPB)	1) $P = P_0 + [R_0]^{-1}_{w\to b}([R1]V_b)$ 2) $[R]_{w\to b} = [DR][R_0]_{w\to b}$	Similar to DRM 3 but when body-centered calculation is preferred
8	DRM (RVB)	1) $P = P_0 + [R_0]^{-1}_{w\to b}([R1]V_b + [R2]A_b)$ 2) $[R]_{w\to b} = [DR][R_0]_{w\to b}$	Similar to DRM 4 but when body-centered calculation is preferred
9	DRM (FVB)	1) $P = P_0 + [R_0]^{-1}_{w\to b}([R1]V_b + [R2]A_b)$	Similar to DRM 5 but when body-centered calculation is preferred

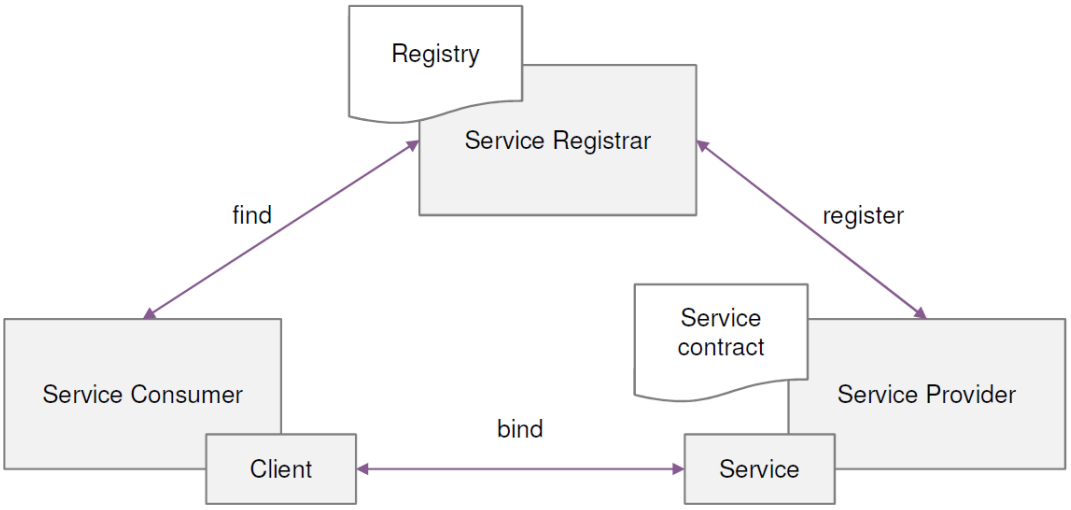
Tabell 2.1 Dødregningsmodellene i Distributed Interactive Simulation (DIS). Tabellen er tatt fra [2].

steg på veien er å kunne inkludere nettleserapplikasjoner. Hvis alt som behøves for å få tilgang på eller delta i en distribuert simulering er en enhet med en nettleser og nettilgang, vil man kunne sørge for at simuleringer er tilgjengelig der brukeren er.

En tjenesteorientert arkitektur består av tre aktører; en tjenestetilbyder, en tjeneste-konsument og et tjenesteregister. Tjenestetilbyderen tilbyr en tjeneste, som den kan registrere i et tjenesteregister. Konsumenten kan finne tjenester i registeret, som den så kan koble seg opp til. Figur 2.2 viser SOA-trekanten.

2.2.1 Implementasjon av tjenesteorienterte systemer

I dette underkapittelet presenteres to ulike måter å implementere tjenesteorienterte systemer på. Det er mange måter å betegne de ulike teknologiene på, og ulike kilder bruker ulike betegnelser.



Figur 2.2 Aktørene i et tjenesteorientert system. Figuren er tatt fra [9].

Her brukes følgende terminologi: Med "Web services" menes SOAP-baserte [13] Web services [14, 15], ofte kalt "Big Web services" eller WS-* [16], og med "RESTful Web services" menes tjenesteorienterte systemer implementert etter prinsippene i REpresentational State Transfer (REST) [17].

Web services er et rammeverk for interoperabel applikasjon-til-applikasjon-interaksjon over et nettverk [18], og er en vanlig måte å implementere tjenesteorienterte systemer på. Dette er tilnærmingen valgt i Natos kjernetjenester [12]. Web services er tungt basert på standarder, og hver standard bygger på eksisterende, velprøvde teknologier og skal være uavhengig av en bestemt plattform eller programmeringsspråk. Web service-rammeverket er modulært, og består av flere ulike standarder for blant annet tjenestekvalitet, sikkerhet, "publish/subscribe" og dynamisk oppdagelse av og tilkobling til tjenester. Brukeren har derfor tilgang på mye avansert funksjonalitet, og kan enkelt hente ut de delene hun har bruk for [18].

Kommunikasjon mellom systemene skjer gjennom utveksling av SOAP-meldinger. SOAP er en protokoll for informasjonsutveksling basert på Extensible Markup Language (XML) [19]. SOAP definerer et meldingsformat og ulike meldingsutvekslingsmønstre, og spesifikasjonen inneholder en prosesseringsmodell som beskriver hvordan en SOAP-melding skal prosesseres. En SOAP-melding består av et meldingshode (en header) og en kropp som kan inneholde vilkårlige XML-elementer. Innholdet i meldingene er bestemt av applikasjonen. SOAP er transportuavhengig og kan sendes over flere transportlagsprotokoller[1] [18], som HTTP [20].

Et viktig prinsipp i tjenesteorientering er å sørge for interoperabilitet ved hjelp av eksplisitt definerte grensesnitt, og her er tjenestekontrakter viktig. En Web service beskrives med Web

[1] SOAP kan også sendes over andre applikasjonslagsprotokoller, se "4. Using Various Protocol Bindings" i [13].

Services Description Language (WSDL) [21]. WSDL er et XML-basert maskinlesbart format som består av to deler; en abstrakt del som beskriver tjenestens grensesnitt, med operasjoner og dataformat, og en konkret del, som har adresserings- og bindingsinformasjon. Den abstrakte delen kan implementeres av flere ulike tjenester, og en tjeneste kan implementere mange ulike abstrakte grensesnitt. Det finnes verktøy for å generere kode fra en WSDL (eller motsatt) for flere ulike programmeringsspråk [18]. En klientapplikasjon kan dermed få generert et kodeskjelett med nødvendige metoder for å kommunisere med en tjeneste. Dette forenkler jobben med både tilkobling og kommunikasjon med tjenesten, siden SOAP-meldinger automatisk genereres fra kode-kallene.

Et alternativ til Web service-teknologien er tjenester basert på REpresentational State Transfer (REST), såkalte RESTful Web services. REST er et paradigme som er tett knyttet til HTTP, og ble i utgangspunktet brukt for å beskrive de grunnleggende konseptene bak weben [22]. REST ser på weben som et nett av hyperlinkede ressurser. En nettleserapplikasjon er en tilstandsmaskin, som ved å følge linker mellom ressurser går fra en tilstand til en annen. Nøkkelabstraksjonen i REST er en ressurs, og hver ressurs har minst et unikt navn, en Uniform Resource Identifier (URI), som brukes som en unik identifikator for ressursen. En ressurs kan ha flere ulike representasjoner i ulike dataformater, og tilgang til ressursene skjer gjennom et uniformt grensesnitt (HTTP-metodene GET, PUT, POST, DELETE). Hver tjeneste tilbyr det samme uniforme grensesnittet til klienter, noe som gir mer skalerbare systemer [22]. Et annet viktig prinsipp er tilstandsløshet; tjenestene holder ikke tilstand for klientene, så all informasjon som trengs for å utføre en operasjon må være inkludert i kallet.

REST og Web services har ulike styrker og svakheter. REST oppfattes ofte som enklere enn Web services (derav navnet “Big Web services”) og kan fungere bedre for enkle, ad-hoc-koblinger mellom systemer [16]. Siden Web services tilbyr flere standardiserte funksjonaliteter kan denne teknologien fungere bedre i store virksomhetssystemer. En svakhet i mange Web service-systemer er at grensesnittet ofte er for tett knyttet opp mot implementasjonen av systemet. Dette kan føre til interoperabilitetsproblemer. Web services har også en tett kobling mellom dataformatet og grensesnittet (begge er definert i tjenestekontrakten, en WSDL), mens REST skiller dataformatet (representasjonen) fra grensesnittet ved å tilby et uniformt grensesnitt. Dette øker skalerbarheten i systemene. Ved å frikoble dataformatet fra grensesnittet har REST-baserte tjenester også større fleksibilitet i representasjonen av ressurser; de kan tilby flere ulike representasjoner av hver ressurs og det er enklere å videreutvikle dataformatet [22].

2.3 Eksisterende teknologi

Det finnes flere måter å inkludere nettapplikasjoner i distribuerte simuleringer. To eksisterende teknologier, basert på henholdsvis Web services og REST, presenteres her.

2.3.1 HLA Web services API

HLA Evolved [3-5] tilbyr et Web service API for HLA-føderater. Ved bruk av dette kan nettbaserte føderater delta i en simuleringsføderasjon, som de kobler seg til via en Web Service

Provider RTI Component (WSPRC). Web service-føderatene er fullverdige medlemmer i føderasjonen og har tilgang til standard HLA-tjenester som tidsstyring, synkronisering og delt tilstand. Web service API-et støtter konseptet om simuleringer som tilgjengelige tjenester og er et skritt på veien mot tjenesteorienterte simuleringssystemer [23].

HLA Web service API-et er beskrevet i WSDL. Dette API-et skiller seg fra HLA API-ene for Java/C++ på flere måter. Blant annet finnes det kun mekanismer for enveis-kall, og dermed må Web service-føderatene selv hente responsene ut av RTI-en, hvilket gir ytelsesproblemer. I tillegg kan ikke eksisterende lokale RTI-komponenter brukes direkte for å koble Web service-føderater til RTI-en, men utbredelsen av verktøy som automatisk genererer klientkode fra en WSDL (se kapittel 2.2.1), burde gjøre det siste til et noe mindre problem [24].

Det er flere problemer med HLA Web service API-et, både skalerbarheten til Web services og ytelsen. Kommunikasjonen skjer ved "request/response" og dataene er formattert med SOAP/XML. Dette gir lavere ytelse enn mer kompakte data sendt over sockets eller med multicast[2]. Lavere ytelse påvirker hele føderasjonen; Web service-føderatet må kanskje operere med lavere tidsoppløsning og det kan komme til å bli en flaskehals for resten av føderasjonen [23]. Operasjonene på lavt nivå som tilbys for å kommunisere med RTI-en kan være for kompliserte å bruke for nettleserapplikasjoner. Disse krever at nettleserapplikasjonene koder og dekoder binære attributt- og parameterverdier, noe som bryter med prinsippene ved tynn-klient design, der tanken er at tjeneren skal gjøre så mye som mulig, og klienten skal motta et format som er enkelt og effektivt å bruke.

2.3.2 RESTful Interoperability Simulation Environment (RISE)

Discrete Event System Specification (DEVS) [25] er en modelleringsformalisme som brukes for diskrete hendelsessystemer. DEVS er en matematisk modelleringsteknikk som lar brukeren spesifisere hierarkiske og modulære modeller, som kan gjenbrukes og kombineres på ulike måter. Det finnes flere rammeverk/verktøy for å implementere og simulere modellene, og det er utviklet komplette verktøysett for modellering og simulering basert på DEVS [26].

RESTful Interoperability Simulation Environment (RISE) [1] er en mellomvare basert på REST-prinsippene, som prøver å løse noen av problemene som gjør at distribuert simulering ikke har fått så stor utbredelse [27]. Man trenger en simuleringsmellomvare som tilrettelegger for løst koblede simuleringskomponenter, som enkelt og raskt kan settes sammen og byttes ut etter behov, også underveis i kjøringen. Mellomvaren må være basert på nett-standarder, for å kunne inkludere nettleserapplikasjoner og andre komponenter/enheter som har tilgang til et nettverk.

RISE ble i utgangspunktet utviklet for å øke interoperabiliteten mellom DEVS-baserte simuleringer, og RISE er brukt i flere eksperimenter der DEVS-modeller ble simulert i nettskyen [28-31]. RISE er en generell mellomvare og pluggbar beholder som kan eksponere ulike simuleringskomponenter som tjenester [32]. RISE tilbyr et uniformt grensesnitt som lar

[2] Multicast er en-til-mange eller mange-til-mange kommunikasjon over et nettverk.

brukeren sette opp en simulering (for eksempel ved å laste opp en DEVS-modell og nødvendig inputdata), starte kjøringen og hente opp resultatene når simuleringen er ferdig. Simuleringskomponentene kan distribueres over flere datamaskiner [32], og simuleringen kan styres av en håndholdt enhet, som en mobiltelefon, eller integreres med andre verktøy. Et problem med denne tilnærmingen er at RISE ikke tilbyr et grensesnitt for interaksjon med simuleringen [33]. RISE kan dermed ikke per i dag brukes til interaktive sanntidssimuleringer, men dette er et tema som det blir forsket på [33].

Et annet mulig problem er den tette koblingen til DEVS. Selv om [1], [31] og [32] beskriver RISE som en generell mellomvare for distribuerte simuleringer som skal kunne brukes for vilkårlige simuleringskomponenter finnes det kun dokumentasjon på implementasjon og testing for DEVS. Det finnes imidlertid mange varianter og utvidelser av DEVS, blant annet for interaktive sanntidssimuleringer: DEVS er brukt sammen med HLA for simulering av militære operasjoner [26].

2.4 Web Live, Virtual and Constructive (WebLVC)

Løsningene som eksisterer i dag basert på Web services og RISE har ikke det som kreves for å delta i interaktive sanntidssimuleringer. RISE har ikke per i dag støtte for interaktive sanntidssimuleringer og har tilsynelatende tett kobling til DEVS-formalismen. HLA Web services API-et har problemer med ytelsen. For hyppig utveksling av store mengder simuleringsdata trengs det en protokoll som har mulighet for interaktive, toveis kommunikasjonskanaler. “Request/response”-kommunikasjonen som brukes av HLA Web services API-et gir ikke denne muligheten, og operasjonene på lavt nivå som tilbys er ikke egnet for nettleserapplikasjoner. I tillegg er SOAP- og XML-meldingene som utveksles mellom simuleringskomponentene tungvinne å bruke for JavaScript-applikasjoner.

