TAMPEREEN TEKNILLINEN YLIOPISTO
Tietotekniikan osasto

HENRI BRAGGE
SHDSL-PÄÄTELAITTEEN WEB-HALLINTASOVELLUS

Aihe hyväksytty osastoneuvoston kokouksessa
12.2.2007

Tarkastaja: Professori Pekka Loula (TTY)

Alkulause

Olen tehnyt diplomityön toimiessani ohjelmistokehittäjänä Design Combus Oy:n palveluksessa Tampereella. Työ käsittelee graafisen käyttöliittymän luomista yrityksen tuoteprojektiin ja sen tarkoitus on helpottaa tuotteen käyttöä. Kiitän esimiestäni Markku Lindelliä projektin toimeksiannosta, työn ohjaajaa Petri Ahosta asiantuntevasta opastuksesta ja työn tarkastajaa professori Pekka Loulaa arvokkaista neuvoista.

Tampereella 1.4.2007

Henri Bragge

Itsenäisyydenkatu 7-9 A 25

33100 Tampere

Puh: +358 50 3086784

E-mail: henri.bragge@gmail.com

Sisällysluettelo

Tiivistelmä

TAMPEREEN TEKNILLINEN YLIOPISTO

Tietotekniikan koulutusohjelma

Tietotekniikka (Pori)

BRAGGE, HENRI: SHDSL-päätelaitteen web-hallintasovellus

Diplomityö, 82 sivua, 18 liitettä

Tarkastaja: Professori Pekka Loula

Rahoittaja: Design Combus Oy

Huhtikuu 2007

Avainsanat: HTTP, palvelin, SHDSL, web

UDK: 004.455.2

Verkon päätelaitteet, kuten reitittimet, kytkimet ja modeemit ovat tietoverkkoja yhdistäviä laitteita, joiden ominaisuuksiin kuuluu pääasiallisesti verkkoliikenteen ohjaaminen, sovittaminen verkosta toiseen ja kommunikointi muiden päätelaitteiden kanssa. Päätelaitteet vaativat käyttöönoton yhteydessä laitteen haltijalta tai ylläpitäjältä yleensä omaan käyttöön sopivien asetusten syöttämistä. Myös ulkoisten olosuhteiden muutos voi luoda tarpeen laitteen asetusten muuttamiseen tai esimerkiksi laitteen ohjelmiston päivittämiseen. Päätelaitteen tarjoamat mahdollisuudet sen asetusten hallintaan riippuvat sen sovellusympäristöstä, mutta yleisimmin käytetään komentorivipohjaista tai web-pohjaista hallintasovellusta tai näitä kahta rinnakkain.

Diplomityössä käsitellään SHDSL (*Symmetric High-speed Digital Subscriber Line*) –päätelaitteeseen toteutettavan web-pohjaisen hallintasovelluksen kehitystä ja kehityksessä esiintyneitä haasteita. Työssä keskitytään erityisesti suunnittelussa esiintyneiden ongelmien taustateorioiden selvittämiseen, joista tärkeimmät ovat HTTP-palvelinsovelluksen toiminta sekä kohdeympäristön laitteiston ja sovellusympäristön ominaisuudet. Työssä käsitellään myös kehitetyn hallintasovelluksen rakennetta, toimintaa ja jatkokehitysmahdollisuuksia. Ennen sovelluksen toteutusta on syytä käydä läpi palvelinta koskeva taustateoria, toteutuksen kohdeympäristö ja tutkia jo toteutettuja sovelluskehyksiä, hallintasovelluksia ja web-palvelimia. Näin voidaan hyödyntää jo olemassa olevaa tietämystä ja varmistaa sovelluksen ajanmukaisuus ja soveltuvuus kohdeympäristöön.

Sovelluksen laatiminen osoittautui haastavaksi, koska sille asetetut vaatimukset monipuolisuuden ja yleiskäyttöisen rakenteen suhteen ja samalla kohdeympäristön asettamat suorituskykyvaatimukset vaativat sovelluksen rakenteen huolellista suunnittelua. Käytännössä lopullista sovellusta edelsi useita prototyyppejä, joista muotoutui hiljalleen halutun kaltainen ohjelmisto.

Abstract

TAMPERE UNIVERSITY OF TECHNOLOGY

Department of Information Technology

Information Technology (Pori)

BRAGGE, HENRI: Web management software for an SHDSL terminal device

Master of Science Thesis, 82 pages, 18 enclosure pages

Examiner: Professor Pekka Loula

Funding: Design Combus Ltd

April 2007

Keywords: HTTP, server, SHDSL, web

UDK: 004.455.2

Network terminal devices, like routers, switches and modems are devices that interconnect computer networks. Their main features include controlling and adapting the network traffic and communication to other network devices. Terminal devices often require management from the owner or the administrator of the device. Management includes tasks like configuring the initial settings, configuring settings to adapt to changes in the environment or upgrading the device's software. Terminal devices offer different management possibilities, depending on the application environment of the device. Most devices offer terminal or Web-based management applications or both of them.

This thesis describes the developing of Web-based management software for an SHDSL (*Symmetric High-speed Digital Subscriber Line*) terminal device. The main emphasis is on the theory behind the problems confronted, the most important of which are the operation of HTTP server software and the software and hardware environments of the target device. The thesis also describes the structure, operation and further concerns of the management software. Before the implementation it is essential to study Web server theory and the features of the target environment and to consider frameworks, management software and Web servers created before. Thus it is possible to utilize existing knowledge and to ensure that the implemented application is modern and appropriate.

The implementation phase was challenging and demanded careful planning because of the requirements for versatile and general-purpose software and the limitations of the target environment. The final product was preceded by numerous different prototypes, from which the application slowly took its final shape.

Käytetyt lyhenteet

ADSL	Asymmetric Digital Subscriber Line
AMPED	Asymmetric Multi-process Event-driven
API	Application Programming Interface
ASP	Active Server Pages
BSD	Berkeley Software Distribution
CGI	Common Gateway Interface
CLI	Command Line Interface
CPU	Central Processing Unit
CSS	Cascading Style Sheets
CVS	Concurrent Versions System
DDR	Double Data Rate
DES	Data Encryption Standard
DMA	Direct Memory Access
DNS	Domain Name System
DSLAM	Digital Subscriber Line Access Multiplexer
DoS	Denial-of-Service
EEPROM	Electrically Erasable Programmable Read Only Memory
FIFO	First In, First Out
FPGA	Field-programmable gate Array
GPL	GNU Public License
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
I2C	Inter-Integrated Circuit
I/O	Input/Output
IETF	Internet Engineering Task Force
IP	Internet Protocol
IPC	Inter-process Communication
ISAPI	Internet Server Application Programming Interface
J2EE	Java 2 Platform Enterprise Edition
JTAG	Joint Test Action Group
LED	Light-emitting Diode
MD5	Message-Digest algorithm 5
MFC	Microsoft Foundation Classes
MII	Media-independent Interface
MIPS	Microprocessor without Interlocked Pipeline Stages
MIT	Massachusetts Institute of Technology
MVC	Model-View-Control
PCI	Peripheral Component Interconnect
PHP	PHP: Hypertext Preprocessor
PKI	Public Key Infrastructure
RAM	Random Access Memory
RC2	Rivest Cipher 2
RC4	Rivest Cipher 4
RFC	Request for Comments
RJ45	Registered Jack 45
RJ6/6	Registered Jack 6/6

ROM	Read-only Memory
RSA	Rivest, Shamir, Adleman algorithm
RSS	Really Simple Syndication
SAPI	Server Application Programming Interface
SHA	Secure Hash Algorithm
SHDSL	Symmetric High-speed Digital Subscriber Line
SNMP	Simple Network Management Protocol
SPED	Single-process Event-driven
SPI	Serial Peripheral Interface
SQL	Structured Query Language
SSL	Secure Sockets Layer
STDIN	Standard Input
STL	Standard Template Library
SVN	Subversion
TCP	Transfer Control Protocol
TLS	Transport Layer Security
UART	Universal Asynchronous Receiver/Transmitter
UDP	User Datagram Protocol
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
VLAN	Virtual Local Area Network
W3C	World Wide Web Consortium
WWW	World Wide Web
XSL	Extensible Stylesheet Language

1 Johdanto

Verkon päätelaitteen hallintamahdollisuus ovat välttämätön osa laitetta, koska laitteet toimivat vain harvoin niihin valmiiksi tallennetuilla oletusasetuksilla. Yrityksille suunnitellut päätelaitteet, kuten tämän projektin kohdeympäristönä oleva SHDSL-päätelaite, eroavat esimerkiksi yksityisasiakkaille suunnatuista ADSL (*Asymmetric Digital Subscriber Line*) –modeemeista siten, että niiden hallinnasta vastaava kohderyhmä voidaan yleensä olettaa olevan yksityisasiakasta teknisesti valveutuneempi. Tällöin niin kutsuttujen velho-ominaisuuksien lisääminen ja ylimääräisten teknisten tietojen piilottaminen voidaan jättää vähemmälle huomiolle. Se ei kuitenkaan vähennä hyvän käytettävyyden merkitystä. Päinvastoin, alansa asiantuntijaa käytettävyysongelmat saattavat häiritä enemmän kuin satunnaista, vain harvoin laitteen ominaisuuksia käsittelevää käyttäjää. Lisäksi opittavuus ja muistettavuus ovat tärkeitä ominaisuuksia käyttäjän tasosta riippumatta.

Käytettävyyden lisäksi sulautetun laitteen verkko-ohjelmiston suunnitelussa on otettava huomioon kaksi tärkeää seikkaa: kohdeympäristön kriittisyys ja sovelluksen joustava suunnittelu. Sulautetun laitteen muistikapasiteetti ja laskentateho on rajoittunut, joten ohjelman on oltava kevyt ja huomioitava myös laitteella olevat muut, ehkäpä sitä tärkeämmät prosessit. Se ei saa myöskään virheen sattuessa lopettaa toimintaansa eikä missään tilanteessa olla vaaraksi laitteen toimintakyvylle. Ohjelman on syytä olla joustava, koska tulevaisuuden tuoteprojektien tullessa ajankohtaiseksi on todennäköistä, että vastaavan kaltaista sovellusta tarvitaan myös niissä.

Työssä perehdytään edellä mainittuihin suunnitteluperusteisiin ja niiden taustalla olevaan teoriaan, joiden perusteella luodaan työssä esiteltävä web-hallintasovellus. Sovelluksen lopullisena tavoitteena on olla helppokäyttöinen, uudelleenkäytettävä ja hyvin kohdeympäristöönsä sulautuva. Sovelluksesta ei tule Iris 440 SHDSL-päätelaitteen hallintamahdollisuuksista ainoa, vaan sen rinnalla toimii myös aikaisemmin toteutettu komentorivipohjainen CLI (*Command Line Interface*) –hallintasovellus. Lisäksi laitteen ominaisuuksia voidaan tarkastella SNMP (*Simple Network Management Protocol*) –managerin avulla. Web-hallintasovelluksen on tarkoitus toimia näiden rinnalla, tarjoten käyttäjälle graafisen ja helppokäyttöisen vaihtoehdon laitteen hallintaan.

Luku kaksi käsittelee HTTP-palvelimeen liittyvää teoriaa: itse protokollaa, prosessien välistä kommunikaatiota, dynaamisen sisällön luomista, prosessimalleja, tietoturvaa ja sovellusarkkitehtuureja. Luku kattaa projektissa tarvittavan palvelinsovelluksia koskevan taustateorian. Luvussa käsiteltävä teoria on tärkeää, jotta voidaan ymmärtää toteutettavan sovelluksen toimintaperiaatteet ja sen kommunikointimenetelmät käyttäjän ja muun ympäristön kanssa.

Luvussa kolme tarkastellaan projektin kohdeympäristöä, joka on yrityskäyttöön suunniteltu Iris 440 SHDSL-päätelaite. Laitteisto- ja suorituskykyseikkojen lisäksi paneudutaan laitteen käyttöjärjestelmään ja sovellusympäristöön sekä jo toteutettuun, web-hallintasovelluksen rinnalla toimivaan CLI-hallintasovellukseen. Uutta sovellusta toteutettaessa on syytä tarkastella edellisen ratkaisun heikkouksia ja vahvuuksia ja

mahdollisuuksien mukaan uudelleenkäyttää sille luotua ohjelmakoodia. Sovelluksen kehitysympäristöä koskien tutustutaan käytettäviin käännöstyökaluihin ja versionhallintamenetelmiin.

Luku neljä esittelee projektin esitutkimusvaiheessa suoritetun palvelinkartoituksen. Siinä tarkastellaan joukkoa HTTP-palvelinsovelluksia, jotka voisivat soveltua hallintasovelluksen ytimenä toimivan palvelimen tehtävään. Palvelinsovellukset jaotellaan määrättyjen jaotteluperusteiden mukaisesti, ja niiden joukosta valitaan tähän tarkoitukseen parhaiten sopivin vaihtoehto. Jaotteluperusteiden pohjalla hyödynnetään edeltävissä luvuissa muodostunutta käsitystä palvelinsovelluksen vaatimuksista sen tarjoamien ominaisuuksien ja toimintatavan suhteen.

Luku viisi käsittelee hallintasovelluksen suunnittelua ja toteutusta. Luvun pääpaino on sovelluksen näkymien suunnittelussa ja sisäisessä toiminnassa. Sovelluksen toteutusta ei käsitellä vaihe vaiheelta, vaan keskitytään lähinnä lopputuloksen tarkasteluun sopivasti rajattuna. Luku etenee työjärjestystä vastaavalla tavalla käsitellen ensin käyttöliittymän ja sen toimintojen suunnittelua, sen jälkeen verkko-ohjelmiston luokka-arkkitehtuuria ja luokkien sisäistä toimintaa, ja lopuksi palvelimen vasteen muodostusta käytännössä.

Luvut kuusi ja seitsemän sisältävät lopputuloksen arviointia sen loppukäyttöön soveltuvuuden ja jatkokehitysmahdollisuuksien kannalta. Erityisesti pohditaan lopputuloksen siirrettävyyttä mahdollisia tulevia tuoteprojekteja ajatellen.

2 HTTP-palvelinsovellus

HTTP-palvelinsovellus on sovellus, joka kuuntelee palvelimen HTTP-porttia (yleensä portti 80) ja vastaa siihen tuleviin pyyntöihin. Ennen vasteen lähettämistä tapahtuu pyynnön vastaanottamisen jälkeen useita eri vaiheita, joita ovat pyynnön jäsentäminen, käsittely, tarvittavan datan haku palvelimelta ja vasteen muodostaminen. Tässä luvussa käsitellään näitä vaiheita ja muita palvelinsovelluksen toimintaan liittyviä seikkoja. Luvussa kerrotaan ensin kommunikointiin käytettävästä HTTP-protokollasta ja sen jälkeen palvelimen toiminnasta käyttöjärjestelmän näkökulmasta. Lisäksi käsitellään palvelinsovelluksen tietoturvaa ja lopuksi verkko-ohjelmiston suunnittelussa käytettäviä sovellusarkkitehtuureita.

2.1 World Wide Web

World Wide Web näki päivänvalon keväällä 1989, jolloin Tim Berners-Lee julkaisi ehdotuksensa uudesta tavasta esittää ja yhdistää tietoa. Tämä aloitti digitaalisessa tiedonsiirrossa uuden aikakauden, joka resurssien yhtenäisen tunnistamisen ja jakamisen sekä hypertekstin avulla mahdollisti äänen, kuvan, tekstin ja muun tiedon jakamisen graafisilla selaimilla. Välitettävän tiedon monipuolisuus loi nopeasti tarpeen myös tiedon dynaamista luomista ja käsittelyä varten. Nykyään se mahdollistaa interaktiiviset sovellukset WWW-palvelimilla, joiden käyttö selaimella internetin yli on yhtä helppoa ja nopeaa kuin perinteisillä sovelluksilla paikallisesti.

2.1.1 HTTP-protokolla

HTTP on WWW:ssä käytetty tiedonsiirtoprotokolla, jonka avulla asiakas ja palvelin neuvottelevat keskenään. HTTP:n kehitti W3C (*World Wide Web Consortium*) ja IETF (*Internet Engineering Task Force*), ja siitä käytetään tällä hetkellä yleisimmin versiota HTTP/1.1, jonka määrittelee RFC 2616 [Ber99]. Eri versiot ovat taaksepäin yhteensopivia, ja versio 1.1 sisältää parannuksia, jotka tehostavat HTTP-liikennettä ja tarjoavat paremman kontrollin WWW-sivujen ulkonäköön [Pus01, s. 127]. HTTP toimii sovelluskerroksella, ja sen alapuolella käytetään yleensä TCP-protokollaa, joka tarjoaa HTTP:lle luotettavan ja yhteydellisen tiedonsiirron. Tämä alikohta perustuu lähteisiin [Ber99] ja [Pel98, s. 19-21].

Pyyntö

HTTP määrittelee joukon metodeja ja näiden vastauskoodeja, joita käytetään käyttäjän ja palvelimen väliseen kommunikointiin. Metodien avulla käyttäjä ilmaisee palvelimelle mitä halutaan tehdä ja sen yhteydessä annetaan yleensä myös kohteen URI. Palvelimen lähettämään vastaukseen puolestaan liittyy aina vastauskoodi, joka kertoo vasteen tilan.

Yksi yleisimpiä HTTP-metodeja on GET, jota käytetään datan noutamiseen määritellystä URI:sta. URI:n lisäksi pyyntöön liitty yleensä otsikkokenttiä, jotka voivat sisältävää tietoa protokollaversiosta tai asiakasohjelmistosta. Esimerkki GET-pyynnön muodosta on esitelty kuvassa 2-1. Kuvan ensimmäisellä rivillä on ensin haluttu metodi, sitten pyydettävä URI ja lopuksi käytetty HTTP-versio. Toisella rivillä on pyynnön

kohde, jonka tarkoitus on erottaa toisistaan samaan IP-osoitteeseen viittaavat DNS-nimet. Vastaavassa muodossa olevia nimi-arvo-pareja voi olla otsikossa myös enemmän.

```
GET /rfc/rfc2616.txt HTTP/1.1
Host: www.ietf.org
```

Kuva 2-1. GET-pyyntö.

GET–metodin lisäksi muita yleisiä metodeja ovat POST ja HEAD. RFC 2616:n määrittelemät metodit näkyvät selitteineen taulukossa 2-1. Selitteestä selviää metodin mahdollisesti tarvitsema lisämääre, joka on tyypillisesti kohteen URI.

Taulukko 2-1. HTTP-metodit.

Metodi	Selite
OPTIONS	Määritetyn URI:in kommunikaatioasetusten haku.
GET	URI:lla identifioidun tiedon haku.
HEAD	Sama kuin GET, mutta haetaan vain otsikkokentät.
POST	Tiedon lähetys URI:lla identifioituun kohteeseen.
PUT	Tiedon tallentaminen URI:lla identifioituun kohteeseen.
DELETE	Poista URI:lla identifioitu kohde.
TRACE	Pyynnön heijastaminen takaisin lähettäjälle.
CONNECT	Dynaamisen tunneloinnin käyttöön varattu metodi

Vaste

Kun HTTP-palvelin vastaanottaa pyynnön, se palauttaa käyttäjälle pyydetyn URI-osoitteen ilmoittaman objektin, eli kuvan 2-1 esimerkissä rfc2616.txt-tiedoston. Koska HTTP ei määrittele lähetettävää sisältöä, se voi olla mitä tahansa digitaalisessa muodossa esitettyä selkokielistä tai binäärimuotoista dataa. Kuvassa 2-2 on esimerkki palvelimen lähettämästä vasteesta. Ensimmäinen rivi kertoo pyynnön tapaan käytettävän protokollaversion, jonka jälkeen on vastauskoodi eli tässä tapauksessa 200. Lisäksi vasteessa näkyy päivämäärä- ja palvelintietoja, sekä tietoa sisällön muodosta ja koosta.

```
HTTP/1.1 200 OK
Date: Tue, 07 Nov 2006 17:31:44 GMT
Server: Apache/2.0.52 (Red Hat)
Last-Modified: Fri, 28 Jul 2006 14:17:09 GMT
Etag: "a963bb-27a-f7c42340"
Accept-Ranges: bytes
Content-Length: 634
Connection: close
Content-Type: text/plain; charset=UTF-8
```

Kuva 2-2. GET-vaste.

Vastauskoodit ovat lajiteltu merkityksensä mukaan samoihin satalukuihin kuuluviin ”tilaperheisiin”. Informatiiviset tilat kuuluvat ryhmään 100, onnistumista ilmaisevat ryhmään 200, uudelleenohjausta ryhmään 300, käyttäjän virhettä ryhmään 400 ja palvelimen virhettä ilmaisevat tilat ryhmään 500. Yleisimpiä vastauskoodeja ovat muun muassa *404 – Not Found, 500 – Internal Server Error* ja *503 – Service Unavailable.*

2.1.2 Dynaaminen web

Pelkästään staattisen tiedon esittäminen WWW-sivuilla voi olla sellaisenaan informatiivista, mutta usein tarvitaan jonkinlaisen interaktiivisuuden tarjoamista käyttäjälle. Se voi tarkoittaa erilaisia painikkeita, lomakkeita ja muita toimintoja, joiden avulla käyttäjä voi vaikuttaa sivun tarjoamaan sisältöön muutoinkin kuin vain yksiselitteisten tilasiirtymien avulla. Tämän toteuttamiseksi tarvitaan jokin logiikka, joka vastaanottaa käyttäjän antamat syötteet ja luo tämän perusteella sille vasteen. Tämä alikohta käsittelee muutamia keinoja palvelimen logiikan toteuttamiseen, ja perustuu lähteisiin [Pel98, s. 478 ja 660-661].

Common Gateway Interface (CGI)

CGI [CoRo99] on palvelimen ja palvelimella suoritettavien ohjelmien välille määritelty tiedonsiirtorajapinta. CGI mahdollistaa dynaamisen sisällön muodostamisen palvelimen käyttöön rajapinnan kautta palvelimella ajettavien sovellusten avulla. Käyttäjälle lähetettävälle sivulle voidaan koota tietoa esimerkiksi skriptikieliä apuna käyttäen. CGI:n etuja ovat yksinkertaisuus, erilliset prosessit ja riippumattomuus palvelimen ohjelmointikielestä ja arkkitehtuurista. Käytännössä tiedon välittäminen CGI-rajapinnan kautta onnistuu erilaisilla ympäristömuuttujilla ja standard-IO-datavirralla.

Vaikka erilliset prosessit voidaan toisaalta laskea eduksi, on niillä myös haittapuolensa. CGI käynnistää jokaiselle sille tarkoitetulle pyynnölle oman prosessinsa, joka vastaa pyynnön käsittelystä ja joka suljetaan käsittelyn päätyttyä. Tämä menettely kuluttaa palvelimen muistia ja vaikuttaa myös suorituskykyyn johtuen jokaiseen pyyntöön liittyvästä prosessin käynnistämisestä ja sulkemisesta. Monta samanaikaista prosessia voi aiheuttaa myös poissulkemis- ja synkronointiongelmia, jos eri prosessit käsittelevät saman muistialueen muuttujia tai yhteistä tietolähdettä.

Koska CGI on riippumaton ohjelmointikielestä, voidaan CGI-sovelluksia toteuttaa millä tahansa kielellä, kunhan palvelimen järjestelmäalusta sitä vain tukee. Tavallisimmin CGI-sovellus on toteutettu jollakin tulkattavalla kielellä, kuten Perl, Python tai Ruby.

Myös käännettävillä kielillä toteuttaminen onnistuu, jolloin voidaan käyttää esimerkiksi kieliä C, C++ tai Java.

FastCGI

CGI:n korvaajaksi kehitetty FastCGI pyrkii vastaamaan edeltäjänsä ongelmiin etenkin muistinkäytön suhteen. Se säilyttää kuitenkin CGI:n hyvät puolet eli yksinkertaisuuden ja riippumattomuuden ohjelmointikielestä. Toisin kuin CGI:ssä, FastCGI:ssä prosessit ovat pysyviä, joten yhden pyynnön käsiteltyään ne jäävät odottamaan seuraavaa [OM96]. FastCGI pyrkii myös vähentämään pyyntöjen käsittelyyn tarvittavien prosessien määrää suorittamalla aikaa vaativat I/O-operaatiot palvelinsovelluksella. FastCGI yhdistää tiedon välittämiseen tarvittavat ympäristömuuttujat ja standard-IO-datavirran yhdeksi datavirraksi, jota voidaan välittää paikallisille prosesseille Unix-soketeilla ja etäkoneilla toimiville prosesseille TCP-protokollalla.

API (Application Programming Interface) –rajapinnat

Korjatakseen muiden sovellusrajapintojen ongelmia, monet palvelimet tarjoavat myös omia rajapintojaan sovelluksien toteuttamiseseen. Tunnetuimpia ovat Microsoftin ISAPI (*Internet Server API*) [MS95] ja Apache API [ASF05]. Myös luvussa neljä esiteltävät GoAhead WebServer [GA00], Klone [KL07] ja Seminole [GS06a] tarjoavat sovellusohjelmointiin omat rajapintansa.

API-sovellusten tärkein etu on nopeus, sillä palvelinsovellukseen linkitetyt ohjelmat toimivat huomattavasti nopeammin kuin esimerkiksi CGI-ohjelmat, koska ohjelma toimii palvelinprosessin yhteydessä ja sitä ei käynnistetä ja suljeta jokaiselle pyynnölle erikseen. Palvelin-API:t tarjoavat usein myös laajemman toiminnallisuuden kuin CGI-rajapinta, koska pyynnön käsittelyn aikana on mahdollista vaikuttaa myös itse palvelimen toimintaan.

Palvelimien tarjoamien API:en haittapuolina on usein ohjelmointirajapinnan monimutkaisuus ja sen rajoittuneisuus tietylle ohjelmointikielelle ja arkkitehtuurille. Koska rajapinnat noudattavat tyypillisesti itse palvelimen toteutuskieltä, on ne useimmin toteutettu C- tai C++-kielellä. Tällöin sovellusohjelmoijalla ei ole toteutuskielen suhteen paljoa valinnanvaraa. Monimutkaisuutta lisää myös API-toteutusten epäyhteensopivuus, koska toteutukset poikkeavat usein laajalti toisistaan. Tämä voi aiheuttaa verkko-ohjelmiston siirrettävyysongelmia, mikäli sen toteutus on vahvasti sidoksissa käytettävän palvelinsovelluksen rajapintaan. Lisäksi ohjelmointirajapinnat ovat vain harvoin avoimia tai edes ilmaisia, kun taas CGI-maailmassa avoimien vaihtoehtojen määrä on lukematon.

2.2 Palvelimen toiminta

Erilaiset HTTP-palvelimet käyttävät palvelun tarjoamiseen erilaisia menetelmiä, mutta eri vaiheiden perusperiaatteet pyynnön vastaanottamisesta vasteen lähettämiseen ovat yleensä samankaltaisia. Unix-pohjaiset HTTP-palvelimet käyttävät pääsääntöisesti hyödykseen Unixin tarjoamia perusmenetelmiä prosessien väliseen kommunikaatioon sekä verkon yli tapahtuvaan kommunikaatioon. Näitä menetelmiä voivat käyttää toiminnot kuten tiedoston luku ja kirjoitus tai verkkokommunikaatio sokettiyhteyden

avulla. Tässä kohdassa käsitellään prosessien välistä kommunikaatiota ja soketteja sekä niiden merkitystä HTTP-palvelimissa. Lisäksi esitellään lyhyesti HTTP-pyyntöön vastaaminen Unixin järjestelmäkutsujen avulla.

2.2.1 Prosessien välinen kommunikaatio

Prosessien yleisin menetelmä keskinäiseen kommunikaatioon on putkien käyttäminen. Putki tarkoittaa yksisuuntaista kommunikointikanavaa, joka mahdollistaa tavuvirran välittämisen kahden prosessin välillä. Putket toimivat FIFO-periaatteella, eli ulos tuleva data luetaan samassa järjestyksessä kuin se on lähetetty. Putken läpi lähetettävä datavirta ei noudata mitään protokollaa, vaan kommunikointiin käytettävä protokolla on määriteltävä itse. Järjestelmä huolehti datan välittämisen luotettavuudesta, eli siitä että dataa ei häviä siirrettäessä. [Ker02a]

Putki luodaan käyttämällä Unixin `pipe()`-järjestelmäkutsua. Tämän avulla saadaan kaksi tiedostokuvainta, joista toista voidaan käyttää putken lukemiseen ja toista kirjoittamiseen. Näistä voidaan kuitenkin käyttää kerrallaan vain toista, jolloin yhden prosessin on tarkoitus käyttää putken lukemiseen ja toisen siihen kirjoittamiseen tarkoitettua kuvainta.

Koska putket ovat yksisuuntaisia, tarvitaan kaksisuuntaiseen kommunikointiin kaksi putkea. Tämä tarkoittaa käytännössä sitä, että kahden prosessin välisessä kommunikaatiossa yksi prosessi hyödyntää kahta putkea, joista yhtä se käyttää tiedon lähettämiseen ja toista vastaanottamiseen. Siirtosuunnan määrittäminen tapahtuu siis suuntaa vastaavaa tiedostokuvainta käyttämällä. Kun vastapuolena oleva prosessi käyttää putkia vastaavalla tavalla, voidaan niillä mahdollistaa kaksisuuntainen kommunikaatio prosessien välillä.

Kaksisuuntaisen kommunikaation toteuttaminen putkien avulla vaatii erityistä tarkkuutta, koska ne joutuvat helposti lukitustilaan (*deadlock*). Tämä tarkoittaa tilannetta, jossa yksi tai useampi prosessi lukittuu odottaessaan jotakin tapahtumaa toiselta prosessilta tai jonkin resurssin vapautumista. Esimerkki lukittumisesta on tilanne, jossa kahden prosessin välillä on kaksisuuntainen yhteys, ja molemmat prosessit asettuvat lukemaan lukemista vastaavan putken tiedostokuvainta. Tällöin dataa ei milloinkaan saavu, vaan prosessit yrittävät lukea toistensa lähettämää dataa ikuisesti. Yleisimmin lukittumista tapahtuu estäviä (*blocking*) operaatioita käytettäessä, koska ne palaavat yleensä vasta kun operaatio onnistuu tai tapahtuu virhe.

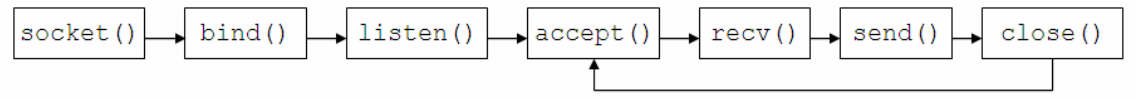
Prosessien lukittumisen mahdollisuus voidaan kuitenkin minimoida suunnittelemalla ohjelmat siten, ettei lukitustilanteita pääse tapahtumaan. Unix tarjoaa myös joitakin järjestelmäkutsuja tämän ongelman ehkäisemiseen. Esimerkiksi `select()`-kutsulla voi valita aktiivisen tiedostokuvaimen siten, että kutsulle syötetään ryhmä valvottavia luku- tai kirjoituskuvaimia ja aika, jonka jälkeen niiden valvonta lopetetaan. Sen avulla prosessi voi esimerkiksi valvoa ryhmää verkkoyhteyksiä, ja arvioida milloin se voi lukea jostakin niistä ilman estämistä [RuCo01, s. 154].

2.2.2 Unix-soketit

Unix-järjestelmissä tietoliikenneprotokollien sovellusrajapintana käytetään useimmiten soketteja, jotka ovat määritelty käytännössä joko Berkeley, System V tai POSIX-standardin mukaisesti. Ne tarjoavat sovellusohjelmoijalle pääsyn kuljetuskerrokselle TCP- tai UDP-protokollan tai suoraan siirtoyhteyskerrokselle IP-protokollan muodossa. Tässä kappaleessa on hyödynnetty Unix-järjestelmäkutsujen manuaalisivuja [OG01].

Soketti määritellään muodollisesti sovellusohjelman kommunikaation päätepisteeksi ja sen alla olevaksi protokollapinoksi. Käytännössä tämä tarkoittaa sitä, että ohjelma voi soketin avulla lukea ja kirjoittaa tietoa verkon yli ja hallita alla olevaa verkkoprotokollaa asettamalla soketille erilaisia asetuksia. Sovellusohjelmoijan kannalta soketti on yleisimmin kaksisuuntainen kommunikointikavana verkon yli, jolle määritetään käytettävä portti ja protokolla. Sokettia voidaan teknisesti verrata levyoperaatioita varten käytettäviin tiedostokuvaimiin. [Ker02b]

Soketin luomiseen ja käyttämiseen hyödynnetään erilaisia järjestelmäkutsuja, joista yleisimmät ovat `socket()`, `bind()`, `connect()`, `listen()`, `accept()`, `send()`, `recv()` ja `close()`. Seuraavaksi havainnollistetaan soketin toimintaa näiden kutsujen avulla. Unix sisältää myös muita sokettien hallintaan liittyviä järjestelmäkutsuja, mutta edellä mainitut lienevät oleellisimmat soketin elinkaaren ja toiminnan havainnollistamisen kannalta. Kuva 2-3 selkeyttää järjestelmäkutsujen järjestystä palvelimen näkökulmasta. Käyttäjän näkökulmasta lohkokaavio on melko samanlainen, mutta `listen()`- ja `accept()`-kutsujen sijaan yhteyden ottamiseen käytetään `connect()`-kutsua ja tyypillinen tiedonsiirto-operaatiot `recv()` ja `send()` tapahtuvat päinvastaisessa järjestyksessä. Kutsuille on yhteistä niiden palauttama numeroarvo, jota tarkastelemalla voidaan päätellä operaation onnistuminen.



Kuva 2-3. Verkko-operaatio sokettien avulla palvelimen näkökulmasta.

Soketin luominen tapahtuu Unixin `socket()`-järjestelmäkutsulla, jonka parametreiksi annetaan käytettävä osoiteperhe, sokettityyppi ja protokolla. Osoiteperheenä käytetään yleensä IP-osoitteille tarkoitettua *AF_INET*-osoiteperhettä. Sokettityyppinä voidaan käyttää esimerkiksi *SOCK_STREAM* TCP-liikennettä ja *SOCK_DGRAM* UDP-liikennettä varten. Protokollana käytetään yleensä nollaa, joka tarkoittaa käytettävän sokettityypin oletusprotokollaa. Kun soketti on luotu, sitä voidaan käsitellä kutsun palauttaman kuvaimen avulla. Dataa voidaan käsitellä tarvittaessa myös sellaisenaan ilman protokollaa.

Kun soketti on luotu, se pitää sitoa haluttuun yhteyteen. Tämän avulla määritetään paikallinen osoite ja etäosoite, eli päätepisteet, joiden välillä soketilla halutaan kommunikoida. Tämä tapahtuu `bind()`-kutsulla, jonka parametreiksi tulee sidottava soketti, osoite ja osoitteen pituus. Kuuntelevan portin tapauksessa yhteyden etäosoite

jätetään yleensä avoimeksi. Osoitteen muodon on noudatettava soketin käyttämää osoiteperhettä, ja IP-osoitteen tapauksessa siinä on määritellään käytettävä IP-osoite ja portti.

Seuraavaksi on valittava, halutaanko ottaa yhteys toiseen päätepisteeseen vai asetetaanko soketti kuuntelutilaan. Yhteys toiseen päätepisteeseen otetaan `connect()`-kutsulla, joka tarvitsee parametreikseen käytettävän soketin, osoitteen ja osoitteen pituuden. Kuuntelutila voidaan asettaa `listen()`-kutsulla, jonka parametreinä on haluttu soketti ja *backlog*-arvo, joka tarkoittaa yhteyspyyntöjonon maksimipituutta.

Sitomisen ja yhteyden ottamisen tai yhteyspyynnön jälkeen sokettia voidaan käyttää datan lukemiseen ja kirjoittamiseen. Tähän käytetään `recv()`- ja `send()`-kutsuja, joiden parametreiksi annetaan käytettävä soketti, merkkipuskuri, puskurin pituus ja lisäasetukset. Luettaessa merkkipuskuriin tallennetaan saatu data ja kirjoittaessa siitä luetaan lähetettävä data. Kuvassa 2-3 palvelin käyttää `recv()`-kutsua pyynnön lukemiseen ja `send()`-kutsua vasteen lähettämiseen.

On huomattava, että `recv()`- ja `send()`-kutsut ovat tarkoitettu yhteydellisen TCP-liikenteen käyttöön. Datagrammipohjaisen UPD-liikenteen välittämiseen käytetään `sendto()` ja `recvfrom()`-kutsuja, jotka vaativat lisäparametrikseen vielä vastaanottajan tai lähettäjän osoitteen. UDP:llä liikennöitäessä ei käytetä `connect()`- ja `listen()`-kutsuja, koska lähettäminen ja vastaanotto tapahtu ilman erillistä yhteydenmuodostusta.

Kun yhteys halutaan lopettaa, kutsutaan `close()`-kutsua. Tämän ainoa parametri on suljettava soketti. `Close()` on yleiskäyttöinen tiedostokuvaimen sulkemiseen tarkoitettu kutsu, joten sitä voi käyttää soketin lisäksi esimerkiksi tiedostokuvaimen sulkemiseen.

2.2.3 Pyyntöön vastaaminen

Yhteydenmuodostus palvelimelle alkaa käyttäjän tekemästä komennosta, kun hän kirjoittaa selaimeen haluamansa palvelimen osoitteen. Tämän jälkeen HTTP-asiakasohjelma luo palvelinohjelmistoon TCP-yhteyden [Pel98, s. 19]. Yhteyden avauduttua välitetty HTTP-pyyntöotsake määrittää pyydetyn sisällön alikohdassa 2.1.1 esitetyllä tavalla. Jos otsakkeessa pyydetty data on staattista, esimerkiksi HTML-tiedosto, se voidaan hakea palvelimen tiedostojärjestelmästä. Muussa tapauksessa palvelin generoi sisällön dynaamisesti.

Yleensä ensimmäinen palvelinohjelman suorittama vaihe on asettua kuuntelemaan haluttua TCP-sokettia. Kuunteleminen tarkoittaa porttiin tulevien yhteyspyyntöjen vastaanottamista, jonka jälkeen pyyntö hyväksytään ja luetaan sen pyyntöotsake. Sen jälkeen suoritetaan tarvittavat vaiheet vasteen laatimiseksi. Tämä voi tarkoittaa esimerkiksi levyltä lukemista tai tietokantakutsujen suorittamista. Yksinkertaisimmillaan vasteen laatiminen tarkoittaa levylle tallennetun HTML-tiedoston lähettämistä, mutta dynaamiset verkko-ohjelmistot edellyttävät monimutkaisempaa logiikkaa, esimerkiksi jollain kohdassa 2.1.2 käsitellyllä tavalla.

Palvelimen toiminnan suunnittelussa on tärkeää ottaa huomioon eri operaatioiden estäminen, kuten alikohdassa 2.2.1 mainittiin. Yleisesti ottaen operaatiot, jotka lukevat dataa tai vastaanottavat yhteyksiä estävät. Estäminen tapahtuu esimerkiksi Unixin `accept()`-kutsun yhteydessä, kun hyväksyttävä yhteys ei ole vielä saapunut tai luettaessa tietoa `read()`-kutsulla. Tehokkaan HTTP-palvelimen pitää ottaa estäminen huomioon siten, ettei se aiheuta koko palvelimen pysähtymistä, joka näkyisi käyttäjälle heikkona vasteena. Ongelma voidaan ratkaista pyynnöille `fork()`-kutsulla käynnistettävillä lapsiprosesseilla, säikeillä tai tapahtumapohjaisella suunnittelulla, johon voidaan käyttää apuna esimerkiksi aktiivisen tiedostokuvaimen valitsevaa `select()`-kutsua. Palvelimen eri prosessimalleja käsitellään tarkemmin seuraavassa kohdassa.

Kun pyyntö on vastaanotettu, sen käsittely voidaan ohjata pyyntöä vastaavalle lapsiprosessille tai säikeelle, joka suorittaa pyynnön käsittelyä vastaavat toimenpiteet. Nämä voivat olla pyynnön antamien parametrien arvioimista, ja esimerkiksi tietokantakutsujen suorittamista parametrien perusteella. Vaste lähetetään käyttäjälle tyypillisesti `send()`-kutsulla, minkä jälkeen soketti voidaan sulkea tai jäädä odottamaan seuraavaa pyyntöä.

2.3 Prosessimallit

Tässä kohdassa käsitellään erilaisia prosessimalleja HTTP-palvelimen näkökulmasta. Mallien perusperiaatteet pätevät mihin tahansa prosesseihin, mutta tässä kohdassa mallien tarkastelun pohjana käytettävät vaiheet tapahtuvat vain ohjelmalla, jolle on ominaista pyynnön vastaanottaminen, prosessointi ja vastauksen lähettäminen, aivan kuten HTTP-palvelimella. Tämä kohta perustuu lähteeseen [Dru99].

Projektin kannalta palvelinsovelluksen prosessimallilla on merkitystä sen muistinkäytön ja kohdeympäristöön integroinnin kannalta. Käyttöjärjestelmän näkökulmasta prosessimallit eroavat toisistaan lähinnä prosessien määrän suhteen, sillä esimerkiksi moniprosessinen sovellus käyttää useampaa yhteaikaista prosessia, kun monisäikeinen tai SPED-mallinen sovellus pärjää yleensä yhdellä. Unix-pohjaisilla käyttöjärjestelmillä uusien prosessien luomista tehostaa tosin `fork()`-kutsu ja sen myötä datan yhteiskäyttöön tarkoitettu kirjoituskopiointi (*copy on write*) [HaJä03, s. 144]. Prosessimallin vaikutus sovelluksen muistikapasiteetin tarpeeseen on siten pieni, koska virtuaalimuistin sivuja varataan muuttujien käytön mukaan, jolloin tärkeämpi muistivaatimukseen vaikuttava tekijä on sovelluksen toteutus ja sen todellinen kuormitus. Tästä huolimatta pienikin prosessimallin tarjoama muistinsäästö koettiin projektissa eduksi, mikä suosi prosessien määrän minimointia.

