Toni Haka-Risku

MVC-arkkitehtuurin toteutus web-sovelluskehyksissä

Tietotekniikan pro gradu -tutkielma

7. lokakuuta 2017

Jyväskylän yliopisto

Tietotekniikan laitos

Tekijä: Toni Haka-Risku

Yhteystiedot: Ag C416.1, toni.haka-risku@student.jyu.fi

Ohjaaja: Ohjaamaton työ

Työn nimi: MVC-arkkitehtuurin toteutus web-sovelluskehyksissä

Title in English: Implementation of MVC-architecture in web-frameworks

Työ: Pro gradu -tutkielma

Suuntautumisvaihtoehto: Kaikki suuntautumisvaihtoehdot

Sivumäärä: 85+1

Tiivistelmä: Tässä tutkielmassa esitellään MVC-arkkitehtuurin toteutusta Python-pohjaisissa web-sovelluskehyksissä. Työssä selvitetään millä tavalla MVC on toteutettu sovelluskehyksissä ja vastaako se alkuperäistä MVC:n toteutusta. MVC:n toteutus määritellään Krasnerin artikkelissa (Glenn E. Krasner & Stephen T.Pope Krasner 1988), joka pohjautuu Reenskaugin alkuperäiseen MVC:n määritelmään. Työssä tutkitut sovelluskehykset ovat Django, Pyramid ja Tornado. Django ja Pyramid eivät toteuttaneet MVC:tä. Tornadon ja web-sokettien avulla MVC on mahdollista toteuttaa.

Avainsanat: LaTeX, gradu3, pro gradu -tutkielmat, kandidaatintutkielmat, käyttöohje

Abstract: This document is a sample gradu3 thesis document class document. It also functions as a user manual and supplies guidelines for structuring a thesis document.

The English abstract of a thesis should usually say exactly the same things as the Finnish tiivistelmä.

Keywords: MVC, Web-sovelluskehys

Esipuhe

Tutkielman aiheen valintaan vaikutti aiempi kokemus ohjelmoinnista niin vapaa-ajalla kuin työelämässäkin. Suurimmassa osassa ohjelmistoprojekteja, joissa olen työskennellyt, on ollut käytössä MVC-pohjainen sovelluskehys työkaluna. Django-sovelluskehyksen sivuilla kuitenkin kyseenalaistetaan MVC:n toteutus sellaisenaan. Tästä sain idean kirjoittaa Pro Gradu tutkielman löytääkseni vastauksen kyseiseen ongelmaan. Jyväskylässä 7. lokakuuta 2017

Toni Haka-Risku

Termiluettelo

Sovellusarkkitehtuuri	kertoo millä tavalla sovellus on rakennettu
URL	määrittää tavan nimetä objekteja ja komponentteja sovelluksessa, jolloin ne voidaan identifioida ja paikantaa (Web Application Development 2007)
MVC	Model-View-Controller sovellusarkkitehtuuri
Sovelluskehys	on sovelluskehitykselle tarkoitettu pohja, joka kutsuu käyttäjän kirjoittamaa ohjelmakoodia ja tarjoaa ratkaisuja yleisimpiin toistuviin ongelmiin
WSGI	määrittää miten web-palvelin kommunikoi web-sovellusten kanssa ja millä tavalla web-sovellukset yhdistetään prosessoimaan pyyntöjä.
Template	Dynaaminen html-pohja, johon voidaan kirjoittaa ohjelmalogiikkaa
Piirikytkentä	Dynaaminen html-pohja, johon voidaan kirjoittaa ohjelmalogiikkaa
Pakettikytkentä	on tiedonsiirtomenetelmä, jossa yhteys data jaetaan paketeiksi. Osapuolien väliseen viestintään ei varata kaistaa, eikä osapuolien välille luoda katkeamatonta yhteyttä (Arlington 1981).
Piirikytkentä	on tiedonsiirtomenetelmä, jossa yhteys pysyy auki riippumatta siitä, onko osapuolien välillä liikennettä. Piirikytkentä on käytössä esim. puhelinverkoissa (Arlington 1981).
Asiakas-sovellus	alustaa yhteyden lähettääkseen pyyntöjä palvelimelle (Fielding 1999a).
HTTP	Kommunikaatio-protokolla Web:lle. Määrittelee kahdeksan perusoperaatiota: OPTIONS, GET, HEAD, POST, PUT, DELETE, TRACE ja CONNECT. Käytetyimmät operaatiot ovat GET ja POST. GET-metodi hakee datan, joka on liitetty pyynnössä annettuun URL:iin. POST lähettää dataa ohjelmalle, joka kuuntelee jotain määriteltyä URL:ia.

Käyttäjä-agentti	on tietokoneohjelma, joka lähettäää pyynnön jollekin tietylle ohjelmalle. Tällaisia ovat esimerkiksi selaimet (Fielding 1999a).
Palvelin	on tietokoneohjelma, joka ottaa vastaan yhteyksiä ja lähettää takaisin vastauksia asiakas-sovellukselle. Jokainen ohjelma pystyy olemaan yhtäaikaa asiakas-sovellus ja palvelin (Fielding 1999a).

Kuviot

Sisältö

1 Johdanto

1994-luvulla Internet tuli julkisesti kaikkien saataville ja tämän jälkeen siitä on tullut alusta lukuisille Web-sovelluksille. Web:n ensimmäiset toteutukset koostuivat suurista määristä dokumentteja, joita pystyttiin yhdistämään toisiinsa hyperlinkeillä. Nämä olivat usein yksinkertaisia tekstitiedostoja, joissa sisältö oli staattista. Yhdessä vuosikymmenessä Web on kuitenkin muuttunut staattisesta arkistosta kattavaksi alustaksi sovellusten toteuttamiseen ja julkaisuun. Uudet web-teknologiat, metodit ja kielet mahdollistavat dynaamisten sovellusten luonnin sekä malintamisen käyttäjien välille. Web-sovelluksissa tullaan kehittymään sitä mukaan, kun uusia teknologioita otetaan käyttöön web-selaimissa (Web Application Development 2007).

Web:n arkkitehtuurissa palvelin-sovellus tarjoaa asiakas-sovellukselle Web-sivuja, jotka voidaan dynaamisesti luoda palvelin puolella ohjelmakoodin avulla. Esimerkiksi selaimen pyytäessä web-sivua tulee pyyntö palvelimelle, jonka perusteella palvelin hakee tietokannasta tarvittavat tiedot ja muodostaa niistä sivun, joka palautetaan vastauksena takaisin selaimelle. Palvelimen puolella toteutettu sivujen prosessointi johti CGI-sovelluksiin (Common Gateway Interface) missä sovelluksen yksittäinen URL yhdistetään ohjelmakoodiin. Sovelluksen kasvaessa palvelimen puolella ohjelmakoodissa generoidut web-sivut osoittautuvat kuitenkin vaikeaksi hallita. Tämän ongelman ratkaisemiseksi kehittyi hybrid-malli, jossa ohjelmakoodin palasia kirjoitetaan tekstejen sekaan, jolloin halutut osat voidaan muuttaa dynaamiseksi. Pyynnön aikana dynaamiset ohjelmakoodin palaset suoritetaan ja yhdistetään dokumenttiin. PHP-ohjelmointikielellä oli tähän merkittävä vaikutus. Ratkaisu teki sivujen uudelleenkäyttämisen helpommaksi, mutta ajan kuluessa huomattiin koodimuutoksien olevan vaikeasti toteutettavissa, kun ohjelmakoodin palaset olivat käytössä kaikkialla sivuilla. Tämä puolestaan johti lopulta malliin, jossa sovelluksen tietty osa hallitsee pääsyä dataan ja toinen osa hajotetaan palasiin, joilla hallitaan sivujen luontia ja käyttäjien näkymiä (Web Application Development 2007, 3.1)

Tyypillinen rakenne Web-sovelluksissa koostuu seuraavista osista:

- HTML-sivujen näyttäminen käyttäjälle

- selaimen ohjelmakoodit, jotka ajetaan selaimen puolella
- palvelimen ohjelmakoodit, jotka ajetaan palvelimen puolella ja tyypillisesti keskustelevat tietokantojen kanssa

Selaimessa käytetty ohjelmointikieli perustuu ECMAScript standardiin, joka on olio-pohjainen ohjelmointikieli ja erityisesti tarkoitettu käyttöön selaimen puolelle. Tunnetuin ECMAScriptin toteutus on JavaScript. Palvelin-puolen ohjelmointikieliä on monia. Tällaisia ovat esimerkiksi PERL, PHP, Python ja Ruby. Palvelin-puolen ohjelmointikielien pohjalta on luotu lukuisia sovelluskehyksiä, joissa pohjana on MVC-arkkitehtuuri. Näistä tunnetuimpia ovat Ruby-kielellä toteutettu Ruby On Rails sekä Pythonilla toteutettu Django (Web Application Development 2007). Tutkimuksessa käydään Pythonilla toteutettu Pyramid- ja Django-sovelluskehyksen MVC-rakenne läpi ja heijastetaan toteutusta alkuperäisen MVC:n määritelmään. Lisäksi toteutetaan MVC-arkkitehtuuri käyttäen Tornado-sovelluskehystä, joka ei ota itsessään kantaa MVC-arkkitehtuuriin.

1.1 Tutkimuskysymys

Tutkimuksen tarkoituksena on selvittää ohjelmistokehityksen ja ohjelmistoarkkitehtuurejen taustoja. Näiden pohjalta tutkitaan millä tavalla Model-View-Controller -arkkitehtuuri (MVC) on toteutettu web-sovelluskehyksissä ja miten se eroaa alkuperäisestä arkkitehtuurin määritelmästä, jonka määrittelee Krasner (Glenn E. Krasner & Stephen T.Pope Krasner 1988) käyttäen pohjana Reenskaugin kehittelemää MVC-arkkitehtuuria (Reenskaug 2003) . Lisäksi pyritään toteuttamaan MVC alkuperäisen määritelmään mukaan Tornado-sovelluskehyksellä käyttäen TCP:n päällä toimivaa web-soketti protokollaa kommunikaatiossa HTTP:n sijaan. Havaintojen pohjalta selvitetään voidaanko Reenskaugin ja Krasnerin alkuperäinen MVC-malli toteuttaa web-sovelluskehyksellä.

1.2 Tutkimuksen rakenne ja aiheen rajaus

Tutkimuksessa käytetään laadullista tutkimusmenetelmää. Hypoteesi on kuitenkin vahvasti läsnä tutkimuksen aikana, koska tutkimus perustuu olettamukseen MVC:n toteutuksesta web-sovelluskehyksissä. Tutkimus ei todista mitään MVC:n aihealueen ulkopuolelta vaan

pyrkii esittämään yleisiä oletuksia MVC:n suhteesta web-sovellusten toteutuksiin sovelluskehykisen avulla. Tutkimuksen päämääränä on joko vahvistaa tai kumota hypoteesi MVC:n soveltuvuudesta web-sovelluskehyksillä toteutettuihin web-sovelluksiin.

Internettiä ja WWW:tä (World Wide Web) varten toteutetuissa web-sovelluksissa toistuvat usein samat toiminnallisuudet, kuten esimerkiksi staattisten resurssejen jakaminen, URL:ien liittäminen ohjelmakoodiin sekä tietoturvan hallinta (Consulting 2005). Tällöin on tärkeää, että Internetin ja WWW:n taustat käydään tutkimuksessa läpi, sillä niiden merkitys on suuri web-sovelluskehyksien olemassaololle. Lisäksi HTTP-protokollalla on suuri rooli MVC:n suhteen, koska se on alunperin tarkoitettu yksisuuntaiseen viestintään, vaikka MVC:n alkuperäisenä ajatuksena on kaksisuuntainen kommunikaatio esimerkiksi Mallin ja Näkymän välillä (Reenskaug 2003). HTTP:n lisäksi käydään läpi Internet Protokollan, TCP:n sekä Web-soketin toteutukset läpi. Näistä web-soketeilla pyritään ratkaisemaan kaksisuuntaisen kommunikaation ongelma MVC:ssä.

Tutkimus aloitetaan kirjallisuuskatsauksella, jossa tarkastellaan mitä aiempaa tutkimusta ohjelmistokehityksestä ja MVC:stä on tehty. Lisäksi käydään läpi mitä lähteitä löytyy Python-pohjaisista web-sovelluskehyksistä sekä ohjelmistoarkkitehtuureista. Web-sovelluskehykset rajataan Pyramid-, Django- ja Tornado-sovelluskehyksiin. Tämän jälkeen tutkitaan Internetin ja WWW:n taustoja sekä avataan ohjelmistoarkkitehtuurejen merkitystä tutkimukselle. Seuraavaksi käydään läpi MVC:n historiaa sekä millä tavalla MVC on tarkoitettu toteutettavaksi. Tässä vaiheessa esitellään jokaisen MVC-komponentin tarkoitus sekä niiden keskinäisen kommunikaation rakentuminen. Lisäksi esitellään Dortmundin yliopistossa kirjoitettu esimerkkiohjelma Smalltalkilla siitä miten MVC:n toteutus tuodaan sovellukseen käytännössä.

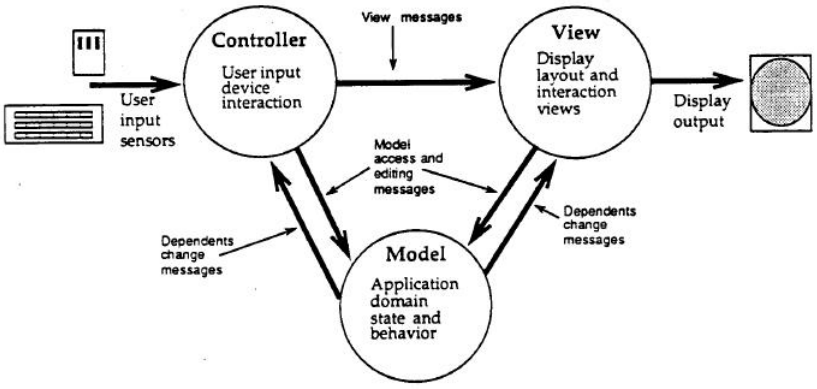
MVC:n tarkastelun jälkeen esitellään tutkimuksessa käytetyt web-sovelluskehykset, joita käytetään apuna MVC:n tutkimisessa. MVC:n toteutus käydään hyvin yksityiskohtaisesti läpi ja sen vertaaminen rajataan Python-pohjaisiin web-sovelluskehyksiin. MVC:stä on olemassa erilaisia versioita, joten sen määrittely tulee rajata tarkasti. Kun puhutaan MVC:stä tarkoitetaan tällä Krasnerin artikkelissa esiteltyjä määrittelyitä MVC:n toteutuksesta (Glenn E. Krasner & Stephen T.Pope Krasner 1988), jotka pohjautuvat Trygve Reenskaugin esittelemään MVC:n määritelmään (Parc 1979a).

Tarkasteltavista sovelluskehyksistä Pyramid ja Django ovat MVC-arkkitehtuuriin pohjautuvia sovelluskehyksiä. Tornado on web-sokettien ympärille toteutettu asynkroninen sovelluskehys, jota käytetään tutkimuksessa toteuttamaan MVC alkuperäisen toteutuksen mukaisesti. Sovelluskehyksistä käydään läpi sen historia sekä yleisellä tasolla mihin käyttötarkoitukseen sovelluskehys on tarkoitettu. Tämän jälkeen verrataan MVC:n toteutusta erikseen Pyramid- ja Django-sovelluskehykseen ja selvitetään millä tavalla niiden sovellusarkkitehtuuri mahdollisesti eroaa MVC:n alkuperäisestä toteutuksesta. Sovelluskehyksien muihin teknisiin ominaisuuksiin ei oteta kantaa. Havaintojen perusteella pohditaan MVC:n mahdollisia ongelmia sovelluskehyksien toteutuksessa ja selvitetään löytyykö sovelluskehyksien arkkitehtuurista jotain yhtenäisiä piirteitä, mitkä ovat kytköksissä MVC:n toteutukseen. Saatujen tulosten pohjalta pyritään kirjoittamaan Tornado-sovellus, joka toteuttaa MVC:n niinkuin se on alunperin tarkoitettu käyttäen web-soketteja. Tutkimuksen lopuksi koostetaan havainnoista yhteenveto, jossa pohditaan saatuja tuloksia ja selvitetään pystytäänkö niiden perusteella vastaamaan tutkimuskysymykseen.

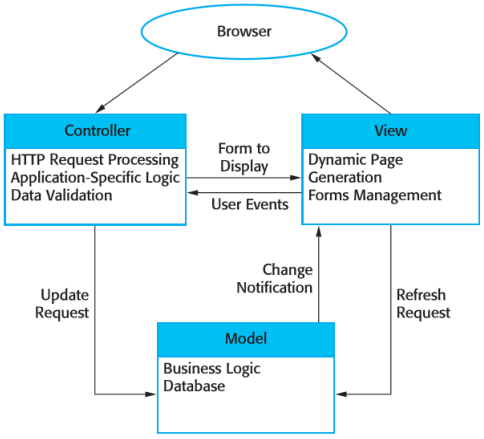
1.3 MVC:n ongelmat web-kehityksessä

Sovellusarkkitehtuurin tarkoituksena on vastata millä tavalla ohjelmiston eri komponentit on organisoitu, rakennettu ja millä tavalla ne kommunikoivat keskenään. Tämä on ensimmäinen vaihe, kun ohjelmistoa lähdetään toteuttamaan. MVC on yksi tapa toteuttaa sovelluksen arkkitehtuuri (Sommerville 2010, s. 148). MVC-arkkitehtuuri on saanut paljon huomiota web-sovelluskehyksien toteutuksissa ja useat web-sovelluskehykset ovat luokiteltu MVC-pohjaisiksi sovelluskehyksiksi (Bestframeworks 2009). Tätä tukee myös Sommervillen teos, jossa väitetään web-sovelluskehysten olevan suurimmaksi osaksi MVC-arkkitehtuurin toteuttavia (Sommerville 2010, s. 432). Se on kuitenkin alunperin tarkoitettu matalan tason käyttöliittymäsovellusten toteuttamiseen, jossa esimerkiksi hallitaan yksittäisiä näppäimistöltä tulleita syötteitä eikä sitä ole suoraan tarkoitettu käytettäväksi web-sovellusten ohjelmointiin. Alkuperäisen MVC:n toteutuksen soveltuvuutta web-ohjelmointiin onkin epäilty. Esimerkiksi Leff soveltaa artikkelissaan MVC:n käyttämistä web-sovelluksissa, mutta samalla esittelee alkuperäisen MVC:n toteuttamisen ongelmana. Tämä johtuu web-sovelluksen jakautumisesta asiakkaan (client) ja palvelimen (server) välille (Avraham 2001). Myös Pyramid-

sovelluskehyksen tekijät kyseenalaistavat MVC-arkkitehtuurin toteutuksen Pyramidissa ja uskovat MVC:n olevan sellaisenaan sopimaton web-ohjelmointiin, vaikka Pyramidin toteutus onkin hyvin lähellä alkuperäistä MVC:tä (Project 2010c). Django on vastaavasti toteutettu MVC:n pohjalta, mutta sen väitetään myös toteuttavan MVC hieman erilailla kuin MVC on alunperin tarkoitettu (Foundation 2016). Sommerville (Sommerville 2010, s. 157) esittelee MVC:n yhtenä arkkitehtuurimallina, jonka hän luokittelee kerrosarkkitehtuurien (Layered architecture) joukkoon. Sommervillen esittelemä MVC-malli kuitenkin poikkeaa Krasnerin (Glenn E. Krasner & Stephen T.Pope Krasner 1988) määrittelemästä MVC-mallista. Alla esitetään Krasnerin määrittelemä alkuperäinen MVC-malli sekä Sommervillen määrittelemä MVC-malli:



Kuvio 1. (G. E. Krasner 1988, s. 5)



Kuvio 2. (Sommerville 2010, s.162)

Krasnerin esittelemässä MVC:ssä Malliin (Model) on rekisteröity Ohjaimia (Controller) ja

Näkymiä (View). Näin Malli pystyy kommunikoimaan muutoksistaan suoraan Ohjaimelle sekä Näkymälle. Sommerville esittelee MVC-mallin web-sovelluksen näkökulmasta, jossa kommunikointi komponettejen suhteen toteutetaan erilailla. Sommervillen mallissa tieto käyttäjän toiminnoista voi tulla Näkymältä itseltään. Käyttäjän pyynnöt voitat tulla niin Ohjaimelle kuin Näkymällekin toisinkuin Krasnerin mallissa, jossa käyttäjän toiminnot tulevat ainoastaan Ohjaimelle. Lisäksi Sommervillen määritelmässä Mallilla ei ole kaksisuuntaista yhteyttä ohjaimeen.

2 Lähdemateriaalin hankinnasta

Kirjallisuuskatsauksessa käydään läpi vaihe vaiheelta, miten lähdemateriaalia kerätään tutkimusta varten. Lähdemateriaalin haku toteutetaan hakukoneilla, jotka ovat tarkoitettu erityisesti tieteellisten artikkeleiden etsimiseen. Tässä tutkielmassa käytetyt hakukoneet ovat seuraavat: IEEE Xplore, ACM Digital Library, Google Scholar sekä joissakin tapauksissa Google:n yleinen hakukone. Yleistä hakukonetta on käytetty esimerkiksi sovelluskehyksien dokumentaatioiden etsintään.

Aluksi muodostetaan kokonaiskuva tuloksista, jolloin silmäillään läpi saatuja artikkeleita. Tässä vaiheessa tarkoitus ei ole vielä valita mitään pohjaksi tutkielmalle, vaan kerätä informaatiota siitä millainen lähdemateriaali on tarjolla kokonaisuudessaan. Saaduista tuloksista poimitaan artikkeleita, jotka sopivat tutkimuksen aihepiiriin. Seuraavaksi artikkeleista valitaan tutkielmalle pohjakirjallisuus. Artikkelit luetaan huolellisesti läpi ja varmistutaan siitä, että ne ovat tieteellisesti päteviä tutkielmaa varten. Erityisesti kiinnitetään huomiota viittauksien määrän valittaessa tärkeimmät lähdemateriaalit. Tutkielmassa esiintyy myös satunnaisia viittauksia, joita ei ole kirjallisuuskatsauksessa mainittu. Tutkimuksen pääkirjallisuus kuitenkin käydään läpi kirjallisuuskatsauksessa. Haussa käytetään seuraavia hakutermejä: "software engineering", "software development", "MVC", "MVC Architecture", "frameworks", "web-frameworks"ja "websocket". Alla esitellyssä taulukossa näytetään hakujen tuottamien tulosten määrä, jossa hakukriteerinä käytetty metatietoja sekä otsikkoa.