Web Live, Virtual and Constructive (WebLVC) er en standard under utvikling i Simulation Interoperability Standards Organization (SISO) Product Development Group (PDG) for WebLVC [34, 35]. Dette arbeidet er ledet av VT MÄK, som har utviklet en WebLVC-tjener det er mulig å kjøpe [36]. WebLVC er designet med tre krav [37]: Den skal ha høy nok ytelse til å kunne brukes av interaktive sanntidssimuleringer, den skal være enkel å bruke for JavaScript-klienter og den skal være fleksibel nok til å støtte ulike mellomvareteknologier for distribuert simulering, som HLA og DIS. Fokuset ligger på nettleserapplikasjoner skrevet i JavaScript. Dette er fordi JavaScript er “språket til weben” [38] og er støttet av alle moderne nettlesere uten bruk av programvareutvidelser.

Den fremtidige standarden består av to deler; en meldingsprotokoll utviklet med tanke på nettleserapplikasjoner, og en arkitektur bestående av en WebLVC-tjener som deltar i distribuerte simuleringer på vegne av klientene sine. Klientene er ikke fullverdige medlemmer av den distribuerte simuleringen, noe som kan være en mulig svakhet ved WebLVC. Tjeneren kan ha flere klienter tilkoblet samtidig, og det er mulig med et distribuert simuleringssystem bestående av bare nettleserapplikasjoner. Tjeneren skal kunne håndtere flere

mellomvareteknologier og oversetter mellom den nettleservennlige meldingsprotokollen brukt av klientene og protokollen brukt i simuleringssystemet.

WebLVC er basert på konseptet om objektmodeller, inspirert av HLA [37]. For å gi enkel tilgang til vanlige simuleringsprotokoller og -arkitekturer har WebLVC en standardobjektmodell som er utviklet for DIS/RPR FOM-føderasjoner. Nettleserapplikasjonene kan abonnere på oppdateringer av interaksjoner og objekter, akkurat som i en vanlig HLA-føderasjon, og kan selv sende oppdateringer til tjeneren. Siden enkelhet og fleksibilitet er viktig, mangler standardobjektmodellen noen viktige HLA-tjenester, som tidsstyring og synkronisering. Dette er en svakhet sammenlignet med Web service HLA API-et. Samtidig er WebLVC spesialisert for tynn-klient JavaScript-applikasjoner, og som diskutert i kapittel 2.3.1 kan det hende det ikke er gjennomførbart for nettleserklienter å bruke lav-nivå HLA-funksjoner. Protokollen er utvidbar, så det er mulig å definere meldinger for tidsstyring og synkronisering i fremtiden.

2.4.1 Eksempel på tidligere bruk av WebLVC

Det finnes en implementasjon av WebLVC, en Commercial off-the-shelf (COTS) WebLVC-tjener utviklet av VT MÄK [36]. Tjeneren består av to komponenter: en WebLVC-tjener som kan brukes i en HLA/DIS-distribuert simulering og et JavaScript-bibliotek for WebLVC.

I mange av dagens systemer for datagenererte styrker (Computer Generated Forces, CGF) er brukergrensesnittet tett koblet til simuleringsmotoren. Mye av funksjonaliteten som tilbys av CGF-systemene er felles, og det er ønskelig å utvikle et felles brukergrensesnitt som kan styre flere ulike CGF-systemer. Det var målet i Improved Control and Visualisation of Computer Generated Forces (ICoViCS) [39], et forskningsprosjekt som ønsket å utvikle et felles brukergrensesnitt for ulike CGF-systemer. Konseptet ble testet gjennom et nettleserbasert brukergrensesnitt som kommuniserte med CGF-systemene over HLA. VT MÄKs WebLVC-tjener ble brukt for å koble nettleserapplikasjonen på HLA, med gode erfaringer. En mer detaljert beskrivelse er gitt i kapittel 6.4.

3 WebLVC

Beskrivelsen i dette kapittelet, i tillegg til implementasjonen av tjeneren beskrevet i kapittel 4, er basert på *WebLVC Draft Protocol Specification Version 0.4* [40]. Dette utkastet består av fire deler:

- En generell introduksjon til WebLVC, der også arkitekturen beskrives.
- En objektmodell-uavhengig del som definerer de grunnleggende meldingstypene.
- En standardobjektmodell basert på DIS og HLA RPR FOM.

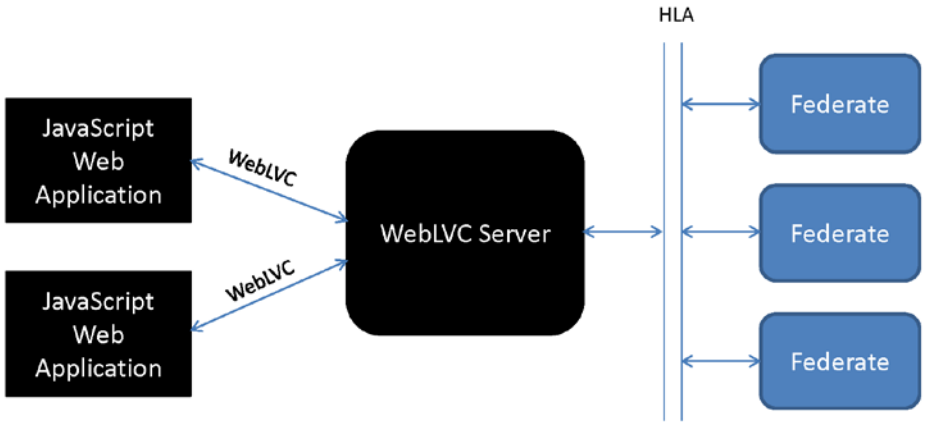
- En del som beskriver hvordan nye WebLVC-meldinger kan genereres basert på DIS eller en HLA FOM.

Alle disse delene er beskrevet i dette kapittelet.

3.1 Arkitekturen

Hovedkomponenten i standarden er en WebLVC-tjener. Siden målet er en standard som er enkel å bruke for nettleserklienter og som kan bruke vilkårlige simuleringsprotokoller og -arkitekturer, ble en gateway-løsning valgt. I følge [41] er en gateway en komponent som oversetter mellom to simuleringssystemer som bruker ulike infrastrukturløsninger. Tjeneren oversetter mellom WebLVC-protokollen brukt av klientene og protokollen/arkitekturen brukt i simuleringssystemet, for eksempel HLA. En illustrasjon av arkitekturen kan sees i Figur 3.1. Protokollen skal være enkel å bruke for nettleserapplikasjoner og mye av kompleksiteten er derfor lagt til tjeneren.

For å oppnå høy ytelse, anbefaler WebLVC at kommunikasjonen mellom klient og tjener går over WebSockets [42]. WebLVC spesifiserer ikke hvilke alternative protokoller som kunne vært brukt. WebSocket er en protokoll basert på Transmission Control Protocol (TCP) [43] som støttes direkte av alle moderne nettlesere. WebSocket åpner en persistent, toveis TCP-tilkobling mellom klient og tjener, noe som gir mulighet for asynkron, interaktiv kommunikasjon med lav tidsforsinkelse/overhead. Dette bidrar til effektiv utveksling av simuleringsdata og muliggjør dermed interaktive sanntidssimuleringer.



Figur 3.1 WebLVC-arkitekturen for en HLA-føderasjon.

3.2 Meldingsformatet

Meldingsprotokollen er utviklet med tanke på at den skal ha høy ytelse og være enkel å bruke for JavaScript-klienter. Derfor er JavaScript Object Notation (JSON) [44] valgt som meldingsformat. JSON er et lettvekts dataformat som er lesbart av både maskiner og mennesker. JSON er naturlig å bruke i JavaScript, siden JSON er streng-representasjonen av et JavaScript-objekt. JavaScript har innebygde funksjoner for å generere og lese JSON, noe som gjør formatet lett håndterlig. Samtidig er JSON helt uavhengig av plattform og programmeringsspråk, noe som gjør det til et ideelt datautvekslingsformat [44]. Dermed kan WebLVC også brukes av andre typer applikasjoner.

3.3 Meldingssemantikken

Protokollen er basert på konseptet om objektmodeller, inspirert av HLA [37]. Ved å bruke ulike objektmodeller kan protokollen støtte ulike datautvekslingsmodeller, og for HLA, ulike FOM-er. Protokollen har en grunnleggende, objektmodell-uavhengig del som beskriver de ulike meldingstypene i protokollen. I den grunnleggende delen beskrives administrative meldinger, feilhåndtering og felles egenskaper som er uavhengig av objektmodellen. Meldingstypene kan utvides med spesifikke objektmodeller, se kapittel 3.4-3.5.

Enhver WebLVC-melding består av et meldingshode (en header) som definerer meldingstypen, en `MessageKind`. Det finnes 13 forskjellige meldingstyper, gitt i Tabell 3.1. Hver meldingstype har flere generelle egenskaper. Noen av meldingstypene skal brukes i et "request/response"-mønster, der klienten ber om informasjon eller en tjeneste og tjeneren returnerer et svar. Dette gjelder meldingstypene for tilkobling til tjeneren, konfigurering, abonnementshåndtering og statuslogging. Meldingstypene for objekter og interaksjoner brukes til å utveksle simuleringsdata og kan sendes av både klienten og tjeneren. De fire øverste meldingstypene er administrative meldinger, som definerer hvordan klienten kan koble seg til og konfigurere tjeneren.

3.3.1 Objekter og interaksjoner

Objekt- og interaksjonsmeldingstypene brukes for å utveksle simuleringsdata. Det finnes tre ulike meldingstyper; `AttributeUpdate`, `ObjectDeleted` og `Interaction`, og disse definerer de ulike typene simuleringsdata som kan utveksles. `AttributeUpdate` brukes for å sende data om et objekt (objektets attributter), og den grunnleggende meldingsspesifikasjonen definerer to viktige egenskaper[3], objektets navn og objektets type. Objektet navns er påkrevd, og alle objekter er unikt identifisert ved navnet sitt. Objekttypen er påkrevd første gang det sendes ut informasjon om et objekt. Et eksempel på den grunnleggende spesifikasjonen av en `AttributeUpdate`-melding kan sees i Figur 3.2.

[3] Med ordet "egenskap" menes her egenskaper i en WebLVC-melding (oversatt fra engelsk "property" [40]). I HLA har objekter attributter, mens interaksjoner har parametere.

Meldingstype	Beskrivelse
`Connect`	Koble til tjeneren
`ConnectResponse`	Tilkoblingsrespons fra tjeneren
`Configure`	Konfigurering av tjeneren
`ConfigureResponse`	Konfigureringsrespons fra tjeneren
`AttributeUpdate`	Utveksle objekt-attributter
`ObjectDeleted`	Slette et objekt fra simuleringen
`Interaction`	Utveksle interaksjoner
`SubscribeObject`	Abonnere på en objekttype
`UnsubscribeObject`	Avslutte abonnementet på en objekttype
`SubscribeInteraction`	Abonnere på en interaksjonstype
`UnsubscribeInteraction`	Avslutte abonnementet på en interaksjonstype
`StatusLogRequest`	Be om å få se statusloggen
`StatusLogResponse`	Respons fra tjeneren på statusloggen

Tabell 3.1 De grunnleggende meldingstypene i WebLVC-protokollen.

```
{
      MessageKind: “AttributeUpdate”,
      ObjectName: “MyObject”,
      ObjectType: “MyObjectType”
}
```

Figur 3.2 Et eksempel på en `AttributeUpdate`-melding.

`ObjectDeleted` brukes når et objekt skal slettes fra simuleringen og her kreves kun objektets navn. `Interaction` brukes for å sende interaksjonsdata, og den grunnleggende spesifikasjonen inneholder en interaksjonstype. Objekt- og interaksjonsmeldingene kan i tillegg inneholde et tidsstempel.

3.3.2 Abonnementshåndtering

WebLVC tilbyr klientene et “publish/subscribe”-paradigme, og det finnes fire meldingstyper for å håndtere abonnementer. Disse gjør klientene i stand til å beskrive hvilke objekter og interaksjoner de er interessert i å få informasjon om. Protokollen legger opp til en relativ avansert mekanisme for abonnementer, der klientene kan bruke filtre for å beskrive hva slags objekt-attributter og interaksjoner de er interessert i. Foreløpig er spesifikasjonen av disse

mekanismene i startgropa og fortsatt under diskusjon i PDG-en [40, 45]. Hvis en klient ikke har definert noen abonnementer, sendes alle objekter og interaksjoner til klienten.

3.3.3 Feilhåndtering

Hver klient har sin egen statuslogg hos tjeneren der det logges implementasjonsspesifikk informasjon. Feil håndteres ved at tjeneren logger feilmeldinger til klientens statuslogg. Dette kan være feil som oppstår i prosesseringen av meldinger, varsler eller annet type informasjon. Klienten kan se sin egen statuslogg ved å sende en `StatusLogRequest`-melding. Alle feil som forekommer i tjeneren etter mottak av meldinger fra klienten logges her. Unntaket er hvis klienten har oppgitt ikke-kompatible verdier i forespørselsmeldinger til tjeneren, for eksempel feil dataformat. Dette vil føre til en negativ respons fra tjeneren. Feilhåndteringsmetoden er en mulig svakhet i protokollen, ved at klienten selv må hente opp informasjon om feil fra tjeneren. For fremtidige versjoner av protokollen burde man vurdere om tjeneren selv skal sende (alvorlige) feilmeldinger til klienten, i det feilen oppstår.

3.3.4 Tilkobling til tjeneren

En klient kobler seg til tjeneren ved å sende en `Connect`-melding. `Connect`-meldingen skal inneholde klientens navn, som fungerer som en unik identifikator for klienten. Meldingen kan også inneholde flere andre meldinger, som konfigurerings- eller abonnementsmeldinger. Disse kan brukes til å konfigurere tjeneren før sesjonen har startet. Ved en tilkoblingsforespørsel ser tjeneren om den kan oppfylle klientens krav og sender en `ConnectResponse`-melding som svar. Tjeneren godkjenner kun tilkoblingen hvis den kan oppfylle alle klientens krav. Når tjeneren har godkjent tilkoblingen, sender den umiddelbart ut nåværende tilstand til alle objekter i simuleringen som klienten er interessert i. Etter at disse objektene er sendt kan den vanlige trafikken begynne. Det er viktig at klienten ikke sender flere meldinger til tjeneren før den har fått svar på tilkoblingsforespørselen. En illustrasjon av denne prosessen er gitt i [40].

Konfigureringsmeldinger brukes til å sette ulike egenskaper hos tjeneren. Tjeneren kan ha flere klienter på en gang, hver med sin egen konfigurering av tjeneren. Den grunnleggende spesifikasjonen gir kun et konfigurasjonsvalg: formatet på tidsstempelet klienten kan sende med objekt- og interaksjonsmeldinger.