Erityisen suuri merkitys prosessimallin valinnalla on sen integroinnin helppoudessa projektin kohdeympäristöön. Kohdeympäristössä erilaisten sovellusmoduulien aikaa vievien operaatioiden vuorottelu tehdään 3.2.1 esiteltävällä tavalla, jossa moduuleja vastaavia tiedostokuvaimia kuuntelemalla. Prosessien määrän lisääminen lisää prosessikohtaisista resursseista johtuen myös kuunneltavien tiedostokuvaimien määrää, joten myös tämä seikka tukee käytettävien prosessien määrän minimointia.

Moniprosessimalli

Moniprosessimallissa käyttäjän pyynnölle määritetään prosessi, joka ajaa kuhunkin pyyntöön liittyvät peräkkäiset vaiheet. Prosessi suorittaa kaikki yhteen HTTP-pyyntöön liittyvät vaiheet ennen uuden pyynnön hyväksymistä. Tämä tarkoittaa yleensä käytännössä sitä, että yksi prosessi huolehtii pyyntöjen vastaanottamisesta ja käynnistää näille yksittäisestä pyynnöstä huolehtivan aliprosessin (Unix-järjestelmissä lapsiprosessi). Kun aliprosesseja käynnistetään useita, voidaan palvella useaa HTTP-pyyntöä samanaikaisesti. Jos jokin prosessi asettuu estämään esimerkiksi levy- tai verkkotoiminnon vuoksi, käyttöjärjestelmä vaihtaa suoritettavaa prosessia, jolloin prosessoriaikaa ei kulu hukkaan.

Jokaisella prosessilla on myös oma osoiteavaruutensa, jolloin yhtäaikaisten HTTP-pyyntöjen suorittaminen helpottuu, koska kukin prosessi voi säilyttää käsiteltävänä olevan pyynnön tilatietoa ilman huolta pyyntöjen välisistä ristiriidoista. Tosin globaaliin dataan, esimerkiksi välimuistiin asetettuihin URL:eihin perustuvien optimointien toteuttaminen saattaa olla vaikeaa. Useimmissa HTTP-palvelimissa tarvitaan myös jonkinlaista tietokantaa, jolloin prosessien tekemien pyyntöjen, tallennusten ja muiden operaatioiden pitää tapahtua tietokannan eheys huomioon ottaen. Tämä vaatii yleensä poissulkemis- ja synkronointiongelmien ratkaisemisen prosessien välisen kommunikoinnin avulla.

Monisäiemalli

Monisäikeinen palvelin toimii periaatteeltaan samoin kuin moniprosessinen, mutta resurssien säästämiseksi pyyntöjen käsittelystä huolehtivat prosessien sijaan säikeet. Kun yksi säie asettuu estämään esimerkiksi I/O-operaation vuoksi, voidaan ohjelman suoritusta jatkaa toisessa säikeessä. Prosessin ja säikeen oleellisin ero on resurssien käytössä: säiekohtaisten resurssien sijaan saman prosessin säikeet jakavat keskenään muun muassa prosessin muistialueen ja tiedostokuvaimet. Tällöin tarvittavien resurssien määrä pienenee ja kontekstin vaihdot eri säikeiden välillä nopeutuvat. Kunkin säikeen paikalliset muuttujat säilyvät omassa pinomuistissaan. Säikeiden ohjaaminen vaatii yleensä jaetun tilatiedon hyödyntämistä ja jaetun dataanpääsyn synkronointia.

Prosessien tavoin yksi säie suorittaa yhden HTTP-pyynnön vastaanottamisen ja hyväksymisen jälkeen kaikki sen tarvitsemat vaiheet ennen uuden pyynnön hyväksymistä.

Säikeiden käyttö vaatii käyttöjärjestelmän tuen, ja säikeistyksestä saatu hyöty riippuu osaltaan tuen toteutuksesta. Jotkin käyttöjärjestelmät tarjoavat myös käyttäjäalueella toimivia säiekirjastoja, jolloin varsinaista kernel-tukea ei tarvita, mutta myöskään saadussa hyödyssä ei päästä natiivin säikeistyksen tasolle.

Single-process Event-driven (SPED)

SPED-malli poikkeaa kahdesta edellisestä mallista siten, että siinä jokaiselle HTTP-pyynnölle käynnistettävän prosessin tai säikeen sijaan sekä pyyntöjen vastaaottamisesta että niiden käsittelystä vastaa yksi prosessi. Tämän palvelinprosessin suoritus pitää tapahtua siten, että se ei missään tilanteessa ajaudu lukitustilaan, jolloin palvelimen

toiminta keskeytyisi. Tämä edellyttää estävien järjestelmäkutsujen käyttämistä siten, että palvelimen tavoitettavuus säilyy. Lisäksi esimerkiksi estävien I/O-operaatioiden sijaan voidaan käyttää niiden asynkronisia vastineita, mikäli käyttöjärjestelmä sellaisia tukee. Unix-järjestelmissä estävien kutsujen yhteydessä on hyödyllistä käyttää `select()`-kutsua, joka mainittiin alikohdassa 2.2.1. Kutsun periaatteena on, että muun toiminnan pois sulkevan odottamisen sijaan ne tarkistavat säännöllisin väliajoin, ovatko suoritettavat I/O-operaatiot aktiivisia, jolloin prosessoriaikaa ei kulu yhden estävän operaation odottamiseen. Lisäksi joillekin estäville kutsuille voidaan asettaa lisäparametri, joka määrää sen palaamaan välittömästi ja operaation onnistuminen voidaan päätellä sen palautearvosta.

SPED-palvelimen toiminta perustuu pyyntökohtaisen tilatiedon säilyttämiseen, jolloin kukin pyyntö voidaan käsitellä ositettuna vaihe kerrallaan. Jokaisella iteraatiokierroksella palvelin voi tarkistaa tulevat yhteydet, lähetyspuskurit tai suoritetut tiedosto-operaatiot, ja valita aktiivinen lähde suoritettavaksi. Suorittamisen jälkeen alustetaan kyseisen pyynnön seuraava vaihe, joka voi olla esimerkiksi HTTP-vasteen lähettäminen.

SPED-palvelimien yksi merkittävä ongelma on paljon prosessoriaikaa vievät I/O-operaatiot, esimerkiksi suuren tiedoston lukeminen. Palvelinprosessi ei saisi yksittäistä pyyntöä suoritettaessa asettua estämään, koska tällöin uusia pyyntöjä ei voida vastaanottaa, mikä saattaa näkyä käyttäjälle huonona saatavuutena tai vasteaikana.

Nykyaikaisissa käyttöjärjestelmissä myös monet CPU-, levy- ja verkko-operaatiot voidaan limittää eli suorittaa ositetusti. Tämä tehostaa entisestään SPED-palvelimen toimintaa ja etenkin vasteaikaa, mutta ongelmana on vanhemmista käyttöjärjestelmistä puuttuva tuki. Muun muassa Linuxin kerneliä 2.6 käyttävissä järjestelmissä asynkroniset levyoperaatiot ovat vakio-ominaisuutena.

Useimmat SPED-palvelimet eivät tuesta huolimatta käytä asynkronisia operaatioita, vaan turvautuvat perinteisiin järjestelmäkutsuihin ja niiden tarjoamaan yhtäaikaisuuden hallintaan. Tästä johtuen SPED-palvelimia käytetään yleensä vain erityissyistä niiden tarjoaman muistinsäästön vuoksi. Ne soveltuvat erityisen hyvin sulautettuihin järjestelmiin, joiden muistikapasiteetti on rajoittunut.

Asymmetric Multi-process Event-driven (AMPED)

AMPED-malli pyrkii hyödyntämään tapahtumaohjatun SPED-mallin olennaisia piirteitä, eli tilatietoon pohjautuvaa suoritusta ja jaettua muistialuetta, ja tukemaan sitä useilla avustavilla prosesseilla tai säikeillä. Apuprosesseilla voidaan huolehtia esimerkiksi tarvittavista I/O-operaatioista. Niiden avulla palvelimen vasteaikaa saadaan parannettua, koska estävät levy- ja verkko-operaatiot voidaan hoitaa omissa prosesseissaan sillä aikaa, kun tapahtumaohjattu pääprosessi hoitaa HTTP-pyynnön käsittelyn muut vaiheet.

AMPED-malli pyrkii säilyttämään SPED-mallin tehon muissa kuin tiedostonlukuoperaatioissa, mutta välttää sen ongelmat esimerkiksi hitaiden levyoperaatioiden ja asynkronisten levyoperaatioiden järjestelmätukiongelmien osalta.

AMPED-mallia voidaan hyödyntää käyttöjärjestelmien tavanomaisilla menetelmillä, joten käyttöönoton helppoudesta ja laajasta järjestelmätuesta ei tarvitse tinkiä.

Unix-järjestelmissä AMPED käyttää standardeja järjestelmäkutsuja kuten `read()`, `write()` ja `accept()` soketteja ja putkia käytettäessä sekä `select()`- tai `poll()`–kutsua testaamaan I/O-operaation valmistumista. AMPED-mallia käytettäessä on oleellista myös tiedostojärjestelmän dataanpääsystä huolehtiminen, aivan kuten moniprosessi- ja monisäiemalleillakin.

Vaikka AMPED-malli ei ole yhtä suosittu kuin muut kohdassa esitellyt prosessimallit, se on vakavasti harkittava palvelinvaihtoehto etenkin sulautetuille järjestelmille, joissa ohjelmalta vaaditaan hyvää suorituskykyä pienellä kuormituksella.

2.4 Palvelimen tietoturva

Minkä tahansa palvelinsovelluksen toteuttamiseen kuuluu usein oleellisena osana tietoturva. WWW-palvelimen tapauksessa se tarkoittaa yleensä palvelimen dokumenttien pääsynvalvontaa ja suurempaa yksityisyyttä vaativissa sovelluksissa, kuten pankkipalveluissa, myös siirrettävän tiedon salaamista. Tässä kohdassa käsitellään muutamaa projektin aikana yleisimmin esiin tullutta tietoturvamenetelmää, jotka liittyvät sekä käyttäjän autentikointiin että tiedon salaamiseen. Autentikointimenetelmistä käydään läpi HTTP-standardin oma autentikointi ja vahvempaan autentikointiin tarkoitettu SSL (*Secure Sockets Layer*) ja sen seuraaja TLS (*Transport Layer Security*). Yleinen tapa vahvan autentikoinnin toteuttamiseen on SSL:n käyttämä julkisen avaimen menetelmä. SSL- ja TLS-protokollien yhteydessä tarkastellaan myös niiden tarjoamia keinoja tiedon salaamiseen.

Kohdan ulkopuolelle jäävät palvelinkohtaiset tietoturvaratkaisut, joita ovat muun muassa IP-pohjainen käytönvalvonta ja tiedostojärjestelmän avulla toteutettu oikeuksien rajoittaminen. Palvelinsovellukselle voidaan toteuttaa myös oma autentimenetelmänsä, jolla hallintasovellus voi varmistua käyttäjän oikeellisuudesta. Näitä seikkoja käsitellään myöhemmin palvelimien esittelyn yhteydessä luvussa neljä. Lisäksi kohdassa käsitellään vain käyttäjän ja palvelinsovelluksen välistä tietoturvaa, vaikka turvallista palvelinta toteutettaessa on otettava huomioon myös palvelimen arkkitehtuuri, ohjelmistokoodin turvallisuus ja yrityksen tai organisaation tietoturvallisuus.

Käyttäjän autentikoinnin perusteella on yleensä tarpeen tehdä myös jonkinlainen autorisointi, joka määrittelee käyttäjän oikeudet palvelimen toimintoihin. SHDSL-laitteen hallintasovelluksessa tämä tarkoittaa käyttäjäkohtaisia laitteen hallintamahdollisuuksia. Autorisointi vaikuttaa tässä tapauksessa palvelinohjelmiston sijaan vain toteutettavan verkko-ohjelmiston toteutukseen, joten siihen palataan tarvittavilta osin luvussa viisi.

2.4.1 HTTP-autentikointi

HTTP-standardi esittelee menetelmän yksinkertaiseen pääsynvalvontaan eli autentikointiin. Standardi sisältää kaksi autentikointiskeemaa: yksinkertaisen *basic*-skeeman ja turvallisemman *digest*-skeeman. IETF on määritellyt nämä menetelmät tarkemmin RFC:ssä 2617 [Fra99]. Tämä alikohta käsittelee HTTP:n tarjoamia

autentikointimenetelmiä, ja perustuu edellä mainitun RFC:n lisäksi lähteisiin [Ker98, s. 148-153] ja [Pus01, s. 140-153].

HTTP-autentikointi perustuu käyttäjän tunnistamiseen käyttäjänimen ja salasanan perusteella, ja sen avulla voidaan suojata palvelimen dokumentteja. Vaikka salasanojen käyttö on inhimillisistä riskitekijöistä johtuen heikko autentikointimenetelmä, se on riittävän vahva moniin sovelluksiin. Vaativammissa sovelluksissa vasta salasanan käyttö jonkin vahvan autentikointimenetelmän, esimerkiksi SSL:n kanssa riittää takaamaan vaaditun tietoturvan.

Basic-autentikointi

Palvelin voidaan konfiguroida käyttämään *basic*-autentikointia, joka tehdään yleensä palvelimen yleiseen asetustiedostoon, johon määritetään hakemistokohtaisesti käytettävä autentikointi ja sallitut käyttäjänimet. Käyttäjien salasanat voidaan asettaa samaan tiedostoon tai erilliseen salasanatiedostoon, joka yleensä niin ikään salataan.

Kun käyttäjä ottaa selainyhteyden suojattuun kansioon palvelimella, palvelin vastaa tähän viestillä, joka ilmaisee, että kansioon pääsy vaatii käyttäjän tunnistamisen. Kuva 2-4 esittelee palvelimelta tulevan autentikointipyynnön, kun käytössä on *basic*-autentikointi. Siitä selviää alikohdasta 2.1.1 tuttujen asioiden lisäksi käyttörajoitusta ilmaiseva vastauskoodi 401, käytettävä tunnistusmenetelmä *basic* sekä *realm*-alue *salainen*. Viimeisen avulla voidaan määrittää ”suojausalueita”, joiden sisällä liikkuessaan käyttäjän tarvitsee tunnistautua vain kerran.

```
HTTP/1.1 401 Unauthorised
Date: Sun, 19 Nov 2006 18:06:11 GMT
Server: Apache/2.0.52 (Red Hat)
WWW-Authenticate: Basic realm="salainen"
Content-Length: 487
Content-Type: text/html; charset=iso-8859-1
```

Kuva 2-4. Palvelimen basic-autentikointipyyntö.

Pyynnön perusteella selain avaa ponnahdusikkunan, johon käyttäjä voi antaa tunnuksensa ja salasanan. Selain lähettää GET-pyyntönsä otsakkeen *Authorization*-kentässä palvelimelle käyttäjänimensä ja salasanansa, johon palvelin vastaa autentikoinnin onnistuessa tavanomaisella OK-viestillä ja sen yhteydessä lähetettävällä datalla.

Basic-autentikoinnin heikkous on se, että käyttäjänimi ja salasana lähetetään GET-viestissä selkokielisenä, jolloin liikennettä kuunteleva hyökkääjä voi helposti saada käyttäjätunnuksen haltuunsa. Tämä voi kuitenkin olla joillekin sovelluksille edellytyksenä, mikäli käyttäjältä vastaanotettua salasanaa on pystyttävä vertailemaan selkokielisenä.

Digest-autentikointi

Basic-autentikoinnin puutteisiin tuo apua *digest*-menetelmä, jonka tarkoituksena on laskea annettujen satunnaisarvojen perusteella verkon yli lähetettävistä tiedoista tiiviste. Myöskään *digest*-menetelmä ei ota kantaa viestien hyötykuorman salaamiseen, mutta autentikointimenetelmänä se korjaa *basic*-menetelmän suurimmat ongelmat.

Toiminnaltaan *digest*-autentikointi on samankaltainen kuin *basic*, eli kun käyttäjä pyytää palvelimelta dokumenttia ilman oikeuksia (oikeaa tunnusta ja salasanaa), palvelin vastaa käyttäjärajoitusta ilmaisevalla 401-sanomalla. Esimerkki sanomasta näkyy kuvassa 2-5.

```
HTTP/1.1 401 Unauthorized
Date: Sun, 19 Nov 2006 18:06:11 GMT
Server: Apache/2.0.52 (Red Hat)
WWW-Authenticate: Digest
                  realm="salainen@palvelin.com",
                  qop="auth,auth-int",
                  nonce="5a4ec0e876a987b7b06ce76c0a7e86b3",
                  opaque="fc67cee59a6c9e7cf574a40a8a5a34"
Content-Length: 487
Content-Type: text/html; charset=iso-8859-1
```

Kuva 2-5. Palvelimen digest-autentikointipyyntö.

Kuvassa näkyy käytettävä *realm*-alue, suojauksen laatu, satunnaismerkkijono tiivisteen laskentaa varten ja *opaque*-merkkijono, jonka käyttäjän pitää palauttaa tiivisteessä muuttumattomana palvelimelle. Tiivisteen laskentaan käytetään MD5-funktiota.

Kun käyttäjä on palauttanut laskemansa tiivisteen ja sen sisältämän käyttäjänimen ja salasanan palvelimelle, laskee palvelin saman itse ja vertaa käyttäjältä saatua tiivistettä tulokseen. Tiivisteen ollessa sama, voidaan varmistua käyttäjän oikeellisuudesta ja lähettää tälle OK-sanoma ja pyydetty data. Tiivistettä ei ole enää mahdollista purkaa selkokieliseen muotoon, joten *digest*-menetelmän käyttö edellyttää, että palvelin tuntee oikean salasanan alkuperäisessä muodossaan. Tällöin esimerkiksi tiivisteiden vertailu kryptattuihin Unix-salasanoihin ei ole mahdollista, koska niiden muuntaminen vertailukelpoiseen muotoon ei käytännössä onnistu.

2.4.2 SSL/TLS

HTTPS:n toteuttamiseen käytetään yleisimmin SSL-protokollaa, joka mahdollistaa turvallisen tiedonsiirron verkko-ohjelmistojen välillä. SSL on internetin yleisin ja ehkäpä tärkein tietoturvaprotokolla [Ker98, s. 296]. SSL on Netscapen kehittämä protokolla, ja siitä käytetään nykyisin versiota 3.0 [Fre06]. Myös SSL:n seuraajaa eli TLS-protokollaa tuetaan laajasti, ja lyhennettä *SSL* käytetäänkin yleisnimityksenä näille kahdelle protokollalle. TLS:ää kehittää IETF, ja siitä käytetään yleisimmin versiota 1.0 [DiAl99], joka julkaistiin vuonna 1999. Keväällä 2006 ilmestyi TLS versio 1.1, jossa on paranneltu muun muassa RSA-algoritmin toimintaa.

Sekä SSL että TLS takaavat tiedonsiirrolle kolme tietoturvan perusominaisuutta: autenttisuuden, luottamuksellisuuden ja tiedon eheyden. Luottamuksellisuudella tarkoitetaan sitä, että tietojärjestelmässä oleva tai siirrettävä tieto on vain sen käyttöön oikeutettujen saatavilla. Autenttisuudella tarkoitetaan sitä, että kommunikoinnin osapuolet voidaan tunnistaa ja eheydellä sitä, että tiedot ovat luotettavia. TLS-protokollan muutokset SSL:ään nähden ovat pieniä, ja se tukee hyvin SSL 3.0:ssa määriteltyjä viestien rakenteita. TLS:ssä on myös poistettu tuki Fortezza-salauskortille. [Ker98, s. 93]

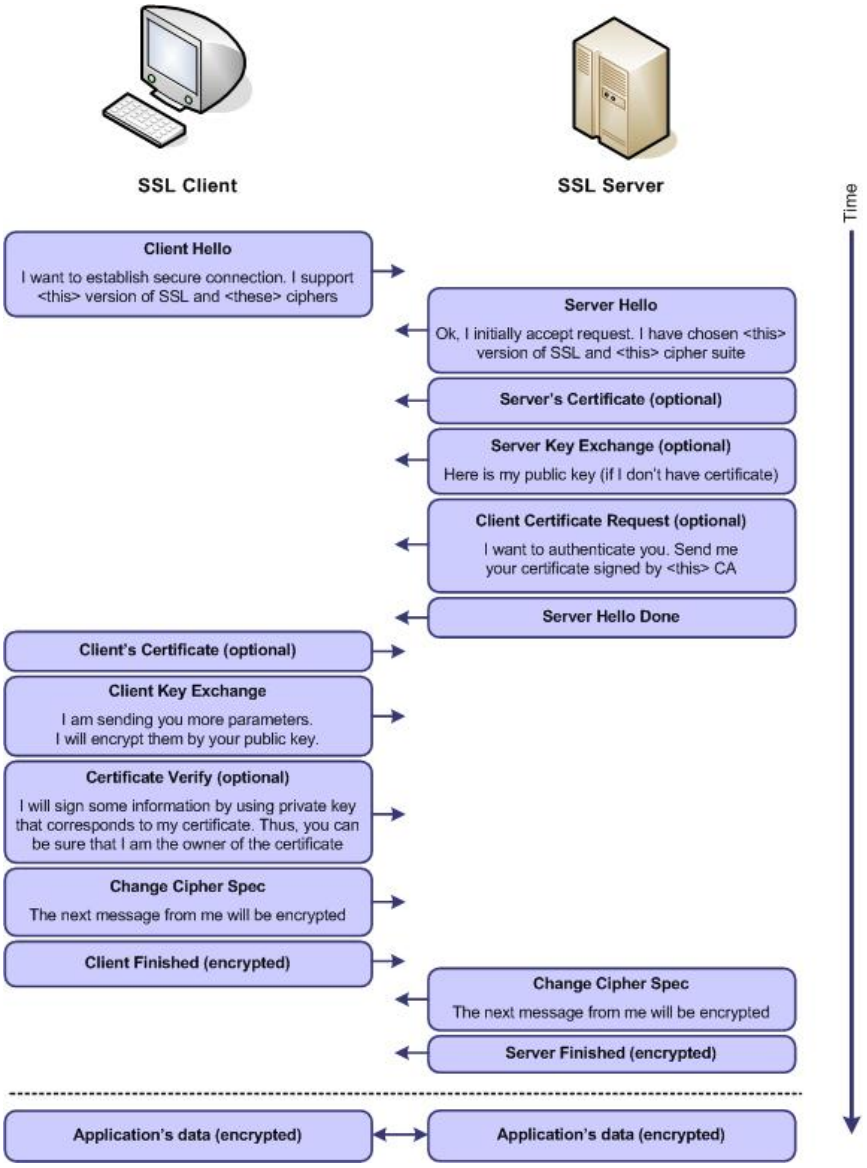
SSL-protokolla

SSL-protokolla toimii kuljetus- ja sovelluskerroksen välissä ja on riippumaton sitä käyttävästä sovelluksesta. SSL tarvitsee oman TCP/IP-sokettinsa, jolle käytetään yleisesti porttia 443. Esimerkiksi SSL-salausta käyttävällä HTTP-palvelimella tämä tarkoittaa SSL-portin käyttämistä yleisen 80-portin sijaan. SSL:ää käytettäessä kaikki sen kautta kulkeva liikenne eli dokumentin URL, sen sisältö, lomakkeiden sisältö, evästeet ja HTTP-otsakkeet ovat salattuja. Satunnaiselle kuuntelijalle selviää ainoastaan keskustelevien osapuolten IP-osoitteet ja SSL-sanomien tyyppi.

SSL mahdollistaa joustavan symmetrisen salaus-, tiiviste- ja autentikointimenetelmän valinnan. SSL voi käyttää salaukseen DES-, 3-DES-, RC2- tai RC4-algoritmeja ja autentikointiin MD5- tai SHA-tiivisteitä sekä RSA-avaimia ja sertifikaatteja. Kommunikointi voi tapahtua myös anonyymisti, jolloin käytetään Diffie-Hellman –avaintenvaihtoalgoritmia. [Ker98, s. 297]

SSL-tapahtuman vaiheet

Seuraavaksi käydään läpi yleisellä tasolla SSL-tapahtuman aikana suoritettavat vaiheet palvelimen ja käyttäjän välillä. SSL-tapahtuman aluksi tiedonsiirron päätepisteet autentikoidaan ja sovitaan salauksessa käytettävä symmetrinen avain. Vaiheet suoritetaan päätepisteiden välillä lähetettävien sanomien avulla, joiden kulkua havainnollistetaan kuvassa 2-6.



Kuva 2-6. SSL-yhteyden muodostaminen [Maj05].

Käyttäjä aloittaa yhteydenoton palvelimeen *Client hello* –sanomalla, joka sisältää sessiotiedon, satunnaisdataa salausta varten sekä tietoa käyttäjän tukemasta protokollasta ja suojausmenetelmistä. Se voi olla myös vastaus palvelimelta tulleeseen *Hello request* –sanomaan, jolla palvelin voi ilmaista pyyntönsä istuntoneuvottelun aloittamiseksi.

Palvelin vastaa käyttäjän onnistuneeseen yhteydenottoon *Server hello* –sanomalla. Virheen tapahtuessa vastataan virhettä ilmaisevalla sanomalla. *Server hello* sisältää tiedot palvelimen määräämistä salaus- ja pakkausalgoritmeista, ja palvelimen oman satunnaisdatan salausta varten. Lisäksi palvelin tarkastaa käyttäjän mahdollisesti antaman sessiotunnuksen, ja palauttaa tarpeen mukaan joko uutta sessiota ilmaisevan arvon tai vanhan, edellistä sessiota ilmaisevan arvon.

Seuraavaksi palvelin lähettää käyttäjälle sertifikaattinsa, joka on valittujen salausasetusten mukainen. Yleisesti käytetään X.509.v3-standardin mukaista sertifikaattia [Ker98, s. 150-152]. Käyttäjä tietää palvelimen tervehdysvaiheen päättyneen *Server hello done* –sanoman perusteella. Mikäli palvelin on tehnyt sertifikaattipyynnön, lähettää käyttäjä palvelimelle oman sertifikaattinsa, joka on muodoltaan vastaava kuin palvelimen sertifikaatti. Sertifikaatin avulla palvelin voi varmistua käyttäjän oikeellisuudesta samalla tavalla kuin käyttäjä palvelimen, jolloin saavutetaan molempien päätepisteiden autentikointi.

Tämän jälkeen käyttäjä lähettää *Client key exchange* –sanoman, joka sisältää avaimen, jolla luodaan lopullinen istuntoavain. Tätä avainta kutsutaan *premaster*-avaimeksi, ja sen muoto ja pituus riippuvat valituista salausalgoritmeista. Lopullista istuntoavainta eli *master*-avainta käytetään kaiken istunnossa siirrettävän tiedon salaamiseen. *Premaster*-avaimen salaaminen PKI (*Public Key Interface*) –menetelmällä ennen sen lähettämistä on ensiarvoisen tärkeää, jotta siirrettävä tieto voidaan salata hyökkääjältä.

Istuntoavaimen lähettämistä seuraa vapaaehtoinen *Certificate verify* –sanoma, jolla varmistetaan käyttäjän lähettämä sertifikaatti. Yhteydenmuodostus päätetään yhteysosapuolten toisilleen lähettämillä *Finished*-sanomilla. Samalla varmistetaan avaintenvaihto- ja autentikointitapahtumien oikeellisuus sanomien sisältämien tiivisteiden avulla. Tämän jälkeen varsinainen istunto voi alkaa, jonka aikana päätepisteet salaavat niiden välillä kulkevan liikenteen yhteisellä, symmetrisellä salausavaimella.

2.5 MVC-sovellusarkkitehtuuri

Samalla kun verkko-ohjelmistojen vaatimukset monipuolisuuden, ulkoasun ja käyttäjäystävällisyyden suhteen kasvavat tiedonsiirtoyhteyksien ja palvelinkoneiden tehostuessa, myös niiden toteuttaminen muuttuu vuosi vuodelta vaikeammaksi. Lisähaasteena verkko-ohjelmiston toteutuksessa on sen käyttämien rajapintojen ja toisinaan myös ohjelmointikielten suuri määrä, koska ohjelmisto neuvottelee ulkomaailmaan käyttäjille ja käyttää usein hyödykseen tietokantaohjelmistoja. Kasvanutta monimutkaisuutta helpottavat parantuneet kehitysympäristöt, jotka tarjoavat virheentarkistusominaisuuksia ja mahdollistavat ohjelmiston eri osien käsittelyn yhtenäisellä tavalla. Lisäksi ohjelmistokehityksessä on vallitsevana pyrkimyksenä laajojen ohjelmistojen pilkkominen pienempiin, helpommin omaksuttaviin osiin, joita voidaan tarvittaessa kehittää myös toisistaan erillään. Modulaarisen suunnittelun mahdollistaa muun muassa oliokielet Java ja C++, sekä proseduraalisilla kielillä kutsurakenteen hierarkinen suunnittelu.

Etenkään verkko-ohjelmoinnissa pelkästään ohjelmointikielen modulaarisuus ei riitä jo siitä syystä, että nykyaikainen verkko-ohjelmisto saattaa käyttää hyödykseen useita eri ohjelmointikieliä: ulkoasuun voidaan käyttää HTML-, JavaScript- ja CSS-kieliä, logiikkaan Javaa, Perliä tai PHP:tä ja lisäksi voidaan tarvita SQL (*Structured Query Language*) –kutsuja tai muuta rajapintaa tietokannan käsittelyyn. MVC (*Model-View-Control)* –arkkitehtuuri pyrkii hajauttamaan nämä osat toisistaan, jolloin esimerkiksi HTML-lomakkeen parametreiksi ei tarvitse kirjoittaa suoraan SQL-kutsuja, ja toisaalta SQL-tietokannassa ei tarvitse huolehtia käyttäjälle lähetettävän vasteen ulkoasusta. MVC-rakenne sopii yleisesti kaikkiin käyttöliittymän, logiikan ja tietokannan sisältäviin

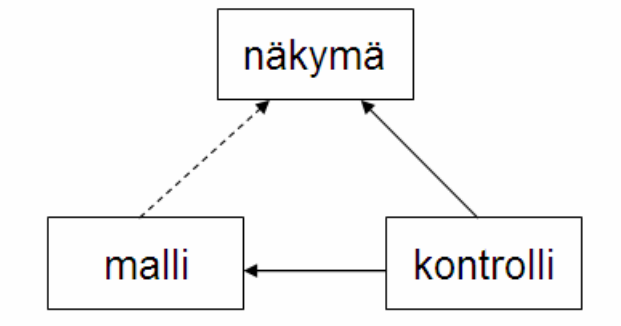
ohjelmistotyyppeihin, mutta tämän projektin puitteissa on tarpeen tarkastella sitä verkko-ohjelmiston näkökulmasta.

MVC-arkkitehtuuriin perustuvia sovelluskehyksiä on toteutettu lukuisia useille eri kielille, ja myös arkkitehtuurista käytettäviä nimityksiä on useita. Sun käyttää MVC:stä nimitystä *Model 2*, jota Sunin J2EE (*Java 2 Platform Enterprise Edition*) –sovelluskehys hyödyntää. Muita tunnettuja MVC-sovelluskehyksiä ovat esimerkiksi Apachen Javalle suunnittelema *Struts*, Rubylle *Ruby on Rails*, Pythonille *Django* ja Microsoftin kehittämä *ASP.NET*. Myös PHP:lle löytyy useita MVC-arkkitehtuurilla suunniteltuja sovelluskehyksiä.

Tässä työssä ei käsitellä konkreettisia sovelluskehyksiä, koska niitä ei projektin toteutuksessa hyödynnetty. Sen sijaan hyödynnettiin HTTP-palvelimen omaa API-rajapintaa ja sivupohjamenetelmää, joihin palataan luvuissa neljä ja viisi. Kuten luvun neljä palvelinvaihtoehtojen kartoituksessa huomataan, suurin osa palvelimista tukee CGI- tai FastCGI-rajapintaa, joiden avulla muun muassa tulkittaville kielille luotujen sovelluskehysten käyttöönotto onnistuisi helposti. Näihin tekniikoihin suhtauduttiin kuitenkin varauksella, koska sovelluskehykset ovat pääsääntöisesti raskaita, eivätkä siten sovellu tämän projektin sulautetulle verkon päätelaitteelle. Tästä huolimatta työssä haluttiin hyödyntää MVC-arkkitehtuurin periaatteita sovelluksen eri osa-alueiden jakamiseksi erillisiin, mahdollisimman itsenäisiin osiin. Tällä saavutetaan ohjelman parempi ylläpidettävyys, siirrettävyys ja sen komponenttien uudelleenkäytettävyys [HaMä98, s. 355].

Muita mainittavia sovellusarkkitehtuureita MVC:n lisäksi ovat muun muassa ohjelma- ja käyttöliittymäosaan sovelluksen jakava yksinkertainen jaottelu (*Simple Separation*), sekä Microsoftin MFC (*Microsoft Foundation Classes*), joka perustuu *Document/View* –jaotteluun.

MVC-arkkitehtuuri jaottelee nimensä mukaisesti ohjelmiston kolmeen eri osaan: malliin (*Model*), näkymään (*View*) ja kontrolliin (*Control*). Näistä mallilla tarkoitetaan ohjelmiston pysyvää, sovellusaluekohtaista dataa, näkymällä käyttäjälle näkyvää osaa ja kontrollilla näiden kahden osan välillä toimivaa logiikkaa. Kuvan 2-7 yksinkertainen lohkokaavio havainnollistaa eri osien vuorovaikutusta.



Kuva 2-7. MVC-sovellusarkkitehtuuri.

Kuvassa yhtenäinen nuoli esittää osien suoraa ja katkoviiva niiden epäsuoraa yhteyttä toisiinsa. Tämä tarkoittaa käytännössä sitä, että kaikki komennot näkymältä mallille ja toisaalta esitettävä tieto mallilta näkymälle kulkevat kontrollin kautta. MVC-arkkitehtuuri mahdollistaa ohjelmiston suunnittelun siten, että muutokset yhdessä ohjelmiston osassa eivät vaikuta ohjelman muuhun toimintaan, edellyttäen että eri osien väliset rajapinnat pysyvät samana.

Malli

MVC:n mallilla tarkoitetaan sovellusaluekohtaista, pysyvää ”tosielämän” dataa, jolla voidaan tarkoittaa esimerkiksi laskutusjärjestelmän kirjanpitomerkintöjä tai webin keskustelualueen käyttäjätietoja. Malli itsessään voi sisältää sen varastoiman tiedon tarvitsemaa logiikkaa, jonka avulla tarjotaan sovelluksen ylemmälle kerrokselle rajapinta tiedon saantia ja manipulointia varten. Lisäksi logiikan avulla tieto pysyy järjestyksessä, jolloin ylemmän kerroksen ei tarvitse huolehtia käsiteltävän tiedon jäsennyksestä ja rajoituksista. Monimutkaisemmissa tietorakenteissa on huolehdittava myös tiedon viite-eheydestä ja transaktioiden synkronoinnista.

Yleisimmin mallina käytetään SQL-kielistä relaatiotietokantaa, kuten MySQL tai PostgreSQL. Mallina voidaan käyttää myös esimerkiksi yksinkertaista tallennukseen käytettävää tiedostoa tai ohjelmistolla manipuloitavaa fyysistä laitetta.

Näkymä

Itse käyttäjälle verkko-ohjelmiston tutuin osa on näkymä. Näkymän tehtävänä käyttäjältä saatujen käskyjen välittäminen kontrollille ja huolehtia kontrollin kautta mallilta saadun datan esittämisestä käyttäjän ymmärtämässä muodossa. Alemmalta tasolta saadun tiedon ei välttämättä tarvitse olla peräisin mallilta, vaan näkymiä voidaan luoda myös suoraan kontrollilta saatujen vasteiden perusteella. Yleisimmin tapahtumiin liittyy kuitenkin myös mallilla olevan datan käsittelyä.

Tyypillinen web-näkymä käyttää tietojen esittämiseen HTML-kieltä, ja mahdollisesti JavaScriptiä näkymälogiikkaa ja CSS- tai XSL-kieltä ulkoasun hienosäätöä varten. Suurin hyöty näkymän eristämisestä muusta ohjelmasta on siinä, että muita osia suunniteltaessa ei tarvitse ottaa kantaa näkymään liittyviin seikkoihin, kuten tiedon asetteluun tai sivun ulkonäköön. Lisäksi web-sivun suunnittelijan ei tarvitse hallita näkymän alla toimivan sovelluslogiikan toteutuskieltä, vaan vain verkkosivujen luomiseen tarkoitetut kuvaus- ja skriptikielet.

Kontrolli

Mallin ja näkymän väliin tarvitaan vielä ohjelman aivot, joille käytetään MVC-arkkitehtuurissa nimeä kontrolli. Kontrolli vastaanottaa käyttäjältä tulevia pyyntöjä ja tarjoaa niihin mallia apuna käyttäen asianmukaisen vasteen. Kontrolli tuntee näkymän tarvitsemat ja mallin tarjoamat palvelut, mutta ei ota kantaa niiden toteutustapoihin, vaan toimii tarvittavana logiikkana näiden osien välillä.

Kontrollin toiminta perustuu yleensä tarvittaviin käsittelijöihin (*handlers*) ja takaisinkutsuihin (*callbacks*), joita näkymä voi käyttää. Käsittelijä vastaanottaa kutsun

ja sen tarvitsemat parametrit, joiden perusteella se käyttää mallin rajapintaa näkymän tarvitseman datan noutamiseen. Kontrolli voi tarvittaessa manipuloida dataa esimerkiksi poistamalla määrätyt tietokentät, mikäli se tietää ettei näkymä tule niitä tarvitsemaan. Kontrolli myös muotoilee mahdolliset logiikassa tai mallissa tapahtuneet virheet näkymän ymmärtämäksi vasteeksi, esimerkiksi ilmaisemalla HTML-lomakkeelle syötetyn virheellisen arvon. Täten alemmalla kerroksella tapahtuneet virheet voidaan esittää käyttäjälle halutulla tavalla. Takaisinkutsujen avulla voidaan määritellä näkymässä esitettävä tieto. Takaisinkutsut ovat yleensä erilaisia silmukka-, ehto- ja arviointikomentoja, joiden perusteella mallilta saatu tieto tulostetaan näkymään. Siten näkymässä voidaan vapaasti määritellä käyttäjälle esitettävä tieto ja sen asettelu.

3 Kohdeympäristö

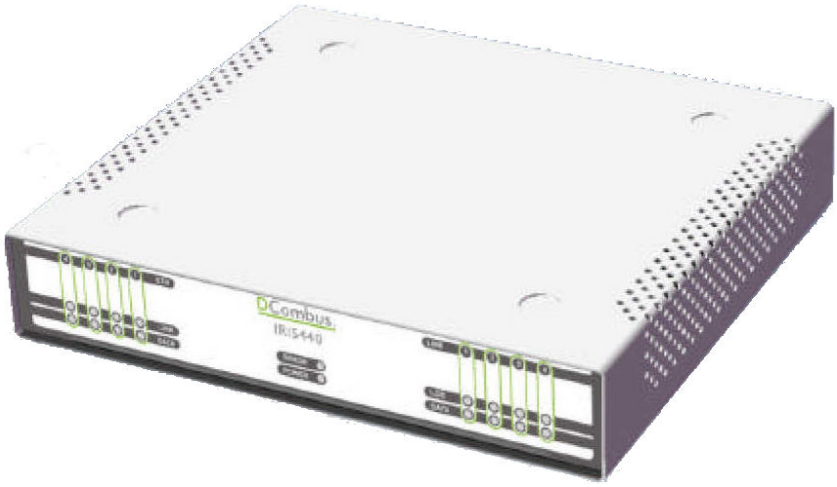
Verkon päätelaitteeseen toteutettavan sovelluksen määrittelyssä ja suunnittelussa ovat tärkeitä kohdeympäristön asettamat reunaehdot. Sulautettujen järjestelmien sovelluskehityksessä tästä tekee erittäin tärkeää se, että laitteisto on yleensä suoritusteholtaan rajoittunut ja sovellusympäristö ominaisuuksiltaan vaatimaton ja ennaltamäärätty. Laitteiston tarjoama laskentateho ja muistikapasiteetti antaa käsityksen sovelluksen vaatimuksista ja rajoittaa esimerkiksi web-hallintasovelluksen ytimenä käytettävän HTTP-palvelimen ratkaisuvaihtoehtoja. Myös hallintasovelluksen vasteaikoihin ja käytettävyyteen liittyviä ei-toiminnallisia vaatimuksia on syytä peilata laitteiston ominaisuuksiin. Laitteistoakin tärkeämpi sovelluksen toteutustapaa rajoittava tekijä on olemassa oleva sovellusympäristö, joka tarjoaa sovelluksen kehittämisessä käytettävän käyttöjärjestelmän, käännösympäristön, ohjelmointirajapinnat ja muut sovellukset ja niiden rajapinnat. Myös sovellusympäristön yleisiä suunnitteluperiaatteita on syytä pyrkiä noudattamaan, vaikka poikkeustilanteissa myös periaatteista joustaminen on mahdollista. Suunnitteluperiaatteisiin kuuluu muun muassa käytettäväksi suositeltu yksiprosessinen ja säikeetön prosessimalli.

Yksi ohjelmiston toteutukseen liittyvä seikka, joka ei suoranaisesti liity toteutuksen kohdeympäristöön, on ohjelmistokoodin tallennukseen, ylläpitoon ja muutosten hallintaan käytettävä versionhallintamenetelmä. Iris 440 –laitteen sovellusympäristön lähdekoodipuun versionhallintaa käsitellään tämän luvun lopussa.