Hakusana	ACM digital library	IEEE Xplore	Google Scholar
"software engineering"	3 639	4 971	55 500
"software development"	21 729	3,045	60 900
"MVC"	20	1 068	5 320
"MVC architecture"	49	308	122
"frameworks"	44 824	174 731	79 700
"web-frameworks"	7 848	50	136
"websocket"	37	934 000	9 340

2.1 Sovelluskehitys ja arkkitehtuurit

Sovelluskehityksen tarkasteluun löytyy runsaasti lähdemateriaalia. Merkittäväksi lähteeksi valitaan Ian Sommervillen kirjoittama *Software Engineering 9th Edition* -kirja, jossa käydään läpi mitä kaikkea isojen ohjelmistojen kehitykseen kuuluu (Sommerville 2010). Kirja on jaettu neljään pääosioon: *Introduction to Software Engineering*, *Dependability and Security*, *Advanced Software Engineering* ja *Software Management*. Tutkimuksessa lähteenä käytetään *Introduction to Software Engineering* ja *Advanced Software Engineering* -osioita. *Introduction to Software engineering* käy läpi ohjelmistojen suunnittelua ja toteutusta yleisesti. Tällaisia ovat esimerkiksi vaatimusmäärittelyt, prosessit, arkkitehtuuri ja testaus. *Advanced Software Engineering* esittelee ohjelmistojen kirjoittamista siten, että ne olisivat mahdollisimman uudelleen käytettäviä ja ylläpidettäviä. Tähän tarjotaan ratkaisuksi erilaisia sovellusarkkitehtuureja, kuten esimerkiksi *Component-based architecture* ja *Service-oriented architecture* (Sommerville 2010).

2.2 MVC

Google Scholarin tuloksista löytyy kolme artikkelia MVC:stä, jotka sopivat lähdemateriaaliksi tutkimukseen. Ensimmäinen artikkeleista on John Deaconin kirjoittama artikkeli, joka tarkastelee lyhyesti MVC:tä (Deacon 2009). Artikkeli on kuitenkin hyvin suppea, mutta selittää tiivistetysti MVC:n idean. Toinen artikkeli on Steve Burbeckin kirjoittama, joka käsittelee MVC:tä sellaisena kuin sitä käytettiin Smalltalkissa (Steve 1992). Burbeckin artikkeliin viitataan monissa MVC:tä käsittelevissä julkaisuissa, joten sen arvo tämän tutkielman pohjakirjallisuudessa on vahva. Viittausten määrä on katsottu hakemalla artikkelia Google Scholarin hakukoneessa. Seuraavaksi kartoitetaan pohjakirjallisuutta käyttäen ACM Digital Library sekä IEEE XPlore -hakukoneita. Kolmas artikkeli Glenn E. Krasnerin kirjoittama julkaisu, jossa esitellään MVC:n toteutusta erilaisissa Smalltalk-sovelluksissa. Julkaisusta löytyy useita versioita, joista tässä tutkielmassa käytetään molempia (Glenn E. Krasner & Stephen T.Pope Krasner 1988) (G. E. Krasner 1988). Tähän artikkeliin on myös viitattu runsaasti, joten se on Burbeckin julkaisun kanssa tärkeimpiä lähteitä MVC:n pohjakirjallisuudessa. Kirjoitushetkellä viittauksia Krasnerin artikkeliin on 2263. Monien MVC-arkkitehtuuria soveltavien artikkeleiden lähdeviitteistä löytyy viittauksia Burbeckin ja Krasnerin artikkeleihin. Tämän

perusteella pystytään toteamaan kyseisten artikkeleiden olevan tieteellisesti päteviä ja tarjoavan kattavan lähdemateriaalin MVC:n pohjaksi. Burbeckin ja Krasnerin kirjoittamien artikkeleiden taustalta löytyy MVC-arkkitehtuurin alkuperäinen kehittäjä Trygve Reenskaug, jonka omia julkaisuja sekä kotisivujen MVC-osiota käytetään lähteenä tutkielmassa (Parc 1978).

2.3 Web-sovelluskehykset & Internet

Internetin taustojen selvitykseen käytetään IETF:n (The Internet Engineering Task Force) dokumentaatioita (Force 1992). Dokumentaatiota käytetään TCP:n, Internet Protokollan, HTTP:n ja Web-sokettien taustojen selvittämiseen. Yksittäisistä Web-sovelluskehyksistä löydetty kirjallisuus on hyvin suppea, eikä tämän varaan voida rakentaa kovinkaan perusteellista tieteellistä pohjaa. Tämän vuoksi tutkimuksessa joudutaan turvautumaan sovelluskehyksien omaan dokumentatioon täydentämään lähdemateriaalia. IEEE Xploren ja ACM Digital Libraryn avulla löytyy kaksi julkaisua, joita käytetään tutkimuksen pohjana sovelluskehyksiä tarkastellessa. Ensimmäinen artikkeli on Okanovicin ja Mateljan kirjoittama artikkeli, jossa esitellään web-sovelluskehyksien suunnittelua (Ockanovic 2011). Se myös sivuuttaa lyhyesti MVC:tä. Toisena artikkelina käytetään ACM:stä tuloksena saatua Iwan Vosloon julkaisua, jossa käydään läpi yleisesti web-sovelluskehyksien rakennetta (Kourie 2008). Lisäksi käytetään IEEE:stä Ahamedin julkaisua, joka esittelee yleisesti asioita joita tulisi ottaa huomioon sovelluskehystä valittaessa (Sheikh I. Ahamed ja Pezewski 2008).

Web-sokettejen lähdemateriaalina käytetään IETF:n dokumentaatiota (Fette 2011) sekä Google Scholarin hakutuloksista löytynyttä artikkelia, jossa toteutetaan reaaliaikainen monitorointisovellus käyttäen web-soketteja (Puranik 2013). Tornado-sovelluskehyksestä itsestään löytynyt materiaali on hyvin suppeaa, jolloin sen esittelyssä käytettävät lähdemateriaalit liittyvät Tornadon käyttämiin teknologioihin. Näistä tärkeimpänä web-soketit sekä asynkroninen siirräntä (Brecht 2006). Pääpainona on kuitenkin web-soketit, jolloin asynkronisen siirrännän läpikäyminen on hyvin pintapuolista.

3 Sovelluskehitys

Katsomalla ympärille nykyajan yhteiskunnassa, huomataan monien asioiden toimivan ohjelmien varassa. Puhelimet, elektroniset kellot, autot ja erilaiset yhteiskunnalliset palvelut sisältävät kaikki ohjelman, jolla toimintoja sekä dataa hallitaan. Nykyajan modernia yhteiskuntaa ei pystytä ylläpitämään ilman sovelluksia. Monet tuotantolinjastot sekä rahoitusjärjestelmät teollisuudessa ovat täysin automatisoituja. Viihdeteollisuus kuten musiikki, pelit, elokuvat ja televisio ovat kaikki monimutkaisten sovellusten varassa. Ihmisten tiedot ovat monissa valtion hallintajärjestelmissä sekä niin julkisissa kuin yksityisissäkin potilastietojärjestelmissä. Nykypäivänä katsottaessa ympärille löydämme lähes poikkeuksetta jonkin asian, jonka taustalla on ohjelma. Suuri määrä ihmisiä kirjoittaa sovelluksia nykypäivänä. Liike-elämässä ihmiset kirjoittavat taulukkolaskentaohjelmia helpottaakseen työtään. Tutkijat ja insinöörit kirjoittavat ohjelmia prosessoimaan kerättyä dataa ja harrastelijat kirjoittavat ohjelmia viihdyttääkseen itseään. Suurin osa sovelluskehityksestä on kuitenkin toteutettu ammattilaisten toimesta tukemaan liiketoimintamalleja, kehittämään laitteita sekä rakentamaan erilaisia palveluita. Ammattilaisten toteuttamat ohjelmistot toteutetaan usein ryhmissä ja ne ovat suunnattu käytettäväksi muille kuin ohjelmoijille itselleen (Sommerville 2010, s.4).

Sovelluskehityksellä tarkoitetaan koko ohjelmiston elinkaaren mittaista tuotantoprosessia alkumäärittelyistä ylläpitoon asti. Se on ammattihaara, joka on erikoistunut tutkimaan ja tuottamaan sovelluksia. Sovelluskehittäjät käyttävät erilaisia teorioita, käytänteitä ja työkaluja ratkaisemaan sovelluskehitykseen liittyviä ongelmia. Sovelluskehitys ei kuitenkaan keskity ainoastaan teknisiin ongelmiin vaan myös projektin hallintaan ja työkaluihin, jotka tukevat sovellusten kehittämistä (Sommerville 2010, s.1-10). Sovelluskehityksessä tarkoituksena on saada aikaan tuloksia

3.1 Sovellukset

Ohjelmistot ovat abstrakteja ja aineettomia. Ne eivät noudata mitään fysiikan lakeja eivätkä ne koostu mistään materiasta tai tuotantoprosessista. Tämä tekee ohjelmistokehityksestä helppoa ja vapaata, koska sille ei ole asetettu mitään luonnollisia esteitä. Rajojen puuttumi-

nen johtaa myös siihen, että ne voivat olla usein erittäin monimutkaisia, vaikeita ymmärtää ja kallista muuttaa. Ohjelmistojen tyypit vaihtelevat sulautetuista järjestelmistä maailmanlaajuisiin tietojärjestelmiin. Tietojärjestelmän kehittäminen organisaatiolle eroaa täysin esimerkiksi ohjainlaitteen ohjelmoinnista. Näitä kahta taas ei voi verrata esimerkiksi grafiikkapohjaiseen pelikehitykseen. Kuitenkin kaikki nämä tarvitsevat ohjelmistokehitystä (Sommerville 2010, s.1-10). Ohjelmistot eivät ole pelkästään tietokoneohjelmia vaan ne koostuvat ohjelman lisäksi dokumentaatioista sekä erilaisista konfiguraatioista, joilla määritellään ohjelma toimimaan tietyllä tavalla. Dokumentaatio sisältää usein tiedon ohjelmiston rakanteesta sekä käyttäjille suunnatun ohjeen miten sovellusta hallitaan ja käytetään (Sommerville 2010, s.1-10).

Sommenville jakaa ammatilaisten kehittämät ohjelmistotuotteet kahteen kategoriaan: geneeriset tuotteet ja räätälöidyt tuotteet. Geneeristen tuotteiden määritelmät ovat sitä kehittävän organisaation hallinnassa. Päätäntävalta ominaisuuksista on siis kehittävällä osapuolella. Räätälöidyissä tuotteissa tilaaja määrittelee ohjelmiston tarkoituksen ja sen ominaisuudet. Näitä kahta tyyppiä voidaan kuitenkin yhdistää, mikä on varsin yleistä ohjelmistoille. Pohjana voi olla geneerinen tuote, johon toteutetaan erilaisia tasoja, jotka ovat määritely tilaajan toimesta (Sommerville 2010, s.1-10).

3.2 Web-sovellukset

World Wide Web:n kehityksellä on ollut tärkeä rooli sovelluksien kehityksessä. Alunperin Web oli yliopistojen käytössä datan tallennusta varten ja sillä oli hyvin vähän vaikutusta sovelluksiin. Webin ympärille kirjoitetut sovellukset olivat käynnissä paikallisissa tietokoneissa ja niihin oli vain pääsy organisaation sisällä. 2000-luvun alussa Web alkoi kehittyä laajemmin. Selaimiin lisätyt ominaisuudet mahdollistivat Web-pohjaisten sovellusten käytön siten, että erillisen sovellukselle tarkoitetun käyttöliittymän sijasta voitiin käyttää selainta. Tämä johti monien erilaisten sovelluksien kehitykseen, jotka tarjosivat innovatiivisia palveluita käytettäväksi selaimen kautta. Webin kehityksen myötä selaimet kehittyivät siten, että ne pystyivät ajamaan myös pieniä sovelluksia paikallisesti. Tämä muutti organisaatioiden sekä liiketoiminnan tapoja käyttää sovelluksia. Sen sijaan, että sovellukset olisi asennettu käyttäjien tietokoneille, ne voitiin asentaa suoraan yhdelle web-palvelimelle. Tämä laski

myös kustannuksia tehdä muutoksia sovelluksiin. Suurin osa on liiketoimien harjottajista on siirtynyt käyttämään web-pohjaisia sovelluksia siellä missä se on mahdollista (Sommerville 2010, s.8).

Seuraava vaihe web-sovellusten kehityksessä olivat web-palvelut. Web-palvelut ovat sovelluskomponentteja, jotka tarjoavat hyödyllisiä palveluja käytettäväksi muille sovelluksille. Siinä missä web-sovellus tarjoaa käyttöliittymän sovellukselle, web-palvelu tarjoaa rajapinnan käyttöliittymäsovellukselle käytettäväksi. Web-palvelu on yksittäinen osa sovellusta ja sovelluksella voi olla monia web-palveluja käytettävänä.

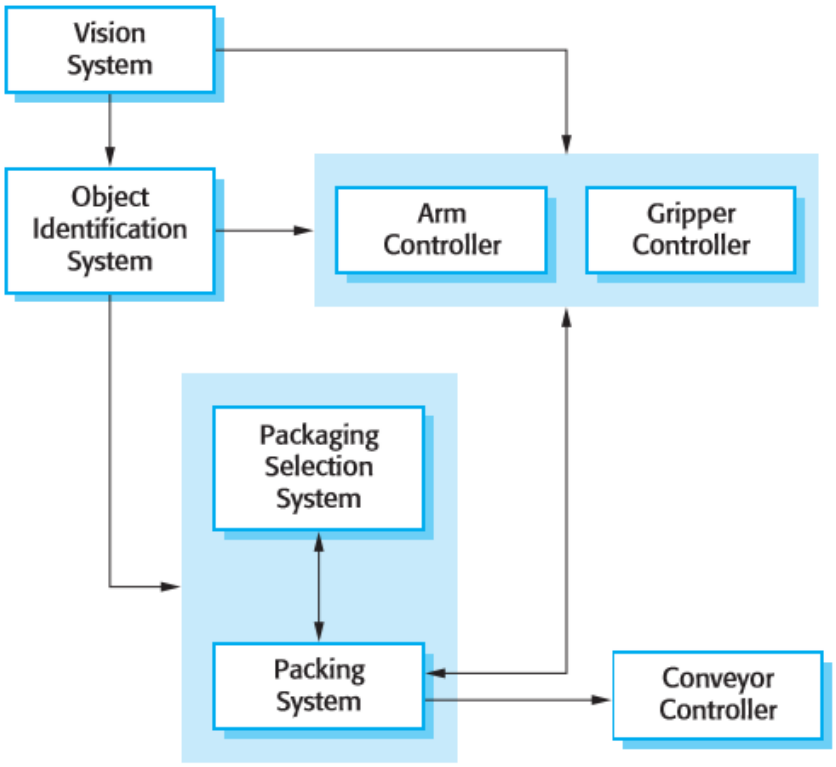
4 Sovelluksien arkkitehtuuri

Arkkitehtuuri määrittää sovelluksen rakenteen, organisoinnin sekä komponenttejen kommunikaation. Suunnittelun aikana arkkitehtuurilla pyritään kuvaamaan se millaisiin osiin sovellus on jaoteltu. Sovelluksen kehityksen alussa on tärkeää, että sovelluksen yleistä arkkitehtuuria noudatetaan. Yksittäisten komponenttejen refaktorointi vaatimusten muuttuessa on usein helppoa, mutta sovelluksen kokonaisarkkitehtuurin muuttaminen on usein kallista. Käytännössä vaatimusten määrittely ja arkkitehtuurin suunnittelu koskettavat toisiaan. Ideaalitilanteessa sovelluksen vaatimukset eivät sisällä mitään arkkitehtuuriin liittyviä suunnitelmia tai vaatimuksia, mutta todellisuudessa tältä ei voida välttyä. Arkkitehtuurin määrittely on usein tarpeen vaatimusten rakentamisessa ja organisoinnissa. Tästä syystä arkkitehtuurin suunnittelu vaatimusmäärittelyjen ohella on usein mahdollista. Tällöin arkkitehtuurista käytetään abstraktia versiota, jossa sovellus on pilkottu erilaisiin osiin vastaamaan vaatimusten määrittelyn ongelmiin. Sommerville jakaa arkkitehtuurin kahteen tasoon: Sovellusarkkitehtuuriin (Architecture in small) ja Järjestelmäarkkitehtuuriin (Architecture in large). Sovellusarkkitehtuurissa keskitytään yksittäisten sovellusten sisäiseen komponenttijakoon sekä rakenteeseen. Järjestelmäarkkitehtuuri keskittyy suurien järjestelmien rakenteeseen. Tällaiset järjestelmät koostuvat useista sovelluksista, muista järjestelmistä ja erilaisista komponenteista. Suuret järjestelmät ovat usein jaettu monille tietokoneille, jotka voivat olla myös useiden eri yritysten omistuksessa (Sommerville 2010, s. 148).

Arkkitehtuurin mallinnusta voidaan käyttää kahdella hyvin erilaisella tavalla. Sitä voidaan käyttää auttamaan kommunikoinnissa osakkaiden sekä suunnittelijoiden kanssa, jolloin järjestelmän yksityiskohdat piilotetaan abstraktejen komponenttejen taakse arkkitehtuurikaaviossa. Yksinkertaisen kaavion avulla pystytään keskustelemaan useiden ihmisten kanssa siten, että yksityiskohdat eivät häiritse kokonaisuuden suunnittelua. Tällöin myös projektiin osallistuvien ihmisten ei tarvitse olla teknologioista tietoisia. Kaavion avulla myös projektin johto näkee avainkomponentit ja pystyy tämän avulla jakamaan tehtäviä kehittäjille. Toinen tapa on käyttää arkkitehtuurin mallinnusta dokumentaationa arkkitehtuurille, joka on jo etukäteen suunniteltu. Tällaisella on tarkoitus kuvata koko järjestelmä yksityiskohtaisemmin. Tällöin esitellään komponentit, niiden yhteydet toisiinsa sekä rajapinnat. Monissa projek-

teissa arkkitehtuurin kuvaus jää usein hyvin abstraktille tasolle, koska aina ei katsota yksityiskohtaisen mallinnuksen antavan tarpeeksi lisäarvoa (Sommerville 2010, s. 150).

Arkkitehtuurit kuvataan usein kaavioilla, joissa komponentit kuvataan lyhyesti nimetyillä lohkoilla. Jokainen lohko esittää yksittäistä komponenttia. Lohkojen sisällä olevat lohkot kuvaavat komponentin hajautusta useampaan alikomponenttiin. Nuolet komponenttejen välillä kuvaavat miten dataa ja signaaleja siirretään komponenttejen välillä. Kaavioiden avulla järjestelmän toteuttamiseen osallistuvat henkilöt ymmärtävät järjestelmän rakenteen korkealla tasolla (Sommerville 2010, s. 150). Sommervillen kirjassa esitellään esimerkkikaavio sovelluksen arkkitehtuurista. Kaaviossa esitellään arkkitehtuurimalli siitä miten pakkausrobottisovellus tulisi toteuttaa.



Kuvio 3. Pakkausrobotin arkktehtuurimalli (Sommerville 2010, s.149)

Kaaviossa esitellään robottisovelluksen jako erilaisiin komponmentteihin. Robotti pystyy pakkaamaan kaikenlaisia tavaroita liukuhihnalta. Se käyttää Näkökomponenttia (Vision System) valitakseen erilaisia tavaroita ja tunnistamaan Identifiointi-komponentilla (Object Identification System) tavaroiden tyypin. Kun tavara on tunnistettu se annetaan Pakkaus-komponenteille

prosessoitavaksi (Packing Selection System ja Packing System). Kun edellä mainituilla komponenteilla on hoidettu tavaran tunnistus sekä pakkaustyypin valinta, käsitellään tavara antamalla vastuu käsivarrelle (Arm Controller) ja kouralle (Gripper Controller), jotka kontrolloivat robotin fysiikka. Lisäksi robotilla tulee olla myös liukuhihnan hallintakomponentti (Conveyor Controller), jotta pystytään myös mahdollisesti pysäyttämään, hidastamaan ja nopeuttamaan linjaston kulkua (Sommerville 2010, s.148)

4.1 Arkkitehtuurin suunnittelu

Arkkitehtuurin suunnittelu on luova prosessi, jossa toteutetaan halutun järjestelmän organisointi siten, että se vastaa niin järjestelmän toiminallisiin kuin ei-toiminnallisiin vaatimuksiin. Toiminallisilla vaatimuksilla tarkoitetaan vaatimuksia, jotka määrävät mihin tarkoitukseen järjestelmä tulee ja millaisia ominaisuuksia se tarjoaa. Ei-toiminnallisilla vaatimuksilla tarkotetaan järjestelmän sisäisiä vaatimuksia, jotka ovat usein näkyvissä vain järjestelmän kehittäjille. Ei-toiminnallisilla vaatimuksilla vastataan siihen miten järjestelmä tulee rakentaa. Toiminnallisilla vaatimuksilla vastataan siihen, mitä järjestelmän tulisi tehdä. Arkkitehtuurin suunnittelu on usein hyvin paljon sidoksissa siihen, millainen järjestelmä on kyseessä. Sommerville määrittelee arkkitehtuurin suunnittelun olevan enemmänkin sarja erilaisia valintoja kuin, että se olisi ennalta määrättyjen aktiviteettien noudattamista. Sommerville esittelee seuraavat yhdeksän yleistä kysymystä, jotka järjestelmäarkkitehtien on kysyttävä suunnitteluprosessin aíkana (Sommerville 2010, s. 151):

1. Onko olemassa sovelluarkkitehtuuria, joka voi toimia valmiina pohjana järjestelmälle?
2. Millä tavalla järjestelmä hajautetaan prosessorejen tai ytimien kesken?
3. Mitä erilaisia arkkitehtuurimalleja käytetään?
4. Minkä fundamentaalin ajatuksen pohjalle järjestelmä rakannetaan?
5. Millä tavalla järjestelmän pääkomponentit hajautetaan alikompoenteiksi?
6. Millaista strategiaa käytetään hallitsemaan komponentteja järjestelmässä?
7. Millainen arkkitehtuurillinen organistointi on paras ei-toiminnallisille vaatimuksille?
8. Millä tavalla arkkitehtuurin suunnittelu evaluoidaan?
9. Miten järjestelmän arkkitehtuuri dokumentoidaan?