3.4 Objektmodeller

Objekt- og interaksjonsmeldingene kan utvides med ulike objektmodeller, basert på datautvekslingsmodellen som brukes i simuleringssystemet. For HLA gjenspeiler objektmodellen føderasjonens FOM, og brukeren kan selv utvikle egne objektmodeller [40]. For å definere en ny objektmodell må brukeren definere nye objekt- og interaksjonstyper og utvide `AttributeUpdate` og `Interaction`-meldingstypene med nye egenskaper. Egenskapene er knyttet til en bestemt objekt- eller interaksjonstype. Brukeren kan også utvide de andre meldingstypene med nye egenskaper.

3.4.1 Standardobjektmodellen i WebLVC

For å gjøre jobben enklere for brukere av tjeneren er det utviklet en standardobjektmodell basert på DIS/RPR FOM. Denne inneholder representasjoner av vanlige RPR FOM-objekter og interaksjoner. Objektmodellen er håndlaget (ikke automatisk generert) og skiller seg derfor fra DIS/RPR FOM på noen viktige områder. Dette er for å følge JSON-konvensjoner og sørge for at datastrukturen som sendes er så enkel som mulig å bruke for nettleserapplikasjoner [40]. Noen av disse valgene er fornuftige, for eksempel ved å bruke arrayer istedenfor strukturer (structs) for entitetstype og posisjonsinformasjon. Ulempen er at oversettelsen fra en DIS PDU/HLA-datatype ikke blir triviell.

Det er definert flere objekt- og interaksjonstyper, gitt i Tabell 3.2. Hvordan `AttributeUpdate` og `Interaction`-meldingstypene utvides med nye egenskaper basert på objekt- og interaksjonstype er detaljert spesifisert i protokollen. Et eksempel på en `AttributeUpdate`-melding er gitt i Figur 3.3. Der kan vi se et `WebLVC:PhysicalEntity`-objekt.

Standardobjektmodellen definerer også flere utvidelser av konfigurasjonsmeldingstypen. Brukeren kan blant annet definere hvilket koordinatsystem som skal brukes, og om tjeneren skal utføre dødregning for klienten. Det er ikke spesifisert i versjon 0.4 av protokollen [40] om dette gjelder dødregning av klientens egne objekter, eller de objektene klienten får tilsendt fra tjeneren. Siden tanken med protokollen er effektivitet, ønsker vi å begrense mengden data som sendes ut på nettverket. Det antas derfor at dette kun gjelder de objektene klienten er interessert i, og at klienten selv er ansvarlig for dødregning av egne objekter.

3.5 Utvidelser av protokollen

Som beskrevet i kapittel 3.4 kan protokollen utvides med nye objektmodeller. Brukeren kan også utvide standardobjektmodellen med flere nye egenskaper. Protokollens siste del inneholder informasjon om hvordan man automatisk kan utvide protokollen basert på DIS eller en HLA

Objekttyper	Interaksjonstyper
`WebLVC:PhysicalEntity`	`WebLVC:WeaponFire`
`WebLVC:AggregateEntity`	`WebLVC:MunitionDetonation`
`WebLVC:EnvironmentalEntity`	`WebLVC:StartResume`
`WebLVC:RadioTransmitter`	`WebLVC:StopFreeze`
	`WebLVC:RadioSignal`
	`WebLVC:TransferOwnership`[4]

Tabell 3.2 Objekt- og interaksjonstyper i standardobjektmodellen.

[4] Denne interaksjonstypen er listet opp i versjon 0.4 av protokollen [40], men er ikke spesifisert med noen egenskaper.

FOM. For HLA er det beskrevet flere standard oversettingsregler fra HLA- til JSON-datatype.

```
{
      MessageKind: “AttributeUpdate”,
      ObjectName: “AMunitionObject”,
      ObjectType: “WebLVC:PhysicalEntity”,
      EntityType: [2,9,225,2,14,2,1],
      Coordinates: {
            DeadReckoningAlgorithm: 5,
            WorldLocation: [22986.70, 639168.56, 5900727.25],
            VelocityVector: [134.68, 881.42, 897.57],
            AccelerationVector: [6.66, -79.21, -92.16],
            Orientation: [-2.60, -0.78, 1.72]
      },
      Marking: “BAL”,
      DamageState: 1,
      EngineSmokeOn: true
}
```

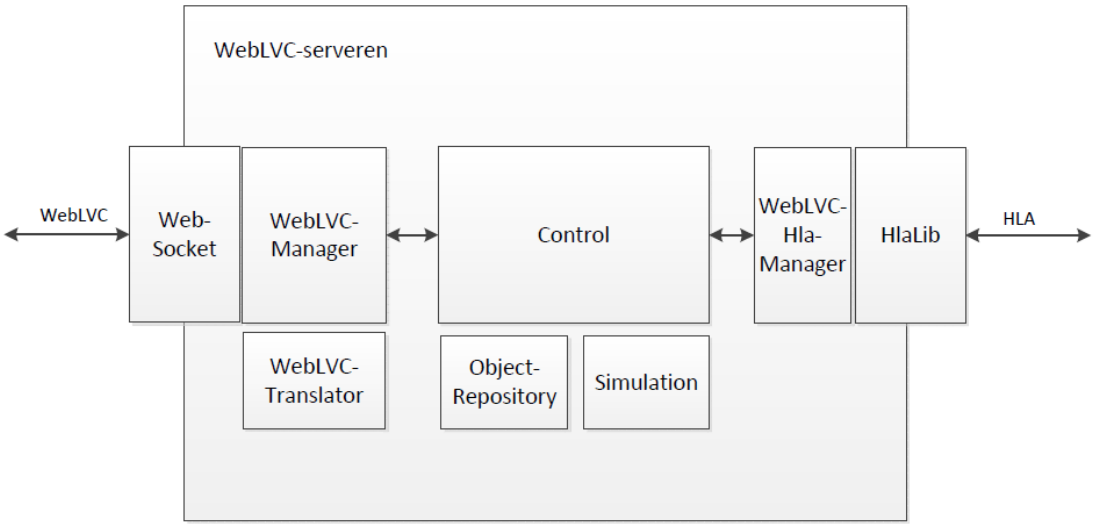
Figur 3.3 En `AttributeUpdate`-melding for et ammunisjonsobjekt.

4 Implementasjon av WebLVC

I dette kapittelet beskrives implementasjon av WebLVC-tjeneren. Dette kapittelet er ment for utviklere som skal utvikle egne WebLVC-tjenere og kan brukes som en dokumentasjon av tjeneren. Formålet med oppgaven var å utvikle en demonstrator som koblet nettleserapplikasjoner til en HLA-føderasjon over protokollen WebLVC. Det har derfor blitt fokusert på standardobjektmodellen og HLA.

4.1 Implementasjon av WebLVC-tjeneren

Tjeneren er implementert i Java (versjon 1.8) og bruker HlaLib [46], et bibliotek utviklet ved Forsvarets forskningsinstitutt (FFI) som forenkler implementasjonen av HLA-føderater. Transportmekanismen brukt fra klient til tjener er WebSocket og dette er implementert med Jetty WebSocket [47]. Tjeneren har to hovedfunksjoner: Den skal ta i mot WebLVC-meldinger fra klienter, konvertere disse, og sende objekt-attributter og interaksjoner ut på HLA-nettet. Den skal også ta i mot objekt-attributter og interaksjoner fra HLA-nettet, konvertere dem til

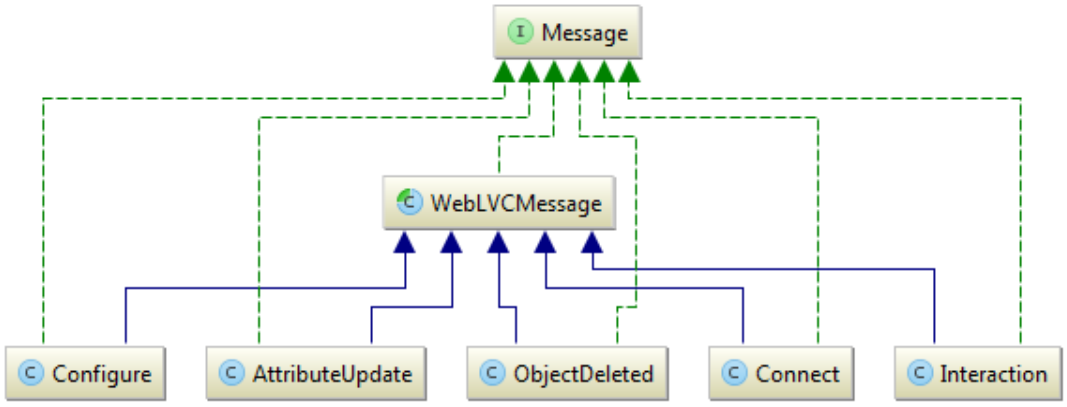


Figur 4.1 *Hovedkomponentene i WebLVC-tjeneren. Klientene kommuniserer med tjeneren med WebLVC over WebSocket, og tjeneren kommuniserer med HLA gjennom HlaLib. Når en WebLVC-melding kommer inn fra en klient, prosesseres den i* `WebLVCManager`*. Her konverteres JSON-strengen til et Java-objekt og meldingen prosesseres. Hvis meldingen inneholder simuleringsdata, konverteres den til dataformatet brukt av HLA. Dette skjer i* `WebLVCTranslator`*. Simuleringsdataene sendes så ut på HLA gjennom* `WebLVCHlaManager`*, som sender og mottar data fra HLA. Det samme skjer i motsatt rekkefølge for data mottatt fra HLA. I* `ObjectRepository` *lagres simuleringsdataene som mottas, både fra HLA og fra klientene.* `Simulation` *utfører dødregning av eksterne objekter på vegne av klientene.* `Control` *administrerer og styrer hele prosessen, som er beskrevet mer detaljert i kapittel 4.1.*

WebLVC-meldinger, og sende dem til klienter som abonnerer på datatypen. Hovedkomponentene i tjeneren kan sees i Figur 4.1.

4.1.1 Prosessering av WebLVC-meldinger

Det er implementert Java-objekter for alle grunnleggende meldingstyper i WebLVC. Det første som skjer når en melding kommer inn fra en klient, er at JSON-strengen leses inn i et Java-objekt tilsvarende en av de 13 meldingstypene, gitt i Tabell 3.1. Dette skjer ved hjelp av Jackson [48], et Java-bibliotek for å konvertere JSON-strenger til Java-objekter. Ved bruk av Jackson må utvikleren selv skrive Java-objektene og definere variabler og metoder som skal tilsvare JSON-egenskapene. Det er implementert en Java-klasse per meldingstype, og alle klassene implementerer grensesnittet `Message`. Deler av klassehierarkiet kan sees i Figur 4.2.



Figur 4.2 Deler av `Message`-hierarkiet. Legg merke til den abstrakte klassen `WebLVCMessage` som implementerer flere av metodene i grensesnittet `Message`, og som de andre klassene arver fra.

For å få programmet til å konvertere JSON-strenger til riktig meldingstype-objekt, må det defineres flere JSON subtyper over `Message`-grensesnittet. Dette kan sees i Figur 4.3. Etter at meldingen er konvertert til riktig Java-objekt legges den i en kø av meldinger. En tråd henter ut og prosesserer meldinger fra denne køen. Prosesseringen styres av klassen `WebLVCManager`.

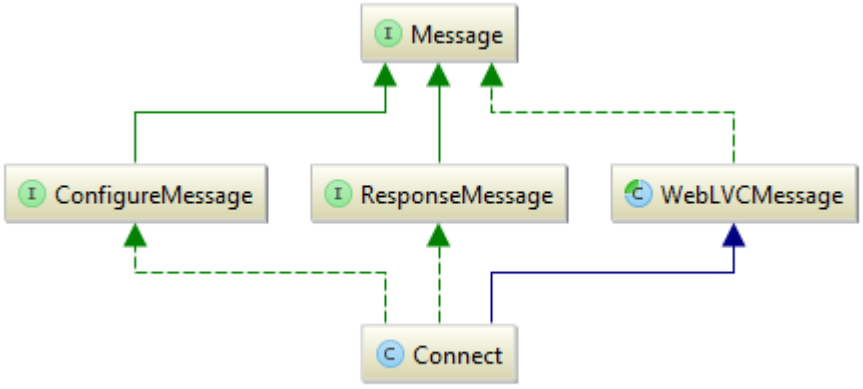
```
@JsonTypeInfo(use=JsonTypeInfo.Id.NAME,
include=JsonTypeInfo.As.PROPERTY, property="MessageKind")
@JsonSubTypes(value = {
        @JsonSubTypes.Type(value=Connect.class, name="Connect"),
        @JsonSubTypes.Type(value=ConnectResponse.class,
name="ConnectResponse"),
        …
})
public interface Message {}
```

Figur 4.3 Grensesnittet `Message` med JSON-subtyper.

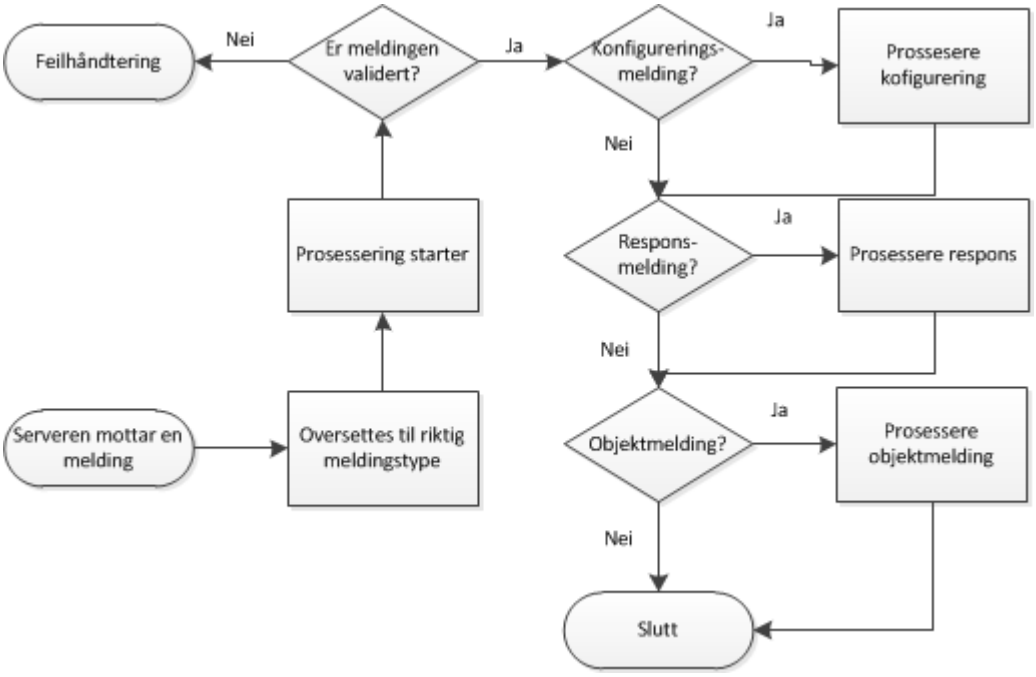
Første steg i prosesseringen av meldinger er validering. Dette skjer ved hjelp av en JavaBean-validator [49]. Her valideres meldingen for egenskaper som er obligatoriske å ha med. Andre feil i meldingen blir oppdaget under konverteringen fra JSON-streng til riktig meldingstype-objekt, eller under den videre prosesseringen. Ved oppdagelse av feil kan en av to ting skje: Ved en tilkoblingsmelding fra en ny klient vil klienten få negativ respons, sammen med flere feilmeldinger. For allerede eksisterende klienter vil feilen logges i klientens statuslogg.