3.1 Laitteisto

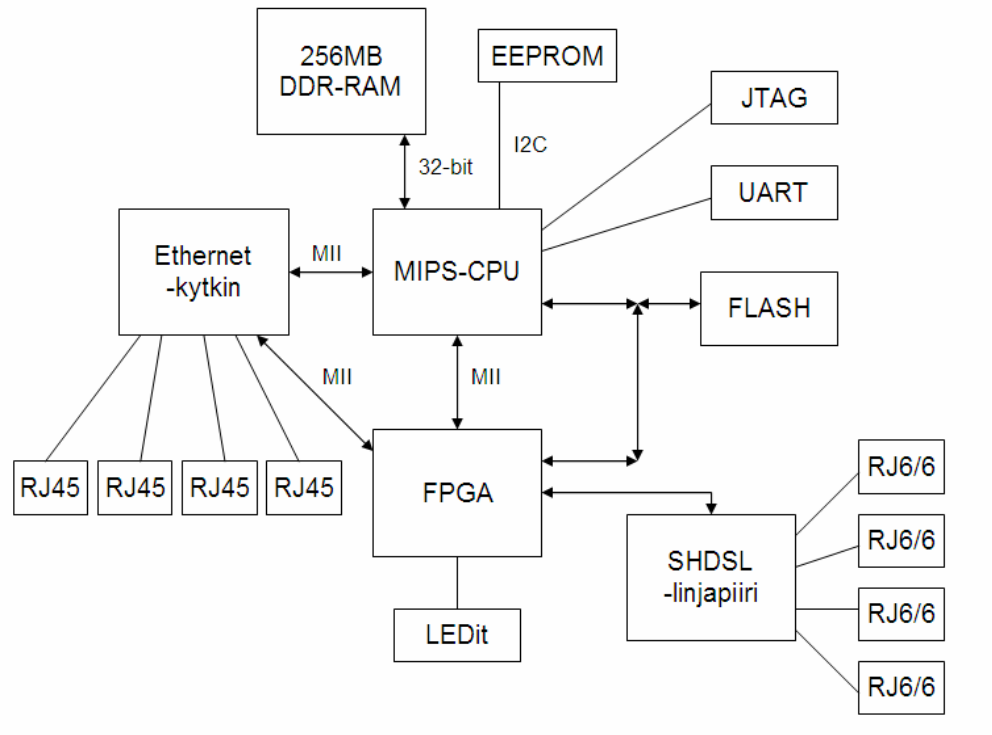
Iris 440 –päätelaitteen toiminnallisuus toteutetaan erilaisilla ohjainpiireillä ja itse prosessorilla. Laitteen keskeisimmät tehtävät ovat käyttöjärjestelmän ajaminen ja dataliikenteen vastaanottaminen ja välittäminen oikealle linjalle sille sopivassa muodossa. Tämä kohta perustuu lähteeseen [DC06].

Iris 440 –laite on asiakaspään- ja *point-to-point* –yhteyksiin tarkoitettu SHDSL-päätelaite, joka tarjoaa asiakkaalle internetyhteyden symmetrisen ja nopean SHDSL-linjan avulla. Laite näkyy kuvassa 3-1. Laite sisältää neljä ethernet-liitäntää ja sarjaporttiliitännän hallintayhteyttä varten. Asiakasyhteyden maksiminopeus on 22,8Mbps ja ethernet-liitäntöjen 100Mbps.



Kuva 3-1. Iris 440 –päätelaite.

Laitteen tärkeimmät osat ovat prosessori, FPGA-piiri sekä ethernet- ja SHDSL-liitännöistä vastaava kytkin ja linjapiiri. Kuvassa 3-2 on yksinkertaistettu lohkokaavio laitteen eri osista ja niiden välisistä liitännöistä.



Kuva 3-2. Iris 440 –laitteen HW-arkkitehtuuri.

Laitteen ethernet- ja SHDSL-liitäntöjen välillä kulkeva liikenne ohjataan aina laitteen prosessorin kautta. Lisäksi datapolkuun kuuluu FPGA-piiri, jonka yhtenä tehtävänä on toimia sovittimena prosessorin ja SHDSL-linjapiirin välillä.

CPU

Laitteen CPU eli keskusprosessori on 32-bittinen ja toimii MIPS-käskykannalla. Se toimii 300MHz kellotaajuudella ja sisältää 16 kilotavua välimuistia. Prosessori sisältää myös DMA- ja PCI-ohjaimen sekä kaksi kappaletta MII-liitäntöjä. Muita liitäntöjä ovat UART, I2C, SPI ja JTAG. UART-liitäntää käytetään laitteessa ohjausväylänä ja I2C-väylää EEPROM-muistin käsittelyyn.

Muistit

Laite sisältää DDR-keskusmuistia, jossa ajetaan ohjelmistokoodi ja joka toimii datan välivarastona. Muistia on yhteensä 256 megatavua, ja se on jaettu kahdelle 32-bittisellä muistiväylällä toimivalle piirille. Pysyvää muistia varten laite sisältää kapasiteeltiltaan 64 megatavun suuruisen flash-muistin. Tähän tallennetaan muun muassa bootloader, Linux-levykuva ja asetustiedostot. Lisäksi laite sisältää pienen EEPROM-muistipiirin, jota käytetään sähköisten tunnisteiden tallennuspaikkana.

Ethernet-kytkin

Laitteessa olevia ethernet-liitäntöjä varten tarvitaan kytkin, joka yhdistää liitännät laitteen prosessorille. Tämä piiri toimii siis haaroittimena neljän RJ45-liitännän ja prosessorin MII-liitännän välillä.

SHDSL-linjapiiri

Ethernet-liitäntöjen lisäksi myös SHDSL-linjoja varten tarvitaan oma linjapiirinsä, joka jakaa FPGA:lta tulevan datavirran omille linjoilleen. Iris 440 sisältää neljä SHDSL-linjaa, jotka on yhdistetty yhdeksi nopeaksi käyttäjälle näkyväksi modeemilinjaksi.

FPGA

FPGA-piirin tarkoituksena on välittää prosessorilta tuleva ethernet-pakettivirta ja kontrollikomennot SHDSL-linjoille sopivaksi datavirraksi ja päinvastoin. Lisäksi FPGA toimii LED-ohjaimena muille kuin ethernet-LED:eille.

3.2 Sovellusympäristö

Iris 440 –laitteen sovellusympäristöllä tarkoitetaan sen käyttöjärjestelmää ja sillä ajettavia prosesseja sekä erilaisia työkaluja. Prosesseja ovat muun muassa laitteen ominaisuuksien hallinnasta ja ympäristön kanssa kommunikoinnista vastaavat prosessimoduulit. Työkaluja ovat Unixin perustyökalut ja erilaiset kääntämiseen ja binäärien hallintaan käytettävät sovellukset. Tässä kohdassa käsitellään sovellusympäristöä projektin kannalta oleellisista näkökulmista. Erityisesti käytettävien ohjelmointirajapintojen ja prosessimoduulien toimintaperiaatteiden tarkastelu on tärkeää, koska ne vaikuttavat oleellisesti myös web-hallintasovelluksen suunnitteluun ja toteutukseen.

3.2.1 Käyttöjärjestelmä

Iris 440 –laitteen käyttöjärjestelmä koostuu Linux-ytimestä ja sulautetuille järjestelmille tarkoitetun BusyBox-ohjelman [Per96] tarjoamista Unix-työkaluista. Lisäksi laitteessa toimii lukuisa määrä erilaisia prosessimoduuleja, joista kukin vastaa jostain laitteen hallintaa tai kommunikointia koskevasta tehtävästä. Prosessimoduuleja ovat esimerkiksi laitteen siltojen tai verkkoliitäntöjen hallintarajapinnan tarjoavat moduulit sekä komentorivi- tai HTTP-yhteyden kautta käyttäjän kanssa neuvottelevat hallintamoduulit.

Ydin

Iris 440 –laitteen käyttöjärjestelmän ytimenä toimii muokattu versio Linux-kernelin versiosta 2.4.20. Ydin on Linux-järjestelmän keskeisin ohjelmistokomponentti, ja sen ominaisuudet määräävät pitkälti myös koko järjestelmän mahdollisuudet [Yag03, s. 156]. Ytimen tehtävä on kommunikoida järjestelmän laitteiston kanssa ja tarjota sovelluksille standardi rajapinta erilaisten laitteiden hyödyntämiseksi. Ydin huolehtii myös järjestelmässä ajettavien sovellusten muistinhallinnasta ja siitä, että jokainen sovellus saa prosessorilta tarvitsemansa määrän laskenta-aikaa.

Erilaisia laiterajapintoja käyttävät laiteajurit voidaan sisällyttää ytimeen joko sisäänrakennettuna tai erillisinä moduuleina. Yksi Linux-ytimen vahvuuksia onkin ytimen modulaarisuus, jolloin moduulit voidaan ladata tai sulkea tarpeen mukaan, ja keskusmuistissa olevan käyttöjärjestelmän ytimen koko pysyy pienenä. Tämä on tärkeää etenkin sulautetulle järjestelmälle. Koska sulautetun järjestelmän muistikapasiteetti on rajoittunut, on ytimestä myös syytä karsia pois käyttämättömäksi jäävät osat. Linux-ytimen sisältämät osat selviävät sen kääntämiseen käytetystä *.config*-tiedostosta, johon on Iris 440 –laitteessa sisällytetty vain välttämättömimmät osat. Suurin osa laitekohtaisista ajureista käännetään erikseen kernel-moduuleiksi lähinnä siitä syystä, että niiden harvinaisuuden tai suljetun koodin vuoksi Linuxin oletusydin ei niitä tarjoa.

Prosessimoduulit

Prosessimoduulit ovat tiettyyn tehtävään suunniteltuja ohjelmia, jotka tarjoavat oman rajapintansa muun sovellusympäristön tai ulkomaailman käytettäväksi. Sulautetulle järjestelmälle on ominaista, että siinä toimivat prosessit ovat reaaliaikaisia, jolloin niiden toteutus poikkeaa perinteisistä järjestelmistä muun muassa siihen liittyvien ajastusvaatimusten ja rinnakkaisuuden hallinnan osalta [HaMä98, s. 306]. Järjestelmän käsiteltäväksi tulevat syötteet ja tietovirrat eivät ole ennustettavissa, mikä edellyttää tiettyjä erityisvaatimusta perinteisiin eräpohjaisiin sovelluksiin verrattuna. Myös reaaliaikajärjestelmässä voi olla erätyyppisiä syötteitä käsitteleviä osia, ja niitä kutsutaan passiivisiksi moduuleiksi.

Iris 440 –laitteen sovellusympäristössä reaaliaikaisuuden asettamat haasteet on ratkaistu prosessimoduulien keskinäisellä kommunikoinnilla ja niiden asynkronisella toimintatavalla. Prosessien välinen kommunikaatio on toteutettu IPC (*Inter-process Communication*) –rajapinnan avulla, jonka toteuttaa erillinen IPC-moduuli. IPC-rajapinnan avulla prosessit voivat välittää toisilleen dataa ja pyyntöjä, ja sen avulla

voidaan huolehtia prosessien keskinäisestä synkronoinnista. Asynkroninen toiminta tarkoittaa tässä tapauksessa sitä, että kukin moduuli rekisteröi itsensä järjestelmän toimintaa ohjaavalle valvontamoduulille, jolle moduuli voi ilmoittaa aktiivisuutensa esimerkiksi I/O-operaation yhteydessä. Moduuli voi muuttua aktiiviseksi esimerkiksi käyttäjältä tulevan syötteen saapuessa.

Moduulien asynkronisuuden tarjoama etu on se, laskentatehoa ei kulu hukkaan yksittäisen moduulin odottaessa jonkin toiminnon tapahtumista, vaan laskenta-aika voidaan sillä aikaa käyttää jossain muualla ja moduuli ilmoittaa aktiivisuudestaan tarpeen mukaan. Moduulien rekisteröityminen valvontamoduulille toteutetaan erityisen luokan avulla, josta moduuli periyttää itselleen oman kuuntelijaluokkansa. Tämän luokan on toteutettava tietty metodi, jota valvontamoduuli kutsuu, kun moduuli on aktiivinen. Tämä metodi käynnistää moduulin toteuttaman toiminnon suorittamisen, mikä voi sisältää esimerkiksi I/O-syötteen lukemisen ja käsittelyn. Kuuntelijaolion rekisteröityminen valvontamoduulille tapahtuu sen tarjoamalla `register_fd()`-metodilla, jolle kuuntelija antaa syötteenä valvottavan tiedostokuvaimen ja itsensä. Valvontamoduulissa kukin moduuli kartoitetaan sitä vastaavaan tiedostokuvaimeen, jonka aktiivisuutta voidaan tarkastella `select()`-järjestelmäkutsun avulla. Tiedostokuvaimen aktiivisuuden perusteella voidaan käynnistää sitä vastaavan kuuntelijaolion suoritusmetodi ja sitä kautta suorittaa moduulin tarjoama toiminto.

Kommunikoinnista laitteen resurssien kanssa vastaa valvontamoduulin ohjaaman vuorottelun johdosta vain yksi moduuli kerrallaan. Tällöin vältytään esimerkiksi kahdelta eri hallintaprosessilta tulevien samanaikaisten hallintapyyntöjen poissulkemisongelmalta, koska tiettyä hallintaobjektia voi käsitellä vain yksi prosessi kerrallaan. Tämä vähentää prosessien välisen kommunikaation tarvetta ja ennaltaehkäisee monimutkaisia rinnakkaisen ohjelmoinnin ongelmia. Asioiden yksinkertaistamiseksi yksittäisten moduulien käyttämäksi prosessimalliksi on ohjeistettu yksiprosessinen ja tapahtumapohjainen, ei mielellään säikeillä toimiva. Useampaa prosessia tai säiettä käytettäessä poissulkemisongelmista olisi huolehdittava myös yksittäisen moduulin sisällä. Eri prosessimallien etuja ja haittoja käsiteltiin kohdassa 2.3.

3.2.2 Config API

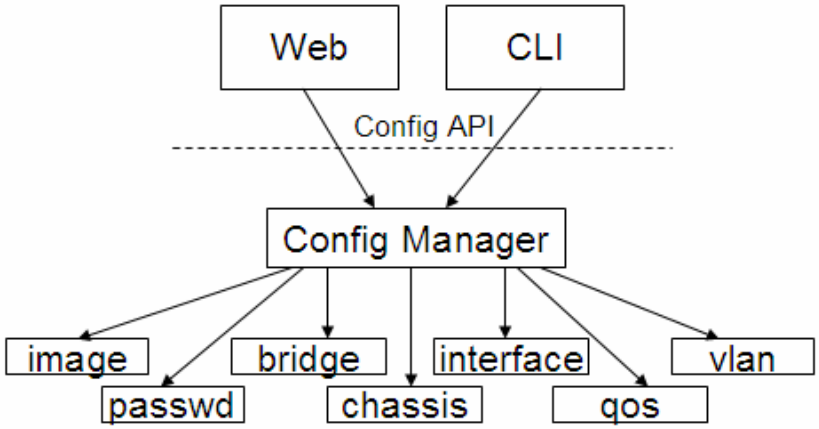
Config API on Design Combus Oy:n alunperin Iris 800 DSLAM (*Digital Subscriber Line Access Multiplexer*) –laitteelle luoma hallintarajapinta, jolla voidaan lukea ja muokata laitteen asetuksia. Rajapinta sisältää metodit eri hallintaobjektien lisäämiseen, päivittämiseen, poistamiseen ja lukemiseen. Näitä kutsuja voidaan käyttää laitteen hallintaan hallintasovelluksen avulla. Iris 800 –laitteen hallinta tapahtuu toistaiseksi ainoastaan CLI-sovelluksella, mutta Iris 440 –päätelaitteelle se on tarkoitus mahdollistaa myös web-pohjaisen hallintasovelluksen avulla.

CLI- ja web-hallintasovellukset ovat laitteen hallintaan tarkoitettuja rinnakkaisia käyttöliittymiä, joista kumpikin sisältää oman logiikkansa syötteiden ja vasteiden välittämiseen käyttäjän ja Config Managerin välillä. Uudelleenkäytön kannalta hallintasovelluksista on järkevintä tehdä keskenään mahdollisimman yhtenevät, vaikkakin terminaalipohjaisen ja web-käyttöliittymän erilaisesta luonteesta johtuen monet sovelluksen elementit poikkeavat toisistaan jo suunnitteluvaiheessa.

Hallintaobjektit ja kutsulogiikka

Laitteella sijaitsevat hallittavat asiat näkyvät sovellukselle hallintaobjekteina. Hallintaobjektit ovat laitteen tarkasteltavia tai muokattavia asioita, kuten silta, verkkoliitäntä tai käytettävän DNS-palvelimen IP-osoite. Jokaisella objektilla on tietty määrä attribuutteja, joista oleellisimpia ovat objektin identifiointiin käytettävät avainattribuutit. Jotkin objektit eivät sisällä lainkaan avainattribuutteja, joka tarkoittaa käytännössä sitä, että tällaisia objekteja voi olla enintään yksi.

Jokaiselle hallintaobjektille toteutetaan *home*, jossa määritetään suoritettava tapahtuma kun jotakin objektin metodia kutsutaan. Erilaisia metodeja ovat *add, update, get_first, get_next* ja *delete*. Jokainen *home* rekisteröidään Config Manageriin, jota käsitellään API-metodien avulla. *Home*-rajapintojen ja niitä vastaavien *manager*-sovellusten käsittelyä Config API-rajapinnan kautta havainnollistetaan kuvassa 3-3. Vaikka rajapinnan metodit ja niiden syötteet ja vasteet ovat pääsääntöisesti kaikille objekteille samat, niin niiden toteutustavat vaihtelevat erilaisista laitteisto- ja ajurirajapinnoista johtuen. Tästä syystä *homet* pitää rakentaa objektin erityispiirteet huomioonottaen. *Home*-toteutusten hallinnasta vastaa *home factory*, joka toimii objektien eräänlaisena abstraktina rajapintana, jota hallintasovellus voi käyttää datan manipulointiin. *Home factory* ohjaa saamansa pyynnön oikealle *homelle*, jolloin hallintaobjekteja voidaan käsitellä samoilla metodeilla objektityypistä riippumatta.



Kuva 3-3. Manager-sovellusten käsittely hallintasovelluksen avulla.

Edellä kuvatusta mekaniikasta johtuen ylemmän tason hallintasovelluksen ei tarvitse olla tietoinen Config Manageriin rekisteröidyistä *homeista*, vaan sen lähettämät pyynnöt voivat olla mielivaltaisia. Config API:lle kohdistettujen kutsujen vastaanotosta vastaa *dispatcher*, joka ohjaa ne oikealle *home factoryyn* rekisteröidylle *homelle*, jolloin kutsu kohdistuu haluttuun objektiin. Täten itse hallintasovelluksen huoleksi jää vain vasteen arvioiminen, kun metodien kohdistaminen oikeaan objektiin oikealla tavalla jää Config Managerin ja objektia vastaavan *homen* huoleksi.

Objektien parametrisointi

Käyttäjäystävällinen hallintasovellus tarvitsee vaadittavan määrän käyttäjälle esitettävää metatietoa erilaisista hallintaobjekteista. Hallintaobjektien perusominaisuuksia ja niiden metatietoa on tarpeellista pystyä täydentämään yhtenäisellä tavalla. Tämän ratkaisuna käytetään def-tiedostoja, jotka ovat yksinkertainen lista attribuutteja, joiden määreinä ovat nimi, tietotyyppi, oletusarvo, pakollisuus ja rajoitukset. Kuvassa 3-4 näkyy katkelma objektin *i24_interface* def-tiedostosta, jossa määritellään laitteen verkkoliitännän hallinta-attribuutit. Esimerkiksi *type*-attribuutti on tietotyypiltään *string*, oletusarvoltaan tyhjä ja pakollinen. Mahdolliset syötteet ovat *ethernet* tai *shdsl*.

```
name: i24_interface
attr: type              string  ""   trueset  ("ethernet|shdsl")
attr: card              int     1    true     range(1,1)
attr: unit              int     0    true     range(1,4)
```

Kuva 3-4. Määritys i24_interface-objektille.

Def-tiedostoihin määritellyt hallintaobjektit ja niiden avulla luotavat *homet* tarjoavat mahdollisuuden hallintasovelluksien geneeriselle toteutukselle. Kun jokainen hallintaobjekti on määritelty yhtenäisellä syntaksilla, voidaan niitä myös käsitellä yhtenäisellä tavalla. Hallintasovelluksen näkökulmasta uuden objektin luominen tarkoittaa yksinkertaisimmillaan uuden objektin def-tiedoston luomista, jonka perusteella generoidaan rajapinta sille määritettyjen arvojen käsittelyyn. Lisäksi objektien muutosten ylläpitäminen on helppoa, koska jokainen muutos, esimerkiksi attribuutin lisääminen, tehdään vain yhteen paikkaan.

3.2.3 CLI-hallintasovellus

Iris 440 –laitteessa toimii Iris 800 DSLAM-laitteen tavoin CLI-hallintasovellus, joka pohjautuu Iris 800:n CLI:n ohjelmakoodiin. Vaikka tämän työn käsittelemä web-hallintasovellus poikkeaa toiminnaltaan CLI:stä, niin siinä voidaan hyödyntää joitakin samoja suunnitteluperiaatteita. Tästä syystä myös CLI:n toiminnan pintapuolinen tarkastelu on paikallaan.

Iris 440:n CLI:hin pääsee käsiksi kirjautumalla laitteeseen *admin*-tunnuksilla, jolloin CLI-sovellus käynnistyy automaattisesti. CLI:hin voi kirjautua myös *monitor*-tunnuksilla, jotka on tarkoitettu laitteen valvontaa varten. CLI-hallintasovellus noudattaa vastaavan tyyppisten sovellusten yleistä toimintatapaa, joista yksi esimerkki on Cisco IOS. Sen ominaispiirteitä ovat hallintaobjektikohtaiset konfigurointitilat ja kumoavina käskyinä toimivat *no*-komennot, joiden avulla esimerkiksi objektin poistaminen tapahtuu.

Jokaista CLI:n komentotyyppiä, esimerkiksi objektin lisäystä, poistoa tai tarkastelua vastaa oma C++-luokkansa, ja eri CLI-komennot ovat näiden luokkien olioita. Komennot luodaan CLI-sovelluksen käynnistyksen yhteydessä ja niitä voidaan määritellä tarvittaessa lisää. Komentojen syntaksi määritellään parametreinä olion

luomisen yhteydessä. Kuvassa 3-5 on esimerkki komennosta, jossa luodaan uusi *add*-komento, jolla voidaan määrittää laitteelle uusi silta eli luoda uusi *bridge*-objekti.

```
new CCliAddCommand(pList->get_object_type("bridge"), ADMIN,
                   ENABLE_MODE_TREE,
                   BRIDGE_CONFIG_MODE, "bridge!Create a bridge! "
                   "name!Bridge name! "
                   "$brname=STR!Bridge name!")
```

Kuva 3-5. Esimerkki komentomäärittelystä.

Komento luo uuden `CCliAddCommand`-olion, joka määrittelee komennon, jonka syntaksi on muotoa *bridge name [nimi]*. Komento toimii *admin*-tilassa oleville käyttäjille, ja sille määritellään myös konfiguraatiotilan indeksi (*ENABLE_MODE_TREE)* ja konfiguraatiotila komennon antamisen jälkeen (*BRIDGE_CONFIG_MODE*). Täten komento on käytettävissä sovelluksen päätilassa, ja komennon antamisen jälkeen CLI menee lisätyn sillan konfiguraatiotilaan. Olion luonnin yhteydessä määritetään myös kullekin parametrille aputeksti, joka toimii ohjeena käyttäjälle. Aputeksti ilmestyy CLI:ssä näkyviin TAB-painalluksella tai kysymysmerkillä.

Koska eri objektityyppejä koskevat komennot luodaan samoilla geneerisillä luokilla, saadaan vaadittavan ohjelmakoodin määrää pienennettyä verrattuna siihen, että jokaisen komennon toteuttava logiikka luotaisiin erikseen.

Komentoluokkien toiminnallisuus vastaa vuorovaikutuksesta Config API:n eli laitteen konfiguraatiorajapinnan kanssa. Tämä tarkoittaa *add*-luokan tapauksessa objektin *homelle* suunnattua *add*-kutsua määrätyillä parametrin arvoilla kuvan 3-6 mukaisesti. Tallennettavan objektin parametrit asetetaan käyttävän antamiksi `SetAttrValues()`-metodin avulla.

```
SetAttrValues(pTree, pParser, pObj, pModule);
pHome->add(ctx, *pObj);
```

Kuva 3-6. Objektin tallennus Config API:n kautta.

Add- eli lisäyskomennon lisäksi muita CLI-komentoluokkia ovat muun muassa poisto-, päivitys-, näyttö-, resetointi- ja asetustiedoston tallennuskomentoluokka. Mitä pienemmällä määrällä luokkia sovellus toimii, sitä vähemmän ohjelmakoodia CLI:n toteuttaminen käytännössä vaatii. Toisaalta komentoluokkien vähentyessä objektikohtaisten ominaisuuksien huomioiminen ja siten myös komentojen käyttäjäystävällisyys vähenee. Tästä syystä luokkien määrän oikeanlainen arviointi on tasapainoilua työmäärän minimoinnin ja riittävän käytettävyyden välillä. Sama käy ilmi myös web-hallintasovelluksen suunnittelussa.

Komentoluokkien, niiden toiminnallisuuden ja komentomäärittelyjen lisäksi CLI vaatii vielä sovelluksen pääosan, jonka tehtävänä on kuunnella CLI:lle tulevia signaaleja

ohjata niiden käsittely sovelluslogiikalle. Tämä osa huolehtii myös valvontamoduulille rekisteröitymisestä kohdassa 3.2.1 esitetyllä tavalla.

3.2.4 Käännösympäristö

Sekä perinteisten että sulautettujen ohjelmistojen suunnittelijat tarvitsevat ohjelmistokoodin kääntämiseen erilaisia kääntäjiä, linkittäjiä ja tulkkeja. Sulautetun ohjelmiston käännöstyökalut poikkeavat perinteisistä siten, että ohjelmisto käännetään yleensä erilaisessa ympäristössä kuin missä lopullista binääriä ajetaan [Yag03, s. 109]. Tämän projektin kohdeympäristö toimii MIPS-prosessorilla, joten tuotettavan binäärin pitää olla MIPS-prosessorin ymmärtämää konekieltä. Koska kehitykseen käytettävät työasemat ovat x86-pohjaisia, tarvitaan ristiinkääntämistä, mikä tarkoittaa ohjelmakoodin kääntämistä kohdeympäristölle tarkoitetulla kääntäjällä ja kohdeympäristön tukemien binäärityökalujen ja kirjastojen käyttämistä.

Iris 440 –laitteen sovellusympäristö koostuu alikohdassa 3.2.1 esitetyllä tavalla käyttöjärjestelmästä ja erilaisista laitteen ja ympäristön kanssa kommunikoivista prosessimoduuleista. Kunkin moduulin tarvitsema ohjelmakoodi sijaitsee sovellusympäristön koodipuussa omassa kansiossaan. Esimerkiksi toteutettavan web-hallintasovelluksen ohjelmistokoodi sijoitetaan kansioon *dslam/src/dcombus/web/*. Koodipuu tarkoittaa tässä tapauksessa hakemistorakennetta, joka pitää sisällään kaiken Iris 440 –laitteen tarvitseman ohjelmakoodin, käännöstiedostot sekä valmiiksi käännetyt sovellukset ja kirjastot.

Iris 440 –laitteen ohjelmistot käännetään GNU-projektin *make*-komentorivityökalulla [FSF97], jolle voidaan määrittää käännettävät ohjelman osat työkalun käyttämään *Makefile*-tiedostoon. Tiedostoon määritetään kääntämisessä tarvittavat työkalut, asetukset ja tiedostot, jolloin itse kääntäminen tapahtuu yksinkertaisilla komentorivikomennoilla, ja käyttäjän ei tarvitse huolehtia kääntämisen yksityiskohdista. Esimerkiksi kääntäjän ja linkittäjän asetukset voidaan määrittää muuttujiin, jolloin asetusten muokkaaminen helpottuu. *Makefile*-tiedostoja voidaan myös ketjuttaa, eli hakemistorakenteen jokainen hakemisto voi sisältää oman käännöstiedostonsa, jota kutsutaan automaattisesti juurihakemistosta tehdyn käännöskutsun yhteydessä, kunhan alihakemisto määritetään juurihakemiston käännöstiedostoon. Kääntäjänä toimii *mipsel-linux-gcc*, joka on MIPS-alustalle suunniteltu versio GNU-projektin GCC-kääntäjästä.

Jokaisen moduulin juurihakemisto sisältää oman *Makefile*-tiedostonsa, joka on yksinkertainen ja sisältää vain kaksi kohdetta: *all* ja *clean*. Kohde *all* tarkoittaa moduulin kääntämistä ja *clean* sen poistamista. Esimerkki tiedoston sisällöstä on esitelty kuvassa 3-7. Kohteella tarkoitetaan *make*-työkalulle määritettävää kohdeparametria.

```
all:
        make –C .. build-web
clean:
        make –C .. clean-web
```

Kuva 3-7. Web-prosessimoduulin Makefile-käännöstiedosto.

Kuvassa 3-7 käytetyt *build-web-* ja *clean-web*-komennot ovat kohteita kuten *all* ja *clean*, ja niiden sisältö on määritelty *module.mk*-tiedostossa. Tämä tiedosto on osa *Makefile*-tiedostoa, joka on eristetty omaksi tiedostokseen käyttäjän ja kääntäjän tarvitsemien kohteiden erottamiseksi toisistaan. *Module.mk*-tiedosto sisältää tiedot käännettäväksi haluttavista tiedostoista, alihakemistoista, ulostuloista, riippuvuuksista ja asetuksista. Moduulikohtaisten käännöstiedostojen yhteydessä on pyrittävä minimoimaan kääntäjän asetusten määrittämistä itse, koska mahdollisimman suuri osa asetuksista on pystyttävä asettamaan lähdekoodipuun juurihakemistosta. Tällöin esimerkiksi lähdekoodin kääntäminen eri kohdealustoille helpottuu, koska käännösasetusten muutokset on tarpeen määrittää vain yhteen *Makefile*-tiedostoon.

Jokainen moduulin lähdekoodia sisältävä alihakemisto sisältää vielä oman *module.mk*-tiedostonsa, johon on listattu alihakemiston sisältämät tiedostot. Nämä alihakemistojen *module.mk*-tiedostot, juurihakemiston *module.mk*-tiedosto ja itse *Makefile*-tiedosto muodostavat yhdessä varsinaisen käännöstiedoston, jonka perusteella käyttäjän *make*-työkalulle antamat *all*- ja *clean*-komennot suoritetaan. Todellisuudessa käyttäjän tarvitsee antaa *make all* –käsky vain koko sovellusympäristön lähdekoodipuun juurihakemistoon, jolloin jokaisen alihakemiston sisältämät käännöstiedostot suoritetaan ketjutetusti.

3.2.5 Versionhallinta

Tässä luvussa käsitellään ohjelmistojen versionhallintaa yleisesti [Mas05] ja tässä projektissa käytettyjen menetelmien näkökulmasta.

Tehokkaaseen ja nykyaikaiseen ohjelmistonkehitykseen kuuluu tärkeänä osana versionhallinta. Versionhallintana voidaan pitää mitä tahansa ohjelmiston kehitysasteesta kertovan metatiedon ylläpitämistä: yksinkertaisimmillaan se voi olla tekstitiedostosta löytyvä tai ohjelman käynnistyessä ilmenevä versionumero. Pelkkä versionumeron ylläpitäminen ei kuitenkaan useimmiten riitä, vaan tarvitaan jokin tapa ylläpitää myös ohjelmistossa tapahtuvia muutoksia ja niiden ajankohtia, ja mahdollisesti tarjota pääsy lähdekoodin edellisiin versioihin. Käsin ylläpidettävä lista muutoksista voi riittää pienelle projektille, mutta ongelmia syntyy jos samaa ohjelmistoa kehittää useampi kuin yksi henkilö. Yleensä edellä mainitut ongelmat ratkeavat keskitettyä versionhallintaa käyttämällä. Se mahdollistaa versiohistorian ylläpitämisen lisäksi yleensä edellisten versioiden lähdekoodin noutamisen ja muutosten täydentämisen selkokielisellä selityksellä. Nykyaikaisten työkalujen tarjoamalla keskitetyllä versionhallinnalla suurenkin ohjelmiston muutokset on helppo pitää koko kehitysryhmän käsiteltävissä ilman pelkoa päällekkäisistä tai toisensa sekoittavista muutoksista. Myös kehityksessä kohdattujen ongelmien ratkaiseminen on helpompaa, kun ohjelmistokoodiin tehtyjen muutosten ja niiden ajankohtien tarkka historia on saatavilla.

Tässä projektissa käytettiin versionhallintaohjelmistoa nimeltä Subversion [CN01]. Versionhallinnassa ylläpidetään Iris 440 –laitteen lähdekoodipuuta useassa eri kehityshaarassa, joita ovat kehitykseen käytettävä *mainline*-haara ja vakaita lähdekoodiversioita ylläpitävät *release*-haarat. Lisäksi versionhallinnasta löytyy *private*-

haara, jossa käyttäjät voivat ylläpitää omaa, ei julkaistavaksi tarkoitettua koodia ja erilaisia testejä.

Tarkoitukseen löytyy myös muita toteutuksia, joista osa on maksullisia ja osa ilmaisia. Toinen suosittu, niin ikään avoin versionhallintasovellus on CVS (*Concurrent Versions System*) [Gru86]. Tässä alikohdassa perehdytään peruskäsitteiden lisäksi lähinnä Subversionilla tapahtuvaan versionhallintaan, koska se on tässä projektissa pääasiallisesti käytössä. CVS ja Subversion ovat toiminnaltaan samankaltaisia, ja Subversionia pidetään yleisesti CVS:n jatkokehitettynä versiona. Subversion on edeltäjäänsä kehittyneempi esimerkiksi transaktioiden toiminnan suhteen, jolloin tiedostoja tallennettaessa käyttäjä voi varmistua siitä, että muutosten tallennus ei jää keskeneräiseksi. CVS:ää käytettäessä saattaa käydä niin, että tallennuksen keskeytyessä esimerkiksi tiedonsiirtovirheen vuoksi vain osa halutuista muutoksista jää voimaan.

Peruskäsitteet

Versionhallinnan keskeisin käsite on tiedon säilytyspaikka (*repository*), joka tarjoaa keskitetyn paikan pääkopion (*master copy*) säilyttämiseen projektin tiedostojen eri versioista. Säilytyspaikka on palvelinsovellus, jonka on syytä sijaita tarkoitukseen pyhitetyllä palvelinkoneella, joka on luotettava, tehokas ja tietoturvaltaan ajan tasalla. Tietoturvan merkitys nousee erityisasemaan etenkin silloin, kun ohjelmistonkehittäjille halutaan tarjota pääsy versionhallintaan myös internetin kautta esimerkiksi kotoaan. Tiedon tallentamiseen voidaan käyttää tavallisia tiedostoja tai esimerkiksi tietokantaa.

Vaikka säilytyspaikkaa voisi teoriassa käyttää myös perinteisena tiedostopalvelimena, on sen perimmäinen tarkoitus nimenomaan tiedostojen versiointi, jolloin edellytyksenä on että sinne tallennetaan vain ohjelmiston lähdekoodia. Myös muu tieto, kuten tekstimuotoiset dokumentit sopivat versioitavaksi, kunhan ne ovat selkokielisiä. Binääridatan muutoksia ei ole järkevää versioida, koska ne eivät ole johdonmukaisia tai ihmiselle ymmärrettäviä. Myös binääritiedostoja voidaan kuitenkin tallentaa, mikä on tarpeen esimerkiksi kääntämisessä käytettäville tiedostoille tai jos tiedoston lähdekoodia ei ole saatavilla.

Ohjelmiston muokkaamista varten käyttäjä tarvitsee itselleen pääkopiosta tehdyn paikallisen työkopion (*working copy*), jota hän voi vapaasti muokata, kääntää ja ajaa. Oletuksena pääkopiosta haetaan yleensä uusin versio, mutta käyttäjä voi valita myös haluamansa vanhemman version. Paikallinen kopio voi olla kopio koko ohjelmistosta, tai vain sen yksittäinen osa. Suurta ohjelmistoa kehitettäessä on luultavasti helpompaa ottaa kopio vain käsittelyn kohteena olevasta kansiosta tai yksittäisestä tiedostosta. Ohjelmisto saattaa sisältää suuren määrän hakemistoja, mutta hyvin suunniteltu ohjelmisto mahdollistaa sen yksittäisten osien kehittämisen muiden osien häiriintymättä, jolloin käyttäjä voi säästää aikaa ja levytilaa käsittelemällä vain tiettyä hakemistoa.

Kun käyttäjä on tehnyt työkopioonsa haluamansa muutokset ja todennut ne kääntyviksi ja mielellään myös toimiviksi, hän voi tallentaa muutokset säilytyspaikan pääkopioon. Versionhallinta kasvattaa versionumeroa automaattisesti tallennuksen yhteydessä, ja kunkin version muutokset merkitään näkyviin. Tallennettaessa käyttäjältä pyydetään yleensä tehdyistä muutoksista lyhyt selkokielinen selitys, jonka perusteella yksittäisen muutoksen tarkoitus ja ajankohta voidaan myöhemmin paikantaa helpommin kuin

kooditason muutoksia tarkastelemalla. Versionumeroiden ilmaisukykyä voidaan parantaa nimeämällä niitä liitteillä (*tags).*

Säilytyksessä oleva pääkopio voidaan myös haarauttaa. Tämä voi olla hyödyllistä esimerkiksi silloin, jos ohjelmistokehittäjä haluaa tehdä ohjelmistoon laajoja muutoksia, jotka voivat vaikuttaa muiden kehittäjien työhön. Design Combus Oy:n versionhallinnassa käytetään kahta kehityshaaraa, joista toinen on tarkoitettu vakaalle tuotantoversiolle ja toinen epävakaalle, kehitysvaiheessa olevalle versiolle. Haarautettua kopiota voidaan kehittää rinnakkain päähaaran kanssa ja yhdistää siihen tarvittaessa takaisin. Tämä helpottaa huomattavasti tilanteita, joissa käyttäjä haluaa tehdä ohjelmistoon epävarmoja tai radikaaleja kokeiluja, ja käyttää siitä huolimatta versionhallintaa.

Subversion

Subversion on monipuolinen versionhallintaohjelmisto, joka tukee perusominaisuuksien lisäksi muun muassa atomisia tallennuksia, HTTP-protokollaa, BerkeleyDB-tallennusta ja symbolisten linkkien versiointia. Lisäksi kopioinnit ja haaroitukset ovat kevyitä ja myös binääridatan tallennus onnistuu.

Peruskäsitteiden yhteydessä mainittujen vaiheiden toteuttamiseen voidaan käyttää Subversionin tapauksessa joko komentoriville syötettäviä komentoja tai tarkoitukseen luotuja käyttäjäsovelluksia, joilla sama voidaan tehdä helposti graafisen käyttöliittymän avulla. Esimerkki hyvästä käyttäjäsovelluksesta on Windowsilla toimiva TortoiseSVN, joka tarjoaa suoraviivaisen käyttöliittymän Subversion-palvelimen selaamiseen ja muokkaamiseen. Myös komentorivin käyttö on intuitiivista ja hyvien ohjeiden ansioista graafista käyttäjäsovellusta ei välttämättä tarvita.

Käyttäjän useimmin tarvitsemat komennot ovat *add, checkout, commit, copy, move, merge* ja *update.* Komentorivin hyväksymä syntaksi on muotoa *svn komento [asetukset] [argumentit]*. Asetukset ovat komennolle annettavia lisäoptioita, joita tarvitaan esimerkiksi käsiteltävän version määrittämiseen. Argumentit taas ovat useimmille komennoille välttämättömät syötteet, ja ne ovat usein pääteltävissä Unixin perustyökalujen argumenteista. Esimerkiksi kansiota siirrettäessä argumenteiksi tulee kopioitavan kansion lähde- ja kohdesijainti.

Käyttäjä aloittaa ohjelmiston muokkaamisen yleensä *checkout*-komennolla, jolla haetaan haluttu kohde palvelimelta. Komento vaatii syötteekseen kohteen osoitteen. Esimerkiksi komento *svn checkout svn://palvelin/home/svn/repository/projekti* hakee oletuksena uusimman version pääkopiosta, mutta asetuksiksi voidaan antaa myös haluttu versio. Kun käyttäjä on tehnyt ohjelmistoon haluamansa muutokset, hän voi tallentaa ne *commit*-komentoa käyttämällä. Komennon yleisin käyttötapa on kutsua komentoa tallennettavaksi haluttavasta paikallisesta kansiosta tai vaihtoehtoisesti antamalla kansio argumenttina. *Copy*- ja *move*-komennot vastaavat saman nimisiä Unix-käskyjä, eli niillä suoritetaan halutun kohteen siirtäminen tai kopioiminen. Molemmat tarvitsevat argumenteikseen sekä lähde- että kohdepolun. *Copy*-komennolla voidaan kätevästi suorittaa ohjelmiston osien haaroittaminen ja *move*-käskyllä kohteiden uudelleennimeäminen.

Loput mainituista komennoista ja muut Subversionin sisältämät komennot on koottu selityksineen liitteeseen 1.

4 Palvelinsovelluksen valinta ja vaatimukset

Jokainen verkko-ohjelmisto tarvitsee tuekseen palvelinsovelluksen ulkomaailman kanssa kommunikointia varten. Tästä syystä projektin esitutkimusvaiheen yksi tärkeä osa-alue oli sopivan HTTP-palvelinsovelluksen etsiminen, joka toimisi toteutettavan hallintasovelluksen. Yhtenä vaihtoehtona olisi ollut palvelimen toteuttaminen itse, mutta jo pelkästään avoimien HTTP-palvelimien määrä on suuri, joten sopivan valmiin toteutuksen löytyminen tuntui todennäköiseltä. Sopivan palvelinsovelluksen löytämistä vaikeuttivat projektin kohdeympäristön asettamat tekniset rajoitteet ja tietoturvavaatimukset. Lisäksi projektille asetettiin tietyt päämäärät verkko-ohjelmiston ylläpidettävyyden suhteen, joten palvelinsovelluksen tarjoamat sovelluskehitystä tukevat rajapinnat tai kehitysympäristöt olisivat hyödyksi.