Vaikka jokainen järjestelmä on uniikki, on niissä paljon yhteistä arkkitehtuurin suhteen. Esimerkiksi tuotteistetut sovellukset rakennetaan yhden pohja-arkkitehtuurin ympärille, josta toteutetaan erilaisia variaatioita vastaamaan erilaisiin vaatimuksiin. Arkkitehtuuria suunnitellessa tulee päättää mitä osia arkkitehtuurista voidaan käyttää uudelleen ja mitkä ovat tarkoitettuja vain tietylle asiakasvaatimukselle (Sommerville 2010, s. 151). Järjestelmän arkkitehtuuri voi pohjautua johonkin tiettyyn malliin tai tyyliin. Arkkitehtuurimalli on kuvaus järjestelmän teknisestä organisoinnista. Tällaisia voivat olla esimerkiksi asiakas-palvelin arkkitehtuuri tai kerrosarkkitehtuuri, johon Sommerville MVC:n luokittelee. Arkkitehtuurin valinnassa tulee ottaa huomioon erityisesti järjestelmän vaatimukset. Sommerville (Sommerville 2010, s.152) esittelee viisi pääarvoa, joiden perusteella arkkitehtuurin valintaa voidaan ohjata:

Suorituskyky	Jos suorituskyky on suurimpana vaatimuksena, tulee kriittisimmät operaatiot olla samalla tietokoneella ja arkkitehtuuri jaettu mahdollisimman suuriin kompoentteihin, jotta kommunikaatio olisi mahdollisimman suoraviivaista. Useiden pienten komponenttejen hajauttaminen usealla tietokoneelle tuo usein ylimääräistä viestintää komponenttejen välillä, mikä hidastaa komponenttejen yhteisiä operaatioita.
Tietoturva	Jos tietoturva on suurimpana vaatimuksena, tulee käyttää kerroarkkitehtuuria, jossa kaikkein kriittisimmät toiminnot on piilotettu alimpiin kerroksiin. Uloimmissa kerroksissa tulee olla tarkat datan validoinnit.
Turvallisuus	Jos tuvallisuuus on suurimpana vaatimuksena, tulee turvallisuuteen liittyvien operaatioiden olla rakennettuna yhden tai mahdollisimman harvan komponentin varaan. Tämän avulla pystytään järjestelmä suojaamaan mahdollisilta järjestelmävirheiltä helposti ja turvallisesti.
Saatavuus	Jos saatavuus on suurimpana vaatimuksena, tulee arkkitehtuuri suunnitella siten, että komponentit tekevät mahdollisimman vähän kriittisiä komponentteja. Tuolloin komponentteja voidaan päivittää ja korvata ilman, että itse järjestelmää tarvitsee

	pysäyttää.
Ylläpito	Jos ylläpito on suurimpana vaatimuksena, tulee komponenttejen olla mahdollisimman itsenäisiä. Datan tuottajien tulee olla erossa datan käyttäjistä ja jaettuja datamalleja tulee välttää.

Yllä esitellyissä arkkitehtuureissa voi kuitenkin olla ristiriitoja. Esimerkiksi suuret komponentit parantavat suorituskykyä ja pienet parantavat järjestelmän saatavuutta. Järjestelmän vaatimukset voivat vaatia kumpaakin, jolloin on tehtävä kompromissieja. Usein kompromissit pystytään saavuttamaan käyttämällä useitai arkkitehtuureja järjestelmän eri osissa (Sommerville 2010, s.152).

4.2 Hajautetut järjestelmät

Sulautetut sekä henkilökohtaiselle tietokoneelle käyttöön tarkoitetut järjestelmät ovat tyypillisesti rakennettu yhden prosessorin varaan. Niitä ei siis ole hajautettu useiden prosessorejen tai ytimien kesken toisin kuin monia suurempia järjestelmiä. Hajautuksen suunnittelu on yksi avaintekijöitä suunnittellessa arkkitehtuuria, jossa järjestelmä on niin suuri, että sen halutaan hajauttaa useampaan alijärjestelmään. Käytännössä lähes kaikki suuret tietokonepohjaiset järjestelmät ovat hajautettuja. Ne näyttäytyvät loppukäyttäjälle kuitenkin yhtenä yhtenäisenä järjestelmänä. Hajautetuissa järjestelmissä komponentit voivat olla useilla eri tietokoneilla. Tällöin tulee erityisesti ottaa huomioon niiden välinen kommunikointi, joka hoidetaan verkon yli (Sommerville 2010, s. 480). Coulouris (G 2005) esittelee seuraavat hyödyt hajautetuista järjestelmistä:

Resurssien jako	Hajautettu järjestelmä mahdollistaa laitteistojen ja sovellusten resurssien jakamisen. Esimerkiksi tulostimet, tiedostot ja levytila voidaan jakaa tietokoneiden kesken verkon yli.
Avoimuus	Hajautetut järjestelmät ovast usein avoimia ja siten käyttävät standardejen mukaisia protokollia, joten järjestelmän komponentta voidaan yhdistellä monelta eri taholta.
Rinnakkaisuus	Prosesseja voidaan ajaa rinnakkain verkossa olevien tietokoneiden kesken.
Skaalautuvuus	Hajautetuissa järjestelmissä mahdollistetaan resurssejen kasvattaminen lisäämällä tietokoneita vaatimusten muuttuessa. Käytännössä kuitenkin verkko, jolla tietokoneet yhdistetään, saattaa rajoittaa järjestelmän skaalautuvuutta.
Virheiden sieto	Useamman tietokoneen käyttö mahdollistaai datan kopioinnin tietokoneiden välillä, jolloin yhden tietokoneen virhetilanteessa voidaan turvautua useampaan muuhun tietokoneeseen.

Suuret hajautetut järjestelmäarkkitehtuurit ovat yllämainittujen hyötyjen takia korvanneet suurimman osan vanhemmista järjestelmistä, jotka kehitettiin 1990-luvulla. On kuitenkin vielä paljon henkilökohtaisia sovelluksia, jotka toimivat yhdellä koneella. Tällaisia ovat esimerkiksi erilaiset kuvankäsittelyohjelmat. Suurin osa sulautetuista järjestelmistä on myös

yhden prosessorin varaan rakennettuja (Sommerville 2010, s. 480).

Hajautetut järjestelmät ovat yleensä monimutkaisempia kuin yhden prosessorin ympärille rakennetut järjestelmät. Tämä tekee niistä vaikeampia suunnitella, toteuttaa ja testata. Erityisen vaikeaa on ymmärtää järjestelmän yhtenäisiä toimintoja, jotka eivät riipu yksittäisitä komponenteista vaan monien komponenttejen tarjoamasta yhtenäisestä toiminnosta. Tämä johtuu kommunikaatiosta niin järjestelmän yksittäisen komponenttejen välillä kuin järjestelmän infrastruktuurista. Esimerkiksi yksittäisen prosessorin ympärille rakennetun järjestelmän suoristuskyvyn perustana on prosessorin nopeus, mutta hajautetun järjestelmän kohdalla tulee ottaa huomioon verkon kuorma, verkon nopeus sekä kaikkien tietokoneiden nopeus, jotka ovat osa järjestelmää. WWW on rakennettu hajautettujen järjestelmien ympärille ja on hyvä esimerkki siitä, kuinka epävakaa ja ennalta-arvaamaton se luonteeltaan on. Vastausajat riippuvat arkkitehtuurista, verkon kuormasta sekä yksittäisten palvelimien kuormasta (Sommerville 2010, s. 481).

4.3 Hajautetut järjestelmät ja Internet

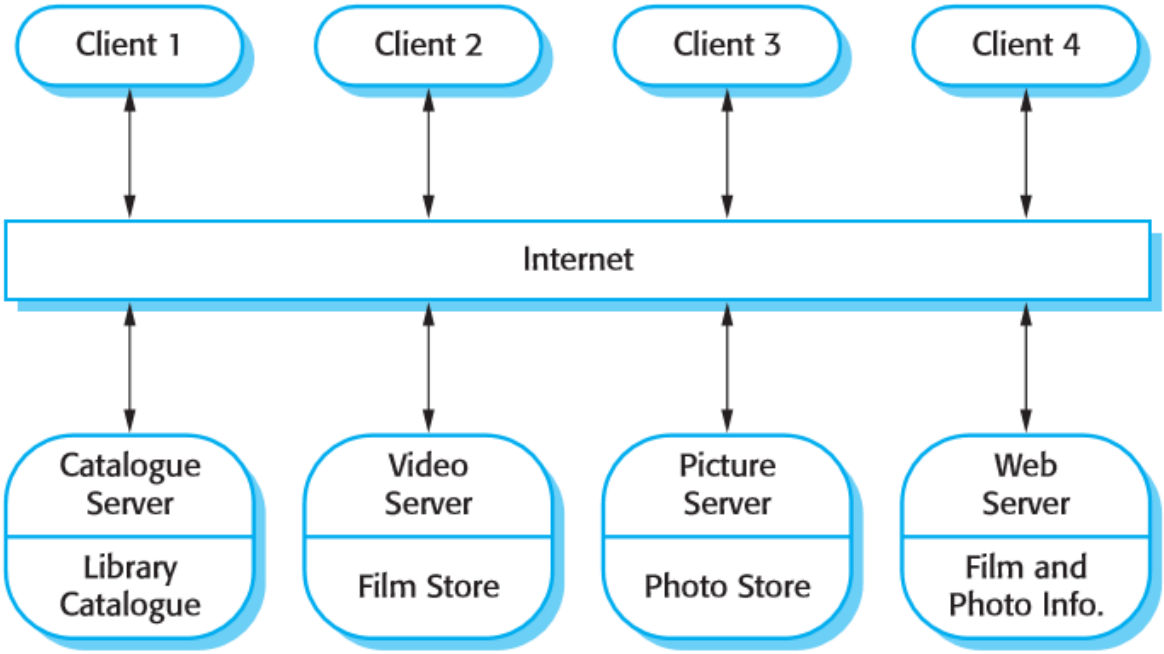
Internetin päälle rakennetut hajautetut järjestelmät käyttävät asiakas-palvelin arkkitehtuuria. Asiakas-palvelin järjestelmissä käyttäjä kommunikoi paikallisesti omalle koneelleen asennetun sovelluksen avulla. Tällaisia ovat esimerkiksi selaimet ja erilaiset sovellukset älypuhelimessa. Kommunikoinnissa käyttäjän sovellus ottaa yhteyden verkon yli toiseen sovellukseen, joka on käynnissä toisella tietokoneella (palvelin). Palvelin tarjoaa erilaisia palveluita, kuten esimerkiksi web-sivuja. Asiakas-palvelin arkkitehtuuri on hyvin geneerinen ja yleisesti käytössä oleva arkkitehtuuri. Se ei rajoitu sovelluksiin, jotka on hajautettu useille tietokoneille, vaan sitä voidaan käyttää myös arkkitehtuurina jakamaan sovellukset loogisesti yhdellä tietokoneella (Sommerville 2010, s.488).

Järjestelmät, jotka käyttävät asiakas-palvelin arkkitehtuuria, koostuvat useista palveluista. Palvelut on usein jaettu myös usealle tietokoneelle. Sommerville jakaa asiakas-palvelin arkkitehtuurin mallin kolmeen osaan (Sommerville 2010, s. 161):

1. Useita palvelimia, jotka tarjoavat palveluita muille komponenteille. Tällaisia ovat esimeriksi tulostin-palvelut tai tiedostojenhallintaan erikoistuneet tiedosto-palvelut.

2. Useita asiakasjärjestelmiä, jotka kutsuvat palvelimien tarjoamia palveluita. Normaalisti on useita asiakasjärjestelmän ilmentymiä, jotka käyttävät rinnakkain palvelimia.
3. Verkko, jonka kautta datan siirto palvelimien ja asiakasjärjestelmien välillä hoidetaan. Suurin osa asiakas-palvein järjestelmistä on hajautettu ja ne luovat yhteydet käyttäen apunaan Internet prokotollia.

Tärkeimpänä etuna asiakas-palvelin arkkitehtuurissa on komponenttejen itsenäisyys sekä hajautus. Palveluja ja palvelimia voidaan vaihtaa tai muokata ilman, että vaikutetaan muihin järjestelmän osiin. Palvelimen ei myöskään tarvitse tietää asiakassovelluksen identiteetistä mitään. Asiakassovellukselle riittää yleensä vain palvelimien ja niissä ajettavien palvelujen nimet. Asiakas kutsuu palvelimen tarjoamaa palvelua käskemällä tätä suorittamaan halutun ohjelmakoodin käyttäen apunaan esimerkiksi HTTP-protokollan trajoamaa viestiä, jota käytetään WWW:ssä. Pääasiallisesti asiakas lähettää pyynnön palvelimelle ja odottaa kunnes palvelin vastaa viestiin.



Kuvio 4. Sommervillen esimerkki elokuvakirjaston asiakas-palvelin arkkitehtuurista (Sommerville 2010, s.162)

Ylläesitetyssä mallissa esitellään monen käyttäjän web-pohjainen järjestelmä elokuvien ja kuvien tarjontan. Järjestelmässä useat palvelimet hallitsevat ja tarjoavat erityyppisiä multimediatiedostoja. Videotiedostojen tarjonta tulee olla erittäin nopeaa ja synkornista. Jotta tähän pystytään tulee videotiedosto pakata pienemmäksi ja tarjota pienemmällä resoluutiolla. Lisäksi videoiden formaatit voivat vaihdella, jolloin videoille tarkoitetun palvelimen tulee pystyä tarjoamaan palvelua, jossa erilaisia videoformaatteja voidaan pakata ja purkaa. Kuvia on kuitenkin sopivaa tarjota suurilla resoluutioilla, joten niiden käsittely voidaan hoitaa erillisellä palvelimella. Näin ei tarvitse kuormittaa yhtä palvelinta kahdella toisistaan poikkeavalla prosessilla. Catalogin tulee pystyä käsittelemään useita erilaisia kyselyitä sekä tarjota yhteyksiä järjestelmiin, jotka hallitsevat esimerkiksi elokuvien tietoja sekä valokuvien myyntiä. Asiakassovellus on yksinkertainen käyttöiittymä, joka on rakennettu web-selaimeen ottamaan yhteyksiä palveluihin (Sommerville 2010, s. 163).

5 Internet

Internet ja sen arkkitehtuuri on syntynyt hyvin pienistä saavutuksista. Internetin taustalla ei ole alunperin ollut minkäänlaista suurempaa suunnitelmaa. Tutkiessamme Internetin arkkitehtuuria tulee muistaa, että tekniset muutokset ovat jatkuvia tietotekniikan alalla. Monet arkkitehtuurilliset ratkaisut, jotka olivat suosiossa muutamia vuosia sitten ovat nyt jo vanhentuneet. Toisaalta ratkaisut, jotka tänä päivänä nähdään hyvinä, voivat jo huomenna olla huonoja. Voidaan sanoa, että ainut asia joka Internetin suhteen pysyy on jatkuva muutos. Internet yhteisöjen monet jäsenet väittävät, että Internetillä ei ole arkkitehtuuria, mutta kuitenkin eräänlaisia perinteitä, johon koko Internet pohjautuu (Carpentern 1996). Väite voidaan kuitenkin kumota, silllä Internet perustuu asiakas-palvelin arkkitehtuurin (Sommerville 2010, s. 162).

Internetin tavoite on tarjota yhteydet, jossa pohjana käytetään Internet Protokollaa. Monien yhteisön jäsenien mielestä tärkeää ei ole se millä tavalla yhteydet luodaan verkon sisällä, vaan se millä tavalla yhteyksien alku- ja loppupää käyttäytyvät. Tähän tärkeimpänä argumenttina pidetään sitä, että useat tarvittavat toiminnallisuudet voidaan toteuttaa vain IP:tä käyttävien systeemien puolella. Jokainen hyvin suunniteltu verkko sisältää mahdollisuuden jollakin tasolla epäonnistua viestin kuljettamisessa. Tällaisen ongelman kanssa tulisi toimia siten, että vastuu kommunikaation eheydestä annetaan ulkopuolisille sovelluksille. Nykyinen Internetin exponentaalinen kasvu näyttää kuitenkin sen, että yhteydet ovat paljon arvokkaampia kuin yksittäiset sovellukset. Tällaiset yhteydet vaativat yhteistyötä palveluntarjoajien ja televiestintäympäristöjen välille (Carpentern 1996).

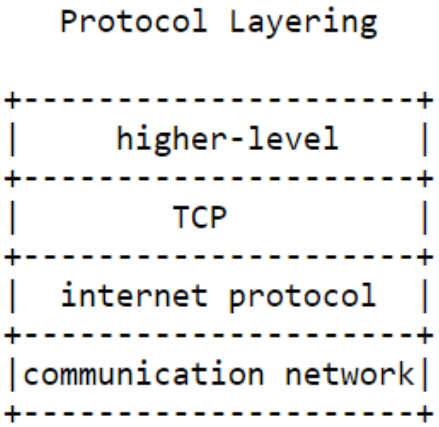
Yleisesti on haluttua, että olisi vain yksi protokolla, jonka kautta yhteydet voidaan muodostaa. Tämän protokollan päälle voidaan kuitenkin rakentaa monia muita protokollia, jotka hoitavat muita erilaisia tarpeita. Käytännössä on ainakin kaksi syytä miksi verkko-protokollien tasoja tarvitaan useita. Esimerkiksi eri IP:stä voidaan haluta vaihtaa toiseen versioon. Myös uudet vaatimukset voivat synnyttää uusia protokollia. Internetin protokollien tulee olla täysin riippumattomia laitteistosta. Tällöin Internet voidaan toteuttaa minkä tahansa digitaalisen teknologian päälle (Carpentern 1996).

5.1 Historia

Internetin historia on lähtöisiin tarpeesta luoda yhteyksiä tietokoneiden välille, joiden avulla voidaan luoda verkkoja. Verkkojen kautta erilaiset sovellukset voivat keskustella keskenään. Tämä on johtanut tarpeeseen luoda standardit, jotka määrittävät sen millä tavalla yhteyksiä tulisi luoda ja ylläpitää. TCP (Transmission Control Protocol) on standardi, joka määrittää yhteyden luonnin tietokoneiden välillä sekä niiden välisen kommunikoinnin. TCP:n avulla sovellukset pystyvät lähettämään dataa toisilleen verkon yli. TCP kehitettiin Yhdysvaltain asevoimien tutkimusorganisaatiossa DARPA:ssa (Defense Advanced Research Projects Agency) (Bush 1981a).

5.2 TCP

TCP on tarkoitettu toimivan eri tasoihin jaetussa protokollien hierarkiassa, joka tukee verkkosovelluksia. Sen tarkoitus on pystyä toimimaan monien alemman tason protokollien päällä ja olla ottamatta kantaa käytetäänkö yhteyden muodostuksessa esimerkiksi paketti- tai piirikytkentää. DARPA määrittää eri protokollien tasot seuraavasti:



Kuvio 5. (Bush 1981a)

TCP on rakennettu Internet Protokollan päälle, joka tarjoaa TCP:lle työkalut määrittää lähettäjän ja vastaanottajan osoitteet erilaisissa verkoissa. Internet Protokolla hoitaa myös mahdollisten TCP-segmenttejen pirstaloinnin ja uudelleenkokoamisen, jolloin TCP:n ei tarvitse

näistä huolehtia. Se myös pitää mukanaan tiedon TCP-segmenttejen järjestyksestä, tietoturvamäärityksistä sekä lajittelusta. TCP tarjoaa rajapinnan luotettavalle yhteydelle useista verkoista koostuvassa ympäristössä (Bush 1981a)

TCP koostuu kahdesta rajapinnasta, joista toinen toimii käyttäjälle tai sovellukselle ja toinen alemman tason protokollille kuten Internet Protokollalle. Sovelluksille tarkoitettu rajapinta koostuu vastaavista kutsuista, joita käyttöjärjestelmä tarjoaa tiedostojen hallintaan. Esimerkiksi se sisältää kutsut yhteyden avaamiseen ja sulkemiseen sekä datan vastaanottoon ja lähetykseen yhteyden läpi. TCP pystyy myös asynkronisesti kommunikoimaan sovelluksien kanssa. Rajapinta alemmille protokollille on yleisesti määrittelemätön, mutta se olettaa rajapinnassa olevan mekansimin, jossa kaksi tasoa pystyvät asynkronisesti siirtämään dataa toistensa välillä. Yleensä kyseinen määrittely oletetaan löytyvän alemman tason protokollalta. TCP on tarkoitettu toimivan geneerisessä ympäristössä, mutta yleisesti voidaan olettaa alemman tason protokollan olevan Internet Protokolla (Bush 1981b).

TCP:n tärkein tehtävä on tarjota turvallinen ja luotettava yhteys kahden prosessin välille. Tarjotakseen tällaisen palvelun Internetin päälle, tulee tietyt osa-alueet toteutua. IETF (Bush 1981b) määrittää TCP:n toteuttamat osa-alueet seuraavasti:

Datan siirto	TCP pystyy lähettämään jatkuvasti paketteja kummankin käytätjän suuntaan. Joskus käyttäjien tulee varmistaa, että kaikki lähetetty data on lähetetty vastaanottajalle. Tähän tarkoitukseen on kehitelty push -toiminto, joka käskee TCP:tä lähettämään datan vastaanottajalle.
Luotettavuus	TCP:n tulee pystyä selviytymään tilanteesta, jossa data on jollain tavalla korruptoitunut tai hävinnyt. Tämä on ratkaistu lisäämällä jokaiseen pakettiin sekvenssinumero ja jokaisessa viestissä vaaditaan kuittausta vastaanottajalta. Jos kuittausta ei jostain syystä saada, lähetetään paketti uudelleen. Vastaanottajan puolella sekvenssinumeron avulla pystytään päättelemään oikea järjestys.
Datavirran hallinta	TCP antaa vastaanottajalle mahdollisuuden rajata datamääriä. Tämä mahdollistetaan antamalla vastaanottajalle mahdollisuus

	lähettää kuittausviestin mukana ikkuna, joka määrittää rajat datalle.
Kanavointi	Koska yhden isäntäkoneen on mahdollista pavella useita TCP-yhteyksiä rinnakkain, tulee TCP:n tarjota lista osoitteita ja portteja yksittäistä severiä kohden. Jokaiselle yhteydelle luodaan soketti (socket). Soketti-pari muodostaa uniikin yhteyden, missä yksittäistä sokettia voidaan myös käyttää moniin eri yhteyksiin rinnakkain. Hyvänä esimerkkinä yhden soketin jakamisesta on erilaiset logitusprosessit, joita käytetään jatkuvasti. Erillisiin portteihin liittäminen hoidetaan aina jokaisen isäntäkoneen toimesta.
Yhteydet	Luotettavuuden ja kanavoinnin toteutus vaatii TCP:tä erillisen tilan alustamista ja ylläpitoa jokaiselle datavirralle. Tietoa soketeista, sekvenssinumeroista ja ikkunoiden kokoa kutsutaan yhteydeksi. Jokainen yhteys määritellään sokettiparin perusteella. Kun kaksi prosessia haluaa kommunikoida kekskenään, kummankin TCP:n tulee ensin luoda yhteys, jolloin kummankin tila alustetaan. Kun yhteys on valmis, se terminoidaan tai suljetaan resurssien vapauttamiseksi. Koska yhteydet luodaan epävakaan internet-yhteyden sekä epäluotettavien prosessejen välille, TCP käyttää aikapohjausta kättely-mekanismia yhteyden muodostamisessa. Tämän avulla vältetään virheelliset yhteydenmuodostukset.
Priorisointi ja tietoturva	TCP:n käyttäjät voivat hallita priorisointia sekä tietoturvaa erilaisilla arvoilla. Tällainen on esimerkiksi TCP-oktetin sisältämät ensimmäiset kolme bittiä, jotka määrittävät oktetin tärkeysjärjestyksen.