Meldingstypene har litt ulike egenskaper/funksjoner. Noen konfigurerer tjeneren, noen krever et svar til klienten, andre sender data ut på HLA-nettverket. For å gjøre prosesseringen lettere er derfor klassene som representerer meldingstypene organisert etter disse egenskapene. Egenskapene er implementert ved hjelp av grensesnitt, og en klasse kan implementere flere

grensesnitt. En `Connect`-melding konfigurerer tjeneren og krever en respons, og klassen implementerer derfor to slike grensesnitt. Dette kan sees i Figur 4.4. Et flytskjema av hele prosessen kan sees i Figur 4.5.



Figur 4.4 Klassen `Connect` implementerer grensesnittene `ConfigureMessage` og `ResponseMessage`.

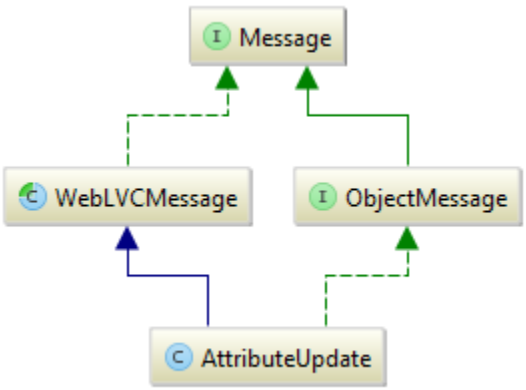


Figur 4.5 Flytskjema for prosessering av meldinger. Her vises de overordnede prosessene som klassen `WebLVCManager` er ansvarlig for.

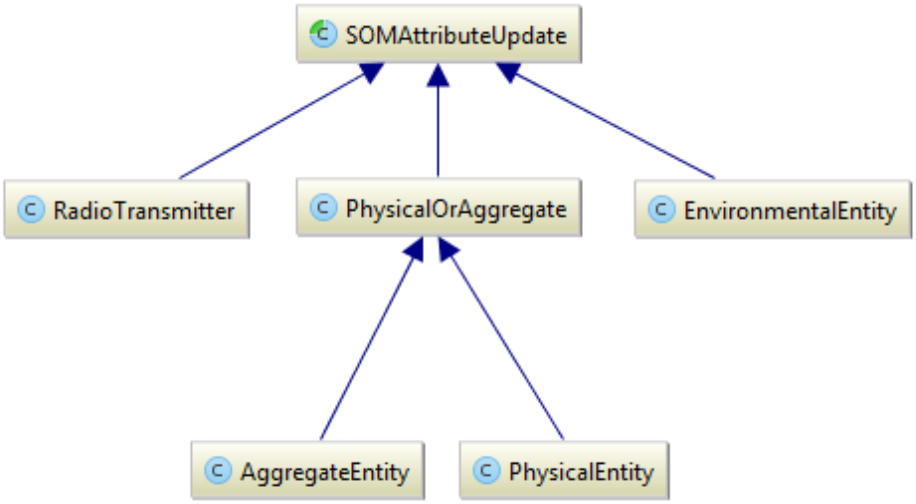
4.1.2 Objekt- og interaksjonsmeldinger

Java-klassene som representerer objekt- og interaksjonsmeldingstypene implementerer grensesnittet `ObjectMessage`, se Figur 4.6, og blir prosessert i flere steg. Meldingene må konverteres fra WebLVC-protokollen til dataformatet brukt av HLA. For å kommunisere med RTI-en brukes HlaLib og dataformatet brukt i HlaLib er klasser av typen `HlaObject` og `HlaInteraction` (heretter betegnet som HLA-objekter og -interaksjoner). Tjeneren skiller mellom eksterne objekter, objekter mottatt fra føderasjonen, og interne objekter, objekter som tilhører en av klientene. Alle objekter er lagret i et sentralt objekt-lager, i klassen `ObjectRepository`, basert på om objektet er eksternt, internt og hvilken klient objektet tilhører. Interaksjoner blir ikke lagret, da disse representerer hendelser og ikke persistente objekter. Objektene lagres som HLA-objekter. For `ObjectDeleted`-meldinger er prosesseringen enkel. Her hentes riktig objekt ut av objekt-lageret før det slettes fra føderasjonen og tjeneren. For meldinger av type `AttributeUpdate` og `Interaction` skjer prosesseringen i flere steg.

Først klassifiseres objektet eller interaksjonen etter riktig type. Her tas det utgangspunkt i standardobjektmodellen, se Tabell 3.2. Etter klassifikasjonen konverteres meldingen til et Java-objekt av riktig type. Det er definert et hierarki av klasser basert på standardobjektmodellen, og Java-biblioteket Jackson brukes for å konvertere til riktig type. Deler av hierarkiet, for objekttyper, kan sees i Figur 4.7. Det er ved konverteringen til riktig objekt- eller interaksjonstype, for eksempel til en av klassene gitt i Figur 4.7, at selve konverteringen av simuleringsdataene skjer, fra WebLVC- til HLA-type. Dette skjer ved at JSON-datatypen konverteres til datatypen brukt i HLA. Jackson kan benyttes til å konvertere enkle datatyper, men klarer ikke komplekse datatyper. Siden standardmodellen er håndskrevet (se kapittel 3.4.1) er ikke alltid oversettelsen fra JSON-type til HLA-type triviell. Noen av klassene for objekt- og interaksjonstypene inneholder derfor metoder som konverter til og fra riktig datatype. Oversettelsesprosessen styres av klassen `WebLVCTranslator`.



Figur 4.6 En `AttributeUpdate`-meldig implementerer grensesnittet `ObjectMessage`.



Figur 4.7 De ulike `AttributeUpdate`-meldingingstypene standardmodellen støtter. Det defineres en `AttributeUpdate`-meldingstype per objekttype.

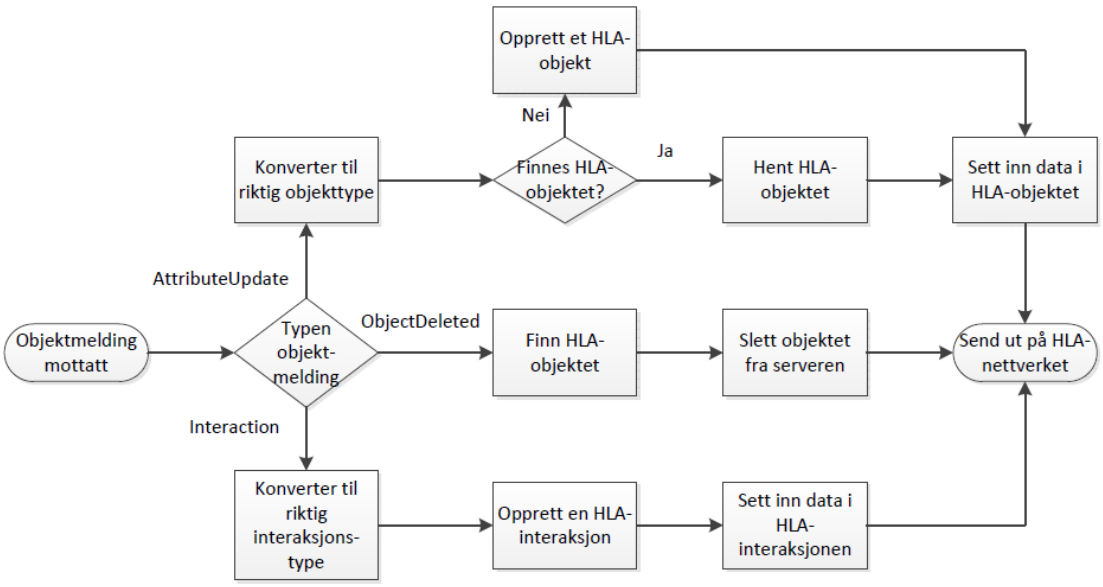
Etter at objekt-attributtene eller interaksjons-parameterne er konvertert oppretter tjeneren et passende HLA-objekt eller -interaksjon. For objekter sjekkes det først om objektet finnes fra før i objekt-lageret. Egenskapene blir lagt inn i HLA-objektet eller -interaksjonen, ved hjelp av Java Reflection API-et[5]. før de blir sendt ut til RTI-en via klassen `WebLVCHlaManager`. Hele prosessen kan sees i Figur 4.8. For objekter og interaksjoner mottatt fra HLA skjer det samme, bare i motsatt rekkefølge.

For kunne bruke flere RPR FOM-attributter og -parametere enn standardmodellen støtter er det også implementert noen serialiseringsklasser, som skal konvertere fra ulike HLA/RPR FOM-typer til en JSON-streng. Disse er basert på standard regler for å konvertere fra RPR FOM-typer til WebLVC-typer, definert i standarden.

4.1.3 Klienter

Hver klient er representert i tjeneren som et Java-objekt med en egen WebSocket-sesjon. All konfigureringsinformasjon for tjeneren er lagret i dette klient-objektet, sammen med informasjon om klientens abonnementer, altså hvilke objekter og interaksjoner klienten er interessert i. Når tjeneren mottar en tilkoblingsmelding av en ny klient opprettes det et nytt klientobjekt. Klientene blir identifisert ved hjelp av et unikt navn og tjeneren forventer alltid en tilkoblingsmelding som første melding fra en ny klient. Hver innkommende melding fra en klient har en referanse til klienten og klientens objekter er lagret etter klient i objekt-lageret.

[5] http://docs.oracle.com/javase/tutorials/reflect



Figur 4.8 Overordnet flytskjema for prosessering av objektmeldinger.

4.1.4 Dødregning

Standardmodellen gir klienter mulighet til å velge om tjeneren skal dødregne objekter mottatt fra HLA. Dette skjer i en separat tråd, i objektet `Simulation`. Denne kjører i sanntid og oppdaterer objektene hvert sekund. For å unngå å sende oppdateringer på stillestående objekter sender tjeneren kun oppdateringer hvis objektet har beveget seg. Alle dødregningsmetodene definert i DIS/RPR FOM er implementert, basert på implementasjonen av metodene utført i forbindelse med det FFI-utviklede tidsstyringsføderatet MAESTRO [50].

4.1.5 Føderasjonsinformasjon

For å kunne tilby klientene et komplett bilde av føderasjonen, abonnerer tjeneren på alle typer objekter og interaksjoner definert i RPR FOM. Tjeneren støtter foreløpig FOM-fila *RPR_FOM_v2.0_draft21_1516-2010.xml*. Som definert i HlaLib-manualen [46] blir denne spesifisert i en fil kalt *HlaLibConfig.xml*. HlaLibConfig-fila som brukes er automatisk generert fra FOM-fila ved hjelp av Java-biblioteket Java Architecture for XML Binding (JAXB) [51], som kan serialisere og deserialisere XML. Dette er gjort ved hjelp av programmet `FomToXml`, som kan finnes i `config`-mappa i kildekoden til tjeneren.

4.2 Begrensninger ved implementasjonen

Det er noen begrensninger i implementasjonen av tjeneren. På grunn av oppgavens omfang har det vært nødvendig å prioritere og det er derfor kun blitt implementert en delmengde av den foreslåtte WebLVC-protokollen. WebLVC-tjeneren implementert her støtter ikke avansert

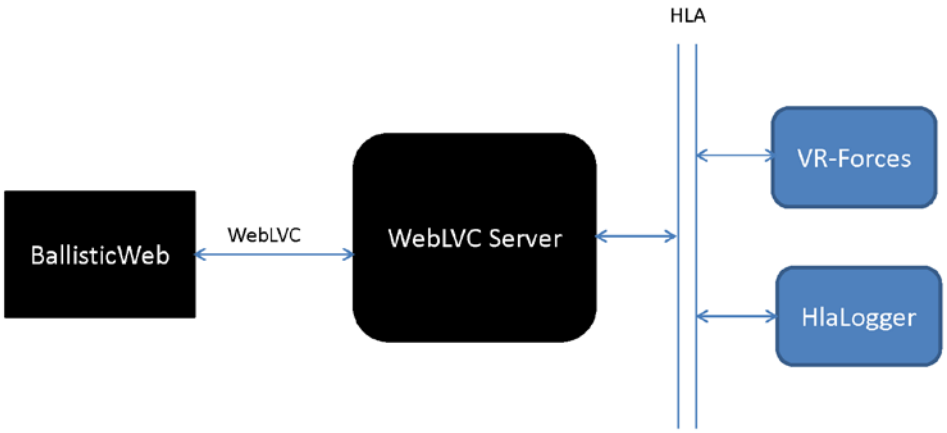
abonnements-håndtering, kun enkle abonnement basert på objekt- og interaksjonstype. Tjeneren er implementert ved hjelp av HlaLib og er derfor tett knyttet til HLA. Det burde allikevel være mulig å sende PDU-er ut på DIS ved hjelp av en ekstern HLA-DIS gateway.

Tjeneren støtter store deler av standardobjektmodellen, med noen viktige unntak, som er beskrevet her. Klientene kan ikke konfigurere dødregningen, ved å sette en terskel på posisjon og en maksimal oppdateringsrate. Standardmodellen gir klienten mulighet til å definere et geografisk interesseområde for objektoppdateringer, ved hjelp av objektene `WorldBounds` eller `ObjectBounds`, men dette støttes foreløpig ikke av tjeneren. Tjeneren støtter to ulike koordinatsystemer, "Earth-Centered, Earth-Fixed" (ECEF) Cartesian og World Geodetic System (WGS) 84. For posisjonsoppdateringer fra en klient i det geodetiske koordinatsystemet konverterer tjeneren posisjonen til ECEF-koordinater før det sendes ut på HLA. Posisjonsoppdateringer fra HLA blir også oversatt før det sendes ut til klienten. Klienter som sender posisjonsoppdateringer i et geodetisk koordinatsystem kan ikke spesifisere hvilken dødregningsmodell (se Tabell 2.1) objektene skal bruke. Dødregnings-modellen til disse objektene settes derfor til "static". Tjeneren støtter meldinger med tidsstempel, men tidsstempelet må være gitt i datatypen double og representere antall sekunder siden simuleringen startet. Tjeneren sender bare tidsstemplene videre med meldingen, ut på HLA eller ut til klientene.