Tämä luku käsittelee näiden vaatimuksien huomioimista teknisesti ja suunnittelullisesti erilaisin kriteerein. Luvussa tarkastellaan näiden kriteerien perusteella valittua palvelimien joukkoa, joista poimitaan sopivimmat ratkaisut. Jäljelle jääneisiin kolmeen palvelinsovellukseen tutustutaan luvussa tarkemmin ja niiden kesken valitaan käytettävä palvelinsovellus.

4.1 Jaotteluperusteet

Alustaviksi HTTP-palvelinsovellusten jaotteluperusteiksi otettiin lisenssiehdot, tuetut ohjelmointikielet, prosessimalli, mahdolliset sivupohjamekanismit, sovelluksen koko ja tietoturva. Näistä tärkeimpinä kriteereinä pidettiin lisenssiehtoja, prosessimallia ja tietoturvaa. Erityisen tärkeää on myös sovelluksen pieni koko, koska sovelluksen loppukäyttöympäristö on sulautettu järjestelmä, jolloin käytettävissä oleva muistikapasiteetti on rajoittunut.

Lisenssiehdot

Yksi tärkeä tekijä web-palvelimien kartoituksessa on niiden tarjoamat lisenssiehdot. Ilman mahdollisuutta käyttää yrityksen liikeideaan soveltuvaa lisenssiä ohjelma ei sovellu käyttöön, vaikka sen muut ominaisuudet täyttäisivät asetetut vaatimukset. Sopiva lisenssi on tässä tapauksessa avoin ja maksuton, joka ei vaadi sillä luodun sovelluksen lähdekoodin julkaisemista. Myös maksulliset vaihtoehdot soveltuvat käyttöön, mikäli ne tarjoavat hinnalleen sopivan vastineen.

Käytännössä lisenssin perusteella täytyi jättää pois kaikki GPL-lisenssin alaiset palvelimet, koska GPL-lisenssi on tarttuva ja näin ollen kaikki siihen toteutettavat muutoksen on julkaistava niin ikään GPL:n alaisena. Lisäksi rajapintojen kautta siihen liittyvä ohjelmakoodi on julkaistava. [FSF91]

Myös suljetun ohjelmistokoodin palvelimia pidettiin alustavasti poissuljettuina, koska haluttiin mahdollistaa palvelimen koodin muokkaus tarvittaessa. Tällöin ohjelmisto on joustava ja kohdeympäristön erityispiirteet voidaan ottaa huomioon itse palvelinsovelluksen toteutuksessa. Tämä helpottaa sovelluksen integrointia olemassa olevaan sovellusympäristöön ja voi mahdollistaa myös tehonparannuksia tai helpotusta ohjelmiston siirtämisessä kohdeympäristöstä toiseen.

Käyttötarkoituksiimme parhaat lisenssit ovat BSD- ja MIT-lisenssit tai niiden johdannaiset, jotka sallivat koodin rajoittamattoman käytön. Koodiin voi tehdä vapaasti muutoksia ja käyttöön otettua koodia ei tarvitse julkaista minkään tietyn lisenssin alaisena, vaan sen voi esimerkiksi halutessaan julkaista suljettuna ja siitä voi periä maksun.

Suurin osa maksullisten palvelimien tarjoajista antavat käyttöön myös ohjelmiston lähdekoodin, ja usein niiden lisenssi mahdollistaa omien ohjelmistojen julkaisemisen ainoastaan suljettuna. Myös tällaiset lisenssit sopivat projektin tarkoituksiin, kunhan lisenssiin ei liity mitään käyttöä hankaloittavia valvontamekanismeja ja lisenssin hinta on kohtuullinen palvelimen tarjoamiin ominaisuuksiin nähden. Koska palvelinsovellusta on tarkoitus käyttää laitteessa, jonka vuosittainen tuotantomäärä on useita tuhansia kappaleita, on toivottavaa, etteivät lisenssikustannukset kasva jyrkästi sovelluksen kohdelaitteiden määrän mukaan.

Tuetut ohjelmointikielet

Koska erilaisten palvelimien skaala on laaja, on odotettavissa, että myös tuettuja ohjelmointikieliä on lukuisia. Suurin osa palvelimista tukee CGI- tai FastCGI-rajapintaa, joiden avulla web-palvelin voi käsitellä pyyntöjä ohjelmointikielestä riippumattomien ympäristömuuttujien ja standard-IO:n kautta. Tavallisen CGI:n ongelmana on sen raskaus, koska jokaista pyyntöä käsittelemään luodaan oma prosessinsa, joka käsittelyn päätteeksi tuhotaan. Vaikka FastCGI vähentää tätä ongelmaa, pidettiin tämän projektin kannalta tarkoituksenmukaisimpana etsiä menetelmä, joka tukisi verkko-ohjelmiston kirjoittamiseen käytettävää ohjelmointikieltä suoraan. Koska CGI:n kautta avautuu mahdollisuus muun muassa PHP-, Python-, ja Ruby-kielille kehitettyjen sovelluskehysten käyttöönottoon, otettiin kuitenkin myös sille tarjottu tuki huomioon.

Koska verkko-ohjelmistolla hallittavan Iris 440 –laitteen Config API –rajapinta on toteutettu C++-luokilla, suoraviivaisin ohjelmointikieli hallintasovelluksen toteuttamiseen olisi C++ tai C. Useimmissa tapauksissa tuettu toteutuskieli tarkoittaa myös itse palvelimen toteutuskieltä, joten muilla kielillä, kuten Javalla toteutetut palvelinratkaisut jätettiin kartoituksen ulkopuolelle. Java jätettiin kielivaihtoehtojen ulkopuolelle, koska Java-virtuaalikonetta ei suorituskyky- ja integrointisyistä haluttu asentaa kohdeympäristöön.

Prosessimalli

Kohdassa 2.3 todettujen seikkojen perusteella kohdeympäristön kannalta sopivin prosessimalli palvelinsovellukselle olisi SPED, eli yhden prosessin tapahtumaohjattu sovellus. SPED-tyyppisen palvelimen käytön oletettiin helpottavan sen integrointia olemassa olevaan sovellusympäristöön ja lisäksi se tarjoaisi pienen säästön palvelimen käyttämän muistimäärän suhteen. Myöskään monisäikeisiä sovelluksia ei pidetty mahdottomana vaihtoehtona, mutta valinnassa pyrittiin yksinkertaisimpaan mahdolliseen ratkaisuun ja siten SPED-malliin.

Ainoastaan yhden, säikeettömän prosessin käytöstä voi yhtäaikaisten pyyntöjen ilmaantuessa aiheutua palvelimen vasteaikaan liittyviä ongelmia, vaikka projektin kohdeympäristössä ei voidakaan odottaa käyttäjiltä tulevien pyyntöjen ruuhkaa. Tämä edellyttää palvelimen huolellista toteutusta ja olosuhteiden muutosten varalle myös jonkin vaihtoehtoisen prosessimallin tarjoaminen on eduksi. Monet palvelimet mahdollistavatkin muiden asetusten ohella myös prosessimallin valinnan ympäristön tarpeille sopivaksi.

Sovellusarkkitehtuuri

Koska hallintasovellus haluttiin luoda sen ylläpidettävyyttä ja uudelleenkäyttöä silmällä pitäen, oli sen toteutuksessa käytettävän sovellusarkkitehtuurin syytä tukea jotain ulkoasun ja sisällön toisistaan irrallaan pitävää arkkitehtuuria. Tällaiseen tarkoitukseen sopisivat monet verkko-ohjelmistojen kehittämiseen tarkoitetut PHP-sovelluskehykset, tai esimerkiksi *Ruby on Rails*, joka on Ruby-kielelle kehitetty MVC-arkkitehtuuria noudattava avoin sovelluskehys. Sovelluskehyksiä ja -arkkitehtuureita käsiteltiin tarkemmin kohdassa 2.5.

Jotkin palvelimet, kuten Klone, Seminole ja GoAhead WebServer tarjoavat verkko-ohjelmistojen kehittämiseen oman ohjelmointirajapintansa, joka luetaan kartoituksessa eduksi, jos se on hyödyksi tämän projektin kohdesovelluksen toteuttamisessa. Palvelinkohtaisen ohjelmointirajapinnan hyödyllisyys riippuu paljolti sen tarjoamien toimintojen soveltuvuudesta projektiin. Rajapinnan olisi syytä tarjota joustava menetelmä geneeristen näkymien luomiseen ja helppo liityntä HTTP-rajapintaan, jolloin tiedonsiirtoprotokollien toimintaan ei tarvitsisi kiinnittää huomiota. Vaikka rajapinnan on syytä olla tarpeeksi laaja, voi liian monipuolinen rajapinta olla vaikeaselkoinen ja hankala käyttää. Erityisen paljon merkitystä on ohjelmointirajapinnan toteutuskielellä, koska käytännössä vain C- ja C++-rajapinnat ovat tässä projektissa käyttökelpoisia.

Toivottavaa olisi myös jonkinlaisen sivupohjamekanismin tarjoaminen, jonka avulla voidaan sisällyttää HTML-tyyppiseen sivupohjaan dynaamisesti haettua tietoa. Lisäksi eri konteksteille, kuten yhteystapahtumille ja pyynnöille kuuluvan datan käsitteleminen erillään pitäisi olla mahdollisimman yksinkertaista. Tämä voi tapahtua esimerkiksi tallentamalla eri kontekstien muuttujien omaan näkyvyysalueeseensa, joiden alustamisesta esimerkiksi uuden istunnon yhteydessä huolehtii palvelinohjelma. Tällä tavoin esimerkiksi GoAheadin web-palvelimessa erityyppisten muuttujien käsittely tapahtuu erittäin suoraviivaisesti.

Koko

Palvelimen teho- ja muistivaatimuksen on oltava mahdollisimman pieni, koska kohdelaitteen suoritinteho ja muistikapasiteetti on rajoittunut. Palvelinsovellusten karsinnassa huomioitiin sen koko, arvioituna karkeasti tar-paketin koon perusteella. Tällä tavoin arviointi on epätarkkaa, joten sitä käytettiin vain viitteellisenä arviona. Kuitenkin esimerkiksi Apache- ja Litespeed-palvelimet oli helppo karsia pois jo pelkästään asennuspaketin koon perusteella. Palvelimen muistivaatimuksen rajoittamiseen auttaa sen modulaarisuus, jolloin palvelimen ohjelmakoodiin voidaan sisällyttää vain tarvittavat osat. Modulaarisuuden ansiosta esimerkiksi alikohdassa 4.3.3

esiteltävä Seminole-palvelin pysyy kohtalaisen laajasta ohjelmointirajapinnastaan huolimatta kevyenä.

Tietoturva

Yksi tärkeä jaottelukriteeri oli tietoturva, koska SHDSL-laite on kriittinen osa verkkoa, jolloin se on pystyttävä suojaamaan mahdollisilta hyökkäyksiltä. Riittävän tietoturvan takaamiseen vaaditaan vähintään jokin autentikointimenetelmä, jolla voidaan varmistua siitä, että vain sallitut käyttäjät pääsevät käsiksi hallintasovellukseen. Tähän tarkoitukseen voidaan käyttää esimerkiksi HTTP-protokollan tarjoamia autentikointiskeemoja, joita käsiteltiin alikohdassa 2.4.1.

Varsinkin etäyhteyksiä varten tarvitaan tuki myös tiedonsiirron suojaavalle salausprotokollalle, joten tuki esimerkiksi SSL- tai TLS-protokollalle lasketaan eduksi. Nämä protokollat tarjoavat mahdollisuuden päätepisteiden autentikointiin julkisen avaimen menetelmällä ja tiedon salaukseen symmetrisellä salausavaimella alikohdassa 2.4.2 esitetyllä tavalla.

Autentikoinnin ja tiedonsiirron salauksen lisäksi hallintasovelluksen toteutuksessa on otettava huomioon autorisointi, eli ohjelman käyttäjäkohtaiset hallintaominaisuudet. Autorisointimekanismit ovat kuitenkin palvelinsovelluksesta riippumattomia, joten niitä käsitellään hallintasovelluksen suunnittelun ja toteutuksen yhteydessä luvussa viisi.

4.2 Palvelinratkaisujen kartoitus

Käyttöön otettavan web-palvelimen valinta aloitettiin etsimällä lupaavilta vaikuttavia vaihtoehtoja internetistä. Yksinkertaisimmillaan tämä tapahtui Google-hakukoneen avulla, jolla löytyi runsaasti tietoa web-palvelimista erilaisilla hakusanoilla, kuten *web server*, *http daemon* ja *httpd*. Hakutuloksia pyrittiin painottamaan kevyisiin palvelinratkaisuihin erilaisilla lisämääreillä, kuten *lightweight* ja *embedded.* Hakukriteerit pyrittiin valitsemaan löyhästi, koska tarkoitus oli löytää mahdollisimman suuri osa tarjolla olevista ohjelmistoista, vaikka ne eivät lopullisten vaihtoehtojen joukkoon päätyisikään. Kaikkien valintaperusteiden ei vielä kartoitusta tehdessä voitu olettaa olleen selvillä, joten karsintaa ei haluttu tehdä epävarmojen ominaisuuksien perusteella.

Tietyt esikriteerit täyttäneet palvelinsovellukset koottiin Excel-tiedostoon, jonka sarakkeet perustuivat kohdassa 4.1 esiteltyihin jaotteluperusteisiin. Taulukko palvelinsovelluksista on esitelty liitteessä 2. Kevyillä esikriteereillä karsittiin pois joitakin kaikkein yksinkertaisimpia, ainoastaan staattisen tiedon näyttämisen mahdollistavia palvelimia sekä raskaampia, tehokkaille palvelinkoneille tarkoitettuja palvelimia. Raskaampien palveliminsovellusten mahdollistama yhtäaikaisten käyttäjien lukumäärä ja esitettävän tiedon monipuolisuus ylittäisivät toteutettavan sovelluksen tarkoituksen, koska tarkoituksena on tehdä kevyt, vain muutaman yhtäaikaisen käyttäjän hallintasovellus.

Kaikki tarkasteluun otetut toteutukset eivät olleet kokonaisia palvelimia. Mukana oli myös muutama koodikirjasto, jotka tarjoavat jonkin mahdollisesti projektiin soveltuvan sovelluskehyksen, mutta vaativat tuekseen erillisen palvelinsovelluksen. Toisaalta

päinvastainen esimerkki on libwebserver-kirjasto, jonka tarkoituksena on tarjota perinteiselle sovellukselle web-palvelimen toiminnallisuus. Libwebserverin käyttö vastaa jossain määrin palvelinkohtaisen API-rajapinnan käyttöä, joskin palvelinominaisuuksien toteuttaminen kirjastona saattaa parantaa verkko-ohjelmiston itsenäisyyttä.

Lisenssin perusteella karsittiin pois vain pieni joukko palvelimia, sillä suurin osa niistä oli joko täysin avoimia tai tarjosivat maksullisen lisenssin. Suurin osa GPL-palvelimista oli myös ominaisuuksiltaan vaatimattomia, jolloin oma verkko-ohjelmisto olisi luotava käytännössä CGI-rajapintaa käyttämällä. Tntnet-palvelimen [Mäk04] lisäksi mikään GPL-palvelimista ei tukenut sulautettua tiedostojärjestelmää, jolloin niillä ei olisi suurta etua myöskään sulautettua kohdeympäristöä ajatellen. Sulautetussa tiedostojärjestelmässä verkko-ohjelmiston staattiset tiedostot käännetään samaan binääriin palvelinsovelluksen kanssa. Tällöin palvelimen käsittely helpottuu ja koko yleensä pienenee. Sulautetun tiedostojärjestelmän käytön edellytyksinä on, sitä tarvitaan vain lukemiseen ja että staattisessa tiedossa ei tapahtu muutoksia.

Suurimmasta osasta palvelimia löytyi myös SSL-tuki, joka oletettiin tarpeelliseksi riittävän tietoturvan takaamiseksi. Suurin osa palvelimista tuki myös HTTP-autentikaatiota ja lähes jokainen Unixin oikeudenhallintamenetelmiä tiedostojen suojaukseen. Harvinaisempia tietoturvamenetelmiä olivat Monkey HTTP-daemonin tukema URL- ja IP-perusteinen esto sekä tietoturvaltaan monipuolisimpien Abyss Web Serverin [AT01] ja Zeus Web Serverin [ZT95] tukema epäilyttävien pyyntöjen esto ja DoS (*Denial of Service*) –hyökkäysyritysten tunnistaminen. Kaksi jälkimmäistä palvelinta olivat tosin tarkasteluun pääsyn rajamailla, koska kumpaankaan ei ole saatavilla lähdekoodia ja etenkin Zeus on projektin tarkoituksiin liian raskas.

Palvelinsovellusten suurimmaksi kompastuskiveksi osoittautui tuki SPED-prosessimallille, koska moni palvelin toimi moniprosessisena ja suurin osa monisäikeisenä. Molempaa mallia pidettiin huonona vaihtoehtona, koska mikäli verkko-ohjelmisto haluttaisiin tehdä palvelimen tarjoamalla ohjelmointirajapinnalla, pitäisi myös siinä itsessään huomioida usean prosessin tai säikeen käyttöön liittyvät ongelmat. Muun kuin CGI-tyyppisen verkko-ohjelmiston toteutuksen kannalta yksinkertaisin vaihtoehto olisi selkeästi SPED-tyyppinen palvelin, koska silloin palvelimen integrointi kohdeympäristöön olisi helpointa. SPED-tyyppisiä palvelimia oli tarkasteltavassa ryhmässä kuitenkin vain yhdeksän, joista osa oli kaiken lisäksi GPL-lisenssin alaisia, joten vaihtoehdot olivat vähäisiä. Näistä vaihtoehdoista löydettiin kuitenkin kaksi ominaisuuksiltaan lupaavaa palvelinta: Klone ja Seminole, joita käsitellään kohdassa 4.3. Samassa kohdassa esiteltävä GoAhead WebServer valittiin lähempään tarkasteluun lähinnä vakuuttavien tosielämän sovellusesimerkkiensä asiosta. Lisäksi sen tukemien tekniikoiden määrä oli sulautetuille järjestelmille suunnitellulle palvelinsovellukselle poikkeuksellisen laaja. Tarkasteluun päätyi yhteensä 36 HTTP-palvelinta, joista tässä luvussa mainittujen kriteerien perusteella varteenotettavia vaihtoehtoja oli lopulta seitsemän kappaletta. Kyseiset palvelimet näkyvät taulukossa 4-1. Taittosyistä johtuen osa alkuperäisen taulukon sisältämistä kriteerisarakkeista on jouduttu karsimaan.

Taulukko 4-1. Ominaisuuksiltaan lupaavimmat palvelinsovellukset.

Ohjelma	Lisenssi	CGI	Templ.	Prosessimalli	Koko (kt)	Tietoturva
Klone	GPLv2/maksullinen	ei	on	SPED	5370	TLS/SSL, session
Seminole	$499-$1999	ei	on	SPED	pieni	SSL
AppWeb/ Mbedthis	maksullinen	on	on	single/multi-threaded	6860	Auth, SSL
GoAhead WebServer	täysin avoin	on	on	single/multi-threaded	2590	Auth, SSL
Lighttpd	BSD (muokattu)	on	ei	SPED	3680	Auth, SSL
thttpd	BSD	on	ei	SPED	550	permissions, SSL
Fusion	maksullinen	-	-	SPED/multi-threaded	11	HTTPS server

AppWeb/Mbedthis [Mbe03] on ominaisuuksiltaan lähes vastaava GoAhead WebServerin kanssa, joskin jälkimmäinen on hivenen kehittyneempi. Lighttpd- ja thttpd-palvelimissa kiinnostavimmat ominaisuudet koskivat PHP-kielisen verkko-ohjelmiston integrointia SAPI (*Server Application Programming Interface*) –moduulien avulla. Verkko-ohjelmiston toteuttamista PHP-kielellä pidettiin kuitenkin toissijaisena ratkaisuvaihtoehtona, koska sille toteutettujen sovelluskehysten vaatimukset etenkin muistinkäytön suhteen ovat yleensä tämän projektin vaatimuksia kevyempiä. Myös projektin kohdeympäristön ennalta määrätyn mallin ja sen rajapinnan ansiosta PHP-sovelluskehyksissä usein käytettävää SQL-tietokantaa ei voida hyödyntää.

4.3 Tarkoitukseen parhaiten sopivat ratkaisut

Luvussa aiemmin esiteltyjen jaotteluperusteiden mukaisen karsinnan jälkeen alkuperäisistä palvelinvaihtoehdoista jäi jäljelle lopulta kolme palvelinsovellusta: GoAhead WebServer, Klone ja Seminole. Tässä kohdassa käydään läpi pääpiirteittäin näiden palvelinsovellusten keskeisimmät ominaisuudet ja toimintaperiaatteet sekä niiden hyvät ja huonot puolet tämän projektin kannalta. Sovellusten toimintaa havainnollistavat esimerkit ovat yksinkertaistettuja, ja niitä käytetään lähinnä palvelimen keskeisen idean havainnollistamiseen, ei niiden toiminnan selvittämiseen perinpohjaisesti. Tässä kohdassa on hyödynnetty Klonen, GoAhead WebServerin ja Seminolen teknisiä dokumentteja [KL07], [GA00], [GS06a] ja [GS06b].

4.3.1 GoAhead WebServer

GoAhead WebServer on avoimeen lähdekoodiin perustuva sulautetuille järjestelmille suunniteltu HTTP-palvelin, jota on sovellettu monissa kaupallisissa päätelaitteissa, kuten Telewellin EA501 ADSL-modeemeissa. Lupaavien sovellusesimerkkiensä ansiosta GoAhead WebServer nousi lukuisten muiden palvelimien joukosta yhdeksi varteenotettavimmista palvelinvaihtoehdoista. Sen tärkeimpiä etuja ovat pieni muistivaatimus, tuetut tietoturvamenetelmät ja niiden käyttöön tarjotut rajapinnat sekä pohjautuminen yleisiin tekniikoihin kuten ASP (*Active Server Pages*), CGI ja JavaScript. Näiden tekniikoiden lisäksi GoAhead tarjoaa suunnittelijalle oman

ohjelmointirajapintansa, joka helpottaa dynaamisen sisällön esittämistä verkkosivuilla. Se käyttää tästä konseptista nimitystä GoForms.

Palvelimen asetukset

GoAhead WebServer ei sisällä varsinaista asetustiedostoa, vaan sen asetukset sijaitsevat käyttöjärjestelmäkohtaisen pääohjelmatiedoston alkuosassa. Esimerkiksi Linux-käyttöjärjestelmälle asetukset merkitään tiedostoon *LINUX/main.c*. Sinne merkitään verkko-ohjelmiston hakemisto, tietoturvasalasana, kuunneltava portti ja uuden kuunteluportin haun yrityskerrat kuvassa 4-1 esitellyllä tavalla.

```
1         static char_t                 *rootWeb = T("web");
2         static char_t                 *password = T("");
3         static int                    port = 80;
4         static int                    retries = 5;
```

Kuva 4-1. Palvelimen asetukset main.c-tiedostossa.

Palvelimen tietoturvaominaisuuksien käyttöönotto ei vaadi erillisiä asetuksia, vaan ne ovat verkko-ohjelmiston käytettävissä erilaisten rajapintojen kautta. Tarvittavat funktiot löytyvät *um.h* ja *websda.h* –tiedostoista, joista ensimmäinen tarjoaa palvelimelle käyttäjänhallinnan ja jälkimmäinen *digest*-skeeman mukaisen käyttäjän autentikoinnin.

ASP

Active Server Pages (ASP) on Microsoftin kehittämä tekniikka, jonka avulla voidaan tarjota verkkosivuilla dynaamista sisältöä. Tämä tapahtuu sulauttamalla tavanomaiseen HTML-koodiin skriptejä, jotka generoivat dynaamisen sisällön ennen sivun lähettämistä käyttäjälle. GoAhead käyttää skriptikielenä JavaScriptin kevennettyä variaatiota nimeltä Ejscript. ASP ei kuitenkaan sido suunnittelijaa tiettyyn skriptikieleen, vaan se voidaan vapaasti valita *script*-direktiivin *language*-parametrilla. Yleisimmin käytetään VBScript-, JScript-, JavaScript- tai Perl-kieliä. Skriptit suoritetaan palvelimella, joten selaimen ei tarvitse tukea käytettävää skriptikieltä.

GoAheadin ohjelmointirajapinta tarjoaa mahdollisuuden hyödyntää ASP-sivujen kautta myös C-funktioita. Tähän käytetään `websAspDefine()`-kutsua, joka vaatii parametreikseen käytettävän C-funktion nimen ja sen ”kutsumanimen” ASP-tiedostosta.

GoForms

Tämän projektin kannalta GoAheadin merkittävin ominaisuus on GoForms, joka mahdollistaa verkko-ohjelmiston logiikan eli kontrollin tehokkaan eristämisen käyttäjän havaitsemasta näkymästä. Perinteisen CGI:n tavoin GoForms käyttää kommunikointiin ympäristömuuttujia, joiden kautta se selvittää esimerkiksi yhteysosapuolen osoitteen ja HTTP-pyynnön parametrit. Toisin kuin CGI, GoForms käsittelee jokaisen kutsun samassa prosessissa, jolloin tarvittava muistimäärä ei kasva suuresti kutsumäärän lisääntyessä. Haluttua GoFormia voidaan kutsua selaimella esimerkiksi muodossa *http://palvelin/goform/omaFormi?nimi=Henri&ika=22*, jolloin pyynnön käsittely

ohjautuu *omaFormi*-nimiselle GoFormille. Esimerkki GoFormista näkyy kuvassa 4-2. Käsittelijä kirjoittaa yhteyskahvaan *wp* siltä saadut parametrit *nimi* ja *ikä*, jonka lisäksi se kirjoittaa HTML-tiedoston otsakkeen ja päätteen sekä vastauskoodin OK. Esimerkin GoForm käyttää ainoastaan GoAheadin omia API-kutsuja, mutta vastaanotettuja parametrejä voidaan käyttää myös monipuolisemmin, esimerkiksi hakemalla henkilölle nimen ja iän perusteella tietokantaan tallennettuja lisätietoja.

```
1     void omaFormi(webs_t wp, char_t *path, char_t *query)
2     {
3       websHeader(wp);
4       websWrite(wp, “Nimi: %s <BR>”, websGetVar(wp, “nimi”, “”));
5       websWrite(wp, “Ika: %s”, websGetVar(wp, “ika”, “”));
6       websFooter(wp);
7       websDone(wp, 200);
8     }
```

Kuva 4-2. Esimerkki GoForm-käsittelijästä.

Käsittelijältä saatu vaste on esitelty kuvassa 4-3. Palvelimen lähettämiä sivun ylä- ja alaosaa voidaan muokata haluttuun muotoon `websHeader()`- ja `websFooter()`-funktioista. Sovellus voi käyttää edellisen kuvan rivin 7 `websDone()`-funktiota myös muiden kuin tässä lähetetyn vastauskoodin 200 palauttamiseen.

```
<HTML>
<HEAD>
<TITLE>GoAhead WebServer</TITLE>
<META HTTP-EQUIV="Content-Type" CONTENT="text/html">
</HEAD>
Nimi: Henri <BR>
Ika: 22
</BODY>
</HTML>
```

Kuva 4-3. Käsittelijältä saatu vaste.

Teknisistä ominaisuuksistaan, avoimuudestaan ja tosielämän sovellusesimerkeistä huolimatta GoAhead ei vakuuttanut suunnittelultaan siinä määrin, että se olisi otettu projektissa lopulta käyttöön. Suurimpana asiana askarrutti Ejscriptin joustavuus riittävän monipuolisten sivupohjien luomiseen eri näkymille. Lisäksi sivupohjan valinta pitäisi toteuttaa käytännössä joko suoralla pyynnöllä, uudelleenohjauksella tai jonkinlaisella include-mekanismilla, joista jokainen on liian kankea tämän projektin tarkoituksiin. Käytettävää sivupohjaa olisi syytä pystyä vaihtamaan yhdestä muuttujasta, jolloin sivupohjan valintaan ei vaikuta ainoastaan saadut parametrit, vaan esimerkiksi datan haussa tapahtunut virhe voi aiheuttaa tietyn sivupohjan latautumisen.

Heikkoutena kahteen muuhun kohdassa esiteltyyn palvelinsovellukseen nähden on myös sivupohjien sijainti omassa hakemistossaan, jolloin niiden syntaksi tarkistetaan vasta ajon aikana. Tällöin monet yllättävät virhetilanteet saatetaan huomata kääntämisen sijaan vasta sovellusta ajettaessa.

4.3.2 Klone

Klone on sulautetuille järjestelmille suunniteltu web-palvelin, joka tarjoaa suunnittelijalle mahdollisuuden hajauttaa palvelimen rakenne näytön, logiikan ja tiedon suhteen erillisiin osiin. Klone tukee SPED-prosessimallia, joka helpottaa ohjelmiston integrointia kohdejärjestelmään ja keventää palvelimen taakkaa muistin käytön suhteen. Ohjelman toimii tarvittaessa myös *fork*- tai *prefork*-moodissa, joten se sopii suurellekin yhtäaikaiselle pyyntömäärälle, jos suorituskyky ei ole ongelma. *Fork*-asetus tarkoittaa moodia, jossa palvelin voi käyttää hyödykseen mielivaltaista määrää lapsiprosesseja, joita se käynnistää uusien pyyntöjen saapuessa. *Prefork*-asetus on moodi, jossa palvelin luo käynnistyessään käyttäjän asettaman kiinteän määrän lapsiprosesseja. Työn kohdeympäristössä yhteyksien määrä oletetaan pieneksi, joten näitä asetuksia ei käytännössä tarvita. Tulevaisuutta ajatellen ylimääräisten vaihtoehtojen löytyminen lasketaan kuitenkin eduksi.

Klone määrittelee myös oman ohjelmointirajapintansa, joka helpottaa verkko-ohjelmiston luomista sen tarjoamien funktioiden avulla. Rajapinta sisältää funktiot muun muassa otsikkokenttien, pyyntöjen, vasteiden, sessioiden, tulosteiden, syötteiden ja erilaisten muuttujien käsittelyä varten. Näiden avulla voidaan luoda helposti esimerkiksi yksinkertainen sessionhallinta ilman, että koko mekanismi jouduttaisiin ohjelmoimaan itse.

Klonen erikoisominaisuus on verkko-ohjelmiston ja palvelinsovelluksen kääntäminen yhdeksi binääriksi, jolloin ohjelmiston sekä dynaaminen että staattinen sisältö sekä itse ohjelmisto käännetään yhdeksi ajettavaksi binääriksi. Tämä binääri on eräänlainen sulautettu tiedostojärjestelmä. Tiedostojärjestelmä käännetään käyttöjärjestelmän omalla kääntäjällä, jolloin saavutetaan ohjelmiston tehokas suoritus ja pieni muistivaatimus. Klonen minimivaatimuksiksi luvataan noin 140kt ROM-muistia ja 70kt RAM-muistia, jotka soveltuvat työn kohdeympäristölle erittäin hyvin.

Käytännössä verkko-ohjelmisto muodostuu kl1-muotoon tallennettavista HTML-sivupohjista, joiden sisään voi kirjoittaa C-kielisiä skriptejä. Skriptin eri osille käytetään neljää erilaista lohkoa, joita ovat liitoslohko, esittelylohko, koodilohko ja tulostuslohko. Kuhunkin lohkoon laitetaan tiettyjä koodin osia, jolloin koodin rakenne selkeytyy. Lohkojen käyttötarkoitukset ovat kuitenkin vain viitteellisiä, joten ohjelmoija voi ottaa niiden käytössä vapauksia. Koodissa voi käyttää myös ulkoisia funktioita tallentamalla ne C-tiedostoina erilliseen paikkaan ja kutsumalla niitä kl1-tiedostosta.

Erilaisten koodilohkojen merkitys on seuraava:

- Liitoslohko: lohkolla voidaan liittää sivuun erillisiä staattisia tai dynaamisia osia muista tiedostoista. Liitoslohkon määritykset tulevat <%@ ja %>-sulkeiden sisään.
- Esittelylohko: varsinaisessa koodissa tarvittavat sisällytykset, funktiot ja globaalit muuttujat esitellään erillisessä esittelylohkossa. Esittelylohkon määritykset tulevat *<%!* ja %>-sulkeiden sisään.
- Koodilohko: jokaiselle pyynnölle suoritettava koodi, sivun eräänlainen "main-metodi" asetetaan koodilohkoon. Koodilohkossa voidaan esittelylohkossa

määriteltyjen ja sen omien muuttujien lisäksi käyttää Klonen esimääriteltyjä funktioita. Koodilohkon sisältö tulee <% ja %>-sulkeiden sisään.
- Tulostuslohko: lohkossa voidaan tulostaa muuttujien arvoja. Tulostuslohkon määritykset tulevat <%= ja %>-sulkeiden sisään.

HTML-sivupohjien ja niiden sisältämien koodilohkojen käyttö on yleisiä web-sivuilla käytettäviä skriptikieliä hallitsevalle suunnittelijalle suoraviivaista. Menetelmän avulla Klone tuo sulautetun järjestelmän verkko-ohjelmiston luomisen askeleen lähemmäs nykyaikaista, suoraviivaista web-ohjelmointia.

Palvelimen asetukset

Klone-palvelimen asetusten määrittäminen tapahtuu helpoiten palvelinohjelmaan sisällytettävästä *kloned.conf*-tiedostosta, joka sijaitsee sulautetun tiedostojärjestelmän *etc*-kansiossa. Yleisimmät huomioitavat asetukset ovat palvelimen tyyppi, prosessimalli, protokolla, portti ja verkko-ohjelmiston juurihakemisto. Kuvan 4-4 esimerkkikonfiguraatiossa määritellään verkko-ohjelmisto nimellä *oma_www*. Prosessimallina on tässä tapauksesa *prefork*, jolloin ohjelma luo aluksi kolme *kloned*-prosessia, jotka huolehtivat palvelimelle tulevista pyynnöistä. Sovellus kuuntelee porttia 80, ja etsii näytettävät sivut juurihakemistosta *www*. Asettamalla konfiguraatioon vastaavalla tavalla useamman eri nimisen verkko-ohjelmiston, voidaan samalla sovelluksella käynnistää monta eri sivustoa.

```
1               server_list     oma_www
2               oma_www
3               {
4                   type    http
5                   model   prefork
6                   addr.type       IPv4
7                   addr.port       80
8                   dir_root        /www
9               }
```

Kuva 4-4. Klone-palvelimen asetustiedosto kloned.conf.

Tiedostoon *kloned.conf* asetetaan myös palvelimen mahdolliset SSL-asetukset, joissa määritellään käytettävät avaintiedostot ja salauksen asetukset. Käytettävä asetustiedosto voidaan määrittää myös palvelimen käynnistämisen yhteydessä, jolloin käytetään palvelinsovelluksen *-f* -vipua asetustiedoston osoittamiseen.

Yksinkertainen pyynnönkäsittelijä

Yksi sivu voi muodostua useasta eri kl1-tiedostosta, joita voidaan liittää mukaan liitoslohkon avulla. Liitosmekanismin avulla voidaan luoda yksinkertainen pyynnönkäsittelijä, joka vastaanottaa HTTP-kutsun halutut parametrit ja valitsee niiden perusteella käyttäjälle lähetettävän sivun. Käytännössä tämä tarkoittaa sitä, että pyynnön parametreja tarkastellaan ehtolauseilla, ja tietyn ehdon täyttyessä sitä vastaava kl1-sivupohja liitetään mukaan.

Seuraava yksinkertainen kl1-tiedosto kuvassa 4-5 sisältää vain HTTP-pyynnön käsittelyn, mutta ei vasteen muodostamista käyttäjälle. Tässä tapauksessa käsittelijä siirtää vastuun tulostuksesta täysin sisällytettävälle sivupohjalle eli kl1-tiedostolle *tpl1* tai *tpl2*.

```
1       <%
2           vars_t *in_args = request_get_args(request);
3           int tpl = vars_get_value_i(in_args, "tpl");
4
5           if (tpl == 1) {                                 %>
6       <%@         include "tpl1.klone"                    %>
7       <%      } else if (tpl == 2) {                      %>
8       <%@         include "tpl2.klone"                    %>
```

Kuva 4-5. Esimerkki yksinkertaisesta dispatcher-skriptistä, joka valitsee sisällytettävän sivupohjan.

Toiminta voi liitoslohkojen jälkeen jatkua myös *dispatch.kl1*-tiedostossa, mutta Klonen tarjoamilla keinoilla tämä on tarpeettoman vaivalloista, koska keinot eri kontekstien muuttujien muokkaamiseen eivät ole kovinkaan monipuoliset. Ainoastaan sessiomuuttujille voidaan helposti asettaa nimi-arvo-pareja *session_t*-tietuetta käsittelevillä funktioilla, kun taas esimerkiksi pyyntökohtaisten asetusten muuttaminen ei vastaavalla tavalla onnistu. Täten pyyntöjen ketjumainen välittäminen eri kl1-tiedostojen välillä ei onnistu halutulla tavalla. Sen sijaan näkymän hallinta pitää siirtää toiselle sivupohjalle ja suorittaa siellä *request_get_args*-funktiolla pyyntöjen haku ja haluttujen tietojen tulostus, kuten kuvan 4-6 *tpl1*-sivupohjassa on esitelty. Siinä HTTP-parametri nimellä *objnm* tallennetaan muuttujaan ja tulostetaan näkymään.

```
1       <html>
2       <head><title>Show</title></head>
3       <body>
4       <%
5             vars_t *args = request_get_args(request); // shortcut
6             char *objnm;
7
8             objnm = vars_get_value(args, "objnm");
9
10            io_printf(out, "<b>Objname:</b> %s<br />", objnm);
11      %>
12      </body>
13      </html>
```

Kuva 4-6. Sivupohja, jonka dispatcher sisällyttää vaadittujen ehtojen täyttyessä.

Vaikka Klonen tarjoamat menetelmät sivupohjien käyttöön ovat yksinkertaisilla sivuilla suoraviivaisia, eivät jotkin sen piirteet täysin soveltuneet projektin tarkoituksiin. Kuten esimerkistä käy ilmi, sivupohjan valinnan suorittava *dispatcher* luodaan Klonen omalla skriptikielellä sen sijaan, että se voitaisiin toteuttaa standardilla C-koodilla. Vaikka palvelimen oman ohjelmointirajapinnan käyttäminen asettaa verkko-ohjelmiston toteutukselle tiettyjä rajoituksia, haluttiin toteutettavan sovelluksen palvelinsidonnaisen

koodin osuus pitää mahdollisimman pienenä. Mikäli jo pelkkä sivupohjan valinta tehdään palvelinkohtaisella skriptikielellä, olisi toteutettava sovellus todennäköisesti hyvin riippuvainen käytettävästä palvelimesta.

Toinen häiritsevä seikka Klonessa on se, että se ei tarkalleen määrittele ohjelman logiikan sijaintia verkko-ohjelmiston arkkitehtuurissa. Kun sivupohjan valinta tehdään pyynnön parametrien vastaanottamisen jälkeen skriptikielellä selkeästi yhdessä sivupohjassa, voidaan haettavan tiedon saantitapa määritellä muualla, esimerkiksi sisällytettävässä sivupohjassa. Tiedonhaku *dispatcher*-sivupohjassa olisi huono vaihtoehto siitä syystä, että silloin sivupohjan lisäksi myös tietoon pääsyyn käytettävä ohjelman osa, esimerkiksi listattavien objektien hausta vastaava funktiokutsu, pitäisi suorittaa *dispatcher*-sivupohjassa, mikä monimutkaistaisi sitä entisestään. Jälkimmäinen vaihtoehto, eli tiedonhaku sisällytettävästä sivupohjasta, taas olisi huono siitä syystä, että näkymät eivät välttämättä mene yksi yhteen tiedon saantimenetelmän kanssa, vaan useat eri näkymät saattaisivat hakea tiedon samalla tavalla. Kumpikaan tapa ei noudata MVC-mallia, joka todettiin projektille soveltuvaksi web-arkkitehtuuriksi kohdassa 2.5. Kolmas, ja tässä tapauksessa ehkäpä järkevin tapa, olisi luoda C-kielellä tiedon hakemiselle oma geneerinen funktionsa, joka hakisi näkymien tarvitsemat tiedot yhtenäisellä tavalla ilman, että näkymän generoinnista vastaavan logiikan tarvitsee tuntea hakemaansa tietoa tai sen saantimenetelmää. Toteutettavassa hallintasovelluksessa tarvittaisiin tiedon hakemisen lisäksi omat funktionsa myös hallintaobjektien lisäys-, muokkaus- ja muille toiminnoille.

4.3.3 Seminole

Kolmesta lupaavimmasta palvelinsovelluksesta sopivimmaksi osoittautui Seminole. Tässä alikohdassa käydään pääpiirteittäin läpi Seminole-palvelimen tekniikka ja perusperiaatteet. Tämän projektin kannalta Seminolen tärkein ominaisuus on näkymän eristäminen ohjelman logiikasta HTML-sivupohjien avulla, joten alikohdan yksinkertaistettu esimerkkiohjelma hyödyntää kyseistä tekniikkaa.