TCP:n oletetaan olevan tavallinen moduuli käyttöjärjestelmässä. Käyttäjien tulee pystyä hallitsemaan TCP:tä niinkuin mitä tahansa tiedostojärjestelmää. Se pystyy myös kutsumaan muita käyttöjärjestelmän toimintoja. Rajapinta itse verkkoon tulee kuitenkin toteuttaa laitteen ajurissa. TCP ei kuitenkaan kutsu ajurin rajapintaa suoraan, vaan käyttää Internet Pro-

tokollaa. Internet protokolla sen sijaan kutsuu mahdollisen alemman tason ajurin rajapintaa. Käyttäjälle tarkoitettu rajapinta toteutetaan TCP:ssä siten, että sen kutsut ovat vastaavia kuin muissakin käyttöjärjestelmän ohjelmien tarjoamissa kutsuissa. TCP:n käyttäjille tarjoamassa rajapinnassa kutsut ovat seuraavat (Bush 1981c):

OPEN	Avaa yhteyden
CLOSE	Sulkee yhteyden
SEND	Lähettää dataa
RECEIVE	Ottaa vastaan dataa
STATUS	Palauttaa yhteyden tilan

Kutsut ovat täysin verrattavissa käyttöjärjestelmän käyttäjäsovelluksien kutsuihin. Tällaisia ovat esimeriksi tiedostojen luku- ja kirjoituskutsut. Rajapinnan tarkoituksena on tarjota kutsuja datagrammejen lähetykseen ja vastaanottamiseen kaikkien osapuolen välillä, jotka on kytketty Internettiin. Kutsujen avulla voidaan viedä parametereissa erilaista tietoa, kuten osoite, palvelun tyyppi, järjestysnumero ja muita datagrammejen hallintaan liittyviä tietoja.

5.3 Internet Protokolla

Internet Protokolla on tarkoitettu käytettäväksi tietokoneiden välisiin verkkoihin, joissa on käytössä pakettikytkentä. Se tarjoaa toteutuksen siirtämään datagrammeja käyttäjien välillä. Se toteuttaa myös pitkien datagrammejen pirstaloinnin pienempiin osiin tarvittaessa, jos dataa halutaan siirtää verkkojen läpi, jossa vain pienemmät paketit ovat sallittuja. Internet Protokolla ei ota kantaa datan luotettavuuteen, datavirran hallintaan tai järjestykseen. Tämä jätetään ylemmän tason protokollien hoidettavaksi. Esimerkiksi TCP-moduuli kutsuu Internet moduulia ottamaan haltuun TCP-segmentin, joka sisältää otsikon ja lähettäjän datan. Otsikko sisältää osoitteen ja muut tarvittavat parametrit. Tämän jälkeen Internet-moduuli luo internet datagrammin ja kutsuu paikallista verkkorajapintaa lähettämään datagrammin eteenpäin. (Southern California 1981)

Internet Protokolla toteuttaa kaksi toiminnallisuutta: osituksen ja osoitteistuksen. Datagrammin otsikoissa tulleiden tietojen perusteella internet moduuli osaa lähettää datan oikeaan

paikkaan sekä pirstaloida datan sopivaksi verkolle. Internet Protokolla käsittelee jokaista datagrammia erillään ilman riippuvuuksia muista datagrammeista. Internet Protokolla käyttää neljää palvelua: Palvelutyyppi, Elinkaari, Vaihtoehdot ja Tarkistussumma. Palvelutyyppi on abstrakti tai geneerinen lista parametrejä, jotka määrittää verkolle minkä tyyppinen palvelu on kyseessä. Yhdyskäytävät käyttävät tätä informaatiota valitsemaan oikeat parametrit tietylle verkolle tai määrittämään seuraavan verkon mihin hypätä. Yhdyskäytävillä tarkoitetaan tässä verkon solmua, joka mahdollistaa siirtymisen toiseen verkkoon. Solmut voivat olla esimerkiksi tietokoneita. Elinkaarella tarkoitetaan ylärajaa, jonka aikana datagrammin tulee saavuttaa määränpäänsä. Jos määränpäätä ei saavuteta, datagrammi tuhotaan. Yläraja määritetään lähettäjän toimesta ja sitä vähennetään siirron aikana. Datagrammi tuhoutuu saavuttaessan arvon nolla (Southern California 1981). Vaihtoehdot ovat käytössä erityisissä tilanteissa. Tällaisia ovat esimerkiksi tietyt ohjaukset verkossa tai lisäykset tietoturvaan. Tarkistussummalla varmistetaan, että datagrammi on siirretty onnistuneesti. Datagrammin sisältö voi kuitenkin sisältää virheitä. Jos Tarkistussummassa on virheitä, datagrammi tuhotaan välittömästi.

Internet Protokolla ei tarjoa luotettavaa kommunikaatioyhteyttä. Se ei sisällä alku- ja loppupään kuittauksia eikä se ota kantaa datan eheyteen. Se ei myöskään sisällä uudelleenlähetyksiä tai muuta datavirran hallintaa (Southern California 1981).

5.4 HTTP-protokolla

HTTP (Hypertext Transfer Protocol) on ollut Internetin WWW:n (World-Wide-Web) käytössä vuodesta 1990 lähtien. HTTP:n yhtenä tarkoituksena on siirtää hyperteksi-dokumentteja järjestelmien välillä. Hypertekstillä tarkoitetaan tietokoneissa käytettyä tekstiä, jossa mahdollistetaan ristiviittauksia dokumenttejen välillä. Näitä viittauksia kutsutaan hyperlinkeiksi (Cailliau 1990). HTTP on sovellustason protokolla ja se käyttää alemman kuljetuskerroksen protokollia yhteyden luomisessa. WWW:ssä HTTP käyttää kuljetuskerroksen TCP protokollaa (Fielding 1991a). HTTP on geneerinen sekä tilaton ja sitä voidaan käyttää muuhunkin kuin hypertekstin siirtämiseen ylätunnisteiden, pyyntöjen sekä virhekoodejen kautta. Tällaisia ovat esimerkiksi nimipalvelimet ja jaetut oliopohjaiset hallintajärjestelmät. HTTP-protokolla mahdollistaa järjestelmien rakentamisen riippumattomana siirrettävästä datasta

(Fielding 1999a). HTTP tuo sovelluksille mahdollisuuden kommunikoida avoimella joukolla menetelmiä ja ylätunnisteita, jotka määrittävät itse HTTP-viestin tarkoituksen. HTTP käyttää URI:n spesifikaatiota määrittämään resurssejen sijainnin (Fielding 1994).

```
generic-message = start-line
                  *(message-header CRLF)
                  CRLF
                  [ message-body ]
start-line      = Request-Line | Status-Line
```

Kuvio 6. HTTP-viesti kokonaisuudessaan (Fielding 1999b)

HTTP-viestit koostuvat asiakaspuolen viesteistä sekä palvelimen vastauksista asiakkaalle. Molemmat viestit sisältävät aloitusrivin, tyhjän tai useamman ylätunnisteen, tyhjän rivin (CRLF) ja mahdollisen lähetettävän viestin. CRLF määrittää rivinvaihdon ja palvelimen on pystyttävä protokollan datavirtaa luettaessa huomioimaan se. Esimerkiksi datavirtaa luettaessa viesti alkaa CRLF:llä se tulee hylätä. HTTP-viesti sisältää erilaisia ylätunnisteita (header), joiden avulla voidaan viedä ylimääräistä tietoa HTTP-pyynnöstä sekä asiakaspäästä. Ylätunnisteita voi olla useita yhdessä HTTP-viestissä. Viestikentässä (message-body) viedään tietyn viestin sisältö, jos sellainen on viestissä mukana (Fielding 1999b). Viestikentän olemassaolo määritetään pituuskentässä (Content-Length) tai kentässä, joka määrittää siirron enkoodauksen (Transfer-Encoding). HTTP-pyynnön ei tule sisältää viestikenttää, jos esimerkiksi enkoodauskenttä löytyy pyynnöstä ja se sisältää minkä tahansa muun arvon kuin "identity". Seuraavaksi esitellään yleisellä tasolla esimerkki millä tavalla asiakas saa HTTP-palvelimelta vastauksena hyperteksti-dokumentin.

Yhteyden luonnissa asiakas luo TCP-IP yhteyden palvelimen ja asiakkaan välille käyttäen osoitteessa määritettyä porttinumeroa. Jos porttinumeroa ei ole annettu HTTP olettaa porttinumeron olevan 80. Asiakas lähettää pyynnön, joka koostuu yhdestä rivistä ASCII-merkkejä. Pyyntö sisältää sanan "GET", välilyönnin sekä ja dokumentin osoitteen. Dokumentin osoitteen tulee olla yksi yhtenäinen sana. Kaikki ylimääräiset sanat, jotka löydetään dokumentin osoitteesta jätetään huomioimatta tai ne tulkitaan HTTP-spesifikaation mukaisesti (Fielding 1991b). Vastaus yksinkertaiseen GET-viestiin voi sisältää esimerkiksi HTML-pohjaisen (Hypertext Markup language) ASCII datavirran. HTML-kieltä käytetään julkaisemaan hyperteksti-dokumenttejä WWW:ssä. HTML:n pohjana on käytetty SGML:ää, joka on standardi määrit-

telemään merkkauskieliä (Fielding 1995). HTML-dokumentit ovat siis SGML-dokumentteja, jotka toteuttavat SGML:n määritelmät. Hyvin toteutetut asiakassovellukset lukevat koko dokumentin mahdollisimman nopeasti, jolloin käyttäjän laukaisemat toiminnot, toteutetaan vasta dokumentin prosessoinnin jälkeen. TCP/IP yhteys suljetaan palvelimen toimesta kun koko dokumentti on siirretty (Fielding 1991a)

HTTP-protokollan alkuperäinen toteutus on tarkoitettu yksisuuntaiseksi, jolloin asiakas-puolen tulee lähettää aina pyyntö palvelimelle vastaanottaakseen dokumentteja. Palvelin-puolen ei ole tarkoitus syöttää tietoa suoraan asiakas-puolelle ilman tämän pyyntöä. tästä syystä asiakas-puolen on aina pollattava tietoja erikseen (polling). Tämä kuitenkin luo paljon kuormaa ja turhia kyselyitä palvelimelle niissä tilanteissa, jossa pyynnölle tarkoitettu vastaus ei ole vielä valmis. Tämän ratkaisemiseksi HTTP-protokollaan on toteutettu mahdollisuus lähettää viestejä palvelimelta asiakkaalle. Tämä mahdollistaa kaksisuuntaista viestintää asiakkaan ja palvelimen välillä, jolloin asiakkaan ei tarvitse pollata jokaista muutosta. Tyypllisimmät toteutukset ovat *HTTP Long Polling* (pitkä pollaus) ja *HTTP-Streaming* (HTTP-virta). Pitkässä pollauksessa palvelin pyrkii pitämään yhteyden auki selaimelle ja vastaamalla vasta, kun pyyntö on prosessoitu. Tämän jälkeen yhteys suljetaan. HTTP-virrassa palvelin pitää pyynnön auki niin kauan kuin vain mahdollista eikä koskaan terminoi pyyntöä tai sulje yhteyttä vaan lähettää vastauksia osissa (Berners-Lee 2011, s. 2). Edellä mainittuja toteutuksia ei kuitenkaan ole tarkoitettu alkuperäiseen HTTP-toteutukseen. HTTP on alunperin tarkoitettu yksisuuntaiseen kommunikaatioon, mistä syystä kaksisuuntaisen kommunikaation lisääminen HTTP:hen tuottaa ongelmia. Alla esitellään kummankin toteutuksen ongelmat.

Pitkä pollaus:

Otsikoiden kuormitus	Jokainen pyyntö on kokonainen HTTP-viesti ja sisältää kaiken datan otsikoita myöten. Tästä syystä lyhyissäkin viesteissä siirrettävät datamäärät ovat isoja, jotka johtavat verkon kuormitukseen.
Maksimaalinen viive	Kun pyyntö on lähetetty asiakkaalle, tulee palvelimen odottaa seuraava pyyntöä ennen kuin voidaan lähettää seuraavaa viesti. Tämä johtaa lukuisiin ajallisesti pidempiin viesteihin.
Yhteyksien luonti	Yhteyksiä joudutaan sulkemaan ja availemaan useasti

Resurssien jakaminen	Käyttöjärjestelmät sekä sovellukset pyrkivät pitämään TCP-yhteyksien resurssit minimaalisina. Tästä syystä vastaan tulee erilaisia rajoituksia palvelimien yhteyksien hallinnassa.
Yhteyksien heikentäminen	Suuressa kuormassa toimiva asiakas ja palvelin pyrkivät alentamaan tehokkuutta viestin hitauden kustannuksella.
Aikakatkaisut	Jokaiselle pyynnölle muodostetaan aikaraja, jolloin yhteys katkaistaan. Tämä raja asetetaan usein hyvin korkealle, mutta selaimissa on kuitenkin asetettu normaaliksi ajaksi 300 sekunttia ja verkkojen infrarstruktuurissa aikaraja on usein paljon pienempi. Tästä syystä yhteyksiä joudutaan väistämättä katkaisemaan pitkien pyyntöjen aikana.
Asiakaspuolen rajoitteet	Asiakas puolella ei ole tapaa ilmoittaa pitkästä pollauksesta vaan se toteutetaan aina palvelimen toimesta. Asiakaspuolen tulee siis olettaa vain pyyntöjen kestävän odotettua kauemmin. Tämä vois olla usein ongelmallista erilaisissa toteutuksissa.

HTTP-virta:

Välipalvelimet	HTTP-protokolla mahdollistaa välipalvelimien kuulumisen mukaan datan lähetykseen asiakkaan ja palvelimen välissä. Välipalvelimilla ei ole vaatimusta suoraan siirtää vastausta asiakkaalle. Välipalvelimien sallitaan myös puskuroida vastaukset ennenkuin ne lähetetäään edelleen eteenpäin. HTTP-virta ei toimi tällaisessa tilanteessa.
Maksimaalinen viive	Täydellisessä verkossa, HTTP-virran keskimääräinen ja maksimaalinen viive on yksi siirto verkossa. Todellisuudessa maksimaalinen viive on kuitenkin surempi johtuen verkon ja selaimien rajoituksista. Selaimien tekniikat, jotka terminoivat HTTP-virran yhteyksiä ovat usein kytköksissä JavaScript tai DOM (Document Object Model) elementteihin, jotka kasvattavat kokoaan, jokaisesta uudesta viestistä. Välttääkseen hallitsemattoman muistin käytön selaimessa, tulee HTTP-virran katkaista

	yhteys välillä ja lähettää uusi pyyntö uuden virran alustamiseksi.
Asiakkaan puskurointi	HTTP:ssa ei ole vaatimusta asiakaspäälle luoda dataa osittaisista HTTP-vastauksista. Esimerkiksi, jos vastaus lähetetään paloissa, jotka sisältävät yksittäisen osan javascriptiä, ei selaimelta vaadita osan ajoa ennenkuin koko vastaus on saatu.
Datan paloittelu	Vastauksien muuttaminen sovellukselle ajettavaksi tulee tehdä sovellustasolla. HTTP-protokollan viestejä ei voida suoraan tulkita sovellusviesteiksi, koska välipalvelimet saattavat muuttaa virran viestejen rakennetta. Tämä ongelma ei ole läsnä pitkässä pollauksessa, koska siinä jokainen viesti on oma kokonainen HTTP-pyyntö (Berners-Lee 2011, s. 2).

Yllä esitetyistä ongelmista johtuen kaksisuuntaista kommunikaatiota HTTP:n päällä ei voi tutkimuksen kannalta luotettavasti käyttää kaksisuuntaisen kommunikaation toteuttamiseen sen epävarmuuden vuoksi.

5.5 Web-soketti protokolla

Koska HTTP:ta ei oltu alunperin tarkoitettu kaksisuuntaiseen kommunikaation, tehtiin useat aiemmat teknologiset ratkaisut tehokkuuden ja toimintavarmuuden väliltä. Tämän pohjalta on kehitetty web-soketti protokolla. Web-soketti protokolla on itsenäinen TCP-pohjainen protokolla. Sen avulla mahdollistetaan kaksisuuntainen kommunikaatio palvelimen ja asiakkaan välillä. Protokollaa voidaan käyttää niin tavallisissa web-sivustoissa kuin peleissäkin Se on suunniteltu erityisesti syrjäyttämään olemassa olevia kommunikointiteknologioita, jotka käyttävät HTTP:ta datan siirrossa. Se on myös luotu toimimaan ympäristöissä, joissa käytetään HTTP:tä. Web-soketteja käytetään oletetusti HTTP-porttejen 443 ja 80 yli. HTTP ei kuitenkaan rajoita web-sokettejen toteutusta. Web-soketti käyttää yksittäistä TCP-yhteyttä lähettäen dataa kumpaankin suuntaan. Protokollassa on kaksi osaa: kättely ja datan siirto (Fette 2011, s. 1.1).

Kättely ei ole riippuvainen HTTP-protokollasta, mutta se on rakennettu toimimaan HTTP-

pohjaisten protokollien kanssa. Tässä tutkimuksessa keskitytään kättelyyn HTTP-protokollan näkökulmasta. All olevassa esimerkissä selain aloittaa kättelyn kutsumalla HTTP-protokollan mukaisesti GET-kutsua. Tämän jälkeen selain vastaa kättelyviestillä, jossa protokolla vaihdetaan käyttämään web-sokettia (Fette 2011, s. 1.1):

```
1       GET /chat HTTP/1.1
2       Host: server.example.com
3       Upgrade: websocket
4       Connection: Upgrade
5       Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
6       Origin: http://example.com
7       Sec-WebSocket-Protocol: chat, superchat
8       Sec-WebSocket-Version: 13
```

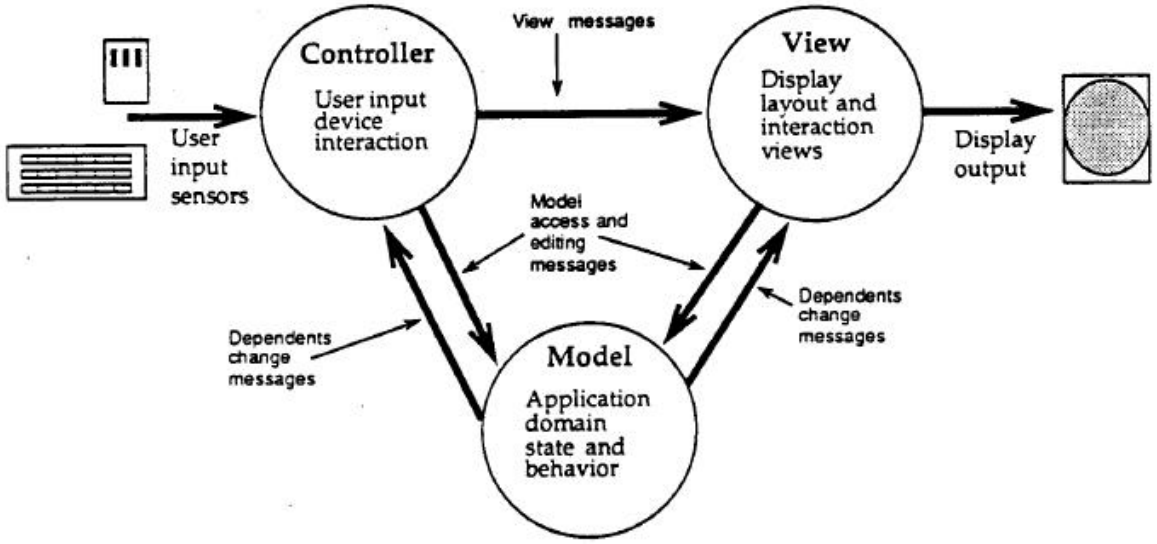
Palvelimelta tuleva vastaus kättelyyn näyttää seuraavalta:

```
1        HTTP/1.1 101 Switching Protocols
2       Upgrade: websocket
3       Connection: Upgrade
4       Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzzhZRbK+xOo=
5       Sec-WebSocket-Protocol: chat
```

Kun asiakas ja palvelin ovat kummatkin lähettäneet kättelyviestinsä ja kättely on onnistunut, alkaa datan siirto. Tuolloin asiakkaan ja palvelimen välillä on yhteys, jonka avulla kummatkin voivat lähettää toisilleen viestejä suunnasta riippumatta (Fette 2011, s. 1.2).

6 MVC

MVC-arkkitehtuurin perusajatus on erottaa käyttöliittymä sovelluslogiikasta ja näin tehdä sovelluksesta helposti ylläpidettävä kolmen eri komponentin avulla: Malli (Model), Näkymä (View) ja Ohjain (Controller). Jokainen komponentti on erikoistunut sovelluksessa johonkin tiettyyn tehtävään. Mallin tehtävänä on hallita sovelluksen tilaa ja vastata sen käsittelemästä datasta ohjaimelle ja näkymälle. Näkymän tehtävänä on taas näyttää sovelluksen käyttöliittymä ja sitä kautta mallin dataa. Ohjaimen tarkoitus on ottaa vastaan syötteitä käyttäjältä käskien mallia ja näkymää muuttumaan tarvittaessa.



Kuvio 7. Model-View-Controller State and Message Sending (G. E. Krasner 1988, s. 5)

Jokaisella komponentilla on oma rajattu tehtävänsä ja ohjelmakoodi tulee jakaa näiden komponentien kesken. Jotta MVC:tä pystyttäisiin käyttämään tehokkaasti, tulee ymmärtää komponenttien työnjako sekä se kuinka komponentit kommunikoivat keskenään (Steve 1992).

Luodessamme MVC-arkkitehtuurin toteuttavia komponentteja, tulee ne periä jostakin abstraktista pohjaluokasta (Model, View tai Controller), joka määrittelee kyseisen komponentin käyttäytymisen MVC:ssä (G. E. Krasner 1988, s. 5). Tässä kappaleessa käydään jokaisen komponentin toteutus erikseen läpi käyttäen ohjelmointikielenä Smalltalkia. Lähteenä käytetään Krasnerin julkaisua (G. E. Krasner 1988).