Tjeneren er tilrettelagt for at det skal være mulig å utvide funksjonaliteten. Tjeneren støtter kun deler av RPR FOM, men har innebygget funksjonalitet som gjør det mulig å utvide standardmodellen med flere egenskaper for hver objekt- og interaksjonstype. Ved å opprette nye objekt- og interaksjonsklasser burde det også være mulig å utvide standardmodellen med flere objekt- og interaksjonstyper.

5 Testføderasjonen

For å teste tjeneren ble det utviklet en testføderasjon. Testføderasjonen består av tre føderater, BallisticWeb, en JavaScript nettleserapplikasjon som simulerer en granat, VR-Forces [52] som simulerer kjøretøy og en logger som logger dataene sendt mellom føderatene. Testføderasjonen kan sees i Figur 5.1. Både WebLVC-tjeneren, VR-Forces og loggeren er koblet på HLA-nettverket via en RTI utviklet av VT MÄK [53], mens BallisticWeb er koblet på føderasjonen via WebLVC-tjeneren.



Figur 5.1 Testføderasjonen.

5.1 Nettleserapplikasjonen BallisticWeb

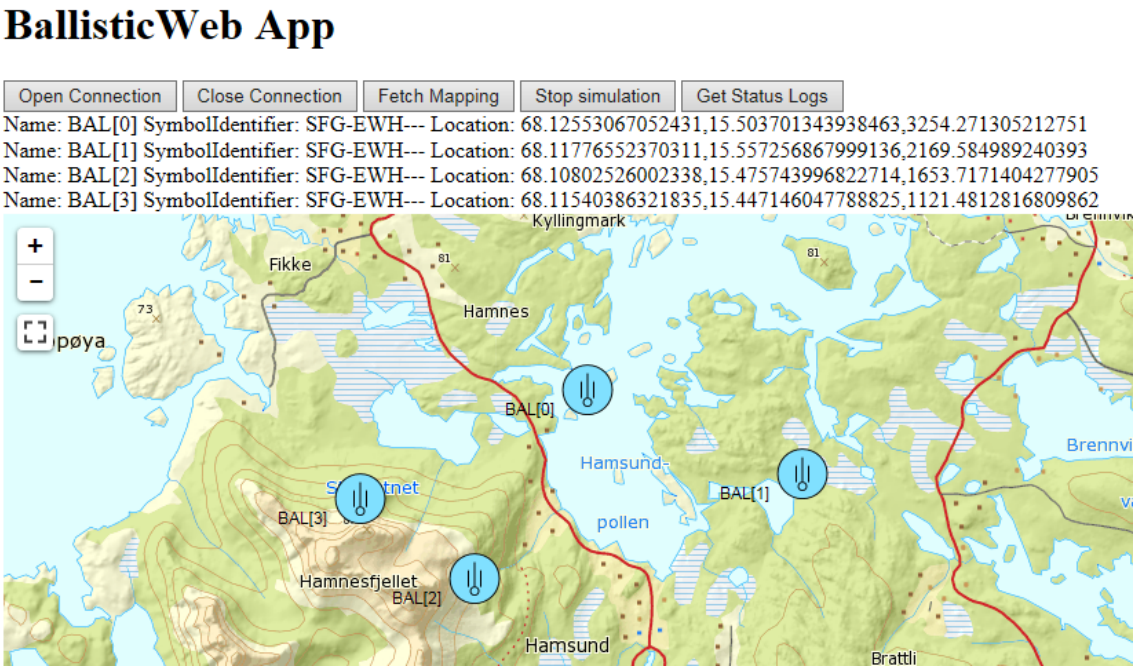
BallisticWeb er en JavaScript-applikasjon som kjører i nettleseren. Applikasjonen har et enkelt brukergrensesnitt, vist i Figur 5.2, som består av et kart og noen knapper som brukeren kan trykke på for å koble seg av og på tjeneren og for å be om statusinformasjon fra tjeneren. Applikasjonen simulerer en granat, og en granat avfyres når brukeren trykker på kartet, fra det stedet brukeren trykket. Applikasjonen tegner også opp enheter mottatt fra tjeneren og sjekker om granaten treffer (kolliderer med) enhetene. Hvis granaten treffer en enhet sendes det en `MunitionDetonation`-interaksjon til tjeneren.

5.1.1 Modellen av granaten

Modellen av granaten er hentet fra [50], der den ble brukt i et ballistikk-føderat laget for å teste MAESTRO. Modellen inkluderer gravitasjon, luftmotstand, jordens sentrifugalkraft og Corioliskraft. Granaten er modellert etter en tysk kanon fra første verdenskrig, og skytes opp med en bestemt vekt (106 kg), fart (1640 m/s) og retning. For en detaljert beskrivelse, se [50].

5.2 Implementasjon av BallisticWeb

BallsticWeb er implementert i JavaScript ved hjelp av rammeverket AngularJS [54], og deler av applikasjonen er inspirert/hentet fra brukergrensesnittet i Simulation-supported Wargaming for Analysis of Plans (SWAP) [55]. AngularJS strukturerer nettleserapplikasjoner etter arkitekturmønsteret Model-View-Controller (MVC), et arkitekturmønster som er mye brukt i utviklingen av grafiske brukergrensesnitt. Rammeverket er data-drevet og tilbyr toveis databinding. Det vil si at AngularJS automatisk oppdaterer brukergrensesnittet når den underliggende datamodellen har endret seg, og motsatt. Presentasjonslogikken, hva slags data



Figur 5.2 BallisticWeb brukergrensesnitt.

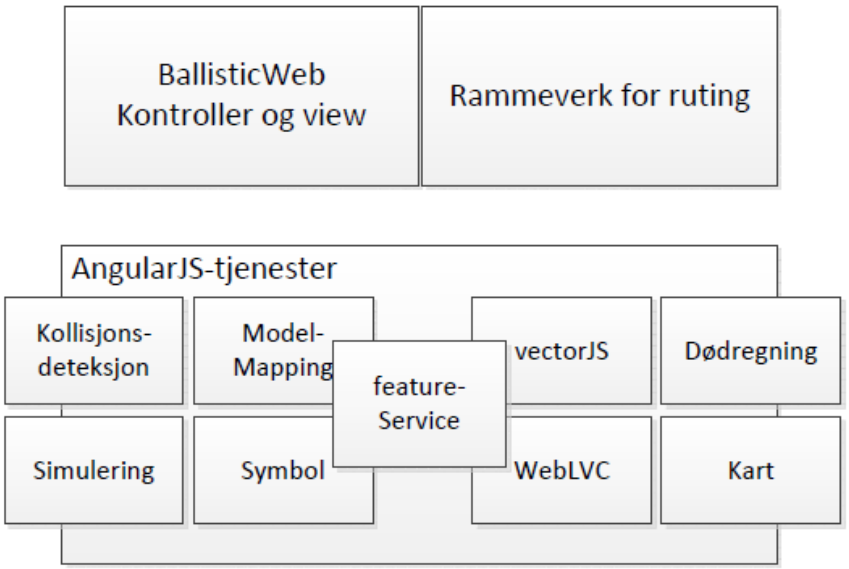
som skal hentes og vises frem, styres av en kontroller. I BallisticWeb er det kun et view (en nettside) og en kontroller som styrer nettsiden, `ballisticWebCtrl`.

AngularJS bruker direktiver til å utvide HTML-vokabularet. Et direktiv er et HTML-element som definerer en gjenbrukbar presentasjonskomponent. Det finnes flere innebygde AngularJS-direktiver, som direktivet `ng-repeat` som kan iterere over en liste. Kartet i BallisticWeb tegnes opp ved hjelp av et egenlaget direktiv. AngularJS har også funksjonalitet for ruting mellom forskjellige sider, og det er definert et rammeverk for ruting i applikasjonen. Dermed vil det være enkelt å utvide applikasjonen senere med flere sider og tilhørende kontrollere.

5.2.1 AngularJS-tjenester

Et annet nyttig AngularJS-konsept er tjenester. Ordet "tjeneste" i AngularJS-sammenheng betyr ikke det samme som en tjeneste i et tjenesteorientert system. AngularJS-tjenester er funksjoner eller objekter som kan holde tilstand over hele applikasjonen, og er ideelt for felles funksjonalitet og datalagring. Tjenestene representerer gjenbrukbare API-er. Alle tjenestene beskrevet i dette underkapittelet er tjenester i AngularJS-forstand.

AngularJS-tjenester brukes ved hjelp av "dependency injection", og utvikleren deklarerer i koden hvilke tjenester hun har bruk for. AngularJS vil så opprette tjenestene ved behov, en instans av hver tjeneste. AngularJS har et sett med innebygde tjenester, i tillegg til at utvikleren kan implementere egne tjenester. Mesteparten av funksjonaliteten i BallisticWeb kommer fra tjenester, og de viktigste av disse er beskrevet i de neste delkapitlene. Figur 5.3 gir en oversikt over de ulike komponentene og de egenlagde AngularJS-tjenestene som finnes i applikasjonen.



Figur 5.3 De ulike komponentene i applikasjonen BallisticWeb.

5.2.2 Kjernefunksjonalitet

Kjernefunksjonaliteten i applikasjonen styres av `featureService`, som er en tjeneste sammensatt av mange mindre tjenester. Denne tjenesten styrer lastingen av kart, tegner opp objekter mottatt fra tjeneren, og setter i gang simuleringen når brukeren trykker på kartet. Den administrerer også data og konfigurasjonen de andre tjenestene trenger for å gjøre jobben sin. Kartet blir tegnet opp ved hjelp av JavaScript-biblioteket Leaflet [56], og kartet som brukes er et rasterkart fra Kartverket som lastes ned fra en enkel kart-tjener [55].

5.2.3 WebLVC

Tilkobling og kommunikasjon med WebLVC-tjeneren skjer over WebSockets. Det er laget to tjenester som håndterer kommunikasjon og mottak av meldinger (WebLVC-tjenestene i Figur 5.3). WebSocket-koblingen opprettes ved hjelp av JavaScripts innebygde WebSocket-funksjonalitet. Når WebLVC-tjenesten mottar en melding fra tjeneren kringkastes denne til resten av applikasjonen. Oppdateringer av enheter i testføderasjonen mottas ved hjelp av `AttributeUpdate`-meldinger fra WebLVC-tjeneren. Ved mottak av en `AttributeUpdate`-melding tegner `featureService` enheten i kartet og sier fra til alle interesserte tjenester at det er kommet et nytt objekt.

5.2.4 Opptegning av enheter

Det er ønskelig at enheter tegnes opp i kartet ved bruk av symboler fra Allied Procedural Publication 6 (APP-6) [57]. Hvert symbol har en tilhørende symbolidentifikasjonskode og

enhetene som mottas over WebLVC har en WebLVC-entitetstype[6] som beskriver hva slags enhet dette er. For å få tegnet opp riktig symbol i kartet må vi derfor oversette fra entitetstype til APP-6-kode. Oversettelsen skjer i en tjeneste ved navn `modelMappingService`, som tar utgangspunkt i en JSON-fil med oversettingsregler. Fila som brukes kan sees i Vedlegg A og finnes også i kildekoden til BallisticWeb. Brukeren kan selv endre fila for å få andre (og bedre) oversettingsregler. Det grafiske symbolet som tegnes på kartet lages av tjenesten `symbolService`, som bruker JavaScript-biblioteket milsymbol [58] for å tegne opp symbolene.

JSON-fila som brukes for oversetting er generert basert på en XML-fil med oversettingsregler. For brukere som vil endre oversettingsfila anbefales det først å lage en XML-fil med oversettingsregler, som så kan oversettes til JSON ved hjelp av Java-programmet `XmlToJson` som finnes i mappa `modelmapping` i kildekoden til BallisticWeb. XML-fila må være i henhold til et XML-skjema, utviklet i forbindelse med [59]. Skjemaet kan finnes i kildekoden til BallisticWeb og kan sees i Vedlegg B.

5.2.5 Simulering og kollisjonsdeteksjon

Simuleringen av granatene kjører i sanntid med et tidssteg på 0,1 sekund. Det er utviklet en simuleringstjeneste som utfører dette, og den innebygde AngularJS-tjenesten `$interval` brukes for å kjøre simuleringen for hvert tidssteg. Ved hvert tidssteg utføres numerisk integrasjon etter den eksplisitte Euler-metoden (forward Euler). Det er utviklet en JavaScript-versjon av deler av Vector3D, et Java-bibliotek utviklet ved FFI [50], for beregninger med vektorer, kvaternioner og koordinattransformasjoner. Utregningene skjer ved hjelp av dette biblioteket.

Det er utviklet en enkel tjeneste for kollisjonsdetektering, som detekterer kollisjoner mellom objekter ved å se om det er overlapp mellom objektene i alle tre retninger (x, y og z). Kartet som brukes i applikasjonen har lav oppløsning, og granatene skytes opp med en forutbestemt fart og retning. For å sørge for at det allikevel kan skje kollisjoner antas det at hvert objekt har en utstrekning på 100 meter i alle tre retninger. Kollisjonstjenesten kalles en gang per tidssteg. Ved kollisjon sendes det ut en `MunitionDetonation`-interaksjon til tjeneren. For hvert tidssteg regner applikasjonen også ut en dødregningsmodell for granatene, gjennom en tjeneste for dødregning.

5.2.6 Oppdatering av brukergrensesnittet

Som nevnt i kapittel 5.1 oppdaterer AngularJS brukergrensesnittet hver gang den underliggende datamodellen endrer seg. Dessverre er det noen begrensninger med hensyn på hvordan dette utføres. For å unngå periodiske oppdateringer oppdaterer AngularJS bare brukergrensesnittet ved noen faste hendelser, som at brukeren gjør en endring i datamodellen, at applikasjonen får en respons på en tjenerforespørsel eller at en av AngularJS-tjenestene `$timeout` eller `$interval` kjøres [54]. Når meldinger mottas fra WebLVC-tjeneren gjennom WebSocket, vil

[6] Dette er den samme entitetstypen som brukes i DIS (DIS enumeration).

dermed ikke applikasjonen oppdatere brukergrensesnittet automatisk. For å løse dette, er det sørget for at AngularJS utfører periodiske oppdateringer. Dette skjer ved å la simuleringen av granatene kjøre i bestemte intervaller med metoden `$interval`, som beskrevet i kapittel 5.2.5. På denne måten vil brukergrensesnittet oppdateres minst like ofte som granatene oppdateres.