Seminole on sulautetuille järjestelmille suunniteltu HTTP-palvelin, jonka etuja ovat erittäin pieni koko, monipuolinen ohjelmointirajapinta ja modulaarinen rakenne. Seminolen korkean tason ohjelmointirajapinta eristää ohjelmoijan matalan tason protokollien yksityiskohdilta, mutta tarvittaessa mahdollistaa niiden muokkaamisen. Yleisiä sovelluskohteita Seminolelle ovat sulautettujen järjestelmien web-käyttöliittymät, etäproseduurikutsujen käsittely, ympäristöönsä mukautuvat aputoiminnot ja perinteiset järjestelmät, joille halutaan mahdollistaa tiedon välitys HTTP-protokollalla. Seminolen toteutuskieli on C++, joka mahdollistaa löyhästi yhdistetyn modulaarisen rakenteen, jolloin käyttämättömistä ominaisuuksista ei koidu ylimääräistä kuormaa. Eri käyttötarkoituksiin suunniteltuja palvelimia varten voidaan luoda omat pyynnön käsittelijänsä, joiden avulla palvelimen käyttöön saa täsmälleen sen tarvitsemat toiminnot. Seminole sisältää joitakin tavanomaisia käsittelijöitä esimerkiksi uudelleenohjausta ja tiedostopalvelinta varten. Omaa sovellusta varten halutunlaisen käsittelijän voi luoda itse. Lukuisten luokkien avulla verkko-ohjelmistossaan voi hyödyntää muun muassa sivupohjia, pyyntöjen ja vasteiden käsittelyä, SSL-salausta ja -autentikointia sekä sessionhallintaa.

Klonen tapaan myös Seminole tarjoaa mahdollisuuden kääntää koko palvelimen hakemistopolku yhdeksi binääriksi, jolloin palvelimen ajaminen on nopeaa ja suuri osa virheistä tulee esiin jo käännösvaiheessa. Lisäksi ohjelmoija voi hienosäätää sovellusta lukuisilla käännösaikaisilla ominaisuuksilla. Kun palvelin ja sille toteutetut verkko-ohjelmistot ovat yhdessä tiedostossa, on sen siirtäminen hakemistosta tai jopa tietokoneesta toiseen vaivatonta. Ympäristön muuttuessa on tietysti huolehdittava binäärin ja järjestelmärajapintojen yhteensopivuudesta ja ajon aikana ladattavien kirjastojen saatavuudesta.

Yksi Seminolen vahvuus on sen siirrettävyys eri alustoille. Sen alustakohtainen koodi on eristetty niin kutsutulle siirrettävyyskerrokselle, jonka avulla se tarjoaa valmiin tuen monille alustoille, kuten POSIX (Solaris, Linux, BSD jne.), Win32, VxWorks, uC/OS2 ja eCos. Vaikka tämän projektin puitteissa ei ole näkyvissä tarvetta muille kuin Linux-alustalle, niin siirrettävyys antaa joustavuutta tulevaisuuteen ja kertoo lisäksi ohjelmiston huolellisesta suunnittelusta. Lisäksi Seminolen kokonaan avoin lähdekoodi antaa suunnittelijalle mahdollisuuden kohdeympäristön ominaispiirteiden huomiointiin.

Palvelimen asetukset

GoAhead WebServerin tapaan myöskään Seminole ei sisällä omaa tiedostoa palvelimen asetuksia varten, vaan tavallisesti ne määritellään käytettävän verkko-ohjelmiston *globals.cpp*-tiedoston *c_servername*- ja *c_serverport*-muuttujilla, jotka tarkoittavat palvelimen host-nimeä ja kuunneltavaa porttia.

Palvelimen tietoturva-asetuksia voidaan hallita *ports/Seminole*-tiedostosta, joka sisältää palvelimen kääntämiseen liittyviä asetuksia. Tiedostosta voidaan asettaa *INC_BASIC_AUTH* ja *INC_DIGEST_AUTH* –muuttujat, joiden perusteella *basic*- tai *digest*-autentikointiskeeman käyttömahdollisuus sisällytetään palvelimen koodiin. Autentikaation varsinainen käyttöönotto tehdään verkko-ohjelmistossa, jonka `HttpdHandler`-luokalle eli pyynnön käsittelijälle luodaan `Authenticator`-olio, jolle määritellään sallitut käyttäjätunnukset ja salasanat sekä käsittelijän kattama *realm*-alue.

Autentikoinnin lisäksi Seminole mahdollistaa myös SSL-salauksen käytön, joka voidaan ottaa käyttöön asettamalla *ports/Seminole*-tiedoston *INC_SSL*-muuttuja ja muokkaamalla verkko-ohjelmiston *main.cpp*-tiedostoa siten, että se kutsuu `Httpd::Start()`-metodia haluttuja SSL-asetuksia vastaavilla parametreilla. Lisäksi *ports/Seminole*-tiedoston *INC_MULTIPLE_TRANSPORTS*-muuttuja pitää olla asetettu, jotta palvelin kykenee ylläpitämään useita erityyppisiä soketteja. Projektissa toteutettu verkko-ohjelmisto käytti kuvassa 4-7 esiteltyjä SSL-asetuksia.

```
pem:web.pem
sock:ssl
rand-file:4096,/dev/urandom
```

Kuva 4-7. Httpd::Start()-metodille annettavat SSL-parametrit.

Parametreilla asetetaan käytettäväksi avaintiedostoksi *web.pem*-tiedosto, josta löytyy käytettävä PKI-sertifikaatti eli palvelimen julkinen ja salainen avain. Soketin tyypiksi annetaan SSL, jotta kuljetettavan tiedon valitsin osaa käyttää SSL-protokollaa tavanomaisen TCP-protokollan sijaan. Salauksessa tarvittavat satunnaismerkkijonot saadaan Linuxin satunnaislukugeneraattorilta.

Sivupohjat

Tärkein seikka, jolla Seminole erottui muista tarkastelluista HTTP-palvelimista ei ollut pieni koko, sulautettu tiedostojärjestelmä tai tietoturvaominaisuudet, vaan sen tarjoamat menetelmät näkymien luomiseen. Seminolen näkymätyypit voidaan toteuttaa HTML-sivupohjina, joissa esitettäväksi haluttu tieto määritetään erilaisten silmukka- ja ehto- ja arviointirakenteiden avulla. Menetelmä on hyödyllinen dynaamisen tiedon käsittelyyn siten, että toteutettavasta verkko-ohjelmistosta saadaan järjevästi ylläpidettävä. Sivupohjat auttavat ylläpidettävyydessä siksi, että niiden avulla sivujen ulkoasun ja asettelun muokkaaminen ei vaadi sovelluslogiikan muokkaamista. Lisäksi sivupohja on usein riippumaton sen tietosisällöstä, jolloin näkymien toteuttamisen vaatima työmäärä ei kasva tietosisällön lisääntyessä. Seminolessa sivupohjien rakenteiden syntaksi tarkistetaan palvelimella käännön aikana, jolloin vältytään ylimääräisiltä ajonaikaisilta virheiltä.

Tiedon näyttäminen sivupohjalla tapahtuu `HttpdFSTemplateShell::Execute()`-metodilla. Tämä metodi tarvitsee parametreikseen pyynnön tilan ja syötettäväksi halutun tietosisällön, jota kutsutaan Seminolen termein symbolitauluksi. Symbolitaulu sisältää näkymässä esitettävän datan, joka voidaan sivupohjassa määrittää esitettäväksi halutulla tavalla. Symbolitaulun erikoistapaus on symbolikartta, joka helpottaa tietueisiin tallennetun datan esittämistä.

Kun `HttpdFSTemplateShell::Execute()`-metodia kutsutaan, se huolehtii kaikesta tulostukseen liittyvistä esivalmisteluista ja prosessoi halutun sivupohjan, joka asettelee näkymään `Execute()`:lle määritetyn symbolikartan sisällön. Kuvan 4.8 esimerkissä esitellään, miten halutun tiedon tallentaminen symbolikarttaan ja näkymän prosessoinnin aloittaminen tapahtuu.

```
1    /* -- demo.cpp --
2     * luodaan uusi käyttäjätili
3     */
4    UserAccount    some_account;
5
6    /* luodaan symbolikartta sym_map, joka saa syötteenään
7     * user_account_map -structissa määritetty rakenne,
8     * symbolien määrä sekä some_account-muuttujan muistiosoite
9     */
10   HttpdSymbolMap sym_map(user_account_map, 2, &some_account);
11
12   /* asetetaan käyttäjätilille nimi ja rooli */
13   some_account.mpName        = "Antti Admin";
14   some_account.mpRole        = "Administrator";
15
16   /* asetetaan käytettävä sivupohja */
17   state.mpFilepath = "/nakyma.thtm";
18
19   /* prosessoidaan tiedot
20    */
21    (void)HttpdFSTemplateShell::Execute(state, &sym_map);
```

Kuva 4-8. Käyttäjätilin tietojen ja sivupohjan asettaminen demo.cpp-tiedostossa.

`Execute()`-metodin ensimmäisenä argumenttina on tietue `state`, joka sisältää tietoa vastaanotetusta HTTP-pyynnöstä ja sen perusteella määritetystä tiedostosta. Käytettävä sivupohja määritetään rivillä 17 asetetulla `state.mpFilePath`-muuttujalla. Muuttuja antaa vapauden käyttää symbolikartan tietosisältöä usealla eri sivupohjalla. Tämä on erityisen hyödyllistä silloin, kun halutaan tarjota käyttäjälle mahdollisuus tietojen tarkasteluun useassa eri muodossa, esimerkiksi HTML-sivuna tai RSS-syötteenä. Kuvassa 4-9 esitellään sivupohja, jossa näkyy miten näytettäväksi halutut arvot määritetään *eval*-komennolla. Komento kertoo sivupohjalla näytettäväksi halutun muuttujan nimen. Muuttujien nimet sivupohjissa on määritetty symbolikartan rakenteessa (`user_account_map`).

```
1               <!-- index.thtm -->
2               <html>
3               <head>
4                    <title>Template Demonstration</title>
5               </head>
6
7               <body>
8                 <h1>User Data</h1>
9
10                <table border="1">
11                  <tr>
12                    <td>User name</td>
13                    <td>%{eval:name}%</td>
14                  </tr>
15                  <tr>
16                    <td>Role</td>
17                    <td>%{eval:role}%</td>
18                  </tr>
19                </table>
20              </body>
21              </html>
```

Kuva 4-9. Sivupohja, joka määrää näytettävän tiedon esitystavan.

Muita mahdollisia direktiivejä ovat *loop*-komento silmukoita varten sekä *if*- ja *ifnot*-komennot ehtolauseita varten. Jokaiselle komennolle voidaan asettaa myös myös attribuutteja, joiden perusteella komennon suorittamisesta vastaava käsittelijä voi toteuttaa erilaisia lisätarkasteluja. Sivupohja muistuttaa hyvin paljon lopullista HTML:ää, jolloin sen toteutus sommittelun ja ulkoasun suhteen riippuu mahdollisimman vähän alla toimivasta kontrollista ja mallista. Sivupohjan näkökulmasta tiedon saantitavalla tai sen alkuperäisellä sijainnilla ei ole merkitystä, vaan olennaisia ovat ainoastaan sen esitystapaan liittyvät seikat.

5 Suunnittelu ja toteutus

Web-hallintasovelluksen suunnittelun ensimmäinen vaihe oli sovelluksen graafisen käyttöliittymän suunnittelu. Tämä suunnitteluvaihe oli ensimmäinen siksi, että sovelluksen asiakasvaatimukset olivat projektin alkuvaiheessa melko hyvin tiedossa, ja ne eivät olisi paljonkaan riippuvaisia projektin teknisestä toteutustavasta. Sovelluksen toiminnallisiin vaatimuksiin perustuva käyttöliittymän suunnittelu oli järkevää tehdä ennen teknistä suunnittelua. Toiminnalliset vaatimukset perustuvat Iris 440 –laitteen hallinnan yleisiin käyttötapauksiin, jotka selvitetään CLI-hallintasovelluksen ominaisuuksien ja laitteen käyttöohjeen perusteella.

Käyttöliittymän suunnittelun jälkeen seuraava vaihe oli sovelluksen luokkien suunnittelu ja toteuttaminen. Myös luokkien tarvitsemat metodit ja tietorakenteet oli suunniteltava. Luokka-arkkitehtuurin suunnittelu tehtiin ennalta suunniteltujen näkymien pohjalta, joiden perusteella pääteltiin tarvittavat luokat. Luokkien tarvitsemat metodit ja tietorakenteet luotiin sovellusmoottorin alapuolella sijaitsevan mallin ja lopullisten näkymien vaatimusten mukaan. Tässä luvussa käsitellään myös hallintasovelluksen integrointia laitteen sovellusympäristöön. Luvun lopussa käsitellään vasteen muodostamisen eri vaiheita verkko-ohjelmiston näkökulmasta.

5.1 Käyttöliittymä

Käyttöliittymän todellisia, konkreettisia näkymiä kutsuttiin näkymiksi ja niiden erilaisia luokkia näkymätyypeiksi. Eri hallintaobjekteja vastaavat konkreettiset näkymät voidaan jaotella niiden tyypin mukaan, joista saadaan hallintasovelluksen vaatimat geneeriset näkymätyypit. Tällaisen jaottelun avulla sovelluksen lopullinen toteutus voitaisiin laatia suunniteltujen näkymätyyppien pohjalta, jolloin vaadittava työmäärä vähenee. Kun jokainen näkymä suunnitellaan johonkin näkymätyyppiin pohjautuen, vaatii uuden saman tyyppisen näkymän lisääminen hallintasovelluksessa vain näkymän parametrien määrittelyn. Näkymän parametreja ovat esimerkiksi hallintaobjektin piilotettavaksi halutut attribuutit tai viitteet muihin attribuutteihin.

Näkymätyyppien suunnittelussa määriteltiin pääasiassa niissä esiintyvien attribuuttien tyypit (ennaltamäärätty avain *fkattr*, avain *kattr* tai tavallinen attribuutti *nattr*), toiminnot ja eri elementtien asettelu. Näkymätyyppien attribuutit merkittiin attribuutin tyyppiä vastaavalla lyhenteellä ja toiminnot painikkeita esittävillä laatikoilla, joissa lukee toimintoa kuvaava teksti. Muita määriteltäviä elementtejä ovat esimerkiksi näkymän otsikko tai sen sisältämät linkit muihin näkymiin.

Käyttöliittymän ulkoasun suunnitelma toteutettiin PowerPoint-esityksenä, jossa jokainen näkymä sijaitsee omalla sivullaan. Suunnitelmassa pyrittiin mahdollisuuksien mukaan sisällöllisesti hyvin lopullista vastaaviin näkymiin, vaikka lopullinen ulkonäkö määräytyisi käytettävien kuvaus- ja skriptikielten (HTML, CSS ja JavaScript) käyttötavan mukaan. Myös teknisiin asioihin, kuten syötekenttien muotoon ja linkkien käyttöön pyrittiin vastaamaan mahdollisimman kattavasti.

5.1.1 Suunnitteluperiaatteet

Hallintasovelluksen käyttöliittymän suunnitteluperusteina käytettiin CLI-hallintasovelluksen osoittamia käyttötapauksia ja niiden vaatimien ominaisuuksien toteuttamista. Käyttötapauksia pidetään yleisesti hyvänä ratkaisuna käyttäjävaatimusten kartoittamiseen, sillä ne osoittavat tehokkaasti ohjelmalta vaadittavat ominaisuudet käyttäjän näkökulmasta [HaMä98, s. 135]. Graafisen käyttöliittymän olisi kyettävä tarjoamaan vähintään samat tarkastelu- ja muokkausmahdollisuudet, kuin mitkä CLI-hallintasovellus tarjoaa. Tämä tarkoittaa pääasiassa siltojen ja niiden porttien, fyysisten liitäntöjen ja niiden VLAN (*Virtual Local Area Network*) –asetusten, laitteen verkkoasetusten, hallinta-asetusten sekä SNMP-asetusten hallintaa ja asianmukaista esittämistä. Samoja käyttötapauksia käytettiin myöhemmin myös käyttöliittymän käytettävyystestissä, jota käsitellään alikohdassa 5.1.5. Käyttötapausten selvittämisessä hyödynnettiin myös Iris 440 –laitteelle laaditun käyttöohjeen esittelemiä konfigurointimahdollisuuksia ja erilaisia käyttöliittymälle esitettyjä toiveita.

Ensisijaisen suunnitteluperusteen valintaan päädyttiin, koska CLI:n vastaanotto asiakkaiden taholta on ollut pääasiassa hyvä, ja koska se esittelee kattavasti Iris 440:n muokattavaksi tai näytettäväksi vaadittavat ominaisuudet. Tästä huolimatta suunnittelun aikana on muistettava, että CLI- ja web-käyttöliittymät ovat toiminnaltaan erilaisia, jolloin CLI:ssä hyväksi todettu ratkaisu ei välttämättä toimi graafisessa käyttöliittymässä. Täten on syytä välttää CLI:n ominaisuuksien orjallista kopiointia ja ottaa web-käyttöliittymän suunnittelussa tiettyjä vapauksia.

Tämän lisäksi yleisenä suunnitteluperiaatteena pidettiin sovelluksen geneerisyyttä, jonka avulla voidaan mahdollistaa sen helppo muokattavuus ja laajennettavuus. Tämä heijastuu käyttöliittymän suunnitteluun näkymätyyppien geneerisyytenä. Näkymätyypit voidaan parametrisoida, ja parametreja muokkaamalla saadaan toteutettua eri objektityyppien vaatimat konkreettiset näkymät. Yksinkertaistettuna käyttöliittymä voidaan jakaa neljään eri näkymätyyppiin: objektin lisäys, muokkaus, näyttäminen ja objektien listaus. Näitä neljää päätyyppiä pyrittiin käyttämään mahdollisimman kattavasti koko hallintasovelluksen laatimiseen, joskin myös muita näkymätyyppejä oli toteutettava erikoisempia näkymiä ja poikkeustapauksia varten.

Varsinaisten suunnittelukriteerien lisäksi ohjelmisto ja sen käyttöliittymä pyrittiin toteuttamaan inkrementiaalisesti, jolloin siihen toteutettaisiin aluksi vain välttämätön perustoiminnallisuus. Näkymätyyppien kartoituksessa pyrittiin löytämään minimimäärä tyyppejä, joilla hallintasovellus voitaisiin toteuttaa käytettävyyden merkittävästi kärsimättä. Näin ohjelmistolle voidaan toteuttaa nopeasti jonkinlainen runko, johon voidaan lisätä ominaisuuksia tarpeen mukaan [HuTh00, s. 48]. Näkymätyyppien määrän minimointi ei saa kuitenkaan vähentää laitteen hallintaominaisuuksia. Kun ohjelman perustoiminnallisuus on valmis, siihen voitaisiin helposti lisätä käytettävyyttä ja toiminnallisuutta parantavia näkymiä ja toimintoja. Tästä huolimatta on selvää, että näkymätyyppien määrä tulee protektin aikana kasvamaan, ja listaa myöhemmin toteutettavista ominaisuuksista ja näkymistä on syytä kerätä työn edetessä.

Aluksi luotu näkymätyyppien minimijoukko oli kompromissi näkymien välisen yhdenmukaisuuden ja sovelluksen riittävän käytettävyyden välillä. Yhdenmukaisuuden kasvaessa eri objektityyppien ominaispiirteiden huomiointi vähenee, jolloin näkymistä

tulee helposti kankeita ja käyttötarkoitukseensa epäjohdonmukaisia. Tämä kuitenkin aluksi sallittiin, koska näkymätyyppien lukumäärän kasvu lisää myös niiden toteuttamiseen vaadittavaa työmäärää.

5.1.2 Näkymien suunnittelu

Käyttöliittymäsuunnitelman ensimmäisissä versioissa pyrittiin luomaan kattava kuva erilaisista näkymistä ja niihin liittyvistä ominaispiirteistä. Vaikka näkymien geneerinen suunnittelutapa pidettiin mielessä alusta lähtien, ei ollut johdonmukaista yrittää heti sovittaa näkymiä joihinkin ennalta suunniteltuihin näkymätyyppeihin. Sen sijaan näkymätyypit haluttiin suunnitella vasta tarvittavien näkymien perusteella, jolloin eri näkymien ominaispiirteet voitaisiin huomioida paremmin. Näiden lähtökohtien mukaisesti käyttöliittymäsuunnitelman ensimmäiset versiot sisälsivät huomattavasti enemmän näkymätyyppejä, kuin mihin lopulta päädyttiin. Näkymätyyppien määrää voitiin jatkossa vähentää yhdistämällä samankaltaisia näkymätyyppejä toisiinsa.

Eräs suunnittelussa ilmaantunut graafisen käyttöliittymän ongelma oli tilanne, missä jokin näkymä sisälsi muokattavia attribuutteja ja samalla siirtymiä muihin näkymiin. Silloin käyttäjälle voi olla epäselvää, katoavatko sivulle syötetyt tiedot siirryttäessä vai jäävätkö ne muistiin odottamaan käyttäjän paluuta. Tästä syystä esimerkiksi lomakenäkymän ei pitäisi samalla sisältää siirtymiä muihin näykymiin, tallennusta ja peruutusta lukuunottamatta. Rajoittamalla tällaisen näkymän siirtymismahdollisuudet vain näihin kahteen, vältytään siltä ongelmalta, joka seuraisi, mikäli sivulta siirryttäisiin muualle sen jälkeen kun jotakin lomakkeen syötekenttää on muokattu. Käytännössä tämä tarkoittaa erillistä hallintaobjektin attribuuttien näyttämis- ja manipulointinäkymää. Lisäksi attribuuttien manipulointi jaettiin vielä erilliseen lisäys- ja muokkausnäkymään. Siirtyminen navigointivalikon painikkeilla kesken muokkauksen tosin onnistuu edelleen, mutta silloin käyttäjän voidaan olettaa ymmärtävän lomaketietojen häviäminen.

Yksi luonnollinen ero CLI- ja web-käyttöliittymässä on se, että kun komentorivillä objektin muokkaaminen ja näyttäminen tapahtuu erillisillä komennoilla, niin graafisessa käyttöliittymässä nämä kaksi asiaa voidaan osittain yhdistää. Liitteessä 3 esitellyn sillan muokkausnäkymän toteutuksessa pitää huomioida tarkasti se, mitkä objektin attribuutit ovat muokattavia ja mitkä ainoastaan näytettäviä. Kuvassa objektin muokattavat attribuutit on merkitty *kursiivilla*. Sillan muokkausnäkymässä sen avainattribuutti eli nimi ei voi olla muokattavissa, koska sitä muutettaessa myös muokattava silta vaihtuisi. Eri näkymäluokat voidaan jaotella neljään pääluokkaan, joita ovat lisäys (*add*), muokkaus (*edit*), lista (*list*), näyttäminen (*show*). Näistä pääluokista on lukuisia variaatioita ja lisäksi tarvitaan joitakin erikoisnäkymiä, kuten staattiset näkymät.

Näkymätyypit

Add-näkymätyyppi on hallintaobjektin lisäämiseen tarkoitettu näkymätyyppi, joka sisältää tallennus- ja peruutuspainikkeet. Näkymän otsikko sisältää lisättävän objektityypin nimen. Näkymässä olevat attribuutit jaotellaan ennaltamäärättyihin avainattribuutteihin (*fkattr*), jotka eivät ole käyttäjän muokattavissa, sekä avainattribuutteihin (*kattr*) ja tavallisiin attribuutti (*nattr*), jotka käyttäjä voi itse valita. Esimerkki tämän tyyppisestä näkymästä on sillan lisäys liitteessä 4.

Edit-näkymätyyppiä käytetään olemassaolevan hallintaobjektin muokkaamiseen, ja se sisältää lisäysnäkymän tapaan tallennus- ja peruutuspainikkeen. Näkymän otsikko sisältää muokattavan objektityypin nimen. Näkymässä olevat attribuutit jaotellaan avainattributteihin (*kattr*), jotka eivät ole käyttäjän muokattavissa ja tavallisiin attribuutteihin (*nattr*), jotka käyttäjä voi itse valita. Esimerkki tämän tyyppisestä näkymästä on sillan editointi liitteessä 3.

List_cut-näkymätyyppiä käytetään hallintaobjektien listaamiseen. Listausnäkymän ominaispiirteinä on, että objektien määrää ei ole erikseen rajoitettu ja näkymätyypin nimen mukaisesti näytettävien attribuuttien määrä on katkaistu. Tämä tyyppi sisältää objektin näyttämiseen, poistamiseen ja lisäämiseen tarkoitetut painikkeet, ja sen otsikko sisältää listan suodatusehdot ja objektityypin nimen. Esimerkki tämän tyyppisestä näkymästä on siltojen listaus liitteessä 5.

List_closed-näkymätyyppi on listausnäkymän variaatio, jonka ominaispiirre on objektien ennaltamäärätty lukumäärä. Tämä tyyppi sisältää objektin näyttämiseen, poistamiseen ja lisäämiseen tarkoitetut painikkeet, ja sen otsikko sisältää listan suodatusehdot ja objektityypin nimen. Esimerkki tämän tyyppisestä näkymästä on ethernet-liitäntöjen listaus liitteessä 6. Iris 440 –laite sisältää neljä ethernet-liitäntää, joten nämä liitännät esitetään näkymässä riippumatta siitä, ovatko ne käytössä vai eivät. Käytössä olevaa liitäntää voidaan tarkastella *show*-painikkeella ja se voidaan poistaa *disable*-painikkeella. Poissa käytöstä oleva liitäntä voidaan aktivoida *enable*-painikkeella.

List_reset-näkymätyyppi on listausnäkymän variaatio, jonka ominaispiirre on objektin nollauksen mahdollistava *reset*-painike. Lisäksi tyyppi sisältää painikkeen objektin näyttämiseen, ja sen otsikko sisältää listan suodatusehdot ja objektityypin nimen. Esimerkki tämän tyyppisestä näkymästä on liitäntöjen laskurien listaus liitteessä 7.

Image_download-näkymätyyppiä käytetään laitteen uuden levykuvan lataamista varten. Tämä tyyppi on luotu saman nimistä näkymää varten, joka sisältää syötekentän levykuvan osoitetta varten ja painikkeen lataamisen aloittamiseksi. Näkymä on esitelty liitteessä 8.

Image_install-näkymätyyppiä käytetään uuden levykuvan asentamiseen laitteen flash-muistille. Tämä tyyppi on luotu saman nimistä näkymää varten, joka sisältää tiedot haetusta levykuvasta ja painikkeen asennuksen aloittamista ja levykuvan tuhoamista varten. Näkymä on esitelty liitteessä 9.

Progress-näkymätyyppi on luotu saman nimistä näkymää varten, jolla ilmaistaan käyttäjälle että jokin toiminto, esimerkiksi levykuvan kopiointi tai asennus on kesken. Esimerkki tyypistä on levykuvan kopiointi liitteessä 10.

Static_config_menu-näkymätyyppi on luotu saman nimistä näkymää varten, joka on asetustiedoston käsittelyyn tarkoitettu valikkonäkymä. Se sisältää painikkeet asetustiedoston tallentamista ja poistamista varten sekä tehdasasetusten lataamista ja uudelleenkäynnistystä varten. Näkymä on esitelty liitteessä 11.

Static_confirm-näkymätyyppi on luotu saman nimistä näkymää varten, joka on tarkoitettu uudelleenkäynnistyksen varmistamiseen. Se sisältää painikkeet toiminnon vahvistamiseen ja peruuttamiseen. Näkymä on esitelty liitteessä 12.

Static_flash_complete-näkymätyyppi on luotu saman nimistä näkymää varten, joka on tarkoitettu levykuvan asentamisen valmistumisen ilmoittamisesta käyttäjälle. Se sisältää valmistumisesta kertovan tekstin ja *reboot*-painikkeen. Näkymä on esitelty liitteessä 13.

Static_front-näkymätyyppi on tarkoitettu etusivua varten, jossa voi näkyä esimerkiksi Design Combus –logo tai tietoa hallintasovelluksesta tai laitteesta. Etusivunäkymä on esitelty liitteessä 14.

Käyttöliittymän ensimmäisissä näkymäsuunnitelmissa käytettiin useissa näkymissä *back*-painiketta helpottamaan näkymien välillä siirtymistä. Esimerkiksi objektin attribuutit näyttävä näkymä, joka ei sisällä lainkaan toimintoja (näkymätyyppi *show_noedit*, esimerkiksi *show SNMP community* liitteessä 15), olisi käytettävyystestin perusteella käyttäjäystävällisempi, jos se tarjoaisi helposti havaittavan poispääsyn näkymästä. Painike koettiin kuitenkin selvästi inkrementiaaliseksi ominaisuudeksi, joten se voitaisiin lisätä tarvittaviin näkymiin myöhemmin. Lisäksi se tulisi olemaan samankaltainen selaimen oman *back*-painikkeen kanssa, jota ei välttämättä ole tarpeen korvata. *Back*-painike ei myöskään ole yhtä helppo toteuttaa kuin esimerkiksi *add*- tai *edit*-näkymissä käytettävä *cancel*, koska toisin kuin *cancelin* tapauksessa, edellinen näkymä ei ole yksiselitteinen. *Add*-näkymässähän *cancel* vie aina kyseisen objektin listausnäkymään ja *edit*-näkymässä objektin näyttämisnäkymään.

Käsiteltävän objektin osoittaminen käyttäjälle

Kun objekti näytetään käyttäjälle tai se on lisättävänä tai muokattavana, pitää olla jokin tapa osoittaa käyttäjälle mikä objekti on kyseessä. Tämä voidaan tehdä esimerkiksi näyttämällä objektin avainattribuuttien arvot otsikossa tai listaamalla kaikki tiedot normaalisti allekkain. Eri menetelmien selkeys on tapauskohtaista, sillä jollekin objektityypille osoittaminen on selkempää otsikon ja toiselle attribuuttilistan muodossa. Näin monipuolinen objektityyppien erityispiirteiden huomioiminen ei kuitenkaan näkymätyyppien yhtenäisyyden vuoksi ollut tarkoituksenmukaista, joten pyrittiin löytämään menetelmä, joka toimii parhaiten kaikkien objektityyppien kanssa.

Vaikka otsikkossa esiintyvät objektin attribuutit kertovat käyttäjälle usein selkeästi mitä objektia ollaan käsittelemässä, voi otsikko muuttua epäselväksi, jos siinä olevien attribuuttien määrä on suuri. Jokin objektityyppi saattaa sisältää enemmän avainattribuutteja kuin muita attribuutteja, jolloin otsikon esittämä attribuuttimäärä olisi attribuuttien kokonaismäärään nähden suhteettoman suuri. Toisaalta myös otsikon muuttaminen ihmisen luettavaan muotoon voi olla hankalaa verrattuna attribuuttien nimi-arvo-parien listaamiseen allekkain. Otsikon käyttöä attribuuttien osoittamiseen tukisi se seikka, että otsikkoa käytetään joka tapauksessa monissa näkymissä objektin identifiointiin objektityypin tai listan suodatinattribuuttien perusteella.

Valittu ratkaisu oli attribuuttien listaaminen nimi-arvo-pareiksi siksi, ettei suunnittelussa tarvitsisi huolehtia avainattribuuttien määrästä muihin attribuutteihin nähden. Lisäksi

esitettävän informaation määrä kasvaa, koska attribuuteista näkyy siten sekä nimi että arvo pelkän arvon sijaan.

5.1.3 Navigointivalikko

Eri hallintaobjekteja koskevien näkymien välillä navigointia varten sivulle toteutettiin navigointivalikko, josta käyttäjä voi valita haluamansa hallintaobjektin. Lähes kaikki valikon painikkeet avaavat listan sitä vastaavan tyypin objekteista muutamaa objektin näyttämisen avaavaa poikkeusta lukuunottamatta. On huomioitava, että navigointivalikko ei ole kaikenkattava näkymien välillä siirtymisen suhteen, vaan osa navigoinnista tapahtuu itse näkymissä olevilla painikkeilla ja selaimen *back*-nappulalla. Valikosta haluttiin tehdä niin itsenäinen kuin mahdollista, mikä tarkoittaa sitä, että valikon kulloinenkin tila olisi sen hetkisen näkymän suhteen staattinen, ja eri painikkeista tapahtuvat toiminnot olisivat aina samat. Tämän takia osa objektien listauspainikkeista, kuten VLAN-liitännät tai siltaan kuuluvat portit, ovat linkkeinä itse näkymissä sen sijaan että niihin pääsisi suoraan navigointivalikosta. Tällöin eri objektien välistä viite-eheyttä ei tarvitse ylläpitää navigointivalikossa, vaan riittää että näkymien objektit tuntevat viitteensä muihin objekteihin. Valikon tilan määräämiseen riittää sen hetkinen objektityyppi, jonka perusteella voidaan arvioida missä valikon haarassa ollaan. Esimerkiksi liitteen 15 kuvassa näytetään laitteelle tallennettu SNMP-yhteisö, jonka perusteella valikolle määräytyy tila, jossa SNMP-haara on avoinna.

Suunniteltaessa tiedettiin, että navigointivalikon toteuttamiseen vaadittavan koodin määrä olisi riippuvainen sen toteutustavasta, ja tarkemmin ottaen sen sitouttamisen asteesta projektin kohdeympäristöön. Yksinkertaisimmillaan valikko voisi olla kovakoodattu valmiiksi, koska käytännössä jokainen tarvittava valikon painike ja sen parametrit tunnetaan etukäteen. Tällöin valikon jokaisella tilalla voisi olla sitä vastaava tunnuksena, joka olisi sitten yhtenä näkymän parametrina näkymää muodostettaessa. Siten jokaista näkymää vastaisi yksikäsitteinen valikon tila, joka voisi käytännössä olla jokaiseen näkymään sisällytettävä valikon HTML-kuvauksen sisältävä staattinen tiedosto.

Toinen vaihtoehto olisi tehdä valikosta älykkäämpi, jolloin siihen toteutettaisiin tuotekohtaista dataa ja näkymän kulloistakin tilaa hyödyntävää logiikkaa. Valikko voisi hakea saatavilla olevat objektityypit ja muodostaa niistä viitemerkintöjä apuna käyttäen sen hetkistä kohdelaitetta vastaava navigointivalikko. Jos valikko tuntisi kulloisenkin näkymän, voisivat kaikki hallintaobjektit olla tavoitettavissa valikosta. Esimerkiksi tietyn ethernet-liitännän VLAN-määritysten listan saisi näkyviin valikosta kyseisen liitännän painikkeen alapuolelta.

Toteutuksessa päädyttiin kuitenkin yksinkertaisempaan, eksplisiittiseen navigointivalikkoon kiireellisen aikataulun ja ennalta tunnetun kohdeympäristön takia. Jos hallittavaksi tulevien laitteiden ominaisuudet vaihtelisivat fyysisesti, niin jonkinlaisen dynaamisen navigointivalikon luominen voisi olla realistisempi vaihtoehto. Käytännössä navigointivalikon eri tiloja esittävät HTML-pohjat tallennettiin omiin tiedostoihinsa, joista kontrolli valitsee näkymää koskevan hallintaobjektin perusteella oikean tiedoston. Näkymää vastaava valikon tila siis lisätään näkymään ohjelman suorituksen aikana. Oletuksena käytettiin valikon perustilaa, jotta mikään näkymä ei vahingossa jäisi kokonaan ilman valikkoa. Huonoa ratkaisussa on sen oletettavasti

heikko yhteensopivuus tulevien tuoteprojektien kanssa, mutta toisaalta uudelleenkirjoitettava koodimääräkään ei ole suuri.

5.1.4 Painikkeiden toiminnot

Jokainen näkymätyyppi sisältää määrätyt toiminnot, jotka käyttäjä voi suorittaa klikkaamalla toimintoa vastaavaa painiketta. Lopullisessa sovelluksessa painikeet ovat lomakkeiden submit- syötteitä, jotka lähettävät lomakkeen parametrit halutulla HTTP-metodilla. Kuhunkin toimintoon liittyy sen tarvitsemat parametrit, joiden muoto on jokaiselle toiminnolle sama. Parametrit muodostuvat toiminnon tyypistä, toiminnon kohteena olevasta objektityypistä ja lisäparametreista. Parametrien lopullinen sisältö riippuu sen hetkisen näkymän sisältämästä datasta, joiden perusteella painikkeella lähetettävä pyyntö muodostetaan. Parametrien tiedot voivat olla peräisin mallilta, käyttäjän täyttämistä syötekentistä tai ne voivat olla kiinteästi määritetty sivupohjaan.

Erilaisia toimintotyyppejä ovat *Show, List, Openadd, Openedit, Add, Update, Delete, Download, Flash, Saveconfig* ja *Removeconfig*. Nämä toimintotyypit kartoitetaan palvelinsovelluksessa niitä vastaaviin toiminto-olioihin, jotka vastaanotettujen parametrien avulla käsittelevät pyynnön. Kontrollin näkökulmasta vastaanotettuja parametrejä tarkastellaan samanarvoisina, ja niiden lopullinen semantiikka riippuu kutsuttavasta toiminnosta. Kontrolli ohjaa HTTP-pyynnön jäsentämisen tuloksena saadut parametrit toimintoa ja objektityyppiä vastaavalle oliolle. Tämä olio tuntee saamiensa parametrien merkityksen ja käsittelee niitä vaaditulla tavalla. Kontrollia ja toimintoluokkia käsitellään tarkemmin kohdassa 5.2.

Seuraavaksi käydään läpi eri toimintojen rooli sovelluksessa ja niiden tarvitsemat parametrit:

- *Show*-toiminto avaa näkymän, jossa näytetään määrätyn objektin attribuutit. Toiminto vaatii parametrikseen objektin yksiselitteisen tunnisteen eli sen tyypin ja avanattribuutit.
- *List*-toiminto avaa määrättyjen suodatinattribuuttien perusteella noudetun objektilistan. Toiminto vaatii parametreikseen suodatinattribuuttien arvot.
- *Openadd*-toiminto avaa määrätyn objektin lisäysnäkymän. Toiminto vaatii parametreikseen objektityypin sekä ennaltamäärätyt avainattribuutit, joita käyttäjä ei voi näkymässä enää muuttaa.
- *Openedit*-toiminto avaa määrätyn objektin muokkausnäkymän. Toiminto vaatii parametreikseen objektityypin ja avainattribuutit.
- *Add*-toiminto tallentaa uuden objektin annetuilla arvoilla. Toiminto vaatii parametreikseen objektityypin, avainattribuutit ja muut määritetyt attribuutit.
- *Update*-toiminto päivittää määrätyn, olemassaolevan objektin halutuilla arvoilla. Toiminto vaatii parametreikseen objektityypin, sen avainattribuutit ja muut määritetyt attribuutit.
- *Delete*-toiminto poistaa määrätyn objektin. Toiminto vaatii parametreikseen poistettavan objektityypin ja avainattribuutit.
- *Download*-toiminto käynnistää laitteen levykuvan lataamisen ja vaatii parametrikseen ladattavan tiedoston URL-osoitteen.

- *Flash*-toiminto käynnistää määrätyn levykuvan asentamisen laitteelle, ja tarvitsee parametreikseen objektityypin ja avainattribuutit.
- *Saveconfig*-toiminto tallentaa hallintasovelluksen nykyiset asetukset, ja se ei tarvitse parametrejä.
- *Removeconfig*-toiminto poistaa nykyiset asetukset, ja myöskään se ei tarvitse parametrejä.

Painikkeet on pyritty sommittelemaan näkymiin siten, että käyttäjä mieltää jotkin toiminnot samankaltaisiksi niiden läheisyyden perusteella [Kal95, s. 156]. Esimerkiksi liitteen 5 *list_cut*-näkymässä tämä näkyy siten, että *new*-painike on erillään vierekkäisistä *show*- ja *delete*-painikkeista. Tämä johtuu siitä, että *new*-painikkeella luodaan uusi objekti, kun taas *show*- ja *delete*-painikkeilla käsitellään määrättyä, jo olemassa olevaa objektia.

Toimintojen suorittamisen jälkeistä näkymää ei voida kertoa yksiselitteisesti, koska se riippuu annettavista parametreista ja voi muuttua käsittelyn kulusta riippuen. Esimerkiksi tietyt objektityypit käyttävät erilaista listausnäkymää kuin muut, ja käsittelyn johtaessa virhetilanteeseen voidaan antaa eri näkymä kuin silloin, jos toiminto onnistuu.

5.1.5 Käytettävyystesti

Käyttöliittymän saadessa lopullista muotoaan, sen soveltuvuus loppukäyttöön kannalta on syytä testata käytettävyystestillä. Jo yksinkertaisen pahvimallin avulla tehdyllä käytettävyystestillä voidaan saada esiin monia käytettävyysongelmia, joita ei suunnitellessa tule ajatelleeksi [HuTh00, s. 53]. Koska käyttöliittymän ideoinnissa ja suunnittelussa oli mukana käytännössä vain kaksi henkilöä, voi siihen helposti jäädä helposti piirteitä, jotka vastaavat liikaa suunnittelijoiden henkilökohtaista näkemystä hyvästä käyttöliittymästä. Lisäksi suunnittelijan kokopäiväinen paneutuminen työhönsä voi estää häntä näkemästä yksinkertaisia virheitä tai puutteita. Esimerkkinä tästä oli näkymä, jossa näytettiin selvästi väärän hallintaobjektin attribuutit, ja suunnittelijan sijaan testihenkilö huomasi huomasi tämän välittömästi käytettävyystestin yhteydessä.

Käytettävyystestauksessa etsitään käyttäjälle päänvaivaa tuottavia ohjelman vikoja ja puutteita tarkkailemalla käyttäjän toimintaa ohjelman parissa [OvRa98, s. 13]. Testattaviksi toiminnoiksi kannattaa valita useimmiten käytettäviä toimintoja, koska niissä käytettävyysparannuksista saavutettu hyöty on suurin [Kuu03, s. 72]. Testaus pyrittiin täten toteuttamaan yleisimpien käyttötapausten perusteella, joiden oletetaan olevan oleellisia laitteen käyttöönoton kannalta. Testausta varten laadittiin kirjallisesti joitakin käyttötapauksia, ja testauksen yhteydessä esiintyneet ongelmat ja huomiot kirjattiin ylös. Käyttötapauksia ei noudatettu orjallisesti, vaan testihenkilö saattoi halutessaan keksiä käyttötapauksia myös itse. Tämä vapaus annettiin, koska testihenkilö oli laitteen asiantuntija ja tiesi mitä asetuksia laitteelle tarvitsisi loppukäytössä asettaa. Myös testihenkilön omat käyttötapaukset ja niissä ilmenneet huomiot kirjattiin ylös vastaavalla tavalla.