Yleisesti MVC-komponenttien toimintaa kuvaavassa esimerkissä käyttäjältä tulee jokin syöte, jonka sillä hetkellä aktiivinen ohjain ottaa vastaan. Syötteen perusteella ohjain lähettää mallille viestin. Malli puolestaan tekee sille määrättyjä operaatioita muuttaen tilaansa ja lähettää edelleen viestin muutoksestaan kaikille siihen liitetyille riippuvuuksille (näkymät ja ohjaimet). Näkymät voivat tämän jälkeen kysyä mallilta sen nykyistä tilaa ja päivittää itsensä, jos siihen on tarvetta. Ohjaimet voivat myös muuttaa tilaansa riippuen mallin tilasta (G. E. Krasner 1988, s. 4).

Suurin merkitys MVC:llä on luoda silta ihmismielen hahmottamalle mallille ja tietokoneessa esiintyvälle mallille. Oikein toteutettuna MVC:n avulla luodaan illuusio siitä, että käyttäjä kommunikoi suoraan mallin kanssa. Todellisuudessa kuitenkin ohjain ja näkymä muodostavat yhdessä rajapinnan sille, miltä malli näyttää ulospäin ja miten sitä käsitellään. Ohjain huolehtii syötteiden vastaanottamisesta ja käsittelemisestä. Näkymä taas huolehtii mallin graafisesta puolesta (Reenskaug 2003, s. 11-12).

6.1 Historia

MVC:n esitteli Norjalainen Trygve Reenskaug ollessaan mukana Xerox PARC -tutkimushankkeessa. Ensimmäinen julkaisu MVC:stä kirjoitettiin vuonna 1978 samassa tutkimuskeskuksessa. Tuolloin julkaisussa esiteltiin kolmen komponentin sijasta neljä komponenttia: Malli (Model), Näkymä(View), Ohjain(Controller) sekä Muokkaaja(Editor). Muokkaaja on väliaikainen komponentti, jonka näkymä luo itsensä ja syötelaitteiden välille. Muokkaaja-komponentista kuitenkin luovuttiin käsitteenä ja se sisällytettiin näkymään ja ohjaimeen (Parc 1978). Alkuperäinen Xerox PARC:n tuottama raportti MVC:stä oli Reenskaugin vuonna 1979 kirjoittama THING-MODEL-VIEW-EDITOR (Parc 1979b). Raportti esitteli MVC:n komponentteja käyttäen hyväksi esimerkkejä Reenskaugin omasta suunnittelutyöstä. Thing-komponentilla mallinnettiin jotakin isompaa kokonaisuutta, joka hallitsee pienempiä kokonaisuuksia. Sitä voidaan ajatella eräänlaisena suurena mallina,joka on jaettu useisiin pienempiin malleihin. Editor-komponentti luo rajapinnan käyttäjän ja yhden tai useamman näkymän välille. Se tarjoaa käyttäjälle sopivan komento-rajapinnan kuten esimerkisi valikon, joka vaihtuu sisällön muuttuessa (Parc 1979b). Reenskaug hylkäsi kuitenkin Editor- ja Thing-komponentin ja päätyi Adele Goldbergin avustuksella termeihin Models-Views-Controllers julkaisten saman

vuoden lopulla raportin, jossa määritellään lyhyesti jokaisen komponentin tehtävä (MODELS-VIEWS-CONTROLLERS) (Parc 1979a). Koska MVC:n historia ja suurin osa MVC:n alkuperäisistä julkaisuista pohjautuvat Smalltalk-ohjelmointikieleen, esitellään myös tässä tutkielmassa MVC:n toteutusta Smalltalkilla. Tämä ei kuitenkaan rajoita tarkastelua, koska arkkitehtuurin idea pysyy täysin samana riippumatta ohjelmointikielestä.

6.2 Malli (Model)

Malli pitää yllä sovelluksen tilaa sekä vastaa sovelluksen tallentamasta datasta. Se voi olla esimerkiksi kokonaislukumuuttuja laskuri-sovelluksessa, merkkijono-olio tekstinkäsittelyohjelmassa tai mikä tahansa monimutkainen olio (G. E. Krasner 1988, s. 3). Kaikkein yksinkertaisimmassa tapauksessa mallin ei tarvitse kommunikoida ollenkaan ohjaimen ja näkymän kanssa, vaan toimia passiivisena säiliönä datalle. Tällaisesta tilanteesta on hyvä esimerkki yksinkertainen tekstieditori, jossa teksti nähdään juuri sellaisena kuin se olisi paperilla. Tässä tapauksessa mallin ei tarvitse ottaa vastuuta kommunikoinnista näkymälle, koska muutokset tekstiin tapahtuvat käyttäjän pyynnöstä. Tällöin ohjain ottaa vastaan käyttäjän syötteet ja voi esimerkiksi ilmoittaa näkymälle muutoksesta, jolloin näkymä päivittää mallin. Ohjain voi myös päivittää mallin ja ilmoittaa tästä näkymälle, jolloin näkymä voi pyytää mallin sen hetkistä tilaa. Kummassakaan tapauksesssa mallin ei tarvitse tietää ohjaimen ja näkymän olemassaolosta (Steve 1992).

Malli ei kuitenkaan aina voi olla täysin passiivinen. Se voi myös muuttua ilman, että se tarvitsee ohjaimen tai näkymän käskyä. Otetaan esimerkiksi malli, joka muuttaa tilaansa satunnaisin väliajoina. Koska malli muuttaa itseään, täytyy sillä olla jokin yhteys näkymään, jotta se voi antaa tiedon muutoksestaan (Steve 1992). Datan kapseloinnin ja ohjelmakoodin uudelleen käytön kannalta ei ole kuitenkaan järkevää, että malli on suoraan yhteydessä näkymään ja ohjaimeen. Ohjaimen ja näkymän tulee siis olla riippuvaisia mallista, mutta ei toisinpäin. Näin mahdollistetaan myös se, että mallilla voi olla useita näkymiä ja ohjaimia (G. E. Krasner 1988, s. 4).

Yleensä mallin tila muuttuu ohjaimista tulleiden käskyjen kautta. Tämän muutoksen tulisi heijastua kaikkiin näkymiin, jotka ovat sidottuja malliin. Tällaisia tilanteita varten kehitet-

tiin riippuvuudet (*dependents*). Riippuvuuksilla tarkoitetaan listaa niistä ohjaimista ja näkymistä, jotka ovat sidottuja malliin. Mallilla tulee siis olla lista riippuvuuksista ohjaimiin ja näkymiin sekä myös kyky lisätä ja poistaa niitä. Malli ei siis tiedä mitään yksittäisistä riippuvuuksista, mutta pystyy kuitenkin lähettämään itsestään muutosviestejä (*change messages*) listassa oleville ohjaimille ja näkymille. Mallin tuottamat muutosviestit voivat olla minkä tyyppisiä tahansa, joten ohjaimet ja näkymät reagoivat niihin omalla määritellyllä tavallaan (Glenn E. Krasner & Stephen T.Pope Krasner 1988, s.2-3).

Mallille määritellään pääluokka *Model* ja tälle viitemuuttuja *dependents*, joka viittaa yhteen riippuvaan komponenttiin tai listaan riippuvista komponenteista. Kaikki uudet mallit tulee periä niiden pääluokasta, jotta saavutetaan sama toiminnallisuus kaikkiin mallikomponentteihin. Komponenttien tieto mallin muutoksista tukeutuu täysin mallin riippuvuusmekanismiin. Kun jokin komponentti luodaan, se rekisteröi itsensä malliin riippuvuudeksi ja samalla tavalla se myös poistaa itsensä (Steve 1992). Näkymät käyttävät riippuuvuusmekanismia päivittääkseen itsensä mallin muutoksien perusteella. Esimerkiksi mallin muuttuessa lähetätetään *changed*, jonka pohjalta jokainen riippuvuus saa *update* -viestin.Viestillä voi olla myös erilaisia parametrejä, joiden perusteella viestiä pystytään tarkentamaan. Esimerkiksi mallin, johon on liitetty useita näkymiä, ei välttämättä tarvitse lähettää kaikille näkymille viestiä muutoksestaan. Se voi välittää viestin mukana parametrina tiedon muutoksesta, jonka perusteella jokainen vastaanottaja voi päättää miten toimia (Steve 1992).

Alkuperäinen *update* -metodi on peritty *Object* -luokasta, eikä se tuolloin tee vielä yhtään mitään. Useimmilla näkymillä se on kuitenkin toteutettu näyttämään näkymä uudestaan kutsuttaessa. Tämä *changed/update* -mekanismi valittin toimimaan kommunikaatiokanavana mallien ja näkymien välille, koska se aiheuttaa vähiten rajoituksia ja esteitä (Steve 1992).

6.2.1 Näkymä (View)

Näkymän tehtävänä on huolehtia graafisesta puolesta MVC:ssä. Näkymä pyytää yleensä mallilta datan ja tämän pohjalta näyttää käyttäjälle käyttöliittymän sovellukseen. Toisinkuin malli, jota pystytään rajoittamattomasti yhdistelemään moniin näkymiin ja ohjaimiin, jokainen näkymä on liitetty yhteen ohjaimeen. Näkymä siis sisältää viitteen ohjaimeen ja ohjain

sisältää viitteen näkymään. Kuten ohjain, näkymä on myös rekisteröity mallin riippuvuuksiin. Kummatkin sisältävät siis myös viitteen siihen malliin, johon ne on rekisteröity (Steve 1992). Jokaisella näkymällä on tasan yksi malli ja yksi ohjain (G. E. Krasner 1988, s. 7).

Näkymä vastaa myös MVC-komponenttien sisäisestä kommunikaatiosta MVC-kolmikon luontivaiheessa. Näkymä rekisteröi itsensä riippuvuudeksi malliin, asettaa viitemuuttujansa viittamaan ohjaimeen ja välittää itsestään viestin ohjaimelle. Viestin avulla ohjain rekisteröi näkymän omaan viitemuuttujaansa. Näkymällä on myös vastuu poistaa viitteet sekä rekisteröinnit (Steve 1992).

Näkymä ei sisällä ainoastaan komponentteja datan näyttämiseen ruudulla, vaan se voi sisältää myös useita alanäkymiä (*subviews*) ja ylänäkymiä (*superviews*). Tästä muodostuu hierarkia, jossa ylänäkymä hoitaa aina jonkun suuremman kokonaisuuden, kuten esimerkiksi näytön pääikkunan. Alanäkymä taas huolehtii jostain pienemmästä yksityiskohdasta pääikkunassa. Näkymillä on myös viite erilliseen transformaatioluokkaan, joka hoitaa kuvan sovittamisen ja yhdistämisen alanäkymien ja ylänäkymien välillä. Jokaisella näkymällä tulee siis olla toteutus, jolla hoidetaan alanäkymien poistaminen sekä lisääminen. Samalla tulee määritellä ominaisuus, jolla sisäiset transformaatiot tuodaan transformaatioluokalle. Tämä helpottaa näkymän ja sen alanäkymien yhdistämistä (G. E. Krasner 1988, s. 8). burbeck havainnollistaa Smalltalkilla kirjoitetulla esimerkillä kuinka MVC-kolmikko luodaan. Esitetyssä esimerkissä on yksinkertaistettu versio MVC-kolmikon luonnista siten, että mukana on myös ylä- ja alanäkymien toteutus.

```
1  openListBrowserOn: aCollection label: labelString initialSelection: sel
2    "Create and schedule a Method List browser for
3    the methods in aCollection."
4    | topView aBrowser |
5    aBrowser ← MethodListBrowser new on: aCollection.
6    topView ← BrowserView new.
7    topView model: aBrowser; controller: StandardSystemController new;
8                label: labelString asString; minimumSize: 300@100.
9  topView addSubView:
10   (SelectionInListView on: aBrowser printItems: false oneItem: false
11   aspect: #methodName change: #methodName: list: #methodList
12   menu: #methodMenu initialSelection: #methodName)
```

```
13    in: (0@0 extent: 1.0@0.25) borderWidth: 1.
14  topView addSubView:
15    (CodeView on: aBrowser aspect: #text change: #acceptText:from:
16    menu: #textMenu initialSelection: sel)
17    in: (0@0.25 extent: 1@0.75) borderWidth: 1.
18    topView controller open
```

Seuraavaksi käydään rivi kerrallaan läpi mitä yllä esitetyssä ohjelmakoodissa tapahtuu. Mallin luonnin jälkeen [5] luodaan viite uudelle *BrowserView* -luokan instanssille [6]. *BrowserView* on peritty *StandardSystemView* -luokasta. Seuraavaksi määritellään malli ja ohjain sekä muuttujat näkymän otsikolle ja koolle [7]. Jos ohjainta ei määritellä erikseen, käytetään näkymän *defaultController* metodia. Riveillä [7-11] luodaan alanäkymä *SelectionInListView* ja riveillä [12-15] luodaan toinen alanäkymä *CodeView*. Lopuksi [16] avataan ohjain, joka käynnistää ikkunoiden piirtämisprosessin.

Näkymät saattavat tarvita myös oman protokollan itsensä näyttämiseen. Kun malli ilmoittaa muutoksestaan, *update* -metodi näkymässä kutsuu *display*, joka puolestaan kutsuu *displayBorder*, *displayView* ja *displaySubviews*. Jos näkymä tarvitsee erityistä käyttäytymistä itsensä näyttämiseen, se toteutetaan edellämainituissa metodeissa. Muuten käytetään pääluokasta perittyjä ominaisuuksia (Steve 1992). Monet näkymät käyttävät myös erilaisia transformaatioinstansseja, joilla hallitaan esimerkiksi näkymän skaalausta ruudulla. Tähän ei kuitenkaan perehdytä sen enempää, koska ne menevät tutkimuksen rajojen ulkopuolelle.

6.2.2 Ohjain (Controller)

Ohjaimen tehtävänä on ottaa vastaan syötteitä sekä koordinoida malleja ja näkymiä saatujen syötteiden perusteella. Sen tulee myös kommunikoida muiden ohjaimien kanssa. Teknisesti ohjaimessa on kolme viitemuuttujaa: malli, näkymä ja sensori (sensor). Sensorin tehtävänä on toimia rajapintana syötelaitteiden sekä ohjaimen välillä. Sensori mallintaa syötelaitteiden käyttäytymistä ja muuttaa ne ohjaimen ymmärtämään muotoon.

Ohjaimien tulee käyttäytyä siten, että vain yksi ohjain ottaa vastaan syötteitä kerrallaan. Esimerkiksi näkymät pystyvät esittämään informaatiota rinnakkain monen näkymän kautta, mutta käyttäjän toimintoja tulkitsee aina vain yksi ohjain. Ohjain on siis määritelty käyt-

täytymään siten, että se osaa tietyn signaalin perusteella päättää tuleeko sen aktivoida itsensä vai ei. Ohjain sisältää toiminnallisuuden jonka perusteella se pystyy päättämään tuleeko hallinta pitää itsellä vai luovuttaa eteenpäin (G. E. Krasner 1988, s. 9). Ohjainten ylimmällä tasolla on *ControlManager*, joka kysyy jokaiselta päänäkymään liitetyltä ohjaimelta erikseen, haluaako tämä ottaa hallinnan. Jos ohjaimen näkymä sisältää kursorin, vastaa ohjain kutsuun myönteisesti, jolloin kyseinen ohjain saa hallinnan. Hallitsevan ohjaimen näkymä kysyy seuraavaksi mahdollisten alanäkymien ohjaimilta samalla tavalla haluaako jokin ohjaimista hallinnan itselleen. Jos myönteisesti vastaava ohjain löytyy, ottaa se uuden hallinnan. Tätä prosessia jatkamalla löydetään matalimman tason näkymä ja sen ohjain ottaa lopullisen hallinnan. Ohjain pitää hallinnan itsellään niin kauan kunnes kursoria liikutetaan näkymän rajoista ulos. Ainoastaan se jonka kohdalla kursori on, vastaa kutsuun ja tuolloin ottaa hallinan. Näkymillä on oikeus kysyä alanäkymiensä ohjaimia. Ohjaimien tehtävänä on kysyä omalta näkymältään onko kursori niiden päällä.

Krasner määrittelee seuraavat metodit, joiden avulla ohjaimet viestivät (G. E. Krasner 1988, s. 9):

isControlWanted - Tuleeko ohjaimen ottaa hallinta.
isControlActive - Onko ohjain aktiivinen.
controlToNextLevel - Luovutetaan hallinta seuraavalle ohjaimelle.
viewHasCursor - Onko ohjaimen näkymässä hiiren kursori.
controlInitialize - Kun ohjain on saanut hallinnan, alustetaan se.
controlLoop - Lähettää *controlActivity* -viesteja niin kauan, kuin ohjaimella on hallinta.
controlTerminate - Lopettaa ohjaimen hallinnan.

Kun ohjain saa hallinnan itselleen, kutsuu se *startUp* -metodia, joka puolestaan kutsuu seuraavia metodeja: *controlInitialize*, *controlLoop* ja *controlTerminate*. Metodit voidaan ylikirjoittaa, jolloin saavutetaan jokin haluttu ominaisuus kyseisessä vaiheessa. Esimerkiksi *controlInitialize* ja *controlTerminate* määräävät mitä tehdään, kun ohjain saa hallinnan tai luovuttaa sen eteenpäin. Ohjaimen hallinnan aikana kutsutaan *controlLoop* -metodia, joka taas kutsuu *controlActivity* -metodia niin kauan kuin ohjaimella on hallinta. Metodi *controlActivity* määrää ohjaimen toiminnan hallinnan aikana (G. E. Krasner 1988, s. 9).

6.2.3 Esimerkkiohjelma

Seuraavaksi esitellään Dortmundin yliopistossa kirjoitettu yksinkertainen esimerkkiohjelma Smalltalkilla siitä miten MVC:n toteutus tuodaan sovellukseen käytännössä. Ohjelmakoodi löytyy myös Krasnerin artikkelista (G. E. Krasner 1988, s. 20). Ohjelmassa toteutetaan yksinkertainen laskuri-ohjelma, joka käyttää MVC-arkkitehtuuria toteutuksessaan. Ohjelmassa esitellään mallina *Counter* -luokka ja näkymänä *CounterView* -luokka. *Counter* perii mallin ominaisuudet ja toimii ohjelmassa yksinkertaisen kokonaisluku-muuttujan ylläpitäjänä. *CounterView* perii näkymän ominaisuudet ja esittää mallin arvon ruudulla. Ohjaimena toimii *CounterController* -luokka, joka perii ohjaimen käyttäytymisen. Ohjain tarjoaa sovellukselle painikkeet, joista voidaan vähentää tai lisätä laskurin arvoa.

Määritellään ensiksi *Counter* -luokka, joka peritään *Model* -luokasta.

```
1  Model subclass: #Counter
2     instanceVariableNames: 'value'
3     classVariableNames: ''
4     poolDictionaries: ''
5     category: 'Demo-Counter'
```

Seuraavaksi määritellään *Counter*-luokalle metodeita, jotka määrävät laskuriarvon alustamisen sekä muokkaamisen.

```
1  Counter methods For: 'Initialize-release'
2  Initialize
3     "Aseta alkuarvoksi 0"
4     self value: 0
5  Counter methodsFor: 'accessing'
6  value
7     "Palauta mallin arvo"
8     ↑value
9  value: aNumber
10    "Aseta mallin arvo"
11    value <- aNumber.
12    self changed "to update displayed value"
13 Counter methodsFor: 'operations'
14 decrement
```

```
15      "Vähennä mallin arvoa yhdellä."
16      self value: value -1
17  Increment
18      "Lisää mallin arvoa yhdellä."
19      self value: value + 1
```

Lisätään luokkaan metodi, jolla itse luokasta saadaan muodostettua instanssi.

```
1  Counter class methodsFor: 'instance creation'
2  new
3      "Palauta uusi instanssi luokasta"
4      ↑super new initialize
```

Seuraavaksi määritellään ohjain (*CounterController*), joka peritään *Controller* luokasta. Luodaan myös ohjaimelle metodit, joiden avulla ohjataan mallia sekä näkymää. Metodeissa toteutetaan valikko, joka tarjoaa mahdollisuuden joko vähentää tai lisätä laskurin arvoa. Kaikki *CounterController* -luokassa käytetyt määrittelemättömät muuttujat peritään yliluokasta.

```
1  Mouse MenuController subclass: #CounterControIler
2      instanceVariableNames: ' '
3      classVariableNames: ' '
4      poolDictionaries: ' '
5      category: 'Demo-Counter'
6  CounterController methodsFor: 'initialize-release'
7  initialize
8      "Alusta valikko, jossa on mahdollisuus vähentää tai
9         lisätä mallin arvoa"
10     super initialize.
11     Self yellowButtonMenu: (PopUpMenu labels:
12                             'Increment\Decrement' withCRs)
13     yellowButtonMessages: #(increment decrement)
14 CounterController methodsFor: 'menu messages'
15 decrement
16     "Vähennä mallin arvoa yhdellä."
17     self model decrement
18 increment
19     "Lisää mallin arvoa yhdellä"
20     self model increment
```

```
21 CounterController methodsFor: 'control defaults'
22 isControlActlve
23     "Ota hallinta kun sinistä nappia ei paineta"
24     ↑super isControlActive & sensor blueButtonPressed not
```

Määrätään näkymä (*CounterView*), joka peritään *View* -yliluokasta. Määrätään myös näkymälle metodit, joiden avulla näytetään mallin tila ruudulla.

```
1  View subclass: #Counterview
2     instanceVariableNames: ''
3     classVariableNames: ''
4     poolDictionaries: ''
5     category: 'Demo-Counter'
6
7  CounterView methodsFor: 'displaying'
8  displayView
9     "Näytä mallin arvo näkymässä"
10    | box pos displayText |
11    box ← self insetDisplayBox.
12    "Asettele teksti näkymään. Asettelu ei
13     ole tutkielman kannalta oleellista."
14    pos ← box origin + (4 @ (box extent y / 3)).
15    displayText ← ('value:', self model value printString)
16                  asDisplayText.
17    displayText displayAt: pos
```

Määritellään *update* -metodi, jotta näkymä pystyy päivittämään itsensä. Metodia kutsutaan yleensä mallin tilan muuttuessa.

```
1  CounterView methodsFor: 'updating'
2  update: aParameter
3    "Yksinkertaisesti päivitä näyttö uudestaan"
4    self display
```

Luodaan myös metodi, joka palauttaa näkymään liitetyn ohjaimen.

```
1  CounterView methodsFor: 'controller access'
2  defaultControllerClass
3     "Palauta näkymään rekisteröity ohjain"
```

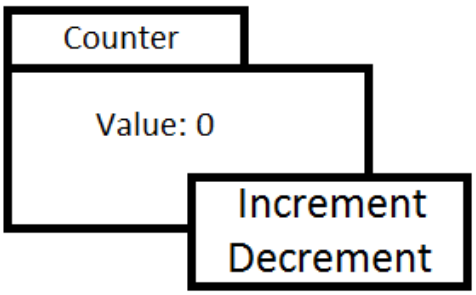
```
4     ↑CounterController
```

Lopuksi tarvitaan metodi, joka luo uuden näkymän sekä rekisteröi mallin ja ohjaimen itseensä. Näkymä näyttää ruudulta samalta kuin kuvassa 2.

```
1  CounterView class methodsFor: 'instance creation'
2  open
3     "Avaa näkymän uudelle laskurisovellukselle. Tässä
4     metodissa nähdään kuinka näkymä huolehtii mallin
5     rekisteröinnistä sekä nähdään kuinka näkymiä voi
6     olla useita sisäkkäin."
7     | aCounterView topView |
8     "Luo laskurinäkymälle uusi näkymä, joka näyttää
9     laskurin arvon"
10    aCounterView ← CounterView new
11    "Asetetaan malliksi Counter -luokan instanssi"
12    model: Counter new.
13    aCounterView borderWidth: 2.
14    aCounterView insideColor: Form white.
15    "Asetetaan ylimmäksi näkymäksi StandardSystemView
16    -luokan instanssi, joka vastaa perinteistä
17 ikkunointimallia"
18    topView ← StandardSystemView new
19       label: 'Counter'.
20    topView minimumSize: 80@40.
21    "Lisätään edellä luotu laskurinäkymä ylinäkymän
22    alanäkymäksi"
23    topView addSubView: aCounterView.
24    "Käynnistetään ohjain"
25    topView controller open
```

6.3 Alkuperäisen MVC:n vaatimukset

Pyramidin sovellusarkkitehtuuri on toteutettu käyttäen pohjana MVC-arkkitehtuuria. Tämä ei kuitenkaan tarkoita sitä, että Pyramid toteuttaisi MVC:n teknisesti sellaisena kuin esimerkiksi Krasner (G. E. Krasner 1988) ja Reenskaug (Reenskaug 2003) määrittelee. Tärkeimpä-



Kuvio 8. Kuva CounterView -näkymästä (G. E. Krasner 1988)

nä vaatimuksena MVC:n toteutukselle on se, että näkymä ja ohjain luovat rajapinnan, jonka kautta käyttäjä keskustelee mallin kanssa. Malli ei saa olla suorassa yhteydessä käyttäjään (Reenskaug 2003, s. 10). Mallin tulee olla riippumaton näkymästä ja ohjaimesta. Sen tulee myös hallita sovelluksen tilaa sekä pystyä antamaan informaatiota sovelluksen tilasta (Steve 1992). Näkymän on keskusteltava mallin kanssa sekä hoitaa mallilta saadun datan graafinen näyttäminen (Glenn E. Krasner & Stephen T.Popez Krasner 1979, s.1). Ohjain puolestaan ottaa vastaan syötteitä ja lähettää viestejä tämän perusteella näkymälle ja mallille (Steve 1992).