5.2.7 Konfigurering av applikasjonen

Applikasjonen tilbyr konfigurering av flere parametre, som hvilken kart-tjener som skal brukes, hvilke koordinater kartet skal åpne med, hvor langt tidssteg simuleringen skal kjøre med og adressen til WebLVC-tjeneren. Dette kan endres i en sentral konfigureringsfil, `app.config.js`.

5.3 VR-Forces

Kjøretøyene granatene skal treffe er simulert i VR-Forces [52], et simuleringsrammeverk for datagenererte styrker utviklet av VT MÄK. En detaljert beskrivelse (av en noe eldre versjon) av VR-Forces er gitt i [60]. Kjøretøyene som brukes er av typen Leopard 2 stridsvogn.

5.4 HlaLogger

HlaLogger er et Java-føderat som abonnerer på objekter og interaksjoner brukt i testføderasjonen. Loggeren skriver ut hvilke objekter og interaksjoner den har mottatt. På denne måten kan utvikleren få oversikt over simuleringsdataene som sendes over RTI-en.

6 Testing og resultater

I denne rapporten er den kommende standarden WebLVC presentert, sammen med utviklingen av en WebLVC-tjener og en tilhørende testføderasjon. I dette kapittelet presenteres resultatene. Det er utført tester av tjenerens funksjonalitet og ytelse, sammen med undersøkelser av ressursbruken til protokollen og andres erfaringer med bruk av WebLVC.

6.1 Testing av tjeneren

Applikasjonen ble testet med tre ulike scenarioer:

- Et enkelt scenario med en granat og et stillestående kjøretøy i VR-Forces.
- Flere granater i høy fart og et stillestående kjøretøy i VR-Forces.

- Flere granater og kjøretøy med høy fart.

Formålet med de ulike scenarioene var å teste den grunnleggende tjenerfunksjonaliteten og teste ytelsen til tjeneren. Et av målene med WebLVC er en standard med høy nok ytelse til å kunne brukes i interaktive sanntidssimuleringer. Tjeneren er en viktig del av WebLVC-konseptet og vil ha stor betydning for ytelsen. WebLVC-tjeneren presentert her er utviklet for testformål og er ikke optimert for høy ytelse. Resultatene vil dermed ikke gi et korrekt bilde på hvor effektiv en WebLVC-tjener kan være, men kan bidra til å gi et bilde på hvor mye utviklingsjobb som kreves for å få en "god nok" WebLVC-tjener.

6.1.1 Testing av grunnleggende funksjonalitet

Tjeneren ble først testet med et enkelt scenario der en granat ble skutt opp mot et stillestående kjøretøy i VR-Forces. Denne testen ble utført for å teste grunnleggende tjenerfunksjonalitet. Kommunikasjonen mellom klient og tjener ble testet, med tilkoblingsmeldinger, sending av statuslogger, oppsett av dødregning og abonnementshåndtering.

Formålet med scenarioet var å teste utveksling av simuleringsdata mellom VR-Forces og nettleserapplikasjonen gjennom WebLVC-tjeneren. Flere objekt- og interaksjonsmeldinger ble sendt, blant annet objektmeldinger av typen `WebLVC:PhysicalEntity` og `WebLVC:EnvironmentalEntity`. Samhandling mellom objektene i nettleserapplikasjonen og VR-Forces ble testet ved hjelp av kollisjoner mellom objektene, ved at en granat traff et kjøretøy simulert i VR-Forces. Ved en kollisjon ble det sendt en `WebLVC:MunitionDetonation` fra nettleserapplikasjonen til VR-Forces. Hvis VR-Forces registrerte kollisjonen ble det sendt tilbake en objektmelding som fortalte om kjøretøyet ble skadet. Ved skadet kjøretøy ble kjøretøyet fjernet fra nettleserapplikasjonen. På denne måten var det mulig å se om den grunnleggende funksjonaliteten til tjeneren fungerte som den skulle.

HlaLib sender av og til feilmeldinger ved mottak av objekter fra VR-Forces (objekter av typen GroundVehicle), men tjeneren klarer allikevel å sende objektet videre til klienten. Det er ikke klart hvorfor denne feilmeldingen sendes. Kollisjonsberegningene fungerte fint i nettleserapplikasjonen og `WebLVC:MunitionDetonation`-interaksjoner ble sendt ut på HLA-nettverket.

6.1.2 Ytelsestester med mange granater

For å teste ytelsen på tjeneren ble testføderasjonen kjørt med et varierende antall granater, mellom 5 og 400 granater. HLA-loggeren ble brukt til å overvåke dataene som ble sendt ut på HLA-nettverket, for å se om dataene kom frem. Nettleserapplikasjonen sendte mellom fem og ti oppdateringer i sekundet per granat (som beskrevet i kapittel 5.1 ble granaten sendt opp med en gitt fart og retning, og det ble sendt ut til sammen 1596 oppdateringer i et objekts levetid). Ytelsen ble målt ved å se på forsinkelsen på mottak av `WebLVC:MunitionDetonation`-interaksjoner hos HLA-loggeren og VR-Forces. Det er derfor flere mulige feilkilder i resultatene, og resultatene vil bare gi et omtrentlig bilde av situasjonen.

For opp til 150 granater ble all informasjon sendt ut på HLA uten (synlige) forsinkelser, og kollisjonen mellom objektene ble registrert samtidig i VR-Forces og i nettleserapplikasjonen. Dette tilsvarer mellom 750 og 1500 innkommende WebLVC-meldinger per sekund. Med rundt 100 granater begynte granatene tegnet opp i VR-Forces å hakke, noe som kan skyldes VR-Forces selv, eller at dataene ble sendt noe forsinket ut fra tjeneren. Ved 150 granater ble interaksjonen sendt ut noe forsinket fra tjeneren, med mindre enn et sekunds forsinkelse. Forsinkelser på et sekund eller mer kom ved simulering av 400 granater. Meldingskøen og prosesseringer av meldinger i tjeneren kan bli en flaskehals ved store mengder innkommende meldinger per tid. Ved bruk av flere tråder som henter ut og prosesserer meldinger kan dette reduseres, men merk at bruk av for mange tråder vil også kunne bli en flaskehals.

Av og til, ved bruk av mange granater, fikk ikke tjeneren registrert objekttypen til en granat (dette sendes bare med første melding) og granaten ble derfor ikke sendt videre ut på HLA. Det viste seg at dette skyldes at tjeneren ikke kan ha flere objekter med samme navn. HlaLib har en mekanisme for gjenbruk av navn fra objekter som er slettet fra føderasjonen, men dette ser ikke ut til å fungere[7]. Derfor vil tjeneren ikke kunne ta i mot flere objekter med samme navn, selv etter at objektet er slettet fra føderasjonen. Granatene i nettleserapplikasjonen blir navngitt etter posisjonen de starter fra, og hvis det blir opprettet flere granater med nøyaktig samme posisjon vil ikke tjeneren godta dette. Neste gang det kommer en oppdatering fra dette objektet, vil tjeneren ikke kjenne igjen objekttypen, og objektet sendes ikke videre. En enkel løsning for å unngå dette problemet er å restarte tjeneren mellom hver test-sesjon. En mer permanent løsning er naturligvis å løse problemet.

6.1.3 Ytelsestester med mange objekter i bevegelse

WebLVC-tjeneren ble også testet med kjøretøy fra VR-Forces i fart. Nettleserapplikasjonen simulerte 50 granater. VR-Forces sender ut flere oppdateringer i sekunder per kjøretøy. Også her ble det testet med gradvis flere kjøretøy i bevegelse, fra 5 til 100 kjøretøy. For mellom 5 og 80 kjøretøy i bevegelse ble oppdateringer om granatene sendt ut uten (synlige) forsinkelser, og kollisjonen ble registrert samtidig i begge føderatene. Ved 80 kjøretøy begynte nettleserapplikasjonen å henge noe ved mange oppdateringer fra tjeneren. Dette skyldes at hver gang en melding mottas kringkastes denne ut i applikasjonen, noe som kan overbelaste applikasjonen. Derfor burde meldingene heller legges i en meldingskø, som blir tømt med jevne mellomrom (for eksempel like ofte som granatene oppdateres). Ved 100 kjøretøy begynte interaksjonene sendt fra applikasjonen å komme noe forsinket frem.

6.1.4 Tjeneren og ytelse

Tjeneren har problemer med ytelsen ved store mengder innkommende meldinger per tid. Det er derfor utført en vurdering av hvilke deler av tjeneren som er aktuelle for optimalisering. I løpet av et objekts levetid blir det sendt mange posisjonsoppdateringer fra nettleserapplikasjonen til tjeneren, og prosesseringstiden til disse meldingene vil få stor betydning for ytelsen. Det er

[7] Det er ikke kjent om problemet ligger i HlaLib eller i RTI-en som brukes, MÄK RTI 4.4.1.c.

derfor fokusert på prosesseringstiden for objektmeldinger. Gjennomsnittlig prosesseringstid for posisjonsoppdateringer er 1 millisekund[8].

Det er i hovedsak to deler av prosesseringen av meldinger som tar tid: valideringen av meldinger og konverteringer av meldinger fra WebLVC- til HLA-format. Konverteringen av meldinger tar lengst tid, særlig for nye objekter med et stort antall attributter. Det innebygde Java-API-et Reflection og Java-biblioteket Jackson brukes for å konvertere meldingene. Her kan det i fremtiden gjøres flere optimaliseringer, for eksempel ved å unngå bruk av Java Reflection, som er lite effektivt å bruke. Tiden det tar å validere meldingen er også en viktig faktor for posisjonsoppdateringer, og valideringen bør derfor også optimaliseres.

6.2 Ressursbruk i WebLVC

Ytelsen avhenger også av ressursbruken til protokollen, eller mengden data som sendes over nettverket. Tabell 6.1 gir størrelsen på et utvalg meldinger sendt fra nettleserapplikasjonen til WebLVC-tjeneren. Tallene er hentet ved bruk av Wireshark [61], et program som analyserer trafikken over et nettverk. Tabellen viser at WebSocket legger på omtrent 62 bytes per melding, i tillegg til størrelsen på JSON-strengene som sendes.

Den totale mengden data sendt over nettverket mellom en klient og en WebLVC-tjener for et objekt avhenger av fire faktorer:

- Størrelsen på første melding, $Load_{AU}$
- Størrelsen på siste melding, $Load_{OD}$
- Størrelsen på en posisjonsoppdatering, $Load_{POS}$
- Antall posisjonsoppdateringer, N

Som man kan se av Tabell 6.1 vil de to første faktorene være neglisjerbare i forhold til de to siste, og den totale mengden data sendt over nettverket for et objekt blir da:

$$Load_{AU} + Load_{POS} \times N + Load_{OD} = O(Load_{POS} \times N) \tag{6.1}$$

For en granat, basert på størrelsene i Tabell 6.1 og med 1594 posisjonsoppdateringer, gir dette omtrent 750 kB data sendt over nettverket.

6.3 Oppsummering av testene

Det er utført flere tester av WebLVC-tjeneren, for å teste både funksjonalitet og ytelse. Tjeneren kan motta og sende flere ulike typer WebLVC-meldinger og klienten kan konfigurere enkle

[8] Tiden er målt ved hjelp av den innebygde Java-metoden `System.nanoTime()` og gjennomsnittet er regnet ut basert på 1594 posisjonsoppdateringer for en granat.

Meldingstype	Payload størrelse (i bytes)	Total størrelse (i bytes)
`Connect`	90	152
`Connect` med konfigurering	343	405
Første `AttributeUpdate`	711	773
`AttributeUpdate` med posisjon	408	470
`ObjectDeleted`	89	149

Tabell 6.1 Størrelsen på et utvalg WebLVC-meldinger sendt fra nettleserapplikasjonen BallisticWeb.

egenskaper hos tjeneren, som abonnement på objekt- og interaksjonstyper og dødregning. Det er derfor mulig for nettleserapplikasjoner å delta i distribuerte simuleringer ved bruk av WebLVC.

Et av protokollens viktigste mål er god nok ytelse til å kunne brukes i sanntidssimuleringer. Ytelsen på tjeneren kan bli et problem for mange objekter med hyppige oppdateringer, som vist i kapittel 6.1.2. Selv om videre arbeid med tjeneren vil kunne gjøre den mer effektiv, se kapittel 6.1.4, vil WebLVC-tjeneren alltid kunne bli en flaskehals for simuleringen. En annen faktor for god ytelse er et effektivt meldingsformat. JSON-strengene som sendes over nettverket vil inneholde noe ekstra metadata i tillegg til simuleringsdataene som sendes, men som vist i Tabell 6.1 burde ikke dette være noe problem for nettverket. Tjeneren vil bli en flaskehals for simuleringen lenge før datamengden over nettverket blir det. Nettleserapplikasjonene er også en viktig faktor for ytelsen, og JSON, som er enkelt og raskt å bruke for nettleserapplikasjoner skrevet i JavaScript, burde bidra til å øke ytelsen til nettleserapplikasjonene.

6.4 Andres erfaringer med WebLVC

ICoViCS [39], se kapittel 2.4.1, hadde positive erfaringer med bruken av WebLVC, og mente WebLVC-tjeneren var avgjørende for prosjektets suksess. FOM-en som ble brukt i HLA-føderasjonen var en utvidelse av RPR FOM, og siden WebLVC-tjeneren til VT MÄK kan oversette vilkårlige FOM-er til JSON og har innebygget støtte for RPR FOM, ble utviklingskostnadene kraftig redusert. Oversettelsen fra RPR FOM til JSON i standardobjektmodellen er imidlertid skrevet for hånd, og det er flere av valgene som er tatt som gjør implementasjonen av tjeneren unødvendig komplisert. Noen forslag til forbedringer er derfor gitt i kapittel 7.

WebLVCs bruk av åpne nett-standarder var også en fordel [39], da disse er støttet av et stort antall nettlesere og burde bidra til lavere utviklings- og vedlikeholdskostnader i fremtiden. WebSocket er støttet av alle moderne nettlesere uten behov for programvareutvidelser, og JSON er et format som er enkelt og effektivt å bruke for nettleserapplikasjoner skrevet i JavaScript. ICoViCS argumenterte også for at WebLVC tilbyr en mer effektiv protokoll enn for eksempel HLA Web service API-et, noe som vil redusere mengden data som sendes over nettverket.

Mengden data sendt over nettverket i testføderasjonen, vist i Tabell 6.1, tyder på at dette stemmer.

Ved de innledende testene merket ICoViCS ikke noe til ytelsesproblemer med tjeneren, uten at de har utført noen formelle ytelsestester. ICoViCS brukte WebLVC for å lage et felles brukergrensesnitt for kontroll og visualisering av datagenererte styrker (CGF-er). Ytelseskravene til dette kan være noe lavere enn kravene som trengs for å simulere enheter direkte i en nettleser. WebLVC er laget for JavaScript-klienter, og JavaScript er heller ikke et programmeringsspråk med høy ytelse. Det kan derfor godt hende at WebLVC egner seg best til å gi brukere tilgang til et simuleringssystem via en nettleser, slik den ble brukt av ICoViCS, og ikke til å simulere enheter direkte i en nettleser.