Käytännössä testaus tapahtui siten, että PowerPoint-muodossa oleva käyttöliittymäsuunnitelma heijastettiin valkokankaalle ja testivalvoja kertoi testihenkilölle mitä tämän pitäisi tehdä. Tämän perusteella testihenkilö kävi läpi ääneen,

mitä näkymien toimintoja hän käyttäisi ja mitä hän syöttäisi lomakkeisiin. Testivalvoja selasi näkymiä sitä mukaa, mihin henkilön kulloinkin suorittama toiminto hänet veisi. Esiintyneiden ongelmien lisäksi valvoja kirjasi ylös testattavan mainitsemia positiivisiksi havaitsemiaan asioita ja muita sivuhuomioita. Testauksen yhteydessä olisi voitu käyttää myös videointia, jolloin testitilanne olisi voitu käydä jälkeenpäin yksityiskohtaisemmin läpi. Tämä koettiin kuitenkin vaikeaksi toteuttaa huonojen teknisten valmiuksien vuoksi ja tarpeettomaksi, koska testivalvoja teki testin aikana huolellisia muistiinpanoja.

Esiintyneitä huomioita

Seuraavat huomiot ovat testauksessa ilmenneitä ongelmakohtia esiintymisjärjestyksessä. Lisäksi on mahdollisesti selvitetty lyhyesti esiin tulleita parannusehdotuksia.

- Ethernet-liitännän *show*-näkymässä testattava ei heti hahmottanut *mode*-attribuutin merkitystä, jota olisi selkeytettävä, jotta käyttäjä ymmärtää sen tarkoittavan linkin nopeutta ja moodia (esimerkiksi 100Mbps full-duplex).
- *Add*-näkymään johtava *new*-painike on merkitykseltään epämääräinen. *Add*-painike voisi näkymän nimen mukaisesti olla testihenkilön mielestä johdonmukaisempi.
- *Add*-näkymän tallennustoiminnon jälkeen tulevasta näkymästä oli epäselvyyttä. Epäselvyys johtui osittain varhaisesta suunnitteluvaiheesta.
- *Add*- ja *edit*-näkymissä virheellisten arvojen ilmaisemista ei ole selvitetty. Latautuuko selaimelle oma virheen ilmaiseva sivunsa vai näytetäänkö ne sivulla jossa syöte on annettu?
- Miten käyttäjä tietää *add*-näkymässä syöteelle vaaditun muodon? Tekstikentässä oleva oletusarvo auttaa, mutta tarpeen lienee lisätä tekstikentän viereen jokin ohjeteksti.
- Pitäisikö ainakin joidenkin näkymien sisältää *back*-painike? Painike saattaisi parantaa käytettävyyttä, koska testihenkilö ei ole tottunut käyttämään selaimen *back*-painiketta CGI-tyyppisiä verkkosovelluksia käytettäessä.
- Kun sillalle lisätään uusi numeroimaton tai VLAN-numeroitu portti, niin linjan oletusarvona oleva *1* voi aiheuttaa epäselvyyttä ethernet-portin yhteydessä.
- Kun käyttäjä lisää uuden hallinta-IP:n, niin kykeneekö hän muistamaan sillalle antamansa nimen, jolle haluaisi IP:n lisättävän? Kelvollinen ratkaisu lienee pudotusvalikko luoduista silloista.
- Onko hallinta-IP:n määrittäminen mahdollistettava jokaiselle portille erikseen? Testattavan mielestä mahdollistaminen on tarpeen siitä huolimatta, että se saattaisi hämmentää uutta käyttäjää.
- Tallennetun konfiguraation poistomahdollisuus on lisättävä. Tämä ominaisuus jäi suunnittelijalta kokonaan huomioimatta, koska ei ollut huomannut kyseistä ominaisuutta CLI:ssä.
- Pitäisikö SNMP-yhteisön *add*-näkymässä näkyä oletusverkkona *0.0.0.0/0* vai *default? Default* lienee kuvaavampi, mutta asia voidaan ottaa huomioon vasta toteutusvaiheessa, kun sisällön generointimenetelmistä on enemmän tietoa.
- Mitä tapahtuu, kun *add*-näkymän lomakkeeseen syöttää tyhjän arvon? Tilanne poikkeaa CLI:stä siinä mielessä, että siinä tyhjän arvon antaminen attribuutille ei ole mahdollista.

- Pitäisikö verkkoliitännän *show*-näkymässä olla linkki kyseisen liitännän pakettilaskuriin? Testihenkilön mielestä laskurin löytäminen voisi olla helpompaa liitännän kautta, mutta suunnitteluperiaatteita noudattaen aluksi käytettäneen CLI:n kanssa yhtenäistä tapaa näyttää laskurit *statistics*-tilan kautta.
- *Show*-näkymien linkit voisivat sisältää sanan *configure* (esimerkiksi *configure VLAN ports* liitännän VLAN-porttien konfigurointiin pelkän *VLAN ports* sijaan), jotta käyttäjälle selviää, että siitä pääsee tarkastelun lisäksi myös manipuloimaan kyseistä objektia.

Testauksessa ilmenneiden ongelmien analysoinnissa oli syytä huomioida, että testattavana ollut henkilö on Iris 440 –laitteen ja erityisesti sen CLI-hallintasovelluksen asiantuntija, kuten aiemmin mainittiin. Tämän johdosta etenkään käyttöliittymän ja hallintaobjektien esittämistavan informatiivisuuden suhteen ei voida testin perusteella tehdä merkittäviä positiivisia johtopäätöksiä.

Monesti käytettävyystestissä nousee esiin enemmän kysymyksiä kuin vastauksia [Kuu03, s. 80]. Testissä tuli esiin paljon asioita, jotka olivat suunnitteluvaiheessa jääneet huomiotta. Monille niistä saatettiin löytää nopeasti yksinkertainen ratkaisu, mutta useat esiin nousseet seikat jätettiin käyttöliittymäsuunnitelman ulkopuolelle. Yksi esimerkki tällaisesta asiasta on *add*-näkymässä tallentamisen jälkeen saatava palaute, jos tekstikentän arvo on tyhjä. Tälle, kuten monille muillekin vastaavanlaisille seikoille tulee helposti mieleen erilaisia ratkaisuvaihtoehtoja, esimerkiksi objektin määrityksessä annetun oletusarvon tallentaminen käyttäjän puolesta tai virheilmoitus. Päätös lopullisesta ratkaisusta kannattanee kuitenkin tehdä vasta toteutusvaiheessa, kun sovelluksen muu rakenne ja ratkaisuvaihtoehdot ovat paremmin selvillä. Tästä huolimatta esiin tulleet seikat on syytä dokumentoida, jotta ne voidaan projektin myöhemmässä vaiheessa palauttaa mieleen.

5.2 Verkko-ohjelmiston arkkitehtuuri

Projektissa toteutettu web-hallintasovellus muodostuu kahdesta pääosasta, joista ensimmäinen on GladeSoft Inc:n tarjoaman Seminole HTTP-palvelimen toteuttama hallintasovelluksen ydin. Se tarjoaa projektin verkko-ohjelmistolle rajapinnan HTTP-pyyntöjen vastaanottamiseen ja vasteiden lähettämiseen. Se sisältää myös sovelluskehyksen, jota voidaan käyttää ohjelmointirajapinnan avulla erilaisten sivupohjien hyödyntämiseen näkymien toteuttamisessa. Seminolea käsiteltiin tarkemmin alikohdassa 4.3.3, jossa havainnollistettiin sen toimintaa esimerkin avulla. Tässä kohdassa käsitellään hallintasovelluksen toista pääosaa, joka on työn kohteena oleva verkko-ohjelmisto. Se sisältää omat tietorakenteensa ja luokkansa mallin eli Config Managerin käyttöön, jota käytetään Config API –rajapinnan avulla. Tietorakenteiden ja luokkien avulla muokataan mallilta saatu data HTML-sivupohjien ymmärtämään muotoon.

5.2.1 Tietorakenteet

Ensimmäinen web-hallintasovelluksen käyttämä tietorakenne, jonka kanssa palvelimelle saapunut HTTP-pyyntö joutuu kosketuksiin on pyynnön parametrien tallentamiseen käytettävä STL (*Standard Template Library*) –kartta, jonka tyyppi on `map<string,`

`string>`. Tyypillä määritetään kartta, jonka jokaisen tietoalkion nimi ja arvo ovat tyypiltään string-merkkijonoja. HTTP-pyynnön parametrit tallennetaan tähän karttaan, ja siitä erotellaan pyydetty toiminto ja objektityypin nimi omiin merkkijonoihinsa. Karttaan säilötty pyynnön loppuosa välitetään toimintoa ja objektityyppiä vastaavalle toiminto-oliolle, jossa sen käsittely suoritetaan.

STL-kartan etuja tietorakenteena ovat automaattinen muistinhallinta ja joustavat käyttömahdollisuudet, joista tässä sovelluksessa käytetyimmät ovat avainperusteinen haku ja iterointi. Automaattinen muistihallinta vähentää ohjelmointityöhön tarvittavaa aikaa ja auttaa vähentämään koodissa esiintyvien huolimattomuusvirheiden määrää.

Web-hallintasovelluksen tärkeimmät tietorakenteet ovat sovelluksen Config API:n kautta mallilta saatavat hallintaobjektit ja niiden sisältämät attribuutit. Hallintaobjekti tarkoittaa jotain Iris 440 –laitteen hallittavaa asiaa, esimerkiksi siltaa tai verkkoliitäntää. Attribuutit ovat hallintaobjektin sisältämiä hallintaominaisuuksia, joita voidaan mahdollisuuksien mukaan tarkastella tai muokata. Config API:lta saatava objekti on olio, joka sisältää joukon julkisia ja yksityisiä metodeja ja muuttujia. Web-hallintasovelluksen kannalta objektin tärkein ominaisuus on sen sisältämät attribuutit, joita se omistaa mielivaltaisen määrän STL-karttaan tallennettuna. Kartan tyyppi on `map<const attr *, const value *>`, eli se sisältää osoittimet attribuutin nimeen ja arvoon. Kartta on objektin yksityinen muuttuja, ja sen sisältämien attribuuttien arvoja voidaan lukea esimerkiksi metodilla `construct_attr_value_clone(const attr *at)`, joka palauttaa parametrina annetun attribuutin arvon. Hallintaobjektin lisäksi web-hallintasovellus tarvitsee käyttöönsä myös muuta tietoa, jota vaaditaan näkymän luomiseen objektityypin ja sille suoritettavan toiminnon vaatimalla tavalla. Nämä vaatimukset koskevat pääasiassa objektin attribuutteja, joiden esitystavan on noudatettava attribuutille määriteltyjä näkyvyysasetuksia ja muokkausmahdollisuuksia. Näitä vaatimuksia varten jokainen mallilta saatava objekti tallennetaan `ViewData`-tietueeseen, joka on esitelty kuvassa 5-1. Rakenne sisältää objektin osoittimen ja tarvittavat lippukartat, joilla objektin kukin attribuutti voidaan ilmaista olevan ennaltamäärätty avain (`fkflags`), virheellisen arvon sisältävä (`errflags`), piilotettava (`hdnflags`) tai näytettävä (`shwflags`). Hallintaobjektin muokkauslomakkeessa kukin attribuutti voidaan asettaa ennaltamäärätyksi avaimeksi eli ei-muokattavaksi tai virheellisen arvon sisältäväksi, jolloin se osoitetaan virheelliseksi. Lomakkeella tallennettavaksi sallitut attribuutit määritellään osoittamalla ne näytettäviksi. Objektien listaamisen ja näyttämisen yhteydessä attribuutit voidaan asettaa piilottaviksi.

```
1          struct ViewData {
2            ViewData() { object = 0; };
3            obj *object;
4            AttrFlags fkflags;
5            AttrFlags errflags;
6            AttrFlags shwflags;
7            AttrFlags hdnflags;
8          };
```

Kuva 5-1. ViewData-tietue hallintaobjektin tallentamiseen.

Jos halutaan tallentaa useampi kuin yksi objekti, kuten esimerkiksi listausnäkymän luomisessa on tarpeen, tallennetaan objektit ja niitä vastaavat `ViewData`-rakenteet STL-listaan, jonka tyyppi on `list<ViewData>`.

`ViewData`-tietueesta löytyvien arvojen avulla näkymä voidaan tehdä ratkaisuja kunkin attribuutin näyttämisen, lukitsemisen tai virheen ilmaisemisen suhteen. `ViewData`-rakenteen sisältämät `hdnflags`- ja `shwflags`-kartat vaikuttavat ristiriitaisilta, mutta niiden molempien olemassaolo selittyy eri tyyppisten toimintojen erilaisella ajattelutavalla attribuuttien näyttämisen suhteen. Esimerkiksi tallennustoiminnossa on ensisijaisen tärkeää, että tallennetaan vain lomakkeessa syötettäväksi määritellyt attribuutit ja hylätään loput. Objektin attribuuttien näyttämisessä tämä ei ole tarpeen, vaan silloin halutaan määritellä piilotettavat attribuutit, koska attribuuttien ei-haluttu näyttäminen ei ole uhaksi laitteen tietoturvalle.

Yksi huomion arvoinen tietorakenne on sovelluksessa käytettävä `CActionKey`-luokka, joka on esitelty kuvassa 5-2. Sitä käytetään erilaisten toimintoluokkien identifiointiin ja se sisältää tiedot kyseisen toiminnon käyttäjämoodista (`m_user`), toiminnon nimestä (`m_action_name`) ja objektityypistä (`m_objnm`) sekä luokan rakentajan ja tuhoajan, vertailumetodin ja erilaisia get-metodeja. `CActionKey`-luokkaa käytetään sovelluksen toiminto-olioden kartoittamiseen tallentamista ja etsimistä varten.

```
1       class CActionKey {
2         enum WebUser m_user;
3         string m_action_type;
4         string m_objnm;
5       public:
6         CActionKey(enum WebUser user, const string action, const
7         string objnm = "");
8         ~CActionKey();
9         int compare_with(const CActionKey &other) const;
10        enum WebUser GetUser() const;
11        string GetActionType() const;
12        const string GetObjnm() const;
13      };
```

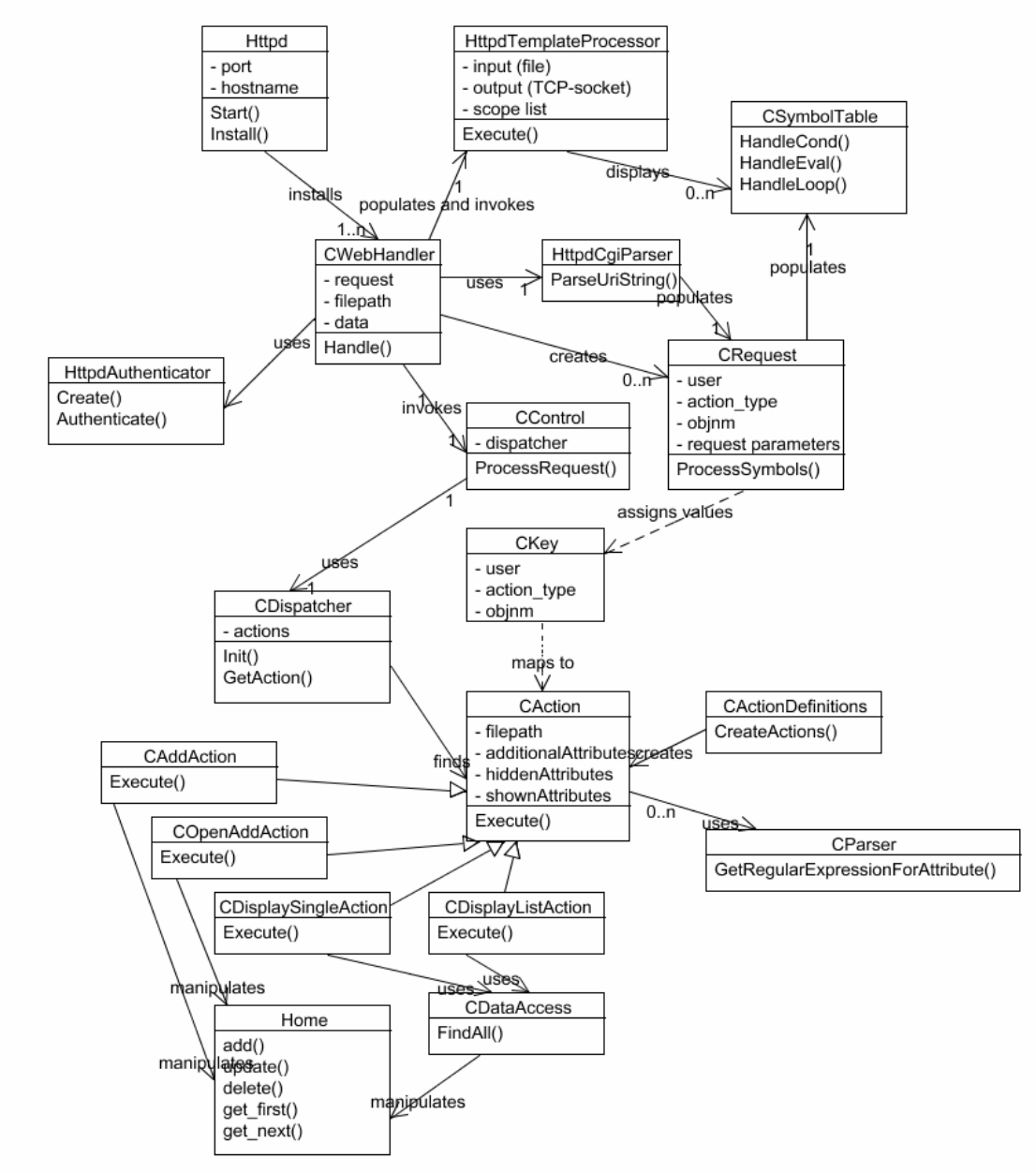
Kuva 5-2. CActionKey-luokka toimintojen identifiointia varten.

Lukuisien toiminto-olioiden tallentaminen tapahtuu omaan STL-karttaansa, jonka tyyppi on `map<CActionKey, CAction>`, eli toimintojen avaimena toimii sen ominaisuuksien perusteella luotu `CActionKey`-olio. Kartasta voidaan helposti hakea saatua HTTP-pyyntöä vastaava toiminto, jolle pyynnön suoritus ohjataan.

5.2.2 Luokat ja metodit

Web-hallintasovellus on jaettu useisiin luokkiin, joiden kesken palvelimen tärkein tehtävä eli HTTP-pyynnön vastaanottaminen, käsittely ja vasteen lähettäminen on jaettu. Luokkakaavio on esitelty kuvassa 5-3. Web-hallintasovelluksen kannalta tärkeitä pääluokkia on viisi, joista periytyy vielä joitakin lisäluokkia. Lisäksi HTTP-palvelimena toimiva Seminole sisältää omat luokkansa, joista suurinta osaa ei kuitenkaan tämän työn

puitteissa ole tarpeen käsitellä. Projektissa toteutettu verkko-ohjelmisto yhdistyy ytimenä toimivaan HTTP-palvelimeen `HttpdFileHandler`- ja `HttpdSymbolTable`-luokista periytettyjen `CWebHandler`- ja `CSymbolTable`-luokkien avulla. Lisäksi HTTP-autentikaatio tapahtuu Seminolen tarjoamalla `HttpAuthentication`-luokalla. Nämä kolme luokkaa ovat käytännössä ohjelman ainoat HTTP-palvelimen omaa ohjelmointirajapintaa käyttävät osat, joten ohjelman muita luokkia voidaan pitää palvelinsovelluksesta riippumattomina.



Kuva 5-3. Web-hallintasovelluksen luokkakaavio.

Luokkien yhteistoiminta perustuu MVC-sovellusarkkitehtuuriin, jota käsiteltiin kohdassa 2.5. Suurin osa luokista edustaa arkkitehtuurin *C*-osaa eli kontrollia.

CWebHandler huolehtii sovelluksen pääsystä ulkomaailmaan, mikä tarkoittaa HTTP-pyyntöjen parametrien jäsentämisestä ja vasteen muotoilu- ja lähetysvaiheen käynnistämisestä. Kommunikoinnista mallin kanssa vastaa CAction-kantaluokasta periytetyt toimintoluokat Config API -rajapinnan kautta. Mallilta saatu tieto välitetään lopulta CSymbolTable-luokalle, joka huolehtii tietojen tarjoamisesta näkymälle sivupohjien ymmärtämässä muodossa.

CWebHandler

HttpdFileHandler-luokasta periytetty CWebHandler-luokka on pyynnön käsittelijä, jonka olioita palvelin luo yhden tai useamman vastaamaan tietyllä etuliitteellä saapuviin HTTP-pyyntöihin. HttpdFileHandler-luokan nimi tulee siitä, että sen avulla palvelinta voidaan käyttää tiedostopalvelimena. Luokan rakentaja vaatii syötteekseen käsittelijää vastaavan etuliitteen ja käytettävän tiedostojärjestelmän tyypin ja oletushakemiston. CWebHandler-olio luodaan palvelimen pääohjelmassa, ja sen elinaika on käytännössä sovelluksen kesto. Erilaisia käsittelijöitä voidaan luoda useampia, joista kukin vastaa tietyllä etuliitteellä saapuvien HTTP-pyyntöjen käsittelystä, esimerkiksi pyyntö osoitteeseen *http://www.palvelin.com/webForm1* ohjautuu etuliitteellä *webForm1* ja osoite *http://www.palvelin.com/webForm2* etuliitteellä *webForm2* rakennettuun käsittelijään. Web-hallintasovelluksessa CWebHandler-käsittelijää käytetään ilman etuliitettä saapuvien pyyntöjen oletuskäsittelijänä.

CWebHandler-oliolle ohjatun pyynnön käsittely tapahtuu useassa eri vaiheessa, joista ensimmäinen on TranslateUri()-metodissa tapahtuva pyynnön jäsennys ja ohjaus kontrollille. Pyydetty toiminto ja objektityypin nimi tallennetaan omiin merkkijonoihinsa, ja loput pyynnöstä tallennetaan STL-karttaan, kuten alikohdassa 5.2.1. kerrottiin. Näiden tietojen perusteella luodaan pyyntökohtainen CRequest-olio, joka välitetään kontrollin rajapinnan kautta eteenpäin.

TranslateUri()-metodissa tapahtuu myös käyttäjän autentikointi, jossa hyödynnetään HTTP-autentikointiskeemoja käyttävää HttpdAuthenticator-luokkaa. Luokan avulla määritetään *realm*-alue, jolle päästäkseen käyttäjän pitää tunnistautua jollakin laitteen *passwd*-moduulilta saadulla käyttäjätunnuksella ja salasanalla. Erilaisia käyttäjätiloja ovat *admin, monitor* ja *engineer,* joista kullekin on määritelty omat sallitut toimintonsa.

Kun kontrolli on käsitellyt pyynnön, CWebHandler-olio hakee pyyntöä vastaavalta CRequest-oliolta kontrollin määrittämän symbolitaulun ja sivupohjan tiedostonimen, joiden osoittimet se tallentaa yksityisiin muuttujiinsa ja jatkaa suoritusta SendFile()-metodiin. SendFile()-metodissa kutsutaan staattista HttpdFSTemplateShell ::Execute()-metodia, jolle annetaan pyynnön tilatietue ja symbolitaulu. Execute()-metodi muodostaa annettujen parametrien perusteella yksikäsitteisen näkymän ja lähettää sen pyynnön tehneelle käyttäjälle. Metodi on vastaava kuin alikohdassa 4.3.3. esitellyssä esimerkissä, paitsi että tässä käytetään symbolitauluna HttpdSymbolMap-luokan sijaan HttpdSymbolTable-luokasta periytettyä luokkaa, jossa jokainen sivupohjalta tullut komento määritellään itse. HttpdSymbolMap-luokan tarkoitus on helpottaa C-kielen rakenteisiin tallennetun tiedon esittämistä näkymässä

esimääritetyillä komentokäsittelijöillä, mutta tässä projektissa käytettävillä tietorakenteilla on helpompaa kirjoittaa komentokäsittelijät itse. Komentokäsittelijät ovat sivupohjien käyttämiä takaisinkutsuja, joiden tehtävänä on tulostaa pyydetyt muuttujat näkymään.

CRequest

`CRequest`-luokka on tarkoitettu pyyntökohtaisen tiedon tallennuspaikaksi, ja siten se sisältää myös pyynnön aikana ylläpidettävän tilatiedon. Tilatietoja ovat esimerkiksi tiedot edellisestä pyynnölle suoritetusta toiminnosta ja pyynnölle määritetty data ja sivupohja. Suorituksen aikana `CRequest`-olio luodaan pyynnön käsittelijässä, jossa sille annetaan alkuarvoksi HTTP-pyynnöstä jäsennetty toiminto, objektityypin nimi ja pyynnön loppuosa.

CControl

`CControl`-luokka sisältää sovelluksen kontrolliosan ytimen. Käsittelijä kutsuu tuntemaansa `CControl`-oliota ja välittää sille pyyntökohtaisen `CRequest`-olion. `CControl` ei tiedä, minkä toimintotyypin kanssa on milloinkin tekemisissä, vaan sen tehtävä on ainoastaan toimittaa saamansa parametrit oikeaan paikkaan, eli `CDispatcher`-oliolta saamalleen toiminto-oliolle. Erityisesti `CControl`-olio ei tiedä mitään pyynnön loppuosan sisältämistä parametreista tai niiden merkityksestä, vaan ne syötetään `CDispatcher`-oliolta saadulle toiminto-oliolle sellaisenaan. `CControl`-olio asettaa pyynnölle myös käsiteltävää objektityyppiä vastaavan navigointivalikon tilan. Vaikka `CControl`-olion rooli pyynnön käsittelyssä ja vasteen muodostamisessa on varsin pieni, on sillä tärkeä tehtävävä pyynnön kuljettamisessa `CDispatcher`-oliolle ja siltä saadun toiminnon onnistuneessa suorittamisessa.

`CControl`-olio luodaan palvelimen käynnistyksen yhteydessä ja tuhotaan sen sulkeutuessa. Se tarjoaa pyynnön käsittelevälle `CWebHandler`-luokalle yksinkertaisen rajapinnan `ProcessRequest()`-metodinsa kautta, joka vaatii syötteekseen pyynnöllä alustetun `CRequest`-olion.

CDispatcher

`CDispatcher`-luokka on tarkoitettu erilaisten toimintoluokkien hallintaan. `CWebHandler`-olion tapaan myös `CDispatcher`-olio luodaan palvelimen käynnistyksen yhteydessä ja tuhotaan vasta sen sulkeutuessa. `CDispatcher::Init()`-metodi luo toimintojen määrittelyolion (`CActionDefinitions`) ja pyytää tätä luomaan sille määritetyt toiminnot. Kun kontrolli kutsuu `CDispatcher::GetAction()`-metodia, se käy läpi määrittelyolion antamat toiminnot ja palauttaa kontrollille sen toiminto-olion osoittimen, joka vastaa saadun pyynnön käyttäjämoodia, toimintotyyppiä ja objektityyppiä. Mikäli näillä kolmella muuttujalla määritellyn `CActionKey`-olion perusteella löydetään useampi kuin yksi toiminto, voidaan oikea toiminto tunnistaa pyynnön loppuosan perusteella. Jos objektityyppiä ei ole määritelty, voidaan palauttaa osoitin tietylle käyttäjämoodille ja toimintotyypille määritellylle oletustoiminnolle. Jos toimintoa ei löydy, palautetaan

osoitin `CAction`-kantaluokan olioon, joka ei tee mitään, mutta jota voidaan käsitellä kuin mitä tahansa muuta toiminto-oliota.

CActionDefinitions

`CActionDefinitions`-luokassa määritellään kaikki web-hallintasovelluksen tarvitsemat toiminnot, joita ovat erikoismäärittelyt ja oletustoiminnot. Erikoismäärittelyt ovat tietylle käyttäjämoodi, toiminto ja objektityyppi –yhdistelmälle määriteltyjä poikkeustoimintoja, joiden avulla voidaan esimerkiksi piilottaa halutut attribuutit jonkin objektityypin näkymästä. Esimerkki erikoismääritellystä `COpenAddAction`-toiminnosta näkyy kuvassa 5-4. Toiminnon sivupohjaksi on määritelty *add.thtm*, käyttäjämoodiksi *ADMIN*, objektityypiksi *bridge* ja piilotettaviksi attribuuteiksi *mac* ja *stpstate*. Viimeisenä oleva ennaltamääritetyt attribuutit on asetettu tyhjäksi, jolloin ainuttakaan attribuuttia ei ole asetettu käyttäjän puolesta. Käyttäjämoodin avulla toteutetaan käyttäjän autorisointi, eli varmistetaan että jokaisella käyttäjällä on käytettävissään vain halutut toiminnot. Esimerkiksi *monitor*-käyttäjälle ei haluta mahdollistaa hallintaobjektien lisäämistä, joten käyttäjämoodilla *MONITOR* ei luoda `COpenAddAction-` tai `CAddAction`-luokan erikois- tai oletusmäärittelyjä.

```
1  new COpenAddAction("/add.thtm", ADMIN,
2                     pList->get_object_type("bridge"),
3                     "mac stpstate",
4                     ""
5                     )
```

Kuva 5-4. Erikoismäärittely COpenAddAction-toiminnolle.

Erikoismäärittelyjen lisäksi määritellään halutuille käyttäjämoodeille ja toimintotyypeille oletustoiminnot, joiden objektityyppi on avoin. Mikäli pyyntöä vastaavaa toimintoa ei löydy, käytetään `CAction`-kantaluokan oliota. `CActionDefinitions`-olio luodaan `CDispatcher::Init()`-metodin yhteydessä, joten olion elinikä vastaa käytännössä sovelluksen elinikää. Tämä tarjoaa tehonsäästöä, koska toiminto-oliot on tällöin luotava vain kerran palvelimen käynnistyksen yhdeydessä. Tämä tarkoittaa myös sitä, että toiminto-olioiden metodien on oltava tilattomia (*stateless, re-entrant*), jotta vältytään yhteisen datan käytöstä aiheutuvilta ristiriidoilta yhtäaikaisia pyyntöjä käsitteltäessä. Toiminto-olion suoritusmetodi ei siis säilytä tietoa sen suoritusten tai HTTP-pyyntöjen käsittelyn välillä, vaan toimii ainoastaan saamiensa syötteiden ja mallilta saamansa datan perusteella. Kaikki pyyntöjen välinen ja toimintojen suorittamisen välinen tilatieto säilytetään näkymässä, `CRequest`-oliossa ja pysyvän tiedon sisältävässä mallissa.

CAction

`CAction` on toimintoluokkien kantaluokka, joka sisältää perittäviä muuttujia ja metodeja, jotka ovat suurimmalle osalle toiminnoista tarpeellisia. Perintä vähentää tarvittavien koodirivien määrää ja siten ohjelmointityön määrää, koska periytetyille toiminnoille tarvitsee määrittää vain kullekin ominaiset toteutustavat ja erityispiirteet. Sovelluksen kontrollin näkökulmasta `CAction`-luokan tärkein metodi on `Execute()`,

joka vaatii syötteekseen `CRequest`-olion ja palauttaa suorituksen onnistumisesta kertovan boolean-arvon. `Execute()`-metodin tarkoitus on piilottaa alleen toiminnot oikeasti suorittavien toimintokohtaisten `ExecuteAction()`-metodien palauttamat virhekoodit, jolloin ylemmän kerroksen päätöksenteko suorituksen onnistumisesta helpottuu. Tämä rajapinta on kaikille `CAction`-luokasta periytetyillä luokilla sama, vaikka niiden sisäinen toteutustapa vaihtelee.

`CAction::Execute()`-metodi palauttaa boolean-arvon, joka kuvaa sitä, onko pyydetyn toiminnon suoritus saatu loppuun. Jotkin pyynnöt vaativat useamman kuin yhden toiminnon suorittamisen, jolloin metodin palauttaman arvon perusteella voidaan päätellä, onko uuden toiminnon hakeminen tarpeen vai voidaanko ohjelman suoritusta jatkaa eteenpäin. Esimerkiksi tallennustoiminnon jälkeen ei kannata palauttaa käyttäjälle tyhjää näkymää, vaan on järkevämpää suorittaa kyseisen objektin `CDisplayAction`, jolla kyseinen objekti voidaan näyttää.

COpenAddAction

Hallintaobjektin lisäyslomakkeen muodostamisesta vastaa `COpenAddAction`, joka luo ensin hallintaobjektin olion annetun objektityypin ja sen ennaltamäärättyjen avainattribuuttien perusteella. Olion perusteella luodaan näkymään lomake hallintaobjektin attribuutteja vastaavilla syötekentillä. Tämä luokka ei tarvitse tietoa laitteelta, mutta se tarvitsee Config API-rajapintaa oikeanlaisen hallintaobjektin olion luomiseen. Hallintaobjektin olio tallennetaan `ViewData`-tietueeseen, jossa on myös muut tarvittavat tiedot lisäyslomakkeen esittämiseen, kuten tiedot ennaltamääritetyistä avainattribuuteista, näytettävät attribuutit ja edellisessä lisäysyrityksessä tapahtuneet virheet.

COpenEditAction

`COpenEditAction` vastaa olemassaolevan hallintaobjektin muokkauslomakkeen muodostamisesta. `ExecuteAction()`-metodissaan `COpenEditAction`-olio luo ensin olion muokattavaksi halutun objektin avainattribuuttien perusteella. Tämän hallintaobjektin olion avulla se hakee mallilta hallintaobjektin arvot, jotka tallennetaan olioon ja `ViewData`-tietueeseen. `ViewData`-tietueessa on tarvittava tieto lomakkeen esittämiseen, ja siitä selviävät näytettävät attribuutit ja edellisessä muokkausyrityksssä mahdollisesti tapahtuneet virheet. Tietojen hakuun käytetään Config API:n `home::get_first()`-metodia, joka tarvitsee syötteekseen avainattribuuttien arvot sisältävän hallintaobjektin olion ja osoittimen, jonne haettava objekti tallennetaan. Hallintaobjektin avainattribuutteja käytetään tiedonhaun suodattimena. Kun sen kaikki avainattribuutit on määritelty, palauttaa malli suodattimen perusteella vain yhden objektin.

CDisplaySingleAction

`CDisplaySingleAction`-luokan tehtävänä on hallintaobjektin tietojen hakeminen mallilta avainattribuuttien perusteella. Tietojen hakuun käytetään `home::get_first()`-metodia. Olion pitää tarkistaa myös hallintaobjektille määritetyt piilotettavat attribuutit ja täydentää niitä vastaavat objektikohtaiset kartat.

`CDisplaySingeAction`-luokan erikoisuus on se, että sen pitää tarkistaa myös haettavien objektien viitteet muihin objekteihin. Viitteiden perusteella luodaan näkymän tarvitsemat linkit viiteobjekteihin. Toiminnon tarvitsemat viitteet määritellään `CActionDefinitions`-luokassa sen erikoismäärittelyn yhteydessä.

CDisplayListAction

`CDisplayListAction`-luokan tehtävänä on hallintaobjektien hakeminen mallilta suodatinattribuuttien perusteella. Ensimmäisen objektin hakemiseen käytetään Config API:n `home::get_first()`-metodia. Seuraavat objektit haetaan `home::get_next()`-metodilla, joka tarvitsee syötteikseen suodatin- ja kohdeobjektin lisäksi vielä edellisenä saadun objektin osoittimen. Päätös seuraavan objektin hakuyrityksestä perustuu hakumetodien palauttamiin virhekoodeihin. Hakua jatketaan, mikäli metodin palauttama virhekoodi on `dc_error_t::OK` ja lopetetaan sen ollessa `dc_error_t::NO_MORE_OBJECTS`. `CDisplaySingleAction`-luokan tavoin myös listauksessa on huolehdittava piilotettavien attribuuttien asettamisesta mahdollisia erikoismäärittelyjä vastaaviksi.

CAddAction

`CAddAction`-luokka vastaa hallintaobjektin lisäämisestä annetuilla attribuuttien arvoilla. `CAddAction`-olio käyttää `ExecuteAction()`-metodissaan Config API:n `home::add()`-metodia, joka tarvitsee syötteekseen tallennettavan objektin. Ennen tallennusta olion on tarkistettava käyttäjän tallennusoikeus asetettaville attribuuteille. Lisäksi tallennettavalle objektille asetetaan sille mahdollisesti ennaltamääritetyt attribuuttien arvot.

CUpdateAction

`CUpdateAction`-luokka muistuttaa toiminnaltaan `CAddAction`-luokkaa, mutta tallentamiseen käytetään nyt `home::update()`-metodia, joka on olemassa olevan hallintaobjektin päivittämiseen tarkoitettu metodi. Mikäli päivitettävää objektia ei tästä huolimatta ole, palauttaa kutsu virhettä ilmaisevan koodin. Tällainen tilanne voi esiintyä esimerkiksi silloin, kun käyttäjä on avannut hallintaobjektin muokkausnäkymän, mutta toinen käyttäjä käy poistamassa saman objektin ennen kuin ensimmäinen ehtii tallentaa tekemiään muutoksia.

CDeleteAction

`CDeleteAction` on `CAction`-luokasta periytetyistä toimintoluokista yksinkertaisin, ja sen tehtävänä on avainattribuuteilla osoitetun hallintaobjektin poistaminen. Tähän käytetään Config API:n `home::delete()`-metodia, joka vaatii syötteekseen poistettavan objektin.

CSymbolTable

`CSymbolTable`-luokka on periytetty Seminolen tarjoamasta `HttpdSymbolTable`-luokasta, ja se vastaa sille annetun tiedon tulkinnasta sivupohjien ymmärtämässä

muodossa. Se sisältää tarvittavan logiikan erilaisten silmukoiden, vertailujen ja arviointien tekemiseen sivupohjalta saatujen komentojen ja kontrollilta vastaanotettujen `ViewData`-tietueiden tarjoaman datan perusteella. Kuvassa 5-5 esitellään `CSymbolTable::HandleCond`-metodi, joka vastaa sivupohjalta saatujen *if*-komentojen vertailusta. Esimerkiksi jos näkymä haluaisi tietoonsa onko sen hetkinen hallinta-attribuutti avain, se voitaisiin selvittää sivupohjakomennolla *%{if:key}%*. Tällöin rivin 3 `HttpdConditionalCommand::Name()`-metodilla saatu komennon nimi *key* palauttaa boolean-arvon muuttujalle `mCurKey`.

```
1   int CSymbolTable::HandleCond(HttpdConditionalCommand *p_cond)
2   {
3      const char *name = p_cond->Name();
4
5      if (strcmp(name,"key")==0) {
6       return (ReturnBool(mCurKey));
7      } else if (strcmp(name,"fkey")==0) {
8        return (ReturnBool(mCurFKey));
9      } else if (strcmp(name,"notbool")==0) {
10       return (ReturnBool(mCurNotBool));
11     } else if (strcmp(name,"checked")==0) {
12       return (ReturnBool(mCurError));
13     } else if (strcmp(name,"hidden")==0) {
14       return (ReturnBool(mCurHidden));
15     } else if (strcmp(name,"shown")==0) {
16       return (ReturnBool(mCurShown));
17     } else {
18       return (HTTPD_TEMPLATE_UNKNOWN_NAME);
19     }
20  }
```

Kuva 5-5. CSymbolTable-luokan konditionaalikäsittelijä.

Seminolen sivupohjan tulkinnasta vastaava olio kutsuu pyynnölle asetetun symbolitaulun konditionaalikäsittelijää eli `CSymbolTable::HandleCond`-metodia. Metodi hakee saadun komennon nimen ja palauttaa sen perusteella valitun muuttujan vertailun tuloksen. Esimerkiksi sivupohjan komento *%{if:key}%* palauttaa arvon *true* tai *false*, riippuen boolean-muuttujan `mCurKey` totuusarvosta.

5.2.3 Integrointi kohdeympäristöön

Prosessimoduulin integrointi kohdeympäristöön vaatii moduulien kantaluokaksi määritellyn `mgr_module_base`-luokan toteuttamista moduulin ominaisuuksien edellyttämällä tavalla. Moduuli periyttää kantaluokasta oman moduuliluokkansa, jonka avulla se paikantaa kohdeympäristön resurssit, kuten Config API:n ja sitä kautta esimerkiksi hallintaobjektien *home*-rajapinnat. Moduuliluokan monimutkaisuus riippuu siitä, paljonko kyseisellä moduulilla on I/O-operaatioita vaativia tiedostokuvaimia, jotka vaativat kommunikointia muun järjestelmän kanssa. Tiedostokuvaimia ovat esimerkiksi tiedostot, soketit ja laitteistokahvat, joille voidaan kohdistaa estäviä I/O-kutsuja. Joitakin Unixin I/O-järjestelmäkutsuja esiteltiin kohdassa 2.2.