Vaikka MVC:tä ei ole tarkoitettu alunperin web-sovelluksiin, voivat ne hyötyä MVC:n arkkitehtuurista. Suurin ongelma MVC:n käyttämisessä web-sovelluskehyksissä on palvelimen (palvelin) ja asiakkaan (client) välinen ositus. Näkymä näytetään aina asiakkaan selaimessa sen omalla päätelaitteella. Malli ja Ohjain taas voivat olla ositettu teoriassa miten vain asiakkaan ja palvelimen välillä. Web-sovelluksissa kehittäjä pakotetaan osioimaan sovellus. MVC:n tulee olla riippumaton osioinnista. Osioinnin ei tule määrittää sovelluksen arkkitehtuuria (Avraham 2001). Osioinnin yhteydessä tulee myös mahdollistaa kaksisuuntainen kommunikaatio palvelimen ja selaimen välille. Reenskaugin mukaan Mallin tulee pystyä lähettämään itsestään muutosviestejä siihen kytketyille komponenteille. Tässä tapauksessa siis web-sovelluskehyksissä toteutettun Mallin tulee lähettää viestejä suoraan Näkymille ja ohjaimille (Reenskaug 2003).

7 Sovelluskehykset

Olio-ohjelmoinnin tärkeimpänä etuna on ajateltu olioiden uudelleenkäyttämistä eri järjestelmien kesken. On kuitenkin huomattu, että oliot ovat usein hyvin pieniä ja suunniteltu erityisesti yhden järjestelmän näkökulmasta. Tästä syystä niiden uudelleenkirjoitus on nopeampaa kuin olemassaolevien olioiden omaksuminen. Olioiden uudelleenkäytettävyyden on todettu parhaiten toimivan sovelluskehysten tuomien abstraktioiden kautta. Sovelluskehys on geneerinen rakennuspohja, jota laajennetaan luomaan vaatimuksien mukainen sovellus (Sommerville 2010, s. 431). Schmidt (D. C. Schmidt 1997) määrittelee sovelluskehyksen seuraavasti: "Sovelluskehys on kokoelma ohjelmisto-artefakteja kuten luokkia, olioita ja erilaisia komponentteja, joitka yhdessä tarjoavat uudelleenkäytettävän arkkitehtuurin samaan perheeseen kuuluvien ohjelmistojen toteuttamiseksi".

Sovelluskehyksen tarjoavat geneerisiä työkaluja sekä ominaisuuksia, joita voidaan käyttää samantyyppisissä sovelluksissa. Esimerikiksi käyttöliittymälle tarkoitettu sovelluskehys sisältää työkaluja erilaisten näkymien toteuttamiseen. Ohjelmistokehittäjän vastuulle jätetään sovelluskehyksen tarjoamien ominaisuuksien laajentaminen sovelluksen vaatimusten mukaiseksi. Sovelluskehys suunnitellaan siten, että se toimii sovelluksen runkona ja sen arkkitehtuuri toteutaan luokkien sekä olioiden kommunikaation pohjalta. Luokkia käytetään usein suoraan sellaisenaan tai niitä laajennetaan käyttämällä erilaisia ohjelmointikielen ominaisuuksia kuten esimerikiksi perintää. Sovelluskehykset rakennetaan lähes aina olio-ohjelmointia käyttäen ja ne ovat lähes aina riippuvaisia kielestä. Sovelluskehyksiä on tarjolla lähes jokaiselle yleisimmälle olio-ohjelmointiparadigman toteuttavalle ohjelmointikielelle. Jopa sovelluskehykset voivat sisältää useita pienempiä sovelluskehyksiä, jossa jokainen on suunniteltu toteuttamaan jonkun tietyn osan sovelluksesta. Sovelluskehystä voidaan käyttää luomaan koko sovellus tai vain tiettyjä osia sovelluksesta kuten esimerkíksi graafisen käyttöliittymän (Sommerville 2010, s. 431).

Schmidt (D. C. Schmidt 1997) jakaa sovelluskehyksien arkkitehtuurit kolmeen luokkaan:

Infrastruktuuri-sovelluskehykset (System infrastructure frameworks) tarjoavat työkaluja hallitsemaan järjestelmätason sovelluksia sekä niiden kommuni-

kaatiota keskenään. Tällaisia ovat esimerkiksi erilaiset kääntäjät ja erilaiset järjestelmätason käyttöliittymät.

Integraatio-sovelluskehykset (Middleware integration frameworks) sisältää kokoelman standardeja sekä luokkia, jotka auttavat komponenttejen väliseen kommunikaatioon sekä tiedonsiirtoon. Tällaisia ovat esimerkiksi Microsoftin .NET ja EJB (Enterprise Java Beans).

Liiketoiminta-sovelluskehykset (Enterprise application frameworks) keskittyvät tiettyjen sovelluksien osa-alueisiin, kuten esimeriksi telekommunikaation tai laskutusjärjestelmien toteutukseen. Ne ovat erityisesti suunnattuja loppukäyttäjälle.

Web-sovelluskehykset ovat nykyaikana yksi tärkeimpiä sovelluskehyksien alalajeja. Web-sovelluskehyksiä, jotka tarjoavat työkaluja dynaamisten web-sovellusten toteuttamiseen, on tarjolla paljon. Yleisesti näiden sovelluskehysten arkkitehtuurina on käytetty MVC-arkkitehtuuria. Tämän takia monet web-sovelluskehykset mielletään MVC-sovelluskehyksiksi, vaikka MVC esiteltiinkin 80-luvulla eri tarkoitukseen. Sovelluskehykset voivat koostua useista arkkitehtuurimalleista. Esimerkiksi MVC-sovelluskehys koostuu Tarkkailu-mallista (Observer Pattern), Strategia-mallista (Strategy Pattern), Yhdistelmä-mallista sekä monista muista, jotka esitellään (Design Patterns: Elements of Reusable Object-Oriented Software) -kirjassa (Gamma 1995). Vaikka jokainen web-sovelluskehys on totetettu hieman erilaisella tavalla, Sommerville esittelee viisi perusominaisuutta, jotka yleensä löytyvät web-sovelluskehyksistä:

Tietoturvan tukemiseksi sovelluskehykset tarjoavat luokkia auttamaan käyttäjänhallinnan toteutuksessa, siten että sovellukseen voidaan luoda erilaisia joituksia käyttäjille.

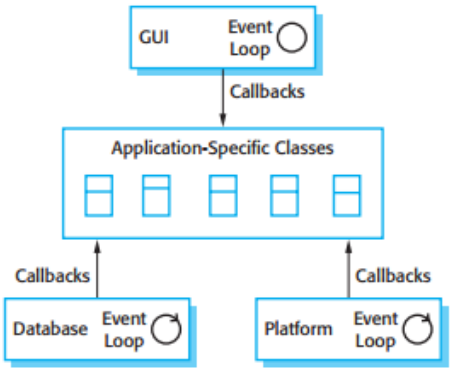
Dynaamisten web-sivujen luontiin tarjotaan työkaluja luoda Templateja (template), joiden pohjalta voidaan data näyttää eritavoilla selaimelle.

Tietokantatuki auttaa kommunikoinnissa tietokannan kanssa. Web-ovelluskehykset eivät yleensä sisällä tietokantaa, mutta ne on usein rakennettu tietyn tietokantatyypin pohjalta. Niillä on erilaisia luokkia, joilla voidaan abstrahoida tietokannan käyttö.

Session-hallintaan on useita työkaluja, joilla sessioita voidaan hallita sekä luoda.

Interaktiivisuus	on monien web-sovelluskehyksien etuna, jolloin voidaan toteuttaa entistä interaktiivisempia sovelluksia käyttäjille (esimerkiksi AJAX-teknologia).

Sovelluskehystä laajennettaessa ei itse sovelluskehyksen koodia muuteta, vaan sovellukseen lisätään luokkia, jotka perivät sovelluskehyksen määrittelemiä ominaisuuksia luokkatasolla. Lisäksi voidaan määritellä takaisinkutsuja (callbacks). Takaisinkutsuja voidaan viedä eteenpäin toiselle ohjelmakoodille kutsuttavaksi. Takaisinkutsuja ajetaan vastauksena ennaltamäärätyille operaatiolle, jotka sovelluskehys tunnistaa. Sovelluskehykseen luodut toiminnot ovat vastuussa sovelluksen kokonaisuudesta, siten että ne kutsuvat sovelluskehyksen käyttäjän määrittelemiä toiminnallisuuksia. Sovelluskehykseen toteutetut oliot ovat vastuussa erilaisten tapahtumien hallinnoinnista ja välittävät näitä tapahtumia edelleen käyttäjän tekemälle sovelluskoodille. Tällaisia toimintoja kutsutaan koukku-metodi (hook method). Niiden tehtävä voi olla esimerkiksi toteuttaa jokin tietty toiminallisuus aina silloin, kun tietokantaan tehdään kirjoitusoperaatioita (Sommerville 2010, s. 433).



Kuvio 9. Esimerkki sovelluksen toimintojen suhteesta sovelluskehyksen takaisinkutsuihin. Sovelluskehyksellä voi olla esimerkiksi metodi, joka hallitsee hiiren tapahtumia. Metodi kutsuu koukku-metodia, joka tulee konfiguroida kutsumaan oikeita metodeja sovelluksesta, jossa hiiren tapahtumat käsitellään. (Sommerville 2010, s.434)

Sovelluskehyksien avulla rakennetut sovellukset voivat toimia uudelleenkäytettävinä pohjina, jollekin tietylle liiketoiminta-alueella tai sovellus-perheelle. Koska näissä sovelluksissa on käytetty jotain tiettyä sovelluskehystä, voidaan sovelluksesta luoda monia instansse-

ja, jotka palvelevat jotain tiettyjä alueita liiketoiminnassa. Koska kaikki sovellukset jakavat saman sovelluskehyksen tavan toteuttaa asioita, on uusien sovellusperheen jäsenten luonti usein hyvin suoraviivaista. Tämä mahdollistetaan ylikirjoittamalla pohjasovellukseen luokkia ja metodeja, joita voidaan muokata ylikirjoittamalla se taas uudessa sovelluksessa. Osa näistä luokista voidaan olla ylimmällä tasolla peritty suoraan sovelluskehyksestä. Sovelluskehykset ovat kuitenkin yleensä hyvin geneerisiä kuin eri liiketoiminta-alueiden sovellukset. Web-pohjaisia sovelluskehyksiä voidaan esimerkiksi käyttää toteuttamaan asiakaspalvelulle tarkoitettuja web-sovelluksia.

Sovelluskehykset ovat tehokkaita sovelluksien uudelleenkäytössä, mutta ne ovat myös kalliita tuoda sovelluskehitykseen mukaan. Sovelluskehykset ovat usein hyvin monimutkaisia ja niiden omaksuninen voi kestää kuukausia. Virheiden etsintä on sovelluskehyksien pohjalta tehdyissä sovelluksissa hastaavaa, koska ohjelmoijan tulee tietää ymmärtää millä tavalla sovelluskehys toimii. Lisäksi voi olla vaikeaa valita oikea sovelluskehys monien joukosta.

8 Python web-sovelluskehykset

Sovelluskehykset ovat suosittuja, koska ne tarjoavat uudelleenkäytettäviä ratkaisuja erilaisiin ongelmiin sovelluskehityksessä. Toimialueesta riippumatta sovelluskehyksiä tulisi käyttää hyväksi kirjottaessa monimutkaisia sovelluksia. Sovelluskehys tuo sovellukseen tason, jossa sovelluksen osat on abstrahoitu erilaisilla luokilla sekä rajapinnoilla, joita voidaan käyttää uudelleen sovelluksen eri osissa. Sovelluskehys ei ole vain kokoelma rajapintoja ja kirjastoja(Sheikh I. Ahamed ja Pezewski 2008). Tärkein ero sovelluskehyksen ja kirjaston välillä on se, että kirjaston ohjelmakoodi kutsustaan aina kehittäjän toimestaa. Sovelluskehyksessä taas kehittäjän ohjelmakoodia kutsutaan aina sovelluskehyksen toimesta (Consulting 2005).

Sovelluskehys ei myöskään generoi koodia. Se käyttää erilaisia komponentteja ja kirjastoja luodakseen infrastruktuurin, jonka päälle voidaan rakentaa sovelluksia sovelluskehyksen ehdoilla. Sovelluskehyksen käyttäminen myös rajoittaa sovelluksen rakennetta ja pakottaa sovelluksen toteuttamaan asioita tietyin ehdoin. Rajoitusten ansiosta sovelluskehittäjä voi keskittyä toimialueeseen liittyviin ongelmiin välittämättä koko sovelluksen yksityiskohtaisesta toteutuksesta (Sheikh I. Ahamed ja Pezewski 2008).

Web-sovelluskehykset ovat sovelluskehyksiä, jotka tarjoavat ratkaisuja helpottamaan websovellusten toteuttamista. Web-sovelluskehyksissä käyttöliittymä näytetään käyttäjille selaimen välityksellä. Sovellus ajetaan joko palvelimella tai suoraan käyttäjän selaimessa. Sovellus määrittää käyttöliittymän sivujen järjestyksen, sisällön sekä mahdollisten toimintojen esittämisen käyttäjälle, jonka kautta käyttäjä voi vaikuttaa palvelimella sijaitsevaan sovellukseen (Kourie 2008).

Yleisimmät teknologiat mitä web-sovelluskehykset tarjoavat ovat rajapinta tietokannalle, template-moottori sekä mahdollisuus käsitellä http-pyyntöjä ohjelmakoodissa. Tietokantarajapinnalla tuodaan sovelluskehitykseen taso, jonka avulla helpotetaan kommunikointia tietokannan kanssa. Yleisimmin käytetty ohjelmointitekniikka tähän on ORM (Object-Relational-Mapping), jolla muunnetaan dataa tietokannan ja ohjelmakoodin välillä. Ohjelmoijalle tämä näkyy ns. virtuaalisena olio-tietokantana, jonka avulla voidaan lukea sekä muokata tietokantaa kutsuilla ohjelmakoodista. Tällöin suoria kyselyitä tietokantaan ei tarvita (Shahram

Ghandeharizadeh 2014). Seuraavassa esimerkissä esitellään miten Djangon ORM:ia käytetään.

```
from django.db import models

class Blog(models.Model):
    name = models.CharField(max_length=100)
  tagline = models.TextField()
```

```
from blog.models import Blog

b = Blog(name='Beatles Blog', tagline='All the latest Beatles news.')
b.save()
```

Esimerkissä luodaan luokka, jolla määritellään taulu SQL-tietokantaan. Model-luokasta periminen mahdollistaa luokasta luotujen instanssien tallentumisen tauluun. Lisäksi luokalle voidaan määrittää attribuutteja, jotka käyttävät Djangon tarjoamia kenttiä. Kentät vastaavat tietokannassa olevia data-tyyppejä. Lopullinen tallentuminen tapahtuu kutsumalla instanssin save() -metodia.

```
beatles_blog = Blog.objects.get(name="Beatles Blog")
```

Yllä oleva rivi hoitaa kyselyn nimen perusteella blog -nimisestä taulusta, luo sen kenttien perusteella instanssin Blog -luokasta ja tallentaa sen muuttujan arvoksi.

Template-moottori on teknologia HTML-sivujen tuottamiseen, jolla generoidaan dynaamisia HTML-sivuja yhdistämällä ohjelmalogiikkaa sekä HTML-kieltä. Alla on esitelty Jinja2-kieli, jota käytetään template-kielenä Flask-sovelluskehyksessä.

```
1 <title>{% block title %}{% endblock %}</title>
2 <ul>
3   {% for user in users %}
4     <li><a href="{{ user.url }}">{{ user.username }}</a></li>
5   {% endfor %}
6 </ul>
```

Yllä esitellyssä esimerkissä luodaan HTML-sivu, jossa tulostetaan otsikko sekä lista url-osoitteita sekä käyttäjänimiä jokaista käyttäjää kohden (Ronacher 2008).

Web-sovelluskehykset voidaan jakaa palvelin-pohjaisiin ja selain-pohjaisiin sovelluskehyksiin. Serveri-pohjaisissa web-sovelluskehyksissä sovelluksen tilaa hallitaan palvelimen puolella. Tuolloin käyttöliittymää perustuu tilaan, mikä on sillä hetkellä palvelimen puolella. Selain-pohjaisissa sovelluskehyksissä sisältö muuttuu selaimen sisällä käyttäjän päässä (Kourie 2008). Tässä tutkimuksessa käsiteltävät sovelluskehykset ovat Pythonilla toteutettuja palvelin-pohjaisia sovelluskehyksiä. Käsiteltävät kehykset ovat Django, Pyramid ja Flask. Django on käsiteltävistä sovelluskehyksistä kaikkein monoliittisin ja tarjoaa eniten ominaisuuksia valmiiksi asennettuna. Pyramidissa taas on vähemmän ominaisuuksia suoraan asennettuna, jonka kautta se pyrkii antamaan käyttäjälle enemmän valinnanvaraa erilaisten teknologioiden valitsemiseen. Flaski taas on mikro-sovelluskehys, joka tarjoaa kaikkein vähiten työkaluja web-kehitykseen näistä kolmesta. Sovelluskehysten koko kertoo myös niiden toteutuksesta: Django (1.7.4) 36 megabittiä, Pyramid (1.5.3) 5.6 megabittiä ja Flask (0.10.1) 1.2 megatavua. Sovelluskehysten koot tarkastettiin komentoriviltä käyttäen linuxin du -työkalua. Kaikkia kolmea sovelluskehystä voidaa laajentaa erilaisilla laajennoksilla.

8.1 Pyramid

Pyramid on Python-pohjainen web-sovelluskehys, jonka tehtävänä on helpottaa web-kehitystä tarjoamalla kehittäjälle valmiita työkaluja avuksi kehitykseen. Pyramid on suunniteltu siten, että kehittäjän ei tarvitse tietää suuria määriä erilaisia malleja ja tekniikoita pystyäkseen tuottamaan web-sovelluksia. Se ei myöskään pakota käyttämään kehityksessä mitään erityistä tekniikkaa, vaan pyrkii olemaan mahdollisimman yksinkertainen ja helposti laajennettavissa erilaisiin käyttötarkoituksiin. Laajentamisella tarkoitetaan erilaisten lisäosien liittämistä Py-

ramidiin. Yksinkertaisuuden ja mimimaalisuuden ansiosta se on myös nopeampi kuin monet muut Python-pohjaiset web-sovelluskehykset. Tämä johtuu Pyramidin poikkeuksellisen pienestä kutsupinosta ajamisen aikana (Consulting 2005).

Pyramid sai alkunsa Pylons-projektista syyskuussa vuonna 2005, jolloin jo yli 30 Python-sovelluskehystä kilpaili käyttäjistä. Ben Bangert ja James Gardner alkoivat yhdessä kehittää sovelluskehystä, josta tuli lopulta Pylons. Alunperin Pylons oli muokattu Myghty Python Templating Framework:n pohjalta tarjotakseen MVC-pohjaisen web-sovelluskehyksen. Myghty -sovelluskehystä ajettiin mod_pythonin päällä, mutta Pylonsin pyrkimys oli käyttää WSGI:tä hyödyntämään joustavaa komponenttipohjaista lähestymistapaa web-sovelluksissa (**pyramid_history**). Pylons projektin tarkoituksena ei ole keskittyä yhden yksittäisen web-sovelluskehyksen kehittämiseen vaan tarjota kokoelma erilaisia teknologioita (Project 2010a). Vuonna 2008 Pyramid tunnettiin nimellä repoze.bfg. Joulukuun alussa tapahtui ohjelmakoodin uudelleen nimeäminen ja ominaisuuksien lisääminen sekä poistaminen (Project 2010b).