ICoViCS hadde bare en ting å utsette på WebLVC: den manglende støtten for avanserte HLA-tjenester. Det kan argumenteres for at det ikke er hensiktsmessig å bruke avanserte HLA-tjenester i tynn-klient nettleserapplikasjoner, samtidig som WebLVC er en utvidbar protokoll. WebLVC-tjeneren som ble brukt fulgte en eldre versjon av spesifikasjonen, og i nyeste versjon er flere avanserte HLA-tjenester inkludert, som abonnementshåndtering og datadistribusjon [40].

7 Mulige forbedringer av protokollen

I første omgang er det flere valg i standardobjektmodellen som gjør det unødvendig komplisert å implementere tjeneren. Standardobjektmodellen er håndlaget og er basert på noen deler fra DIS og noen deler fra RPR FOM. Problemet er at konverteringen mellom WebLVC-type og RPR FOM-type ikke alltid er konsekvent for egenskaper som ligner på hverandre. Navnene gitt til egenskapene i WebLVC og RPR FOM stemmer heller ikke alltid overens, noe som gjør at implementasjonen av tjeneren krever detaljert og spesialisert kode. Det burde derfor tas en gjennomgang av objekt- og interaksjonstypene, for å se om det er mulig å strømlinjeforme konverteringen av datatyper, og da særlig konverteringen av egenskapsnavn. En mulighet er å se om konverteringsreglene definert i protokollen i større grad kunne vært brukt i standardobjektmodellen.

Valget av topp-nivå objekt- og interaksjonstyper (Tabell 3.2) gjør tjeneren mer komplisert, ved at riktig HLA-subtype må bestemmes ut i fra andre deler av meldingen. Dette vises særlig i interaksjonstypen `WebLVC:RadioInteraction`, som skal representere fire ulike radiointeraksjoner. Problemet er at de ulike radiointeraksjonene har få felles egenskaper, og de egenskapene som har likt innhold har ofte ulike navn. Da interaksjonstypen definert i protokollen skal kunne konverteres til alle de fire radiointeraksjonstypene, må flere av egenskapene konverteres ulikt avhengig av hvilken subtype interaksjonen skal konverteres til. Dette skaper unødvendig komplikasjon, både for implementasjonen av tjeneren og for brukere

av protokollen som skal forstå hvilke WebLVC-egenskaper som konverteres til hvilke HLA-egenskaper. Dette bryter også med retningslinjene for bruk av RPR FOM gitt i [62], der det er spesifisert at kun løvnoder (de dypeste subtypene) skal publiseres i en RPR FOM-basert HLA-føderasjon. Ved å definere et hierarki av objekt- og interaksjonstyper gjør man tjeneren mindre kompleks, uten å øke kompleksiteten for nettleserapplikasjonene, da hver applikasjon kun trenger å forholde seg til de objekt- og interaksjonstypene den har bruk for.

For klienter som sender posisjonsinformasjon i et geodetisk koordinatsystem har ikke klienten noen mulighet til å spesifisere hva slags dødregningsmodell objektene den sender ut skal beregnes med. Dette burde vært et mulig valg, siden klienter som sender posisjonsinformasjon i et geodetisk koordinatsystem også kan delta i en HLA-føderasjon, ved at tjeneren konverterer til ECEF-koordinatsystemet før objektene sendes ut på HLA-nettverket.

Tjenerens feilhåndtering, slik den er spesifisert i protokollen, består for det meste av å logge feil til klientens statuslogg. Klienten kan så be om denne statusloggen, og slik få tilgang til eventuelle feilmeldinger. Dette gjør klientene selv ansvarlige for å sjekke om det er skjedd en feil hos tjeneren under prosesseringen av en melding, noe som bidrar til at det kan ta lengre tid før en feil dukker opp hos klienten. Ved alvorlige feil hadde det vært nyttig at tjeneren sendte en feilmelding til klienten, slik at klienten kunne reagert på feilen så fort som mulig.

8 Konklusjon

Ved bruk av WebLVC er det mulig å inkludere nettleserapplikasjoner i distribuerte simuleringer. I denne rapporten er det vist at det er mulig å skyte en granat simulert i en nettleserapplikasjon mot et kjøretøy simulert i VR-Forces, med datautveksling gjennom HLA og en WebLVC-tjener. Protokollen er svært enkel å bruke på klientsiden og WebLVC-meldingene kommer på et naturlig format for JavaScript-klienter.

Meldingsformatet og tilkoblingen som brukes mellom klientene og WebLVC-tjeneren gir høy nok ytelse til å kunne brukes i interaktive sanntidssimuleringer. WebLVC-tjeneren derimot, som konverterer fra et dataformat til et annet, kan bli en flaskehals for simuleringen. Ved flere tusen innkommende meldinger i sekundet vil WebLVC-tjeneren bli en flaskehals og sende meldinger forsinket ut på HLA-nettverket. WebLVC-tjeneren beskrevet her er i første omgang ikke utviklet for å være rask eller effektiv (den er utviklet for testformål), og det er naturlig å anta at tjeneren kan gjøres raskere eller mer effektiv ved videre utvikling.

Standarden er kompleks å implementere på tjener-siden, og brukere av protokollen vil være avhengig av enten å sette av mye ressurser til å implementere en tjener, eller kjøpe en ferdig utviklet tjener, for eksempel den som tilbys av VT MÄK [36]. Særlig standardobjektmodellen

har mange detaljerte krav som krever spesialisert implementasjon. Her kunne det med fordel vært gjort noen forenklinger som ville gjort tjeneren raskere å implementere, uten at det går for mye ut over klienten. Et eksempel på dette er objekt- og interaksjonstypene, som ikke korresponderer helt med objekt- og interaksjonshierarkiet brukt i RPR FOM. Det burde være mulig å utvide objekt- eller interaksjonstypene til å stemme med hierarkiet, uten å gjøre ting for komplisert for klienten. På denne måten vil man følge gjeldende konvensjoner [62] og samtidig redusere kompleksiteten i tjeneren, noe som vil kunne gi en mer effektiv protokoll.

I det videre standardiseringsarbeidet med WebLVC bør det undersøkes om det går an å redusere kompleksiteten til tjeneren, og da særlig standardobjektmodellen, uten at kompleksiteten til klientene går for mye opp, slik at klientene selv blir flaskehalser for føderasjonen.

Referanser

[1] K. Al-Zoubi and G. Wainer, "RISE: A general simulation interoperability middleware container," *J. Parallel Distrib. Comput.*, vol. 73, no. 5, pp. 580–594, May 2013.

[2] IEEE, *IEEE Standard for Distributed Interactive Simulation – Application Protocols*, IEEE Std 1278.1-2012 (Revision of IEEE Std 1278.1-1995), 2012.

[3] IEEE, *IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) – Framework and Rules*, IEEE Std 1516-2010 (Revision of IEEE Std 1516-2000), August 2010.

[4] IEEE, *IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) – Federate Interface Specification*, IEEE Std 1516.1-2010 (Revision of IEEE Std 1516.1-2000), August 2010.

[5] IEEE, *IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) – Object Model Template (OMT) Specification*, IEEE Std 1516.2-2010 (Revision of IEEE Std 1516.2-2000), August 2010.

[6] F. Kuhl, R. Weatherly, and J. Dahman, *Creating Computer Simulation Systems: An Introduction to the High Level Architecture*. Upper Sadie River, NJ: Prentice Hall PTR, 1999.

[7] NSA, *STANAG 4603 (Edition 2) – Modelling and Simulation Architecture Standards for Technical Interoperability: High Level Architecture (HLA)*, 2012.

[8] R. M. Fujimoto, “DVEs: Introduction,” in *Parallel and Distributed Simulation Systems*. New York, NY: John Wiley & Sons, Inc., 1999, ch. 7, pp. 195–221.

[9] J. E. Hannay, K. Bråthen, and O. M. Mevassvik, “Simulation architectures and service-oriented defence information infrastructures – preliminary findings,” Forsvarets forskningsinstitutt, FFI-rapport 2013/01674, 2013.

[10] SISO, *Standard for Real-time Platform Reference Federation Object Model (RPR FOM), Version 2.0*, SISO-STD-001.1-2015, 2015.

[11] M. N. Huhns and M. P. Singh, “Service-oriented computing: Key concepts and principles,” *IEEE Internet Comput.*, vol. 9, no. 1, pp. 75–81, Jan. 2005.

[12] NATO C3B, *Core Enterprise Services Standards Recommendations - The Service Oriented Architecture (SOA) Baseline Profile, Version 1.7*, 2011.

[13] W3C, “SOAP version 1.2,” http://www.w3c.org/TR/soap, 2007, Accessed January 2016.

[14] W3C, “Web Service Architecture,” http://www.w3c.org/TR/ws-arch, 2004, Accessed January 2016.

[15] W3C, “Web Services Glossary,” http://www.w3c.org/TR/ws-gloss, 2004, Accessed January 2016.

[16] C. Pautasso, O. Zimmermann, and F. Leymann, “RESTful web services vs. "big" web services: Making the right architectural decision,” in *Proceedings of the 17th International Conference on World Wide Web*, Beijing, China, Apr. 2008, Conference Proceedings, pp. 805–814.

[17] R. T. Fielding and R. N. Taylor, “Principled design of the modern web architecture,” *ACM Trans. Internet Technol.*, vol. 2, no. 2, pp. 115–150, May 2002.

[18] F. Curbera, M. Duftler, R. Khalaf, W. Nagy, N. Mukhi, and S. Weerawarana, “Unraveling the Web services web,” *IEEE Internet Comput.*, vol. 6, no. 2, pp. 86–93, Mar. 2002.

[19] W3C, “Extensible Markup Language (XML) 1.1 (second edition),” http://www.w3c.org/TR/xml11, 2006, Accessed January 2016.

[20] R. Fielding *et al.*, “Hypertext Transfer Protocol – HTTP/1.1 - Request for Comments: 2616,” http://tools.ietf.org/html/rfc2616, 1999, Accessed March 2016.

[21] W3C, “Web Services Description Language (WSDL) 1.1,” http://www.w3c.org/TR/wsdl, 2001, Accessed January 2016.

[22] S. Vinoski, “REST eye for the SOA guy,” *IEEE Internet Comput.*, vol. 11, no. 1, pp. 82–84, Jan. 2007.

[23] B. Möller and S. Löf, “A management overview of the HLA Evolved Web service API,” in *Proceedings of the 2006 Fall Simulation Interoperability Workshop*, Orlando, Florida, Sep. 2006, no. 06F-SIW-024.

[24] B. Möller and C. Dahlin, “A first look at the HLA Evolved Web service API,” in *Proceedings of the 2006 European Simulation Interoperability Workshop*, Stockholm, Sweden, Jun. 2006, no. 06E-SIW-061.

[25] G. A. Wainer, *Discrete-event modeling and simulation: a practitioner’s approach*. Boca Raton, Florida: CRC Press, 2009.

[26] T. G. Kim *et al.*, “DEVSim++ toolset for defense modeling and simulation and interoperation,” *J. Defense Modeling and Simulation: Applicat., Methodology, Technol.*, vol. 8, no. 3, pp. 129–142, Jul. 2011.

[27] K. Al-Zoubi and G. Wainer, “Distributed simulation using RESTful Interoperability Simulation Environment (RISE) middleware,” in *Intelligence-Based Systems Engineering*, A. Tolk and L. Jain, Eds. Berlin Heidelberg: Springer-Verlag, 2011, ch. 6, pp. 129–157.

[28] E. Mancini, G. Wainer, K. Al-Zoubi, and O. Dalle, “Simulation in the cloud using handheld devices,” in *2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, Ottawa, Canada, May 2012, pp. 867–872.

[29] J. Ribault and G. Wainer, “Simulation processes in the cloud for emergency planning,” in *2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, Ottawa, Canada, May 2012, pp. 886–891.

[30] A. Jeffery, J. Panke, N. Eaket, and G. Wainer, “Mobile simulation with applications for serious gaming,” in *Proceedings of the 2013 Summer Computer Simulation Conference*, Toronto, Ontario, Canada, Jul. 2013, pp. 27:1–27:8.

[31] S. Wang and G. Wainer, “A simulation as a service methodology with application for crowd modeling, simulation and visualization,” *Simulation*, vol. 91, no. 1, pp. 71–95, Jan. 2015.

[32] K. Al-Zoubi and G. Wainer, “Distributed simulation of DEVS and Cell-DEVS models using the RISE middleware,” *Simulation Modelling Practice and Theory*, vol. 55, pp. 27–45, Jun. 2015.

[33] E. Hosang and G. A. Wainer, “An architecture to facilitate interoperability of Discrete Event System Specification and Coalition Battle Management Language simulation

models,” *J. Defense Modeling and Simulation: Applicat., Methodology, Technol.*, vol. 13, no. 1, pp. 43–65, Jan. 2016.

[34] SISO. (2016) WebLVC PDG - Web Live, Virtual, Constructive. [Online]. Available: http://www.sisostds.org/StandardsActivites/DevelopmentGroups/WebLVCPDG.aspx

[35] SISO, *Product Nomination for WebLVC Protocol*, SISO-PN-012-2014, 2014.

[36] VT MÄK. (2016) Web technology for live, virtual and constructive simulation. [Online]. Available: http://www.mak.com/products/web-mobile

[37] L. Granowetter, “The WebLVC protocol: Design and rationale,” in *Proceedings of Interservice/Industry Training, Simulation & Education Conference (I/ITSEC)*, Orlando, Florida, Dec. 2013, no. 13163.

[38] D. Crockford, *JavaScript: The Good Parts*. Sebastopol, California: O’Reilly Media, 2008.

[39] K. Ford, C. Schröeder, L. Simpson, N. Smith, and C. Struselis, “Practical experiences of using the WebLVC standard to support a common GUI framework for controlling disparate CGFs,” in *Proceedings of the 2013 Fall Simulation Interoperability Workshop*, Orlando, Florida, Sep. 2013, no. 13F-SIW-014.

[40] SISO WebLVC Product Development Group, *WebLVC Draft Protocol Specification Version 0.4*, 2015.

[41] A. Tolk, “Challenges of distributed simulation,” in *Engineering Principles of Combat Modeling and Distributed Simulation*. Hoboken, New Jersey: John Wiley & Sons, Inc., 2012, ch. 11, pp. 187 – 208.