Web-hallintasovelluksen integrointi kohdejärjestelmään oli helppoa, koska sille ei toteutettu järjestelmän kanssa kommunikoivia tiedostokuvaimia. Tästä syystä sovelluksen integroimiseksi riitti moduuliluokassa suoritettava Config API –rajapinnan noutaminen, jonka avulla voitaisiin hakea HTTP-pyyntöjen kautta konfiguroitavaksi pyydettävien hallintaobjektien *home*-rajapinnat. Kuvassa 5-6 on esitelty web-hallintasovelluksen `main()`-metodi, jonka tarkoitus on luoda moduuliluokan olio ja alustaa se. Rivillä 3 luodaan palvelinolio, joka sisältää tarvittavat metodit HTTP-pyyntöjen vastaanottamiseen ja vasteiden lähettämiseen. Rivillä 7 suoritettavassa `dc_module_web::init()`-metodissa suoritetaan kantaluokan `init()`-metodin kutsuminen ja Config API –rajapinnan noutaminen. Tämän jälkeen rivillä 8 alustetaan luotu palvelinolio, mikä tarkoittaa sulautetun tiedostojärjestelmän alustamista ja HTTP-pyynnön käsittelijöiden asettamista palvelinoliolle. Toteutettu web-hallintasovellus käytti yhtä pyynnönkäsittelijää, jonka tehtävänä oli pyynnön parametrien jäsentäminen ja välittäminen alemman tason sovelluslogiikalle. Lopuksi palvelin käynnistetään `Httpd::Start()`-metodilla, jolloin se jää ikuiseen silmukkaan vastaanottamaan ja käsittelemään HTTP-porttiin saapuvia pyyntöjä.

```
1          int main(int argc, char *argv[]) {
2
3            gp_WebServer = new Httpd(c_servername, c_serverport);
4            DC_DEBUG_INIT("web");
5            dc_module *mod = new dc_module_web();
6            if(mod) {
7              mod->init(argc, argv);
8              Initialize_The_Web_Server();
9              if (!gp_WebServer->Start()) {
10               exit(1);
11             }
12             delete mod;
13           }
14           return (0);
15         }
```

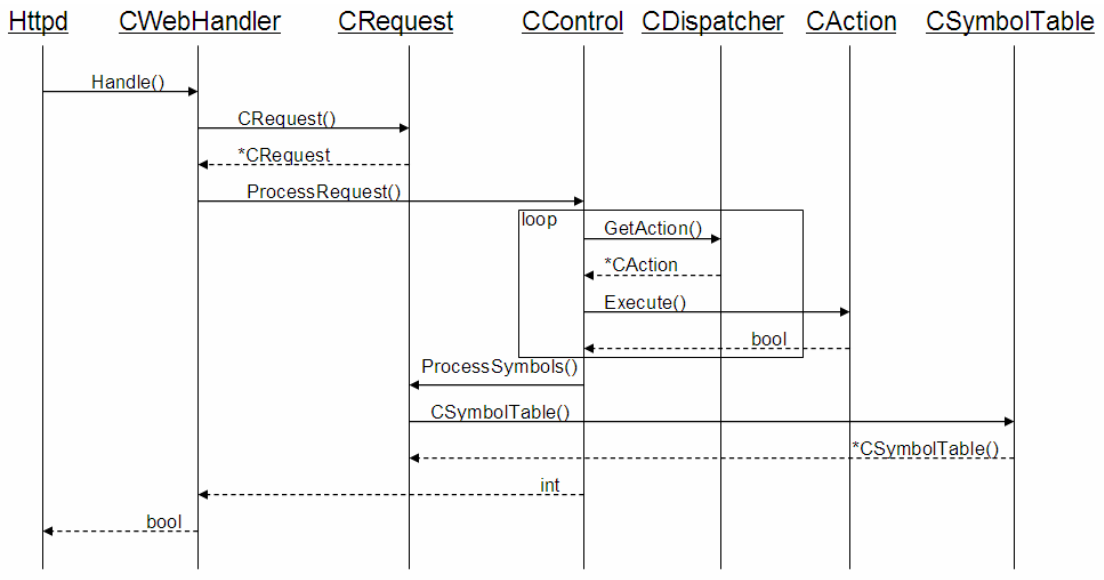
Kuva 5-6. Web-hallintasovelluksen main()-metodi.

Laajemman integroinnin toteuttamiseksi hallintasovellus voisi käyttää `dc_module_base::run()`-metodia, joka rekisteröi moduulin järjestelmän sisäiselle valvontamoduulille. Tällöin voitaisiin käyttää esimerkiksi IPC-viestejä hallintasovelluksen asetusten ajonaikaiseen muokkaamiseen. Hallintasovelluksen ensimmäiseen versioon riitti kuitenkin perustoiminnallisuuden toteuttaminen.

5.2.4 Vasteen muodostaminen

Edellisessä alikohdassa esitellyistä luokista luodut oliot suorittavat web-palvelimen tärkeimmän tehtävän, joka on HTTP-vasteen muodostaminen saadun pyynnön perusteella. Suorituksen kulku on monivaiheinen, ja siihen liittyvät oliot muodostuvat noin kymmenestä luokasta, mikäli lasketaan vain toteutetun verkko-ohjelmiston kannalta oleelliset luokat. Tyypillinen yksinkertaistettu tapahtumasekvenssikaavio olioiden välisistä suhteista on esitetty kuvassa 5-7. Siinä esiteltyjen olioiden lisäksi suoritukseen liittyy lukuisia Seminolen sisäisten luokkien olioita, jotka vastaavat

palvelimen alemman tason toiminnasta ja sivupohjien prosessointiin liittyvistä tehtävistä. Seminolen omia luokkia ei kuitenkaan muutamaa poikkeusta lukuunottamatta käsitellä.



Kuva 5-7. Tapahtumasekvenssikaavio HTTP-pyynnön käsittelystä.

Kun HTTP-pyyntö saapuu palvelimelle `CWebHandler`-luokalle osoitetulla etuliitteellä, pyyntö ohjataan siitä luodulle oliolle ja tarkemmin sanottuna sen `Handle()`-metodille. `Handle()`-metodi käy läpi käsittelijäluokalle ominaiset pyynnön prosessoinnin vaiheet, joita käyttämämme `HttpdFileHandler`-käsittelijäluokan `Handle()`-metodilla ovat `TranslateUri()`- ja `ProcessUri()`-metodit. Toteutetun verkko-ohjelmiston käsittelijä toteutti näistä `TranslateUri()`-metodin ja `ProcessUri()`-metodin kautta kutsuttavan `SendFile()`-metodin. Ensimmäisen metodin tehtävänä on jäsentää saadun HTTP-pyynnön parametrit verkko-ohjelmiston ymmärtämään muotoon. Jälkimmäinen vastaa edellisissä vaiheissa määritetyn tiedoston lähettämisestä käyttäjälle ja siihen palataan alikohdan lopussa. Kuvassa 5-7 näkyy ainoastaan `Handle()`-metodi luokan julkisen rajapinnan mukaisesti.

Edellä esitetty osuus vastaa verkko-ohjelmiston palvelinsidonnaista osaa, joka on tässä tapauksessa toteutettu Seminole-palvelimen API-rajapintaa vastaavalla tavalla. Verkko-ohjelmiston tärkein osuus tapahtuu sen kontrollissa, jossa tapahtuu neuvottelu mallin kanssa pyydetylle toiminnolle osoitetun toiminto-olion avulla. Käytännössä pyynnön suoritus ohjataan `CWebHandler`-oliolle määritetylle `CControl`-oliolle, jonka `ProcessRequest()`-metodia käsittelijä kutsuu. Käsittelijä antaa `ProcessRequest()`-metodille jäsentämänsä toiminnon, objektityypin nimen ja pyynnön loppuosan, jotka se tallentaa pyyntökohtaiseen `CRequest`-olioon. Tehokkuussyistä kahdesta viimeisestä välitetään vain osoitin kokonaisen olion sijaan. Metodin suorittamisen jälkeen `CRequest`-olioon pitäisi olla tallennettuna näkymään luomiseen tarvittava sivupohja ja symbolitaulu. Metodin toiminta näkyy kuvassa 5-8.

```
1       int CControl::ProcessRequest(CRequest *req) {
2           CAction *acn = 0;
3           bool done = false;
4
5           while (!done) {
6             acn = m_dp->GetAction(req);
7             done = acn->Execute(req);
8           }
9
10          req->ProcessSymbols(GetMenuState(req
11                    ->GetObjnm(), *req->GetParams()));
12
13          return (0);
14      }
```

Kuva 5-8. CControl::ProcessContent()-metodi.

`ProcessContent()`-metodi luo aluksi tyhjän `CAction`-osoittimen. Lisäksi se luo boolean-muuttujan virheen tarkastelua varten ja alustaa sen *false*-arvolla, joka osoittaa että toimintoa ei ole vielä suoritettu. Seuraavaksi suoritus siirtyy *while*-silmukkaan, jossa tapahtuu tarvittavan toiminto-olion osoittimen noutaminen `CDispatcher`-oliolta `acn`-muuttujaan rivillä 6 ja sen `Execute()`-metodin kutsuminen rivillä 7. Noudettava toiminto-olio määräytyy `CRequest`-olion parametrien perusteella. Silmukan tehtävä on tarkastella `Execute()`-metodin palauttamaa virheen ilmaisevaa boolean-arvoa `done` ja virheen tai uuden toimintomäärityksen ilmaantuessa hakea uusi toiminto. `Execute()`-metodi piilottaa alleen varsinaisen toiminnon suorittavan `ExecuteAction()`-metodin ja sen palauttamat monipuolisemmat virhekoodit. Todellisten virhekoodien arviointi on erityisen tärkeää esimerkiksi lomakkeiden syötteiden tallentamisen yhteydessä, sillä lomakenäkymään pitää pystyä palaamaan, mikäli käyttäjän antamat syötteet ovat virheellisiä. Lisäksi jos *show*-näkymälle ei löydetä näytettävää objektia, voidaan sen sijasta näyttää jokin muu näkymä, esimerkiksi objektin lisäyslomake.

Kun pyynnön suoritus on saatu päätökseen, palauttaa toiminto-olion `Execute()`-metodi arvon *true*, jolloin päästään ulos *while*-silmukasta. Sen jälkeen kutsutaan `CRequest`-olion `ProcessSymbols()`-metodia rivillä 10, jonka syötteeksi annetaan navigointivalikolle määritetty tila. Metodin avulla `CRequest`-olio luo itselleen `CSymbolTable`-olion, jolle se antaa objektityypin nimen, navigointivalikon tilan ja toiminnolta saamansa datan. `CSymbolTable`-olion tehtävänä on mallilta saadun tiedon muuntaminen sivupohjien ymmärtämään muotoon erilaisten virhe- ja näkyvyystietojen sekä objektien attribuuttien perusteella. Tämän jälkeen pyynnön käsittely on kontrollin osalta saatu päätökseen ja prosessoinnin onnistumistumista ilmaiseva *int*-arvo voidaan palauttaa pyynnön käsittelijälle.

Prosessointikutsun tehnyt `CWebHandler`-olio jatkaa suoritustaan ja tallentaa pyynnön perusteella saadun sivupohjan tiedostonimen sille tarkoitettuun muuttujaan. Viimeinen olion suorittama vaihe on sen `SendFile()`-metodi, jossa suoritetaan käsittelijälle määrätyn tiedoston lähettäminen käyttäjälle. Koska tiedosto on tässä tapauksessa Seminole-näkymäkutsuja sisältävä HTML-sivupohja, kutsutaan staattista `HttpdFSTemplateShell::Execute()`-metodia, joka prosessoi käsittelijälle

määritetyn sivupohjan ja `HttpdSymbolTable`-olion osoittaman datan perusteella yksikäsitteisen näkymän.

6 Työn tulokset

Tämän diplomityön kannalta hallintasovelluksen riittävä valmistumisaste saavutettiin, kun web-hallintasovelluksen toiminnallisuus vastasi laajuudeltaan suurimmalta osin CLI-hallintasovelluksen toiminnallisuutta. Tämä edellyttää, että suurin osa näkymien vaatimista sivupohjista ja sovellusmoottorin tarvitsemista ominaisuuksista on toteutettu, vaikka niiden hyödyntämisessä olisi vielä puutteita. Tässä luvussa tarkastellaan tällä valmistumisasteella saavutettuja tuloksia.

6.1 Soveltuvuus loppukäyttöön

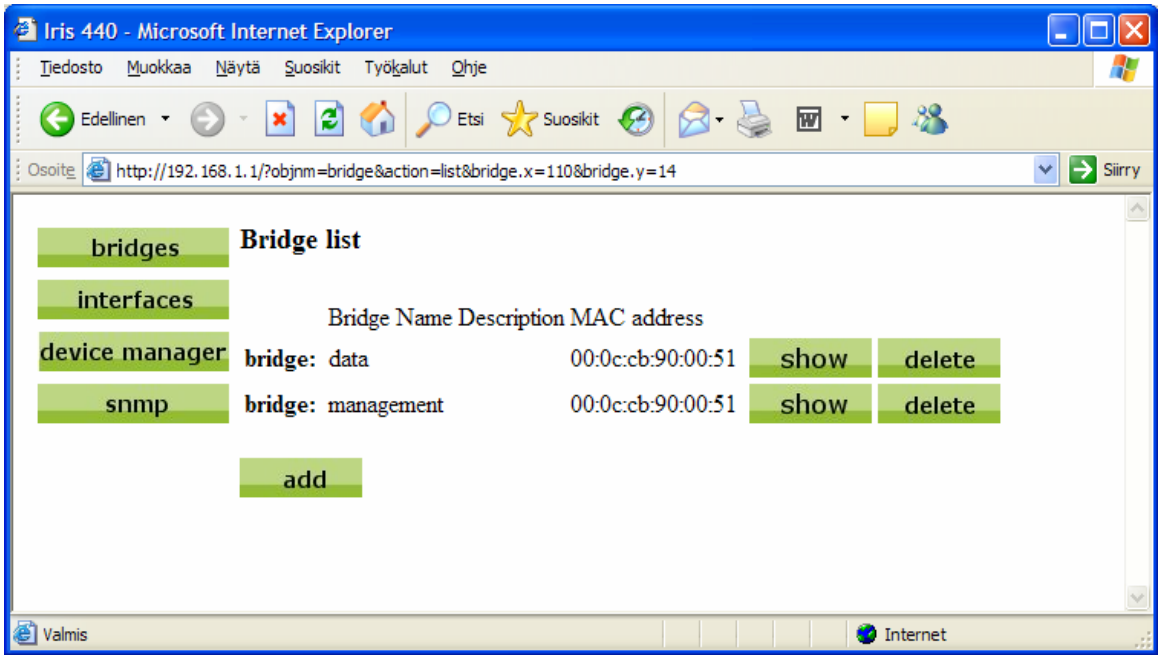
Web-hallintasovelluksen tavoitteena on tarjota käyttäjälle helppokäyttöinen, selaimella toimiva hallintakäyttöliittymä, joka vastaa monipuolisuudeltaan olemassa olevaa CLI-hallintasovellusta mahdollisimman laajasti. Näiden seikkojen kannalta sovelluksen toteutus onnistui kattavasti, muutamaa toistaiseksi puuttuvaa ominaisuutta lukuunottamatta. Graafinen käyttöliittymä tarjoaa käyttäjälle helposti opittavan ja muistettavan sekä yhdenmukaisen ja selkeän mahdollisuuden laitteen hallintaan, ja se sopii varsinkin laitteen asetuksia vain harvoin muokkaavalle käyttäjälle. Toisaalta CLI-hallintasovellus säilyy käytössä edelleen, joten edistynyt käyttäjä voi yhä hyödyntää tehokkaita komentorivikäskyjä.

Hallintasovellus takaa verkon päätelaitteelle sen tarvitseman tietoturvan käyttäjän autentikoinnin ja mahdollisen SSL-salauksen avulla. Jokaiselle kirjautumisoikeuden omistavalle käyttäjälle on määritelty oikeustaso kolmesta eri vaihtoehdosta: *ADMIN*, *MONITOR* ja *ENGINEER*. Käyttäjälle näkyvät valvonta- ja hallintaominaisuudet määräytyvät käyttäjän tason mukaan, jolloin esimerkiksi *monitor*-käyttäjälle on mahdollista vain laitteen asetusten tarkastelu. SSL-salaus ei ole käytössä oletuksena, mutta se voidaan helposti ottaa käyttöön palvelimen käynnistyksen yhteydessä, mikäli etähallinnan käyttö tai muulla tavoin epäluotettava käyttöympäristö vaatii tiedon luottamuksellisuuden varmistamisen. Toistaiseksi SSL-salauksen käyttöönotto onnistuu vain käännösaikaisilla asetuksilla, mutta tarvittaessa käyttöönotto voidaan mahdollistaa myös käynnistysparametrien avulla.

Toimintojen autorisointi perustuu autentikoinnin tarjoaman käyttäjän tunnistamiseen ja toiminnolle määritettäviin oikeustasoihin. Autorisointi perustuu toiminnoille määritettävään oikeustasoon, jonka perusteella toiminnon käyttö ohjautuu käyttöoikeuden omistavalle käyttäjälle. Esimerkiksi sivulla 65 kuvassa 5-4 määritellyssä toiminnossa oikeustaso on *ADMIN*, jolloin toiminnon käyttö onnistuu vain laitteen hallintakäyttäjän tunnuksilla. Autorisoinnin avulla voidaan helposti määrittää toiminnon eri oikeustasoille erilaiset näkymät asettamalla kutakin oikeustasoa vastaavalle toiminnolle oma sivupohjansa.

Hallintasovellukseen kirjauduttuaan käyttäjä päätyy etusivulle, jossa on vasemmassa reunassa navigointivalikko ja sivulla näkyy tietoja laitteesta. Navigointivalikon painikkeista käyttäjä pääsee muutamaa poikkeusta lukuunottamatta haluamansa hallintaobjektin listausnäkymään, josta esimerkkinä on siltojen listausnäkymä kuvassa 6-1. Listausnäkymä sisältää painikkeet olemassa olevien siltojen näyttämiseen ja poistamiseen sekä uuden sillan lisäämiseen. Listassa näkyvien hallinta-attribuuttien

lisäksi jokainen silta sisältää myös muita attribuutteja, joita käyttäjä voi tarkastella haluamansa sillan *show*-näkymässä, joka on esitelty liitteessä 16. Kaikkia attribuutteja käyttäjä ei kuitenkaan hallintasovelluksen kautta näe, vaan osa niistä on asetettu kokonaan piilotettavaksi toimintojen erikoismääritysten yhteydessä. *Show*-näkymän *edit*-painikkella käyttäjä pääsee muokkaamaan kyseistä siltaa sen *edit*-näkymään, joka on esitelty liitteessä 17. Sillalle muokattavaksi asetetut attribuutit ovat erilaiset kuin näytettäväksi asetetut, joten esimerkiksi sen MAC-osoite ei ole käyttäjän muokattavissa, vaan se määräytyy laitteelle asetetun laiteosoitteen mukaisesti.



Kuva 6-1. Laitteen siltojen listausnäkymä.

Näkymien välisten siirtymien johdonmukaisuus ja suvujuus on hallintaobjektikohtaista, joten erilaiset näkymätyypit toimivat joillekin objekteille paremmin ja toisille huonommin. Keskimäärin näkymät ovat kuitenkin selkeitä ja intuitiivisia, jolloin navigointi ja käyttäjän tarvitsemien toimintojen löytäminen tapahtuu nopeasti. Kun hallintasovelluksen perustoiminnallisuus on kunnossa, voidaan sen käytettävyyttä ja näkymien monipuolisuutta parantaa luomalla uusia näkymätyyppejä, joissa erilaisten hallintaobjektien ominaispiirteet huomioidaan paremmin.

Myös hallintasovelluksen suorituskyky ja muistivaatimus vaikuttivat alustavien testien perusteella hyvältä. Verkko-ohjelmiston nopeutta arvioitiin mittaamalla sen kuluttamaa aikaa pyynnön välittämiseen HTTP-rajapinnan ja laitteen Config Managerin välillä. Mittausten avulla todettiin, että verkko-ohjelmiston kuluttama aika pyyntöjen käsittelyyn oli vain murto-osa Config Managerin käyttämästä ajasta, joten se ei missään tapauksessa muodostuisi pullonkaulaksi pyynnön käsittelyssä. Liitteessä 18 on esitelty yhden sillan sisältävän listausnäkymän muodostamiseen kulunut aika. Tuloksista huomataan, että Config Manager käytti tiedonhakuun IPC-viestien avulla noin 1,5 sekuntia, jolloin itse verkko-ohjelmiston kuluttama aika on noin 0,6 sekuntia. Suurin osa tästä ajasta kuluu hallintaobjektien käsittelyyn Config API –kutsujen avulla, joiden

suoritusnopeuteen verkko-ohjelmisto ei voi vaikuttaa. Jos kokonaissuoritusajasta vähennetään ensimmäisen ja viimeisen Config API –kutsun välillä kuluva aika, on verkko-ohjelmiston kuluttama aika vain joitakin kymmeniä millisekunteja. Mittaustuloksia on syytä arvioida suhteellisina, koska käytetyn laitteen kellotaajuus oli vain 200MHz lopullisen kohdeympäristön 300MHz kellotaajuuden sijaan. Laitteen Flash-muistia sovellus vie kokonaisuudessaan noin 800 kilotavua, joista 124 kilotavua muodostuu HTML-painikkeiden kuvatiedostoista. RAM-muistia sovellus käyttää noin 12,5 megatavua, josta suurin osa koostuu sen lataamista dynaamisesti linkitetyistä kirjastoista.

6.2 Jatkokehitys

Kun hallintasovelluksen viimeistely Iris 440 SHDSL-laitteelle on saatu valmiiksi, on edessä sen sovittaminen myös Iris 800 DSLAM-laitteelle. Iris 800 on keskuspään x86-pohjainen Linux-käyttöjärjestelmällä toimiva DSLAM-laite, joka on tehokkaampi ja liitännöiltään laajempi kuin asiakaspään Iris 440 –laite. Hallintasovelluksen kannalta laitteet ovat hyvin samankaltaiset, sillä myös Iris 800 –laitteen sovellusympäristö perustuu moduuleihin kohdassa 3.2 esitetyllä tavalla, ja suurin osa moduuleista toimii samalla tavalla kuin tämän projektin kohdeympäristössä. Suurin ero laitteiden välillä on liitäntöjen määrässä, sillä Iris 440 –laite sisältää neljä SHDSL-linjaa kytkettynä yhdeksi portiksi, kun taas Iris 800 –laite voi sisältää niitä maksimissaan 32 kappaletta. Lisäksi ethernet-liitäntöjä on nykyisessä kohdeympäristössä neljä ja Iris 800 –laitteessa vain yksi. Sovelluksen toiminta on lopulta myös testattava huolellisesti molemmissa ympäristöissä. Testaamisen perustana voidaan käyttää käyttöliittymän suunnittelussa hyödynnettyjä käyttötapauksia.

Hallintasovelluksen siirtäminen samankaltaiselle laitteelle pitäisi olla helppoa, koska sen kaikki toiminnot ovat muokattavissa määritysten avulla, jolloin esimerkiksi hallittavien SHDSL-linjojen määrän muuttaminen nykyisestä yhdestä 32:een vaatii ainoastaan linjojen lukumäärää hyödyntävien toimintojen uudelleenmäärittämisen. Mikäli hallintaobjektien muoto ja niiden käsittelyyn käytettävät rajapinnat pysyvät muuttumattomina, ei verkko-ohjelmiston koodiin tarvitse puuttua. Toimintoluokkien metodit toimivat hallintaobjektien tietosisällön ja määrän suhteen täysin annettujen parametrien, saatujen HTTP-pyyntöjen ja laitteelta saatujen tietojen perusteella.

Hallintasovelluksen suunnittelussa ja toteutuksessa pyrittiin alusta lähtien mahdollisimman alustariippumattomaan ratkaisuun, jolloin ohjelmiston siirtäminen laitteelta toiselle ei pitäisi olla ongelma, mikäli laitteen hallintarajapinta säilyy ennallaan. Vaikka lopullinen sovellus sisältää objektityyppikohtaisia toimintoja ja näkymiä, niin tarvittaessa hallintaobjekteja voidaan käsitellä myös geneerisillä oletustoiminnoilla ja näkymillä ilman erikoismäärittelyjä. Tämä tarkoittaa sitä, että ohjelmistoa ei ole sidottu tiettyihin objektityyppeihin tai niihin liittyviin rajoituksiin, kuten mahdollisten SHDSL- tai ethernet-liitäntöjen määrään. Teoriassa ohjelmisto voidaan siirtää uuteen tuoteprojektiin sellaisenaan ja käyttää pelkkiä oletustoimintoja, ja lisätä objektityyppien toiminnoille erikoismäärityksiä ja niihin liittyviä sivupohjia vasta myöhemmin. Tietoturvasyistä hallintaobjektien muokkaamiseen tarkoitetut toiminnot eivät tosin oletuksena ole käytössä, joten pelkkien oletustoimintojen avulla onnistuu vain objektien tarkastelu.

Lisäksi web-hallintasovellus saatetaan tulevaisuudessa siirtää myös täysin erilaiselle laitealustalle tai käyttöjärjestelmälle. Hallintasovelluksen käyttämä palvelinsovellus Seminole tukee siirrettävyyttä modulaarisuutensa ja erityisesti tarjoamansa siirrettävyyskerroksen avulla. Siirrettävyyskerros tarkoittaa muusta ohjelmasta eristettyä ohjelman osaa, joka sisältää kaiken sovellusalustakohtaisen koodin. Tämän koodin kääntämisessä hyödynnetään sovellusalustakohtaisia asetustiedostoja, joiden avulla ohjelma voidaan helposti muuntaa halutulle alustalle yhteensopivaksi. Seminole sisältää valmiin tuen Unixin lisäksi muun muassa MacOSX-, Windows- ja eCos-järjestelmille.

7 Yhteenveto

Tässä diplomityössä tutkittiin sulautetun järjestelmän web-hallintasovelluksen suunnittelun ja toteuttamisen eri vaiheita. Tuloksena syntynyt hallintasovellus sisälsi kolmannen osapuolen toteuttaman HTTP-palvelinsovelluksen ja itse toteutetun verkko-ohjelmiston. Työtä varten perehdyttiin web-ympäristöjen keskeisiin tekniikoihin palvelimen toiminnan ja tietoturvan sekä verkko-ohjelmiston toteuttamisen kannalta. Erityisen syvällistä tutustumista vaadittiin kohdeympäristönä toimivan Iris 440 –laitteen sovellusympäristön ja HTTP-palvelimena käytetyn Seminolen ohjelmointirajapintoihin.

Web-hallintasovellus tehtiin jo toteutetun CLI-hallintasovelluksen rinnalle tarjoamaan Iris 440 –laitteen ylläpitäjälle mahdollisuus laitteen asetusten tarkasteluun ja hallintaan web-selaimen avulla. Graafisella hallintasovelluksella voitaisiin tarjota käyttäjäystävällinen vaihtoehto monen vierastaman terminaalikäyttöliittymän rinnalle. Käyttäjäystävällisyydestä huolimatta hallintamahdollisuuksien monipuolisuudesta ei haluttu tinkiä, vaan lähtökohtaisesti selaimella piti pystyä suorittamaan vähintään kaikki se mitä terminaalillakin. Ainoaksi puutteeksi CLI-hallintasovellukseen verrattuna jäi valmiiden konfiguraatioasetusten syöttäminen tai vieminen ulos sovelluksesta. Se jätettiin tässä vaiheessa suunnittelun ulkopuolelle teknisten eroavaisuuksien johdosta.

Käyttöliittymän suunnittelu tehtiin pääasiassa CLI-hallintasovelluksen ominaisuuksia tarkastelemalla ja muuntamalla sen tarjoamia komentoja ja hallintamoodeja web-käyttöliittymälle sopivaksi. Täten esimerkiksi CLI:n *bridge configuration* –moodi voidaan rinnastaa web-sovelluksen *bridges*-välilehdeksi ja *interface configuration* –moodi *interfaces*-välilehdeksi. Käyttöliittymän näkymätyyppejä vastaavien HTML-sivupohjien toteuttaminen oli helppoa, koska sivupohjakomentojen käyttö oli samankaltaista kuin web-suunnittelussa yleisesti hyödynnettävillä skriptikielillä.

Työn selkeästi vaikein osuus oli verkko-ohjelmiston kontrolliosuuden toteuttaminen. Toteuttamisen alkuvaiheessa laadittiin erilaisia prototyyppejä, joiden pohjalta ohjelmistoa ryhdyttiin toteuttamaan helpoimmalta vaikuttavaan suuntaan. Lopputuloksena syntynyt sovellusmoottori vastasi lopulta kohtalaisen hyvin sille asetettuja geneerisyyden ja laajennettavuuden vaatimuksia. Erilaiset toiminnot ovat helposti määriteltävissä ja käytettävät sivupohjat ovat erillään sovelluslogiikasta, joten hallintaobjektien tai näkymien muutokset eivät vaadi muutoksia sovelluslogiikassa. Toimintomäärityksiä käsiteltiin sivulla 65 ja Seminole-palvelimen sivupohjia sivulla 49. Määrittelyissä ja näkymissä esiintyvä toisto on pyritty minimoimaan, jolloin tarvittavat muutokset voidaan toteuttaa mahdollisimman pienellä työmäärällä.

Vaikka myös valmiiden sovelluskehysten käyttöä harkittiin, niin toteutuksessa päädyttiin lopulta verkko-ohjelmiston toteuttamiseen tyhjästä, lukuunottamatta näkymien luomiseen käytettyä sivupohjamenetelmää. Verkko-ohjelmiston luokka-arkkitehtuuri ja luokkien sisäinen toiminta luotiin joitakin Seminolen tarjoamia luokkia lukuunottamatta itse. Käytetty ohjelmointikieli oli sulautetuille järjestelmille nopeuden kannalta oivallisesti soveltuva C++, joka ei välttämättä ole verkko-ohjelmiston toteutuskieleksi suoraviivaisin. Ongelmia esiintyi muun muassa sovelluksen käyttämien tietorakenteiden yhteydessä, joista monet kirjoitettiin projektin edetessä uudelleen. Ohjelmointityötä helpottivat muun muassa STL-kirjaston tarjoamat valmiit

tietorakenteet. Toteutuksessa olisi saatettu päätyä johonkin korkeamman tason kieleen, kuten Javaan ja sille toteutettuihin sovelluskehyksiin, mikäli Java-virtuaalikoneen käyttö kohdeympäristössä olisi ollut mahdollista.

Lähdeluettelo

[ASF05], The Apache Software Foundation (2005), Apache API Notes, http://httpd.apache.org/docs/1.3/misc/API.html. (Haettu 18.3.2007).

[AT01], Aprelium Technologies (2001), Abyss Web Server, http://www.aprelium.com/abyssws. (Haettu 19.3.2007).

[Ber99], Berners-Lee, T. et al. (1999), Hypertext Transfer Protocol – HTTP/1.1. IETF RFC2616, http://www.ietf.org/rfc/rfc2616.txt. (Haettu 7.11.2006).

[CN01], CollabNet, Inc. (2001), Subversion, http://subversion.tigris.org. (Haettu 19.3.2007).

[CoRo99], Coar, K., Robinson, D., (1999), The WWW Common Gateway Interface 1.1 Revision 3. http://cgi-spec.golux.com/draft-coar-cgi-v11-03.txt. (Haettu 17.2.2007).

[DC06], Design Combus Oy (2006), Iris 24 HW-arkkitehtuuri.

[DiAl99], Dierks, T., Allen, C., (1999), The TLS Protocol, version 1.0. IETF RFC2246, http://www.ietf.org/rfc/rfc2246.txt (Haettu 18.11.2006).

[Dru99], Druschel, Peter et al. (1999), Flash: An Efficient and Portable Web Server. USENIX Annual Technical Conference Monterey, California, USA, June 6–11, 1999. http://www.usenix.org/events/usenix99/full_papers/pai/pai.pdf. (Haettu 6.1.2007).

[Fra99], Franks, J. et al. (1999), Hypertext Transfer Protocol – HTTP Authentication: Basic and Digest Access Authentication. IETF RFC2617, http://www.ietf.org/rfc/rfc2617.txt. (Haettu 19.11.2006).

[Fre06], Freier, Alan et al. (1996), SSL 3.0 Specification. Netscape Internet Draft, http://wp.netscape.com/eng/ssl3/. (Haettu 18.11.2006).

[FSF91], Free Software Foundation, Inc. (1991), GNU General Public License. http://www.gnu.org/licenses/gpl.txt. (Haettu 19.2.2007).

[FSF97], Free Software Foundation, Inc. (1997), GNU Make. http://www.gnu.org/software/make. (Haettu 19.3.2007).

[GA00], GoAhead Software Inc. (2000), GoAhead WebServer 2.1 datasheet, http://data.goahead.com/webserver/WS-Datasheet5-00.cra.pdf. (Haettu 7.1.2007).

[Gru86], Grune, Dick (1986), Concurrent Versions System, http://www.nongnu.org/cvs. (Haettu 19.3.2007).

[GS06a], GladeSoft, Inc. (2006), Seminole Datasheet, http://www.gladesoft.com/products/seminole/semdatasheet.pdf. (Haettu 20.11.2006).

[GS06b], GladeSoft, Inc. (2006), Seminole Developer's Guide, http://www.gladesoft.com/products/seminole/semman.pdf. (Haettu 20.11.2006).

[HaJä03], Haikala, Ilkka, Järvinen, Hannu-Matti (2003), Käyttöjärjestelmät. Helsinki: Talentum. 242 s.

[HaMä98], Haikala, Ilkka, Märijärvi, Jukka (1998), Ohjelmistotuotanto, 6. painos. Helsinki: Suomen ATK-kustannus Oy. 389 s.

[HuTh00], Hunt, Andrew, Thomas, David (2000), The Pragmatic Programmer. Boston: Addison-Wesley. 321 s.

[Kal95], Kalimo, Anna (1995), Graafisen käyttöliittymän suunnittelu. Helsinki: Tieke RY. 229 s.

[Ker02a], Keren, Guy (2002), Unix Multi-Process Programming and Inter-Process Communications (IPC), http://users.actcom.co.il/~choo/lupg/tutorials/multi-process/multi-process.html. (Haettu 26.11.2006).

[Ker02b], Keren, Guy (2002), Network programming under Unix systems, http://users.actcom.co.il/~choo/lupg/tutorials/internetworking/internet-programming.html#sockets. (Haettu 26.11.2006).

[Ker98], Kerttula, Matti (1998), Tietoverkkojen tietoturva. Helsinki: Oy Edita Ab. 510 s.

[KL07], KoanLogic (2007), Koanlogic Klone tutorial, http://www.koanlogic.com/kl/cont/gb/html/klone-tute.html. (Haettu 14.1.2007).

[Kuu03], Kuutti, Wille (2003), Käytettävyys, suunnittelu ja arviointi. Helsinki: Talentum. 191 s.

[Maj05], Maj, Arthur (2005), Apache 2 with SSL/TLS: Step-by-Step, http://www.securityfocus.com/infocus/1818. (Haettu 18.2.2007).

[Mas05], Mason, Mike (2005), Pragmatic Version Control using Subversion. Dallas: The Pragmatic Programmers. 207 s.

[Mbe03], Mbedthis Software LLC (2003), AppWeb/Mbedthis, http://www.mbedthis.com. (Haettu 19.3.2007).

[MS95], Microsoft Corporation (1995), Internet Server API, http://msdn.microsoft.com. (Haettu 19.3.2007).

[Mäk04], Mäkitalo, Tommi (2004), Tntnet Web Server, http://www.tntnet.org. (Haettu 19.3.2007).

[OG01], The Open Group (2001), The Single Unix Specification, Version 3, http://www.opengroup.org/unix/online.html. (Haettu 19.3.2007).

[OM96], Open Market, Inc. (1996), FastCGI: A High-Performance Web Server Interface, http://www.fastcgi.com/devkit/doc/fastcgi-whitepaper/fastcgi.htm. (Haettu 18.3.2007).

[OvRa98], Ovaska, Saila, Räihä, Kari-Jouko (1998), Käytettävyystestaus. Sytyke RY, Systeemityö 4/98. s. 13-14.

[Pel98], Peltomäki, Juha (1998), WWW-ohjelmointi. Jyväskylä: Teknolit. 722 s.

[Per96], Perens, Bruce (1996), BusyBox, http://www.busybox.net. (Haettu 18.3.2007).

[Pus01], Puska, Matti (2001), Linux palvelimena. Helsinki: Satku. 304 s.

[RuCo01], Cobert, Jonathan, Rubini, Alessandro (2001), Linux Device Drivers: Second Edition. Sebastopol, CA: O'Reilly & Associates Inc., 564 s.

[Yag03], Yaghmour, Karim (2003), Building Embedded Linux Systems. Sebastopol, CA: O'Reilly & Associates, Inc., 391 s.

[ZT95], Zeus Technologies Ltd. (1995), Zeus Web Server, http://www.zeus.com/products/zws. (Haettu 19.3.2007).

Liitteet

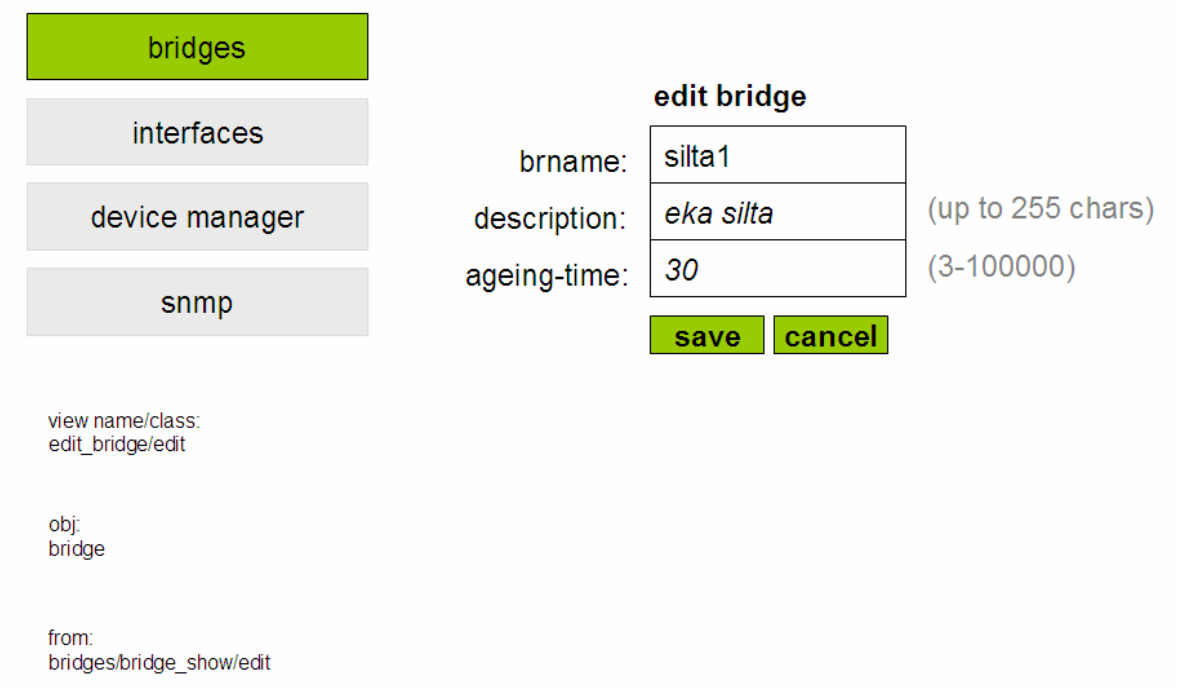
Liite 1. Subversion-komennot selityksineen (suluissa mahdolliset synonyymit).

add – tallenna kohde versionhallintaan.
blame (praise, annotate, ann) – tulosta tiedoston tai URL:n sisältö ja tekijän merkintä.
cat – tulosta tiedoston tai URL:n sisältö.
checkout (co) – kohteen haku paikalliseen hakemistoon.
cleanup – putsaa työkansio mm. lukituksista ja tallentamattomista muutoksista.
commit (ci) – työkansion muutosten tallennus versionhallintaan.
copy (cp) – tee kohteesta kopio haluttuun sijaintiin.
delete (del, remove, rm) – poista kohde.
diff (di) – näytä kahden kohteen välinen ero.
export – luo versioimaton kopio pääkopiosta.
help (?, h) – näytä ohje.
import – tallenna versioimattoman kopion muutokset versionhallintaan.
info – näytä kohteen tiedot.
list (ls) – tulosta kansion sisältö.
lock – lukitse haluttu kohde, eli estä muiden käyttäjien siihen kohdistuvat tallennukset.
log – näytä halutun kohteen muutosten selitykset.
merge – yhdistä halutut kohteet.
mkdir – luo uusi kansio.
move (mv, rename, ren) – siirrä kohde haluttuun sijaintiin.
propdel (pdel, pd) – poista kohteen ominaisuustiedot.
propedit (pedit, pe) – muokkaa kohteen ominaisuustietoja.
propget (pget, pg) – hae kohteen halutut ominaisuustiedot.
proplist (plist, pl) – hae kohteen kaikki ominaisuustiedot.
propset (pset, ps) – aseta kohteelle haluttu ominaisuustieto.
resolved – poista kohteen ristiriitamerkinnät.
revert – peru kohteelle tehdyt muutokset.
status (stat, st) – näytä työkopiossa olevan kohteen tila.
switch (sw) – tallenna työkopio uuteen säilytyspaikkaan.
unlock – avaa haluttu työkopio tai URL.
update (up) – päivitä versionhallinnan muutokset työkopioon.