Koska Pyramid pyrkii tarjoamaan vain välttämättömimmät työkalut web-sovelluksien kehitykseen, sen kehittäjät ovat päätyneet web-kehityksessä neljään yleisimpään ongelmaan ja tarjoavat niihin ratkaisun Pyramidissa:

URL Mapping - URL:ien liittäminen ohjelmakoodiin.

Template - Tuodaan sovelluksen näkymä selaimelle käyttäen template-kieltä, jolla määrätään näkymän rakenne. Kieli on usein HTML:än mukana tuotuja loogisia ilmaisuja, joiden perusteella template-engine rakentaa HTML-sivun selaimelle. Templatejen avulla pystytään erottamaan käyttöliittymä sovelluslogiikasta tehokkaasti.

Security - Perinteiset tietoturvaongelmat tulee olla ratkaistuna valmiiksi jo sovelluskehyksessä. Tämä ei kuitenkaan tarkoita sitä, että kehittäjä voisi täysin unohtaa tietoturvan merkityksen.

Static Assets - Staattisien resurssejen jakaminen niille tarkoitettuihin paikkoihin tiedostorakenteessa.

Yllä määriteltyjen neljän ongelman lisäksi Pyramid tarjoaa lisäosien kautta monia erilaisia työkaluja, joiden avulla pystytään laajentamaan sen ominaisuuksia (Consulting 2005). Tutkielman aiheen rajauksen vuoksi ei kuitenkaan käydä läpi yksityiskohtaisemmin Pyramidin

toteutusta ja siihen liitettävissä olevia lisäosia, vaan keskitytään tarkastelemaan MVC:n toteutusta Pyramidissa.

8.2 Django

Django on web-sovelluskehys, joka sai alkunsa kehitysryhmässä Kansaksen osavaltiossa Yhdysvalloissa 2003, kun web-kehittäjät Adrian Holovaty ja Simon Willison alkoivat käyttää Pythonia web-kehityksessä. The World Online -ryhmä (WO), joka oli vastuussa muutamaamasta paikallisesta uutissivustosta, menestyivät ympäristössä, jossa oli tiukat aikarajat. Journalistit vaativat ominaisuuksien ja kokonaisten sovelluksien valmistumista muutamassa päivässä ja joskus jopa tunneissa. Holovaty ja Willison kehittivät web-sovelluskehyksen, jonka avulla he pystyivät vastaamaan journalistisen ympäristön haasteisiin. Kesällä 2005 he saivat kehitettyä sovelluskehyksen siten, että se oli käytössä suurimmassa osassa World Onlinen sivustoja. Tuolloin mukaan kehitykseen tuli Jacob Kaplan-Moss. Kehittäjät päättivät julkaista heinäkuussa 2005 sovelluskehyksen nimellä Django jazz-kitaristi Django Reinhardtin mukaan (Adrian Holovaty 2009a).

Django tarjoaa samat välttämättömät työkalut kuin Pyramidissa. Se tarjoaa myös ylläpitäjille suunnatun työkalun, josta voidaan hallita sovellusta käyttöliittymätasolta. Lisäksi se tarjoaa lomake-työkalut, käyttäjätasoisen autentikaation sekä tietokanta-abstrahoinnin. Tietokanta-abstraktiolla tarkoitetaan sisäänrakennettua virtuaalista ympäristöä tuomaan yhteys tietokantaan olio-ohjelmoinnin kautta (Object-relational mapping)(Adrian Holovaty 2009b). Siinä missä Pyramid pyrkii tarjoamaan kehittäjille valinnanvaraa erilaisten komponenttejen suhteen, Django tarjoaa kokonaisvaltaisen ratkaisun sisältäen kaikki tarvittavat työkalut web-sovellusten rakentamiseen. Djangoon on tarjolla myös paljon erilaisia paketteja täydentämään sitä.

8.3 Tornado

Tornado on Python-pohjainen web-sovelluskehys , joka pystyy skaalautumaan kymmenille tuhansille yhtäaikaisille yhteyksille. Se käyttää asynkronista siirräntää (asynchronous non-blocking I/O) sekä mahdollistaa web-sokettejen käytön (Authors 2009b). Siirräntä tarkoittaa

tässä yhteydessä luku- ja kirjoitusoperaatioita levyllä. Suurella kuormalla toimivat synkroniset TCP/IP-pohjaiset sovellukset voivat jättää useita syklejä prosessorin laskennasta väliin, joka johtuu siirrännän hitaudesta. Nämä syklit voitaisiin käyttää sovelluksen koodin ajamiseen. Mitä enemmän pyyntöjä tulee samanaikaisesti, sitä enemmän prosessorin syklejä jää väliin ja järjestelmään tulee ylimääräistä kuormaa. Siirrännät paljon hitaampia kuin prosessorin laskentateho. Asynkronisessa siirrännässä pyyntöjä voidaan ottaa vastaan käsiteltäväksi, vaikka edellistä pyyntöä ei ole vielä suoritettu loppuun (Brecht 2006, s. 1). Tornado ei myöskään toteuta MVC-mallia, vaan tarjoaa vapaammat kädet kehittäjälle toteuttaa haluamansa arkkitehtuuri.

9 MVC & Pyramid

Pyramidin kehittäjien dokumenteissa esitellään Pyramid MVC-kehyksenä, mutta samalla myös kyseenalaistetaan tämä väite. Erityisesti mallin ja ohjaimen määritelmä puuttuu (Consulting 2005). Tässä osiossa käsitellään Pyramidia MVC:n näkökulmasta. Tarkastelua varten toteutetaan Pyramidilla vastaava laskuri-sovellus kuin Krasnerin julkaisussa käyttäen Pyramidin tarjoamia ominaisuuksia (G. E. Krasner 1988). Sovellus rajataan käyttämään SQL-tietokantaa mallin datan tallennukseen sekä *URL dispatch* -tekniikkaa (n.d.). Sovelluksen pohjalta tarkastellaan MVC:n kannalta kolmea oleellisinta tiedostoa: *views.py*, *models.py* ja *template.pt*. Näin tutkielman tarkastelu pystytään rajaamaan mahdollisimman pienelle alueelle, jolloin tutkielma on helpompi keskittää tarkastelemaan yksittäistä tekniikkaa.

Tässä osiossa tarkastellaan Pyramid-sovelluksen tiedostoja sekä niiden sisältöjä MVC-komponentteina. Erityisesti keskitytään ohjaimen ja näkymän toteukseen. Samalla tutkitaan voidaanko Pyramid havaintojen perusteella luokitella MVC:n toteuttavaksi sovelluskehykseksi. Koska Pyramidissa MVC:n määrittely on hyvin epävakaalla pohjalla, tulee tehdä selvästi mitä ominaisuuksia komponenteilta vaaditaan. Esimerkiksi *views.py* ja *template.pt* -tiedostot ovat tarkoitettu toimimaan yhdessä näkymänä, jolloin ohjaimen toteuttamat tehtävät sisällytettäisiin näkymään. MVC:tä tutkiessa täytyy sovelluksen komponentit kuitenkin jakaa kolmeen osaan. Tiedosto *views.py* sisältää paljon ohjaimelle yhteisiä piirteitä, joten se erottuu selvästi *template.pt* -tiedostosta. Tutkielmassa oletetaan, että *models.py* sisältää mallin, *views.py* ohjaimen ja *template.pt* näkymän.

9.1 Tiedostojen tarkastelu

```
1 # Tiedosto: models.py
2 class Counter(Base):
3     __tablename__ = 'counter'
4
5     # Asetetaan mallille atribuutit, jotka
6     # vastaavat mallin tilasta.
7     id = Column(Integer, primary_key=True)
8     name = Column(Unicode(255), unique=True)
```

```
9       value = Column(Integer)
10
11      # Määritellään luokalle konstruktori,
12      # joka saa parametreiksi nimen ja alkuarvon.
13      def __init__(self, name, value):
14          self.name = name
15          self.value = value
16
17      def increment(self):
18          self.value += 1
19
20      def decrement(self):
21          self.value -= 1
22
23  # Luodaan instanssi mallista ja rekisteröidään
24  # se sovellukseen.
25  def populate():
26      session = DBSession()
27      model = Counter(name=u'counter', value=0)
28      session.add(model)
```

Models.py -tiedosto sisältää malliin liittyvän ohjelmakoodin. Jokaiselle mallille luodaan aina oma luokkansa, jossa määritellään mallin ominaisuudet. Malli rekisteröidään *populate* -funktion kautta, jota kutsutaan Pyramidin toimesta. Ohjelmakoodista nähdään, että malli ei luo minkäänlaista riippuvuutta näkymään tai ohjaimeen. Se myös pitää huolen datan käsittelystä. Malli on siis Pyramidissa itsenäinen komponentti, joka huolehtii sovelluksen tilasta. Tämän perusteella malli toteutuu Pyramidissa MVC-arkkitehtuurin mukaisesti. *Views.py* -tiedostossa määritellään näkymän ohjelmakoodi. Pyramidissa on konkreettisesti määritelty ohjelmakooditasolla vain malli ja näkymä. Jokaista näkymää kohden on oma funktio, joka ottaa vastaan *request*-olion. Request-oliossa tuodaan sovellukselle kaikki tieto käyttäjästä ja sovelluksen viesteistä. Tämän perusteella tulkitaan request-oliossa tuotu data käyttäjän syötteiksi. Vaikka *views.py* nimetään Pyramidissa näkymäksi, on se toteutukseltaan hyvin lähellä ohjainta. Tästä syystä tarkastellaan funktion toteutusta mahdollisena ohjaimena. Tästä eteenpäin puhutaessa ohjaimesta Pyramidissa, tarkoitetaan sillä *views.py* -tiedoston sisältämää funktiota.

```
1  # Tiedosto: views.py
2  @view_config(route_name='counter_view',
3    renderer='templates/counter.pt')
4  def counter_view(request):
5      dbsession = DBSession()
6
7      # Rekisteröidään malli.
8      counter = dbsession.query(Counter).filter(
9    Counter.name==u'counter').first()
10     try:
11         request.params['minus'].
12         counter.decrement()
13     except KeyError:
14         pass
15
16     try:
17         request.params['plus']
18         counter.increment()
19     except KeyError:
20         pass
21
22     # Palautetaan laskurin arvo, joka tulkitaan
23     # ja näytetään template.pt -tiedostssa
24     return {'value': counter.value}
```

Määritellään funktiolle URL-osoite sekä liitetään siihen template-tiedosto (2). Tämän jälkeen rekisteröidään malli mukaan funktioon (8). Koska tarkastelemme funktiota ohjaimena, voimme tulkita templaten näkymäksi. Tällöin funktioon rekisteröidään malli sekä näkymä, jolloin rekisteröinnin puolesta se toteuttaa ohjaimelle tarkoitetut ominaisuudet MVC:ssä. Funktioon lisätään myös toiminto laskurin vähentämiselle. Mallin *value* -arvoa muutetaan, kun request-oliosta löytyy tietty parametri. Funktio palauttaa paluuarvona mallin arvon, joka tuodaan käsiteltäväksi templateen. Funktio siis ottaa vastaan syötteitä ja niiden perusteella lähettää viestejä mallille sekä näkymälle. Tämän perusteella todetaan, että se täyttää rajauksessa määrätyt ohjaimen ominaisuudet.

Templatessa yhdistetään HTML-merkkauskieli ja sovelluksen ohjelmakoodi. Tästä generoi-

daan HTML-sivu, joka näytetään selaimelle. Koska malli sekä ohjain on jo määritelty, täytyy selvittää täyttääkö template näkymälle määritellyt ominaisuudet.

```
1  <body>
2    <h1>${value}</h1>
3    <form action="./" method="get">
4      <button type="submit" name="plus" value="plus">
5    Increment </button>
6      <button type="submit" name="minus" value="minus">
7    Decrement </button>
8    </form>
9  </body>
```

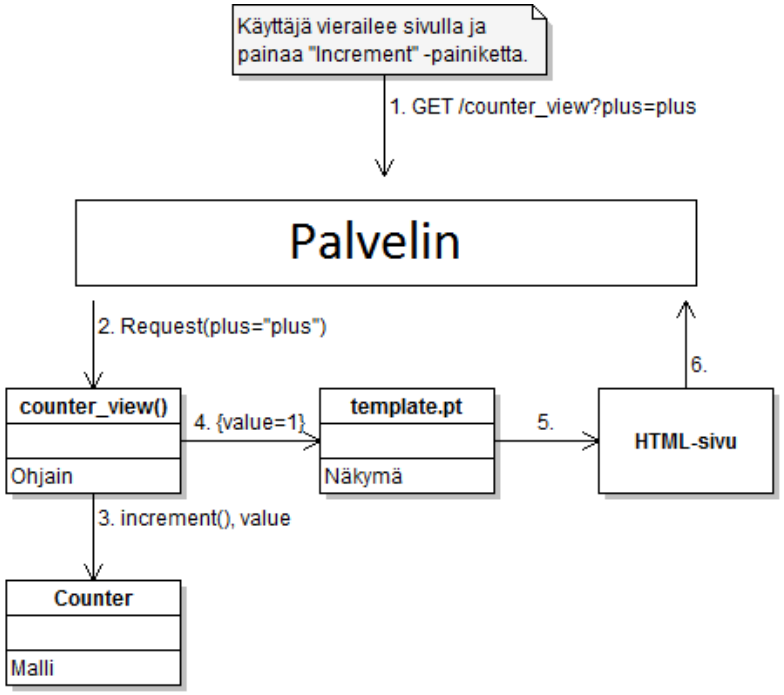
Templatessa luodaan lomake kahdelle painikkeelle, joista kumpikin lähettää lomakkeen eteenpäin *counter_view* -funktiolle. Lomakkeen tiedot tulevat funktiolle request-oliossa, joka sisältää tässä tapauksessa *plus*- tai *minus*-parametrin riippuen siitä kumpaa painiketta on painettu. Lomakkeen tiedot lähetetään samaan osoitteeseen (3), mistä sitä on alunperin kutsuttukin. Eron tuo kuitenkin request-oliossa tuodut parametrit. Otsikossa (2) tuodaan näkyviin laskurin sen hetkinen arvo, joka saadaan tietoon ohjaimelta.

Templatessa hoidetaan sovelluksen graafinen puoli, joten se vastaa ominaisuuksiltaan näkymää.
Ongelmaksi muodostuu kuitenkin näkymän ja mallin välinen kommunikointi. Näkymä ei ole yhteydessä malliin suoraan, vaan tarvitsee ohjaimen kautta tiedon mallin tilasta. Näkymä ei siis sellaisenaan toteuta sille asetettuja ominaisuuksia.

9.2 Sovelluksen toiminta

Alla olevassa kuvassa esitellään visuaalisesti miten laskurisovellus muodostaa HTML-sivun, kun käyttäjä painaa sivulla *Increment* -painiketta. Kuvan vaiheet toteutetaan numerojärjestyksessä alkaen ensimmäisestä.

1 Palvelimelta pyydetään HTTP-protokollan mukaisesti sivua *plus* -parametrilla.
2 Palvelin pyytää sovellukselta sivua. Parametri tuodaan sovellukselle *request*-oliossa.



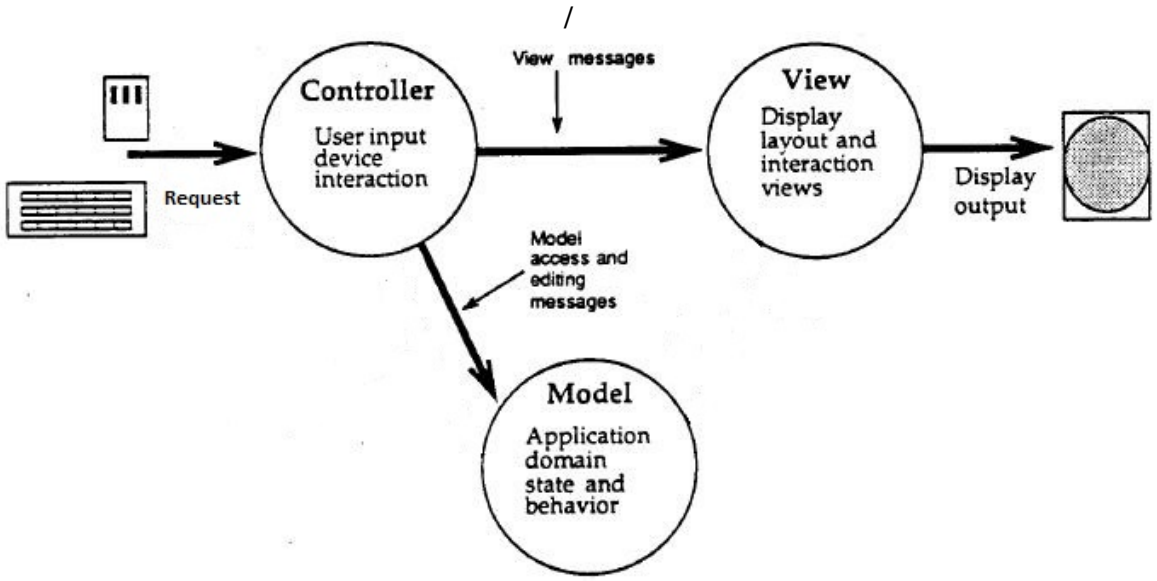
Kuvio 10. Laskurisovelluksen toiminta

```
3 Ohjain käskee mallia muuttamaan tilaansa ja pyytämään samalla tiedon
    muutoksen jälkeisestä arvosta.
4 Ohjain palauttaa mallin arvon, joka käsitellään templatessa.
5 Templatessa generoidaan HTML-sivu, joka tuodaan palvelimelle
    vastauksena.
```

9.3 Yhteenveto

Malli sekä ohjain toteutuvat Pyramidissa MVC:n mukaisesti yksittäisinä komponentteina, mutta kommunikaatio näiden välillä ei mene MVC:n mukaisesti. Malli on itsenäinen komponentti, jolla ei ole riippuvuutta näkymään tai ohjaimeen. Se myös huolehtii sovelluksen käsittelemästä datasta ja vastaa tarvittaviin pyyntöihin. Se ei kuitenkaan pysty kertomaan ohjaimelle ja näkymälle omista muutoksistaan, koska kaikki muutokset tulevat näkymälle asti vastauksena vasta HTTP-pyynnön mukana. Ohjain taas huolehtii request-oliossa tulevista syötteistä ja vaikuttaa malliin sekä näkymään. Näkymä toteuttaa sovelluksen graafisen näyttämisen selaimelle, mutta ei toteuta sille määrättyjä sääntöjä. MVC:ssä näkymän tarkoitus on kommunikoida suoraan mallin kanssa. Tämä ei kuitenkaan onnistu Pyramidissa, jossa

näkymä on yhteydessä vain ohjaimeen. Pyramidin MVC-toteutusta tarkastellessa tulee ottaa huomioon yksittäisten komponenttien toteutus sekä niiden välinen yhteistyö. Yksittäiset komponentit toteutuvat Pyramidissa MVC-arkkitehtuurin mukaisesti, mutta niiden välinen yhteistyö ei toteudu. Pyramidin MVC-komponenttien kommunikointi voidaan esittää käyttäen pohjana muokaten Krasnerin kommunikaatiomallia (G. E. Krasner 1988). Alla esitellyssä kuvassa havainnollisestetaan, kuinka mallin ja näkymän kommunikaatio puuttuu täysin ja kaikki data tuodaan ohjaimen kautta. Lisäksi Malli ei pysty kommunikoimaan muutoksista suoraan ohjaimelle ja näkymälle vaan tarvitsee aina erillisen HTTP-pyynnön.



Kuvio 11. Pyramidin kommunikointi MVC-komponenttien kesken. Kuva on muokattu Krasnerin esittelemästä kommunikointimallista (G. E. Krasner 1988)

Yllä esitetyssä kommunikaatiomallissa ohjain saa vastaan request-olion, jossa tuodaan kaikki tarvittava tieto käyttäjästä. Tämän perusteella ohjain ohjaa mallia sekä muuttujaa. Samalla se pyytää mallilta tietoa sovelluksen tilasta ja välittää tiedon näkymälle. Näkymä taas välittää ohjaimen tuoman datan käyttäjälle graafisena.

Alunperin Pyramidin dokumentaatiossa kyseenalaistettiin mallin sekä ohjaimen toteutus (Consulting 2005). Tutkimuksen pohjalta voidaan kuitenkin todeta, että ongelmaksi ei muodostu yksittäisien komponenttien toteutus, vaan komponenttien välinen kommunikointi. Erityisesti näkymän ja mallin yhteistyö jää kokonaan puuttumaan, jolloin saadaan ristiriita MVC:n

alkuperäisen määritelmän kanssa (Glenn E. Krasner & Stephen T.Popez Krasner 1979, s. 1). Tämä johtuu siitä, että Pyramidissa *views.py* -tiedoston sisältämä ohjelmakoodi on nimensä mukaisesti tarkoitettu näkymäksi ja *template.pt* -tiedosto katsotaan osaksi samaa komponenttia. Tutkimuksen tuloksien perusteella voidaan kuitenkin todeta, että *views.py* -tiedoston näkymä-funktio toteuttaa kaikki ohjaimelle määritellyt ominaisuudet. Template puolestaan hoitaa sovelluksen graafisen puolen, joten sen ominaisuudet ovat mahdollisimman lähellä näkymää. Template ei kuitenkaan riitä toteuttamaan näkymän ominaisuuksia, koska se on täysin riippuvainen ohjaimesta. Lisäksi Malli ei pysty kommunikoimaan suoraan näkymälle ja ohjaimelle ilman, että ohjain/näkymä pyytää Mallilta nykyistä tilaa.

10 MVC & Django

Django luokitellaan MVC-arkkitehtuurin toteuttavaksi web-sovelluskehykseksi, mutta sen toteutus kuitenkin poikkeaa alkuperäisestä MVC-toteutuksesta samalla tavalla kuin Pyramidissa. Djangosta löytyvät samat tiedostot ja luokat MVC:n toteuttamiselle, kuin Pyramidista. Siinä on vastaavasti *models.py*, *views.py* sekä *template.html*. Näistä *models.py* sisältää mallit, *views.py* ohjaimet ja *template.html* näkymät selaimelle. Tässä tutkielmassa käytetään Djangon versiota 1.11. Seuraavaksi esitellään lyhesti Mallin, Näkymän sekä Ohjaimen toteutus Djangossa, jotka on myös esitelty Djangon dokumentaatiossa https://docs.djangoproject.com/en/1.11/top based-views/intro/.