[42] I. Fette and A. Melnikov, “The WebSocket protocol - Request for Comments: 6455,” http://tools.ietf.org/html/rfc6455, 2011, Accessed January 2016.

[43] IETF, “Transmission Control Protocol - Darpa internet program protocol specification,” http://tools.ietf.org/html/rfc793, 1981, Accessed January 2016.

[44] JavaScript Object Notation (JSON). (2015) Introducing JSON. [Online]. Available: http://json.org

[45] K. Snively and B. Dillman. (2015) WebLVC subscriptions. [Online]. Available: http://-discussions.sisostds.org/index.htm?A1=ind1509&L=SAC-PDG-WebLVC#4

[46] A. Alstad, “HlaLib v3.0 - user guide,” Forsvarets forskningsinstitutt, FFI-notat 2010/0871, 2010.

[47] Eclipse. (2016) Jetty. [Online]. Available: http://www.eclipse.org/jetty

[48] FasterXML. (2016) Jackson. [Online]. Available: http://github.com/FasterXML/jackson

[49] Red Hat, "JSR 349: Bean Validation 1.1," http://www.jcp.org/en/jsr/detail?id=349, 2009, Accessed February 2016.

[50] M. L. Asprusten, "Management of Execution and Simulation Time Rate Organiser (MAESTRO)," Forsvarets forskningsinstitutt, FFI-notat 2015/00865, 2015.

[51] Sun Microsystems Inc., "JSR-222: Java Architecture for XML Binding (JAXB) 2.0," http://jcp.org/en/jsr/detail?id=222, 2009, Accessed February 2016.

[52] VT MÄK. (2015) VR-Forces. [Online]. Available: http://www.mak.com/products/-simulate/vr-forces

[53] VT MÄK. (2015) MÄK RTI. [Online]. Available: http://www.mak.com/products/link/-mak-rti

[54] S. Seshadri and B. Green, *AngularJS: Up & Running*. Sebastopol, California: O'Reilly Media, 2014.

[55] G. K. Svendsen, K. M. Fauske, and S. Bruvoll, "Funksjonalitet i SWAP," Forsvarets forskningsinstitutt, FFI-rapport 16/00531, 2016.

[56] V. Agafonkin and OpenStreetMap Contributors. (2016) Leaflet. [Online]. Available: http://leafletjs.com/

[57] NSA, *STANAG 2019 – NATO Joint Military Symbology APP-6(C)*, NATO, (NATO Unclassified), 2011.

[58] M. Beckman. (2016) milsymbol.js. [Online]. Available: http://spatialillusions.com/-milsymbol/

[59] V. B. Kvernelv, A. Skjeltorp, M. L. Asprusten, R. A. Løvlid, and T. K. Thoresen, "Øvelse simulatorføderasjon, Rena leir, oktober 2015," Forsvarets forskningsinstitutt, FFI-rapport 2015/02480 (Begrenset), 2016.

[60] Å. Hjulstad, "Simuleringsrammeverk for datagenererte styrker – erfaringer med og tilpasninger i et COTS-produkt: VR-Forces," Forsvarets forskningsinstitutt, FFI-notat 2005/01788 (Unntatt offentlighet), 2005.

[61] G. Combs *et al.* (2016) Wireshark. [Online]. Available: http://www.wireshark.org

[62] SISO, *Standard for Guidance, Rationale & Interoperability Modalities (GRIM) for the Real-time Platform Reference Federation Object Model (RPR FOM), Version 2.0*, SISO-STD-001-2015, 2015.

Forkortelser

API	Application Programming Interface
APP	Allied Procedural Publication
CGF	Computer Generated Forces
COTS	Commercial off-the-shelf
DEVS	Discrete Event System Specification
DIS	Distributed Interactive Simulation
ECEF	Earth-Centered, Earth-Fixed
FOM	Federation Object Model
HLA	High Level Architecture
HTML	HyperText Markup Language
HTTP	Hypertext Transfer Protocol
ICoViCS	Improved Control and Visualisation of GCFs
IEEE	Institute of Electrical and Electronics Engineers
IETF	Internet Engineering Task Force
INI	Forsvarets Informasjonsinfrastruktur
I/ITSEC	Interservice/Industry Training, Simulation & Education Conference
JAXB	Java Architecture for XML Binding
JSON	JavaScript Object Notation
kB	Kilobyte
MAESTRO	Management of Execution and Simulation Time Rate Organiser
MVC	Model-View-Controller
NSA	Nato Standardization Agency

PDG	Product Development Group
PDU	Protocol Data Units
REST	REpresentational State Transfer
RISE	RESTful Interoperability Simulation Environment
RPR FOM	Real-time Platform Reference Federation Object Model
RTI	Run-Time Infrastructure
SISO	Simulation Interoperability Standards Organization
SOA	Service Oriented Architecture
SOM	Simulation Object Model
STANAG	Standardization Agreement
SWAP	Simulation-Supported Wargaming for Analysis of Plans
TCP	Transmission Control Protocol
URI	Uniform Resource Identifier
W3C	World Wide Web Consortium
WebLVC	Web Live, Virtual, Constructive
WGS 84	World Geodetic System 84
WS	Web service
WSDL	Web Services Description Language
XML	Extensible Markup Language

Vedlegg

A JSON-fil med oversettingsregler


```
{"symbol":"G-------", "kind":[
  {"domain":[
    {"category":[
      {"subCategory":[{"symbol":"G-EVAI--","id":0},
{"symbol":"G-EVATM-","id":1}, {"symbol":"G-EVATM-","id":2}
      ],"id":1},
      {"subCategory":[{"symbol":"G-EVES--","id":3},{"symbol":"G-
EWO---","id":9}
      ],"id":2},
      {"subCategory":[{"symbol":"G-EVU---","id":0}
      ],"id":8}
    ], "id":1},
    {"symbol":"A-------","category":[],"id":2},
    {"symbol":"S-------", "category":[],"id":3},
    {"symbol":"U-------","category":[],"id":4},
    {"symbol":"P-------","category":[],"id":5}
  ],"id":1},
  {"domain":[
    {"category":[
      {"subCategory":[{"symbol":"G-EWH---","id":14}
      ],"id":2}
    ],"id":9}
  ],"id":2},
  {"domain":[
    {"category":[
      {"subCategory":[{"symbol":"G-EWRR--","id":0}
      ], "id":1}
    ],"id":1}
  ],"id":3},
  {"domain":[
    {"category":[
      {"subCategory":[{"symbol":"G-UCAAO-","id":3},{"symbol":"G-
UCAT--","id":2},{"symbol":"G-UCECW-","id":10},{"symbol":"G-UCFM-
-","id":80},{"symbol":"G-UCIL--","id":98},{"symbol":"G-UCIM--
","id":42},{"symbol":"G-UCIZ--","id":4},{"symbol":"G-UCRVA-
```

```
","id":99},{"symbol":"G-UCRVM-","id":96},{"symbol":"G-UCRVM-
","id":97},{"symbol":"G-UCR---","id":30},{"symbol":"G-UCEC--
","id":54},{"symbol":"G-UAA---","id":57},{"symbol":"G-EWZ---
","id":59},{"symbol":"G-USM---","id":61},{"symbol":"G-USS---
","id":63}
      ], "id":3},
      {"subCategory":[{"symbol":"G-UCF---","id":7},{"symbol":"G-
UCFHE-","id":58},{"symbol":"G-UCDMLA","id":68}
      ],"id":4},
      {"subCategory":[{"symbol":"G-UCAT--","id":2},{"symbol":"G-
UCEC--","id":4}
      ],"id":14},
      {"subCategory":[{"symbol":"G-UCEC--
","id":50},{"symbol":"G-USS---","id":55}
      ],"id":6},
      {"subCategory":[{"symbol":"G-UCIZ--
","id":51},{"symbol":"G-UUA---","id":52},{"symbol":"G-UCR---
","id":53},{"symbol":"G-UAA---","id":56},{"symbol":"G-UCIZ--
","id":60},{"symbol":"G-UCFM--","id":62},{"symbol":"G-UCAT--
","id":64},{"symbol":"G-UCFRM-","id":65}
      ],"id":5},
      {"subCategory":[{"symbol":"S-CLFF--
","id":66},{"symbol":"S-CALS--","id":67}
      ],"id":7}
    ],"id":1}
  ],"id":11}
]}
```

B XML-skjema for oversettingsregler

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema attributeFormDefault="unqualified"
elementFormDefault="qualified"
xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:element name="modelMapping" type="modelMappingType"/>

    <!-- We want the symbol type to only accept six-character
strings, and the second character should always be a *.
    This is because the second character should be exchanged for
a different character later.
    Symbol is defined as its own type because we want all levels
in the DIS-enumeration hierarchy to have a fallback symbol,
    in case a mapping to a lower level in the hierarchy doesn't
exist-->
    <xs:simpleType name="symbolType">
        <xs:restriction base="xs:string">
            <xs:pattern value="[A-Z, \-][\*][A-Z, \-][A-Z, \-
][A-Z, \-][A-Z, \-][A-Z, \-][A-Z, \-]" />
        </xs:restriction>
    </xs:simpleType>

    <xs:complexType name="modelMappingType">
        <xs:sequence>
            <xs:element name="symbol" type="symbolType"
minOccurs="1" maxOccurs="1" />
            <xs:element name="kind" type="kindType"
minOccurs="0" maxOccurs="unbounded" />
        </xs:sequence>
    </xs:complexType>

    <xs:complexType name="kindType">
        <xs:sequence>
            <xs:element name="symbol" type="symbolType"
minOccurs="0" maxOccurs="1" />
            <xs:element name="domain" type="domainType"
minOccurs="0" maxOccurs="unbounded" />
        </xs:sequence>
        <xs:attribute name="id" type="xs:int" use="required" />
    </xs:complexType>

    <xs:complexType name="domainType">
```

```
        <xs:sequence>
            <xs:element name="symbol" type="symbolType"
minOccurs="0" maxOccurs="1" />
            <xs:element name="category" type="categoryType"
minOccurs="0" maxOccurs="unbounded" />
        </xs:sequence>
        <xs:attribute name="id" type="xs:int" use="required" />
    </xs:complexType>

    <xs:complexType name="categoryType">
        <xs:sequence>
            <xs:element name="symbol" type="symbolType"
minOccurs="0" maxOccurs="1" />
            <xs:element name="subCategory"
type="subCategoryType" minOccurs="0" maxOccurs="unbounded" />
        </xs:sequence>
        <xs:attribute name="id" type="xs:int" use="required" />
    </xs:complexType>

    <xs:complexType name="subCategoryType">
        <xs:sequence>
            <xs:element name="symbol" type="symbolType"
minOccurs="1" maxOccurs="1" />
        </xs:sequence>
        <xs:attribute name="id" type="xs:int" use="required" />
    </xs:complexType>
</xs:schema>
```

About FFI

The Norwegian Defence Research Establishment (FFI) was founded 11th of April 1946. It is organised as an administrative agency subordinate to the Ministry of Defence.

FFI's MISSION

FFI is the prime institution responsible for defence related research in Norway. Its principal mission is to carry out research and development to meet the requirements of the Armed Forces. FFI has the role of chief adviser to the political and military leadership. In particular, the institute shall focus on aspects of the development in science and technology that can influence our security policy or defence planning.

FFI's VISION

FFI turns knowledge and ideas into an efficient defence.

FFI's CHARACTERISTICS

Creative, daring, broad-minded and responsible.

Om FFI

Forsvarets forskningsinstitutt ble etablert 11. april 1946. Instituttet er organisert som et forvaltningsorgan med særskilte fullmakter underlagt Forsvarsdepartementet.

FFIs FORMÅL

Forsvarets forskningsinstitutt er Forsvarets sentrale forskningsinstitusjon og har som formål å drive forskning og utvikling for Forsvarets behov. Videre er FFI rådgiver overfor Forsvarets strategiske ledelse. Spesielt skal instituttet følge opp trekk ved vitenskapelig og militærteknisk utvikling som kan påvirke forutsetningene for sikkerhetspolitikken eller forsvarsplanleggingen.

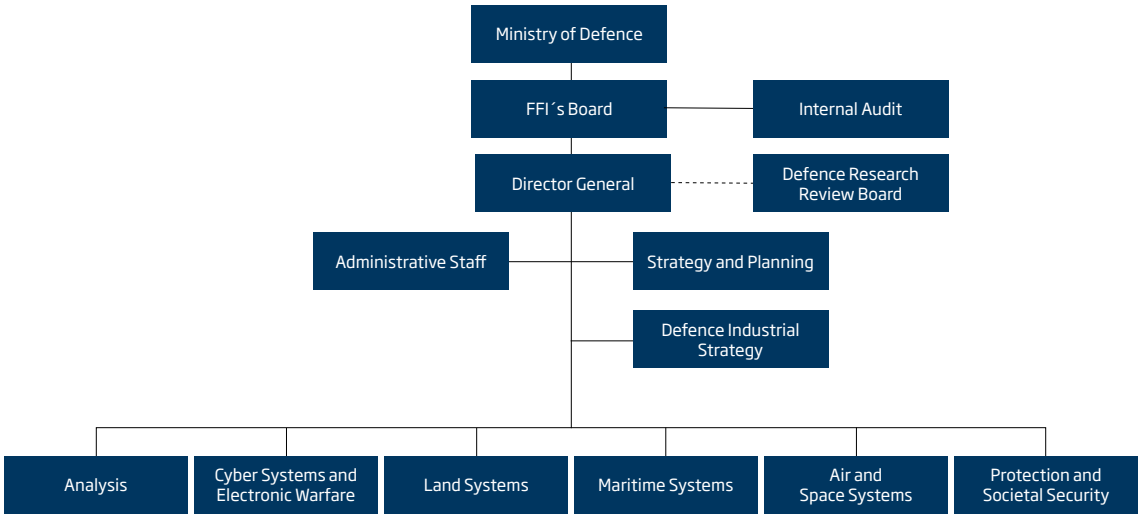
FFIs VISJON

FFI gjør kunnskap og ideer til et effektivt forsvar.

FFIs VERDIER

Skapende, drivende, vidsynt og ansvarlig.

FFI's organisation



Forsvarets forskningsinstitutt
Postboks 25
2027 Kjeller

Besøksadresse:
Instituttveien 20
2007 Kjeller

Telefon: 63 80 70 00
Telefaks: 63 80 71 15
Epost: ffi@ffi.no

Norwegian Defence Research Establishment (FFI)
P.O. Box 25
NO-2027 Kjeller

Office address:
Instituttveien 20
N-2007 Kjeller

Telephone: +47 63 80 70 00
Telefax: +47 63 80 71 15
Email: ffi@ffi.no

FFI Forsvarets forskningsinstitutt
Norwegian Defence Research Establishment