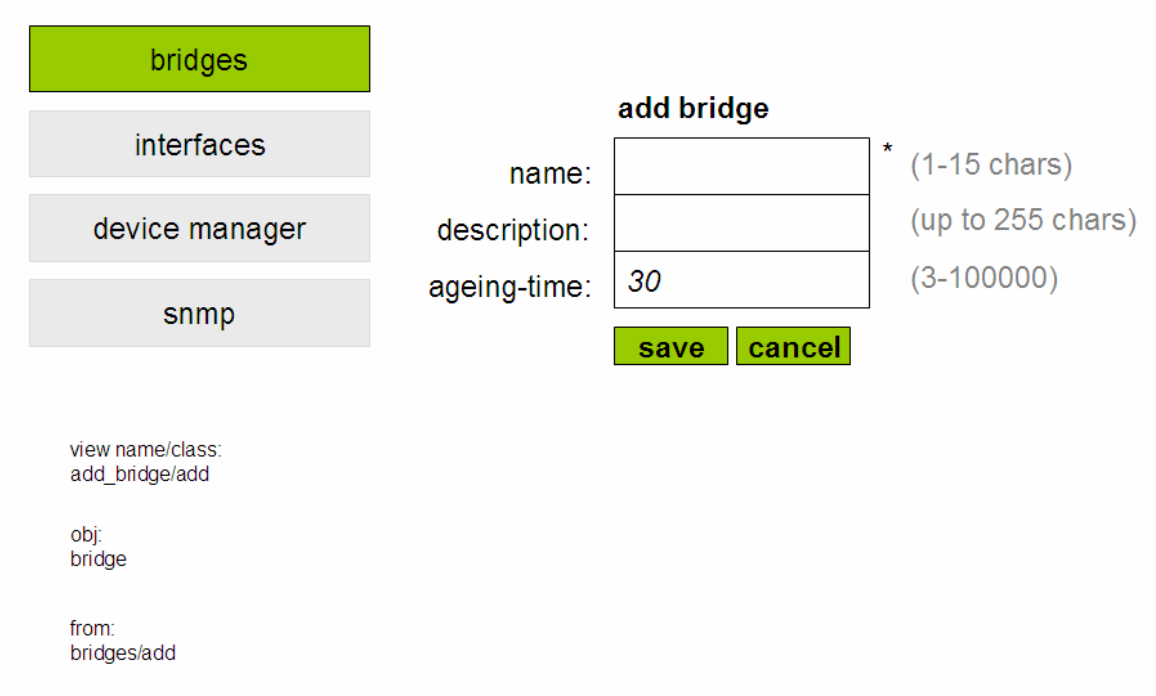
Liite 2. Kartoituksessa mukana olleet HTTP-palvelimet.

Ohjelma	Lisenssi	Prosessimalli	Koko (kt)	Tietoturva
Abyss	Freeware/maksullinen ($60)	multi-threaded	840	Access, Anti-hack protection
axTLS Embedded SSL	LGPL	multi-threaded	2030	SSL
Anti-web HTTPD	GPLv2	SPED	190	?
Apache 2.2	Apache License	multi-threaded	29670	Auth, SSL/TLS, .htaccess
AppWeb/Mbedthis	maksullinen (~$5/lisenssi 1000kpl)	multi-threaded	pieni	Auth, SSL
Barracuda	maksullinen	multi-threaded	pieni	Auth, HTTPS, SSL/TLS (SharkSSL), session
Bauk	"BSD" (VIL)	single-process tai multi-process	940	Auth, access restriction
Boa	GPL	SPED	490	fs permissions
Caudium	GPL	multi-threaded	18768	Auth, session
Cherokee	GPLv2	?	7430	TLS/SSL (GNUTLS, OpenSSL)
Clearsilver	Apache License	?	2610	?
CoreHTTP	AFL (Academic Free License)	SPED	190	?
Embedded HTTP server	LGPL	ei määritelty	1480	HTTPS
fnord	GPL	multi-threaded	130	?
Fusion	maksullinen	SPED/multi-threaded	7KB-11KB ROM	HTTPS server
Hydra Embedded Web Server	opensource	?	erittäin pieni	?
Hydra Web Server	GPL	multi-threaded	1200	TLS/SSL
KLone	GPLv2/maksullinen (~2e/lisenssi)	SPED	5370	TLS/SSL, session, encryption
Kritton	GPL	multi-process	740	tbd
libwebserver	LGPL	ei määritelty	1010	SSL
Lighttpd	BSD (muokattu)	SPED	3680	Auth, SSL
Litespeed	0 - $799 (2CPU)	?	14700	TLS/SSL
mathopd	"BSD"	single-process	253	?
micro_httpd	BSD	kernel inetd	20	HTTPS mahdollinen (stunnel)
mini_httpd	BSD	multi-process	170	Auth, SSL
muhttpd	MIT	?	90	SSL
Monkey HTTP-daemon	GPL	multi-threaded	330	Deny by URL & IP
Null httpd	GPL	multi-threaded	120	?
Seminole	$499-$1999	SPED	pieni	SSL
shttp	MIT	SPED	152	SSL
thttpd	BSD	SPED	550	permissions
Thy	GPL	?	1890	TLS/SSL
tntnet	GPL	multi-threaded	3350	SSL
xs-httpd	GPL	multi-process	1090	SSL
xweb	GPL	?	110	?
Zeus Web Server	maksullinen ($1700/2CPU)	SPED	suuri	SSL, anti-DoS, request filtering, htaccess

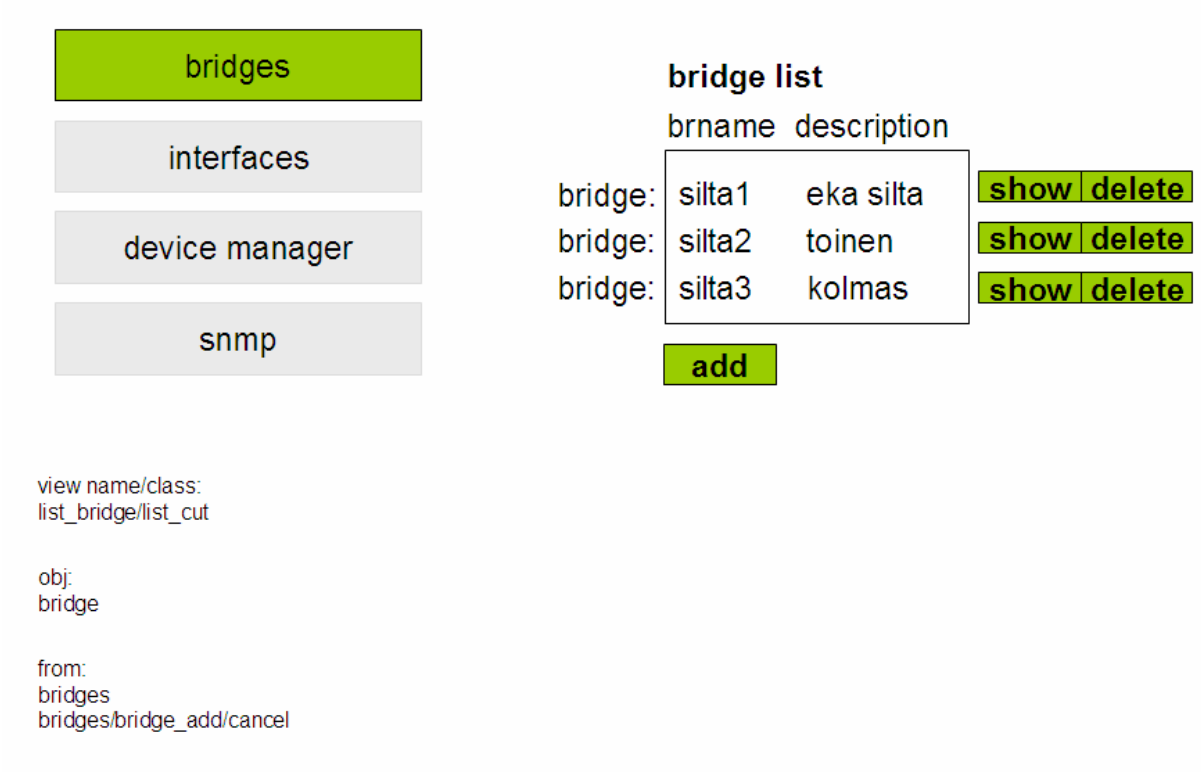
Liite 3. Käyttöliittymäsuunnitelman sillan editointinäkymä.



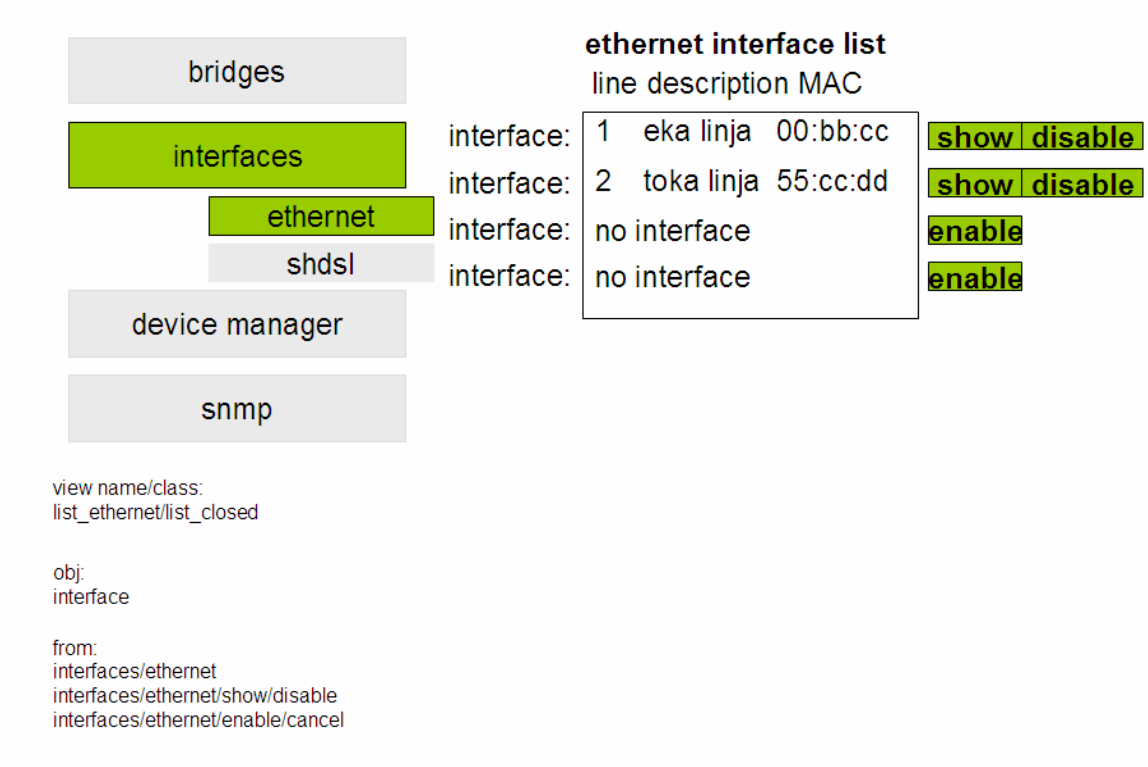
Liite 4. Käyttöliittymäsuunnitelman sillan lisäysnäkymä.



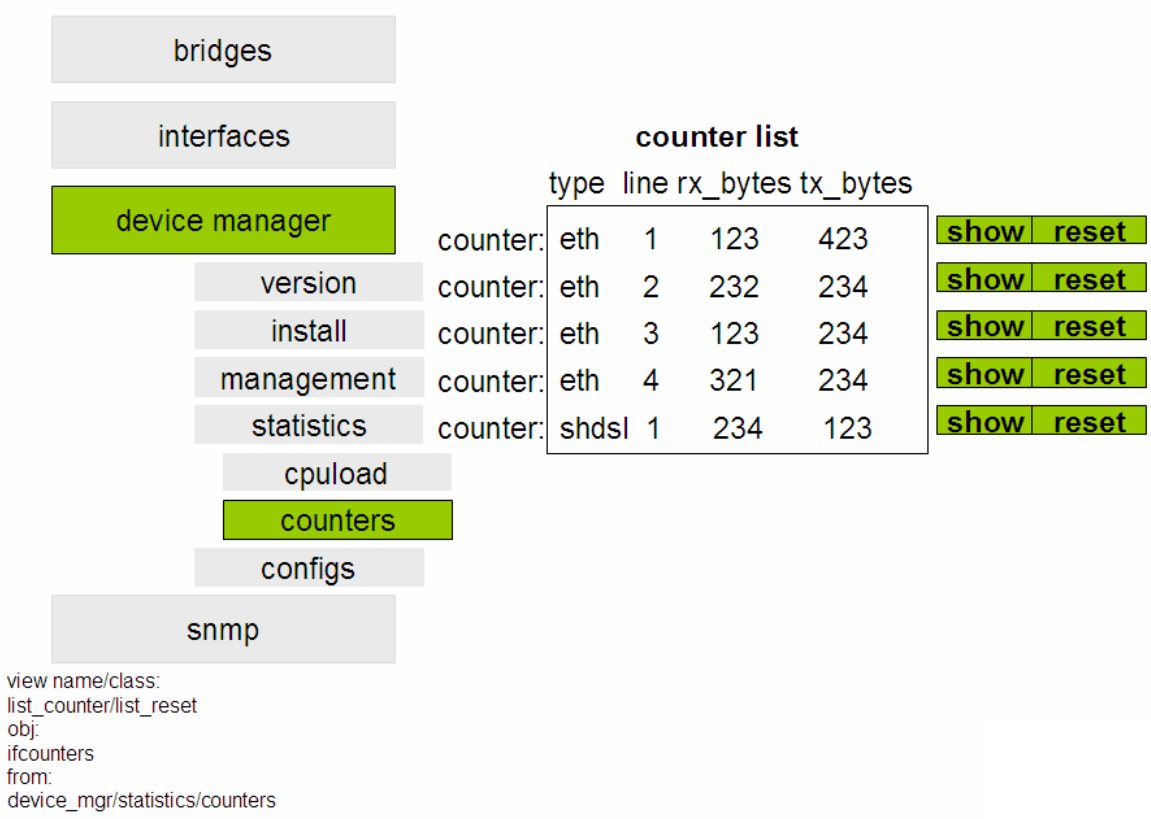
Liite 5. Käyttöliittymäsuunnitelman siltojen listausnäkymä.



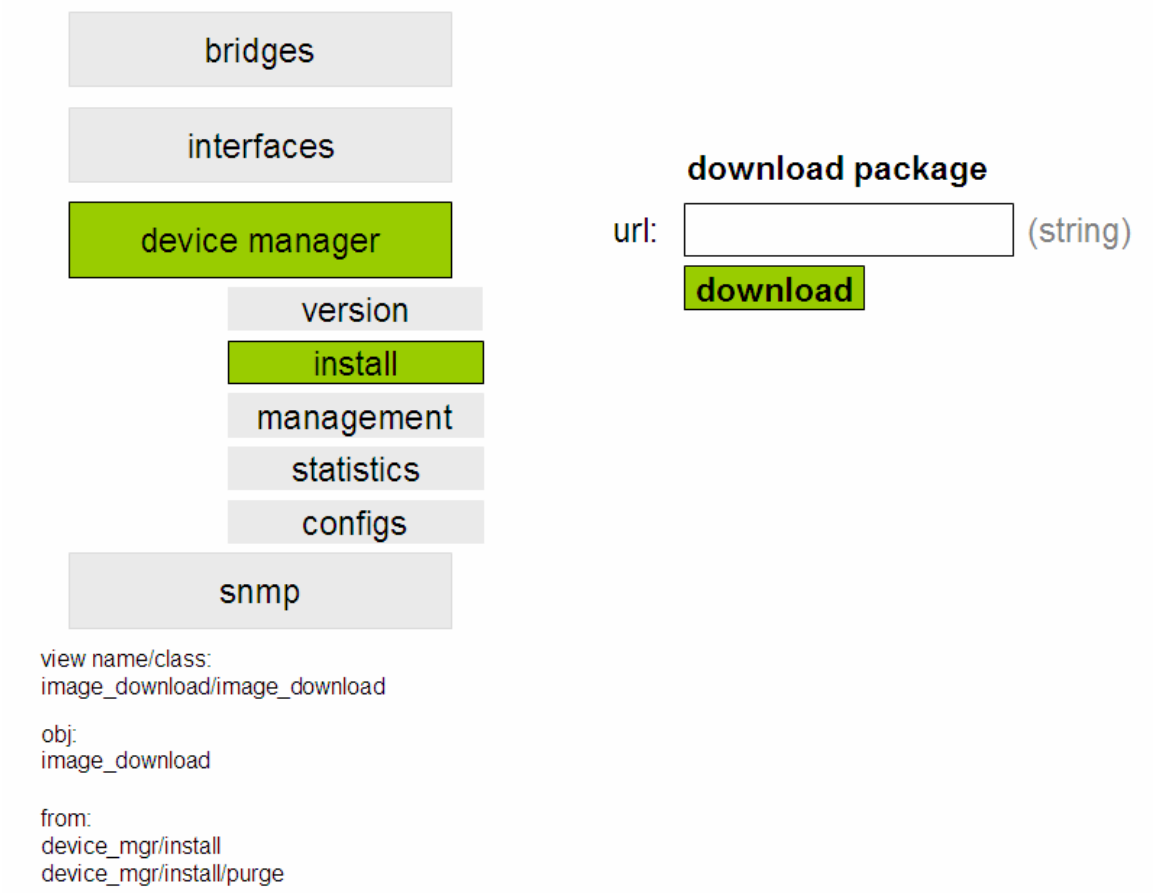
Liite 6. Käyttöliittymäsuunnitelman ethernet-liitäntöjen listausnäkymä.



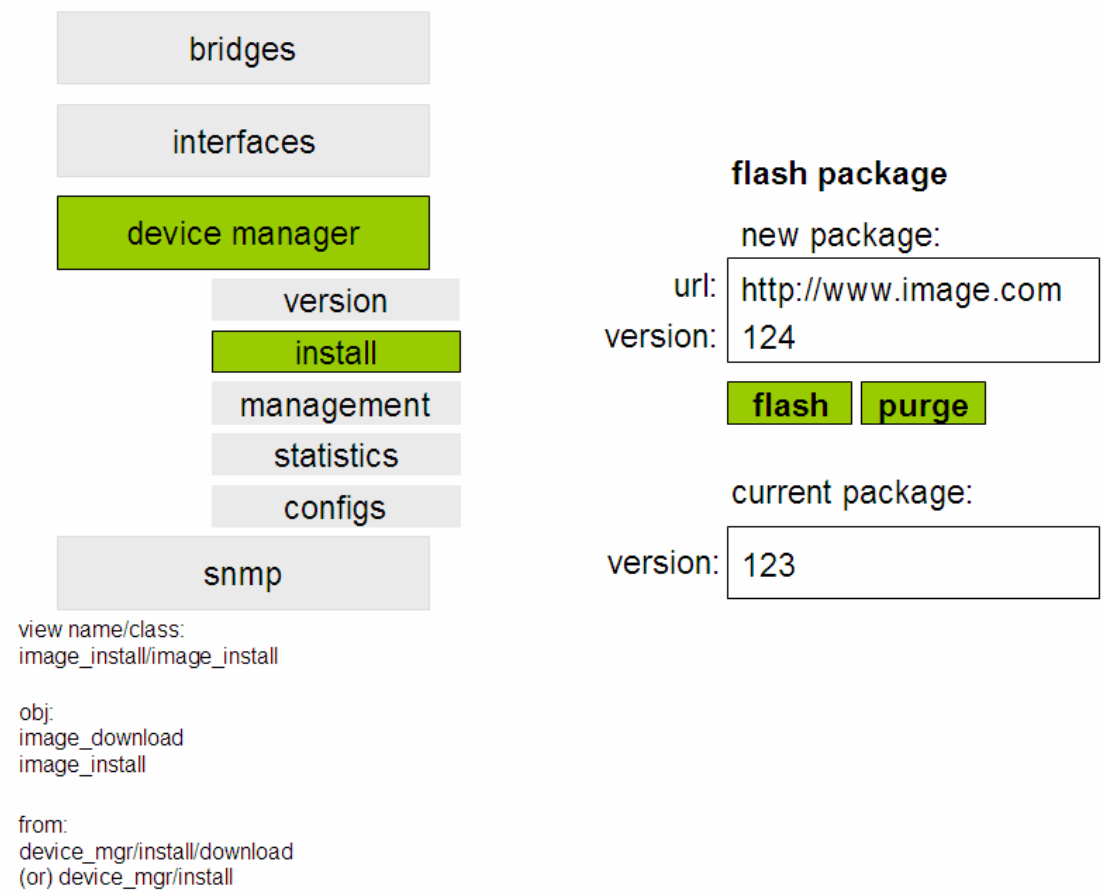
Liite 7. Käyttöliittymäsuunnitelman näkymä pakettilaskurien listauksesta.



Liite 8. Käyttöliittymäsuunnitelman levykuvan latausnäkymä.

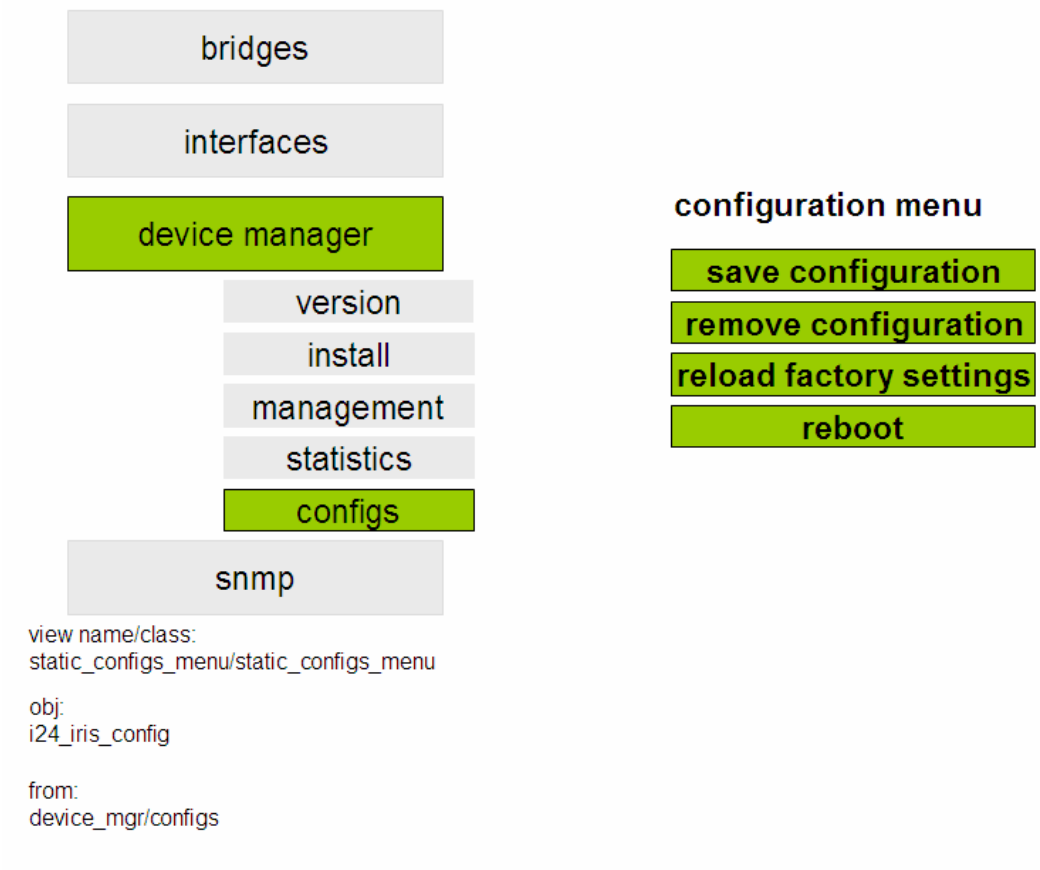


Liite 9. Käyttöliittymäsuunnitelman levykuvan asennusnäkymä.



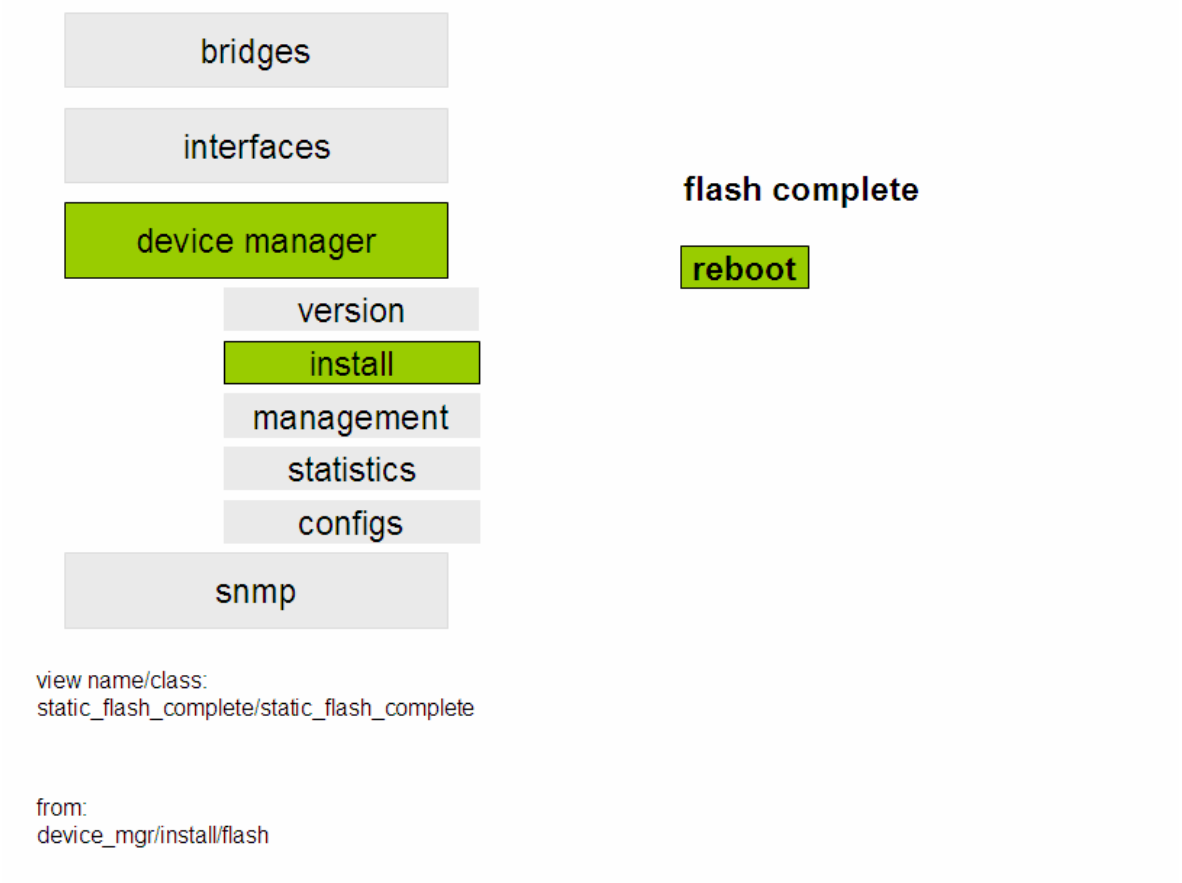
Liite 10. Käyttöliittymäsuunnitelman näkymä levykuvan keskeneräisestä asennuksesta.

Liite 11. Käyttöliittymäsuunnitelman laitteen konfiguraatiovalikko.

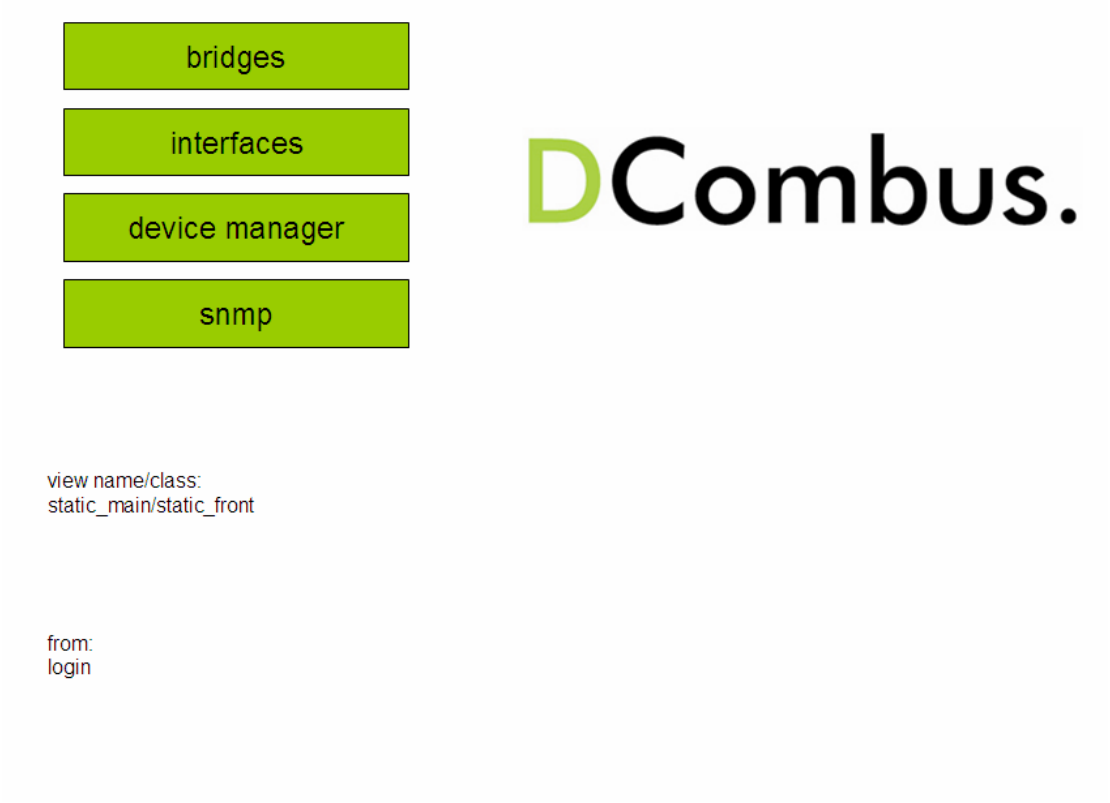


Liite 12. Käyttöliittymäsuunnitelman näkymä pyynnön vahvistamiskyselystä.

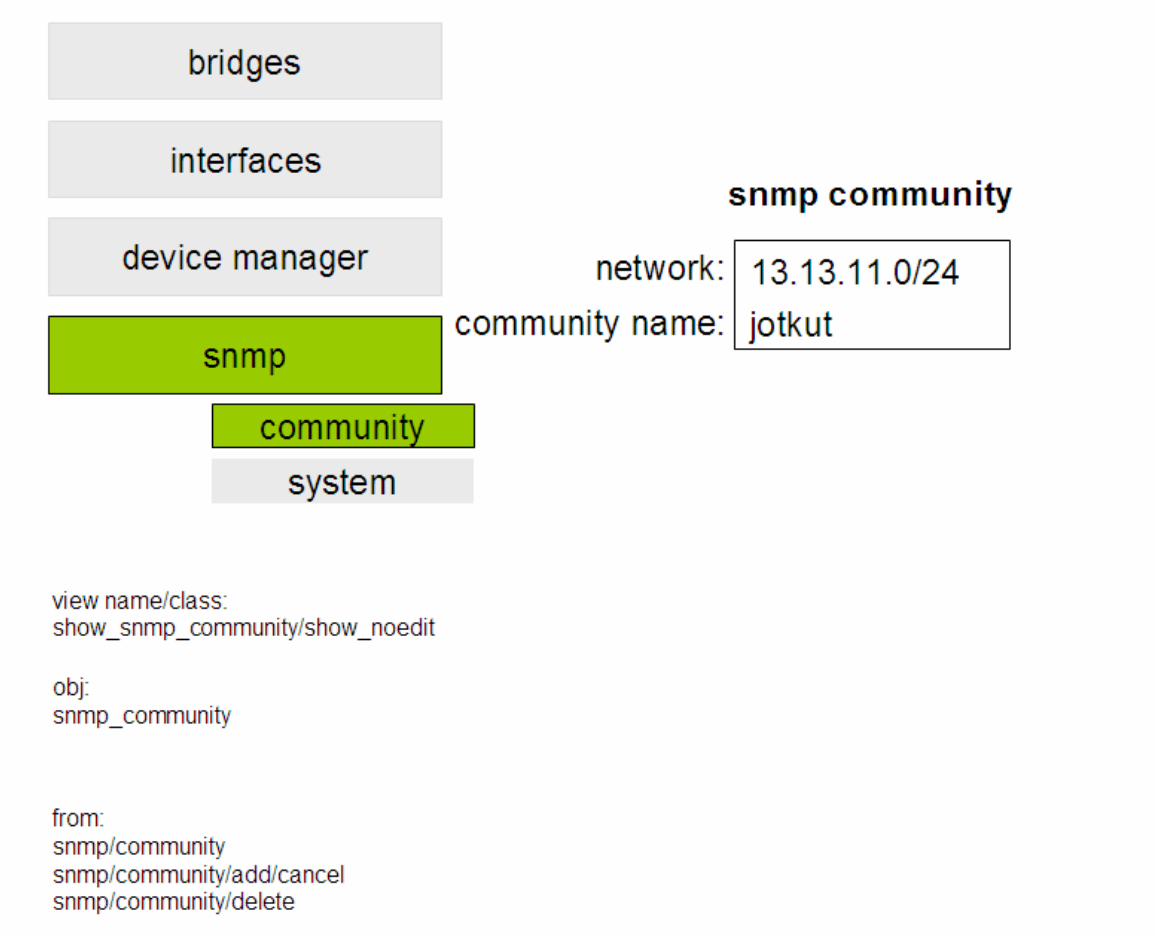
Liite 13. Käyttöliittymäsuunnitelman näkymä levykuvan asennuksen valmistumisesta.



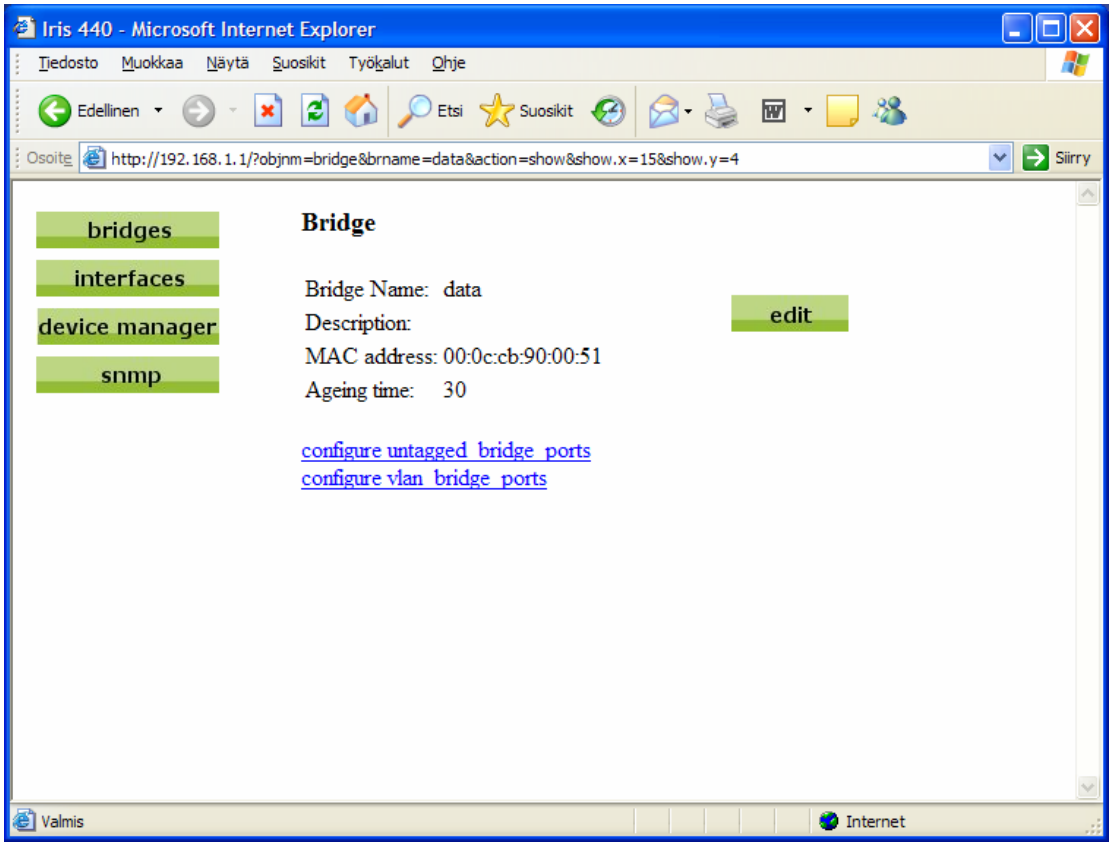
Liite 14. Käyttöliittymäsuunnitelman etusivunäkymä.



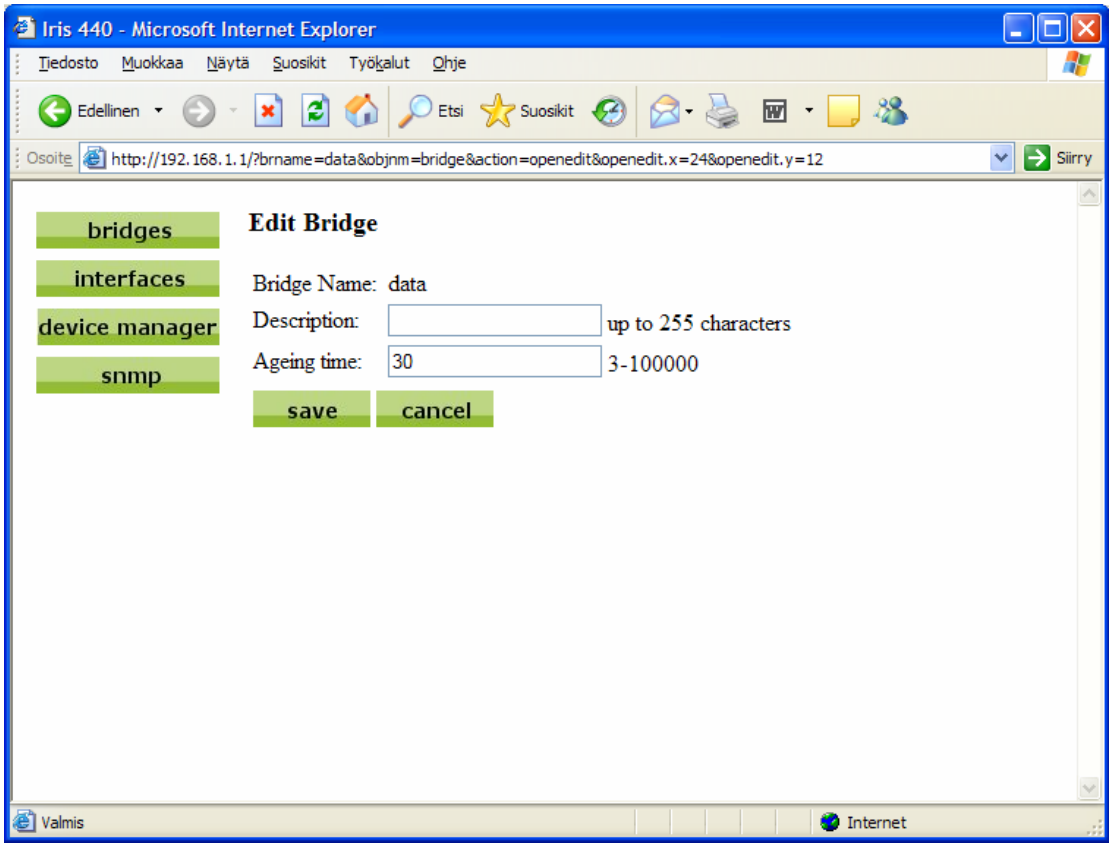
Liite 15. Käyttöliittymäsuunnitelman SNMP-yhteisön näyttäminen.



Liite 16. Sillan näyttäminen lopullisessa web-hallintasovelluksessa.



Liite 17. Sillan muokkaaminen lopullisessa web-hallintasovelluksessa.



Liite 18. Yhden hallintaobjektin listaamiseen kuluva aika.

Mittaus 1:

```
00:05:56,939 [1024] TRACE <> - timing: starting process
00:05:56,989 [1024] TRACE <> - class obj * obj::clone() const
00:05:56,989 [1024] TRACE <> - class dc_error_t obj::serialize() const
00:05:56,989 [1024] TRACE <> - static class dc_error_t
obj::deserializer()
00:05:56,999 [1024] TRACE <> - class dc_error_t
remote_home::get_first() const
00:05:57,019 [1024] TRACE <> - <--- send msg
00:05:57,869 [1024] DEBUG <> - >--- recv msg

00:05:58,159 [1024] TRACE <> - class obj * obj::clone() const
00:05:58,159 [1024] TRACE <> - class dc_error_t obj::serialize() const
00:05:58,189 [1024] TRACE <> - static class dc_error_t
obj::deserializer()
00:05:58,209 [1024] TRACE <> - class dc_error_t
remote_home::get_next() const
00:05:58,269 [1024] TRACE <> - <--- send msg
00:05:58,899 [1024] DEBUG <> - >--- recv msg

00:05:58,959 [1024] TRACE <> - class dc_error_t obj::serialize() const
00:05:58,969 [1024] TRACE <> - static class dc_error_t
obj::deserializer()
00:05:59,069 [1024] TRACE <> - timing: process done

IPC delay: 1,48s
total: 2,13s
```

Mittaus 2:

```
00:09:42,499 [1024] TRACE <> - timing: starting process
00:09:42,499 [1024] TRACE <> - class obj * obj::clone() const
00:09:42,499 [1024] TRACE <> - class dc_error_t obj::serialize() const
00:09:42,509 [1024] TRACE <> - static class dc_error_t
obj::deserializer()
00:09:42,509 [1024] TRACE <> - class dc_error_t
remote_home::get_first() const
00:09:42,529 [1024] TRACE <> - <--- send msg
00:09:43,379 [1024] DEBUG <> - >--- recv msg

00:09:43,659 [1024] TRACE <> - class obj * obj::clone() const
00:09:43,669 [1024] TRACE <> - class dc_error_t obj::serialize() const
00:09:43,699 [1024] TRACE <> - static class dc_error_t
obj::deserializer()
00:09:43,709 [1024] TRACE <> - class dc_error_t
remote_home::get_next() const
00:09:43,769 [1024] TRACE <> - <--- send msg
00:09:44,409 [1024] DEBUG <> - >--- recv msg

00:09:44,469 [1024] TRACE <> - class dc_error_t obj::serialize() const
00:09:44,469 [1024] TRACE <> - static class dc_error_t
obj::deserializer()
00:09:44,569 [1024] TRACE <> - timing: process done

IPC delay: 1,49s
total: 2,07s
```