Mallin toteutetaan luomalla luokka, joka peritään *models.Model* luokasta. Alla esitetyssä esimerkissä luodaan malli, jolla voidaan tallentaa henkilön etu- ja sukunimi.

```
1 from django.db import models
2
3 class Person(models.Model):
4     first_name = models.CharField(max_length=30)
5     last_name = models.CharField(max_length=30)
```

Ohjaimen toiminta toteutuu Djangossa näkymien kautta vastaavasti kuin Pyramidissakin, koska näkymät käsittelevät pyyntöjä ja syötteitä, jotka tulevat selaimelta. Tästä syystä myös ohjaimen määritelmä toteutuu sovelluksehyksessä määriteltyjen näkymien kautta https://docs.djangoproj based-views/intro/.

```
1 from django.http import HttpResponse
2
3 def my_view(request):
4     if request.method == 'GET':
5         # <view logic>
6         return HttpResponse('result')
```

Näkymät toteutetaan vastaavasti selaimelle käyttäen templateja. Djangossa on sisäänrakennettuna oma template-kieli, mutta se on hyvin samankaltainen esimerkiksi Jinja2 template-kielen kanssa.

```
{% extends "base_generic.html" %}

{% block title %}{{ section.title }}{% endblock %}

{% block content %}
<h1>{{ section.title }}</h1>

{% for story in story_list %}
<h2>
  <a href="{{ story.get_absolute_url }}">
    {{ story.headline|upper }}
  </a>
</h2>
<p>{{ story.tease|truncatewords:"100" }}</p>
{% endfor %}
{% endblock %}
```

10.1 Yhteenveto

Djangossa määritellään MVC samalla tavalla kuin Pyramidissakin. Tästä syystä voidaan todeta, että vaikka MVC-komponenttejen vaatimukset toteutuvat yksittäin, tulee samat ongelmat vastaan kuin Pyramidissakin. Kommunikaatiomalli on ristiriidassa MVC:n alkuperäisen toteutuksen kanssa, mikä johtuu HTTP:n päälle rakennetusta toteutuksesta. Lisäksi näkymä ei kommunikoi suoraan mallin kanssa, vaan kommunikaatio mallille toteutetaan ohjaimen kautta.

11 Web-soketit & Tornado

Pyramid-sovelluskehystä ja MVC:tä tutkiessa todettiin MVC:n alkuperäisen toteutuksen vaativan reaaliaikaista kommunikoita mallin sekä näkymän välillä siten, että malli kommunikoi suoraan näkymälle muutoksistaan. Tämä ei pelkällä Pyramidilla toteutetun web-sovelluksen avulla onnistu, koska kommunikointi toteutetaan aina HTTP-pyynnön kautta ja muutoksien näkeminen vaatii aina uuden HTTP-pyynnön. Web 2.0 teknologiat kuten AJAX (Asynchronous JavaScript and XML) ovat tuonneet uuden tavan loppukäyttäjille kommunikoida web-sovelluksien kanssa. Sen sijaan, että muodostetaan useita sivuja sekä takaisinkutsuja tuomaan sisältö loppukäyttäjälle, voidaan sisältöä tuoda reaaliaikaisesti. Loppukäyttäjän ei tällöin tarvitse päivittää sivua nähdäkseen sisällössä muutoksia. AJAX vaatii kuitenkin useiden päivityspyyntöjen lähettämistä palvelimelle yhteyden aikana, jotta käyttäjälle voidaan päivittää sisältöä. Tätä kutsutaan pollaukseksi (polling). Tästä syystä jokainen tapahtuma luo ylimääräistä kuormaa palvelimelle(Puranik 2013). AJAX ei siis riitä toteuttamaan MVC:ssä mallin suoraa kommunikointia näkymälle ja ohjaimelle, koska erillinen pyyntö päivitysten tarkastamiseen tarvitaan edelleen. AJAX:n kanssa ei kuitenkaan tarvita sivun lataamista uudelleen muutoksien näkemiseksi, koska pollaus toteutetaan taustalla yhteyden aikana. Websocketit ratkaisevat pollauksen ongelman ja mahdollistavat yhteyden molempiin suuntiin (Fette 2011, s. 1.1).

11.1 Esimerkkisovellus

Tornadon dokumentaatioissa luodaan yksinkertainen web-sovellus käyttäen web-soketteja (Authors 2009a). Sovelluksessa lähetetään selaimesta viesti palvelimelle, joka kaiuttaa viestin takaisin selaimelle.

```
1 import tornado.ioloop
2 from tornado import websocket
3
4 class EchoWebSocket(websocket.WebSocketHandler):
5     def check_origin(self, origin):
6         return True
7
```

```
8       def open(self):
9           print("WebSocket opened")
10
11      def on_message(self, message):
12          self.write_message(u"You said: " + message)
13
14      def on_close(self):
15          print("WebSocket closed")
16
17  def make_app():
18      return tornado.web.Application([
19          (r"/websocket", EchoWebSocket),
20      ], debug=True)
21
22  if __name__ == "__main__":
23      app = make_app()
24      app.listen(8888)
25      tornado.ioloop.IOLoop.current().start()
```

Jokaista web-sockettia varten luodaan erillinen käsittelijä, joka peritään WebSocketHandler-luokasta. Luokasta tulee ylikirjoittaa *on_message*-, *on_close*- ja *open*-metodit. Tietoturvan kannalta *check_origin* tulisi toteuttaa tarkastamaan asiakaspuolen yhteyden identiteetti, mutta tässä toteutuksessa ei oteta kyseiseen metodiin kantaa. *Open* -metodia kutsutaan yhteyden avaamisessa, *on_message* käsittelee viestin ja *on_close* kutsutaan yhteyden sulkemisen yhteydessä. Metodissa *make_app* liitetään käsittelijä polkuun */websocket*. Sovelluksen käynnistyttyä löytyy web-soketti osoitteesta *ws://localhost:8888/websocket*.

Alla esitellään selaimeen toteutettu JavaScript-koodi, jonka avulla web-sokettiin yhdistetään:

```
1  <script>
2    var ws = new WebSocket("ws://localhost:8888/websocket");
3
4    ws.onopen = function() {
5       ws.send("Hello World!");
6    };
7    ws.onmessage = function (evt) {
8       alert(evt.data);
```

```
9   };
10  </script>
```

Koodin ajaminen selaimessa luo HTTP:n kautta kättelyn ja tämän jälkeen siirtää viestinnän web-soketti protokollalle. Koodissa lähetetään yksinkertainen "Hello World!"viesti yhteyden avautuessa Tornadossa luotuun käsittelijään. Käsittelijässä *on_message* -metodi saa kyseisen viestin ja lähettää tämän takaisin selaimelle web-soketin yhteyttä pitkin. JavaScriptillä toteutetussa koodissa käsitellään viesti *onmessage* -metodissa, jossa viestin sisällön perusteella näytetään selaimessa ponnahdusikkuna.

12 MVC:n toteutus Tornadolla

Tornadon avulla toteutetaan vastaava laskurisovellus kuin aikaisemminkin Pyramid- ja Smalltalk-osioissa. Tällä kertaa kuitenkin HTTP:ta käytetään vain kättelyssä ja kommunikaatio komponenttejen välillä sovelluksen ajonaikana hoidetaan web-sokettejen avulla. Sovelluksessa ohjain ja näkymä toteutetaan selaimen puolella käyttäen HTML:ää sekä Javascript ohjelmointikieltä. Malli kirjoitetaan palvelimen puolelle, jossa käytetään Tornadoa. Ohjain voidaan myös toteuttaa Palvelimen puolelle, jolloin ohjaimen logiikka voidaan piilottaa käyttäjältä. Tässä tutkimuksesa keskitytään toteutukseen, jossa ohjaimen logiikka on nähtävillä web-selaimen puolella. Tutkimuksen kananlta oleellisinta on kuitenkin selvittää voidaanko web-sokettejen avulla toteuttaa MVC web-sovelluksessa. Jotta esimerkkisovellus saadaan esiteltyä selkeästi, on joitain paloja ohjelmasta jätetty näyttämättä. Täysin toimiva sovellus löytyy liitteet -osiosta.

12.1 Mallin toteutus

```
1  class Model(websocket.WebSocketHandler):
2      counter = 0
3
4      def increase(self):
5          self.counter += 1
6
7      def decrease(self):
8          self.counter -= 1
9
10     def on_message(self, message):
11         {"get": lambda: True,
12          "increase": self.increase,
13          "decrease": self.decrease}[message]()
14
15         self.write_message(str(self.counter))
```

Malli määritellään yhdessä luokassa, joka peritään *WebSocketHandler* -luokasta, joka löytyy Tornadon tarjoamasta *websocket* -kirjastosta. Luokalle alustetaan *counter* -muuttuja, jo-

ka määrittää mallin tilan. Lisäksi luokalle määritellään metodit *increase* ja *decrease*. Näiden metodejen avulla muutetaan laskurin tilaa. Lisäksi tulee ylikirjoittaa *WebSocketHandler* -luokasta peritty *on_message* -metodi. Sen tarkoituksena on käsitellä web-sokettiyhteyden kautta tulevat viestit. Viestin perusteella, joko vähennetään (decrease) tai kasvatetaan (increase) laskurin arvoa. Yhteyden muodostuksen aikana lähetetään näkymältä mallille myös "get-viesti, jossa nykyinen arvo päivitetään selaimelle. Jokaisella kutsulla *on_message* lähettää viestin yhdistetyille soketeille, jossa lähetetään mallin nykyinen arvo. Alla esitellään ohjelmakoodit, joilla Tornadon web-palvelin alustetaan ja käynnistetään vastaanottamaan yhteyksiä:

```
1 def make_app():
2     return tornado.web.Application([
3         (r"/model", Model),
4     ], debug=True)
5 
6 if __name__ == "__main__":
7     app = make_app()
8     app.listen(8888)
9     tornado.ioloop.IOLoop.current().start()
```

Web-soketit rekisteröidään Tornadolle antamalla *Application* -luokan instanssille parametrina lista luokkia, jotka on peritty *WebSocketHandkler* -luokasta. Lisäksi jokaista luokkaa kohden annetaan polku, josta sokettiin voidaan ottaa yhteyttä. Varsinainen sovellus rakennetaan *make_app* -funktion paluuarvon perusteella, joka palauttaa instanssin Tornado-sovelluksesta. Sovellus alustetaan kuuntelemaan porttia 8888, jonka jälkeen käynnistetään silmukka kuuntelemaan tapahtumia. Tämän jällkeen Malliin voidaan ottaa yhteyttä osoitteesta *ws://localhost:8888/mod* web-soketin avulla, jossa *localhost* viittaa lokaalin osoitteeseen. Lisäksi malli voi lähettää viestejä sokettiin kytketyille ohjaimille ja näkymille.

12.2 Ohjain

```
1 var modelSocket = new WebSocket("ws://localhost:8888/model");
2 var controller = (function (modelSocket) {
3     return {
4         increase: function () {
5             modelSocket.send('increase');
```

```
6           },
7           decrease: function () {
8               modelSocket.send('decrease');
9           }};
10  });
```

Ohjain toteutetaan web-selaimen puolella, jolla on viite Mallin sokettiin. Ohjaimella on vastaavat metodit kuin Mallilla (*increase* ja *decrease*). Kummankin toteutuksessa lähetetään joko viesti "increase"tai "decrease", joka käsitellään Mallin puolella kasvattaen tai vähentäen laskurin arvoa. Ohjaimelle voidaan myös antaa viite näkymään, jos ohjaimen tulisi kommunikoida näkymän kanssa. Kuitenkin toteutetussa laskurisovelluksessa tälle ei ole tarvetta, koska muutokset tilasta tulevat Mallin kautta. Lisäksi Mallilta tulevat viestit voidaan käsitellä ohjaimessa *modelSocket* -instanssin kautta, mutta tähänkään ei ole tarvetta, koska viestejä ei tarvitse Ohjaimessa käsitellä. Riittää, että ohjain vain lähettää viestejä napin painalluksista mallille.

12.3 Näkymä

```
1  <div id="counterView">
2      <div id="number"> </div>
3      <button id="increase" onclick="ctrl.increase()"> increase </
           button>
4      <button id="decrease" onclick="ctrl.decrease()"> decrease </
           button>
5      <script>
6          var ctrl = controller(modelSocket);
7          var view = view(modelSocket);
8          view.refresh();
9      </script>
10 </div>
```

Yllä esitellään HTML:llä kirjoitettu näkymä, jossa määritellään divisioona (*div*) tunnisteella *counterView*. Tunnisteen avulla voidaan yksilöidä divisioona, jolloin sitä on helpompi käsitellä Javascriptillä. Päädivisioonan sisälle lisätään kaikki mitä halutaan näytettävän laskurissa. Laskurin kaksi painiketta *increase* ja *decrease* liitetään ohjaimen funktioihin. Jokainen painallus kutsuu ohjaimessa määriteltyjä *increase* ja *decrease* funktioita riippuen valitusta

painikkeesta. Muuttujat *ctrl* ja *view* alustetaan funktioilla, joiden kautta websocket -yhteys suoritetaan. Lisäksi tulee varmistaa, että yhteys on muodostettu varmasti ennenkuin mitään toimintoja voidaan tehdä. Tästä syystä *refresh* -funktiossa on toteutettu tarkistus yhteyden tilalle ja sitä kutsutaan alustamisen mukana.

```
1  var view = (function (modelSocket) {
2       modelSocket.onmessage = function (evt) {
3       var element = document.getElementById("counterView").
            firstElementChild;
4       element.textContent = parseInt(evt.data);
5     };
6
7    function waitForSocketConnection(socket, callback){
8       setTimeout(
9           function(){
10              if (socket.readyState === 1) {
11                  if(callback !== undefined){
12                      callback();
13                  }
14              } else {
15                  waitForSocketConnection(socket,callback);
16              }}, 5);
17      }
18      return {
19          refresh: function () {
20              waitForSocketConnection(modelSocket, function() {
21                  modelSocket.send('get');
22              });
23          }};
24  })
```

Yllä esitellään useita funktioita joiden avulla yhdistetään näkymä vastaanottamaan websocket-yhteydestä tulevia viestejä. Lisäksi määritellään *refresh* -funktio, jota kutsumalla varmistetaan yhteyden alustus. Alustuksen yhteydessä myös lähetetään mallille viesti, jossa pyydetään lähettämään alkutila. Serveriltä tulevat viestit käsitellään *onmessage* -funktiossa, jota kutsutaan aina yksittäisen viestin saapuessa. Viestin sisältö löytyy *evt*:n *data* -muuttujasta. Muuttujan oletetaan olevan kokonaislukuarvo, joten se puretaan sellaiseksi ja asetetaan *coun-*

terView divisioonan ensimmäiseen lapsielementtiin, joka on tässä tapauksessa *number* -tunnisteella esiintyvä divisioona. Serveriltä tullut viesti laitetaan tällöin suoraan divisioonan sisältöön, jolloin muutos näkyy suoraan selaimessa.

12.4 Yhteenveto

Tornado ei ota kantaa MVC:n toteutukseen, joten se antaa vapauden sovelluksen kehittäjälle määrittää sovelluksen arkkitehtuurin rakenne. Jakamalla sovellus Malliin, Näkymään ja Ohjaimeen siten, että Ohjain ja Näkymä toteutetaan selaimen puolella ja Malli palvelimen puolella on mahdollista toteuttaa MVC-arkkitehtuuri niinkuin se on tarkoitettu. Komponenttejen kommunikaatio hoidetaan web-sokettejen kautta, jolloin mahdollistetaan kaksisuuntainen kommunikaatio komponenttejen välillä.

13 Tulokset

Djangossa ja Pyramidissa ei toteudu MVC:n alkuperäinen määritelmä. Kummankin MVC-toteutuksessa kommunikaatio Mallin ja Näkymän välillä toteutuu ohjaimen kautta, vaikka alkuperäisessä määrittelyssä myös Mallin tulisi olla suoraan yhteydessä Näkymään. Lisäksi vaaditaan kaksisuuntainen kommunikaatio komponenttejen välillä ja tämä ei toteudu HTTP:n kautta kommunikaation toteuttavissa web-sovelluksissa. Voidaan siis väittää, että mikä tahansa HTTP:n päälle rakennettu web-sovellus ei voi toteuttaa alkuperäistä MVC:n määritelmää. Web-sokettejen avulla ratkaistaan kaksisuuntaisen kommunikaation ongelma MVC:ssä ja Mallin yhteys Näkymään voidaan toteuttaa sovelluskehyksellä, jossa arkkitehtuurin toteutus on jätetty toteutettavaksi erikseen. Tornadon ja web-sokettejen avulla voidaan toteuttaa MVC alkuperäisen määritelmän mukaan.

Lähteet

Adrian Holovaty, Jacob Kaplan-Moss. 2009a. *Django's History*. http://www.djangobook.com/en/2.0/chapter01.html#django-s-history.

———. 2009b. *The Django Book*. http://www.djangobook.com/en/2.0/.

Arlington, Wilson Boulevard. 1981. "TRANSMISSION CONTROL PROTOCOL". https://www.ietf.org/rfc/rfc3439.txt.

Authors, The Tornado. 2009a. "The WebSocket Protocol". http://www.tornadoweb.org/en/stable/websocket.html.

———. 2009b. *Tornado Web Server*. http://www.tornadoweb.org/en/stable/index.html.

Avraham, Avraham Leff James T. Rayfield. 2001. "Web-Application Development Using the Model/View/Controller Design Pattern: 1.2, Web.Applications and the MVC Design". http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=950428.

Berners-Lee, T. 2011. "Known Issues and Best Practices for the Use of Long Polling and Streaming in Bidirectional HTTP". https://tools.ietf.org/html/rfc62026.

Bestframeworks. 2009. *List of Python MVC-frameworks by bestwebframeworks.com*. http://www.bestwebframeworks.com/compare-web-frameworks/python/.

Brecht, Tim. 2006. "Evaluating network processing efficiency with processor partitioning and asynchronous I/Os". http://dl.acm.org/citation.cfm?id=1217961&CFID=752791733&CFTOKEN=42693117.

Bush, R. 1981a. "TRANSMISSION CONTROL PROTOCOL". https://tools.ietf.org/html/rfc793#section-1.1.

———. 1981b. "TRANSMISSION CONTROL PROTOCOL". https://tools.ietf.org/html/rfc793#section-1.4.

Bush, R. 1981c. “TRANSMISSION CONTROL PROTOCOL”. https://tools.ietf.org/html/rfc793#section-2.3.

Cailliau, T. Berners-Lee R. 1990. “WorldWideWeb: Proposal for a HyperText Project”. https://www.w3.org/Proposal.html.

Carpentern, B. 1996. “Architectural Principles of the Internet”. https://www.ietf.org/rfc/rfc1958.txt.

Consulting, Agendaless. 2005. *Pyramid Introduction.* http://www.kemeneur.com/clients/pylons/docs/pyramid/narr/introduction.html.

D. C. Schmidt, M. E. Fayad. 1997. “Object-Oriented Application frameworks”. http://dl.acm.org/citation.cfm?id=262798.

Deacon, John. 2009. “Computer Systems Development, Consulting and Training Model-View-Controller (MVC) Architecture”.

Fette, I. 2011. “The WebSocket Protocol”. https://tools.ietf.org/html/rfc6455.

Fielding, R. 1991a. “The Original HTTP as defined in 1991”. https://www.w3.org/Protocols/HTTP/AsImplemented.html.

———. 1991b. “The Original HTTP as defined in 1991”. https://www.w3.org/Protocols/HTTP/HTTP2.htmll.

———. 1994. “Uniform Resource Locators (URL)”. https://tools.ietf.org/html/rfc1738.

———. 1995. “SGML Media Types”. https://tools.ietf.org/html/rfc1874.

———. 1999a. “Hypertext Transfer Protocol – HTTP/1.1”. https://tools.ietf.org/html/rfc2616.

———. 1999b. “Hypertext Transfer Protocol – HTTP/1.1”. https://tools.ietf.org/html/rfc2616#page-31.

Force, The Internet Engineering Task. 1992. “The Internet Engineering Task Force”. https://tools.ietf.org/.

Foundation, Django Software. 2016. *FAQ: General.* https://docs.djangoproject.com/en/1.10/faq/general/n.

G, Coulouris. 2005. *Distributed Systems: Concepts and Design 4th edition.* Addison Wesley.

Gamma, E. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesle.

Kourie, Iwan Vosloo & Derrick G. 2008. "Server-Centric Web Frameworks: An Overview". http://dl.acm.org/citation.cfm?id=1348246.1348247&coll=DL&dl=ACM&CFID=509430230&CFTOKEN=54230385.

Krasner, Glenn E. 1988. "A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk-80 System". http://www.create.ucsb.edu/~stp/PostScript/mvc.pdf.

Krasner, Glenn E. Krasner & Stephen T.Pope. 1988. "A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80". http://www.ics.uci.edu/~redmiles/ics227-SQ04/papers/KrasnerPope88.pdf.

Krasner, Glenn E. Krasner & Stephen T.Popez. 1979. "Models-Views-Controllers". http://heim.ifi.uio.no/~trygver/1979/mvc-2/1979-12-MVC.pdf.

Loreto, S. 2011. "Bidirectional HTTP". https://tools.ietf.org/html/rfc6202.

Ockanovic, T. Mateljan, V. Okanovic'. 2011. "Designing a New Web Application Framework". http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5967262.

Parc, Xerox. 1978. "MVC Xerox Parc". http://heim.ifi.uio.no/trygver/themes/mvc/mvc-index.html.

———. 1979a. "Xerox Parc THE Original MVC reports". http://heim.ifi.uio.no/~trygver/2007/MVC_Originals.pdf.

———. 1979b. "Xerox Parc THING-MODEL-VIEW-EDITOR". http://heim.ifi.uio.no/trygver/1979/mvc-1/1979-05-MVC.pdf.

Project, Pylons. 2010a. *About Pylons.* http://www.pylonsproject.org/about/pylons.

———. 2010b. *About Pyramid.* http://www.pylonsproject.org/projects/pyramid/about.

———. 2010c. *Pyramid introduction.* http://www.kemeneur.com/clients/pylons/docs/pyramid/narr/introduction.html.

Puranik, Darshan G. 2013. "Real-Time Monitoring using AJAX and WebSockets". http://ieeexplore.ieee.org/abstract/document/6601579/.

Pyramid Documentation, URL Dispatch. n.d. http://docs.pylonsproject.org/projects/pyramid/en/latest/narr/urldispatch.html.

Reenskaug, Trygve. 2003. "The Model-View-Controller (MVC) Its Past and Present". http://heim.ifi.%20uio.no/~trygver/2003/javazone-jaoo/MVC_pattern.pdf.

Ronacher, Armin. 2008. *Jinja2.* http://jinja.pocoo.org/docs/dev/.

Shahram Ghandeharizadeh, Ankit Mutha. 2014. "An Evaluation of the Hibernate Object-Relational Mapping for Processing Interactive Social Networking Actions". http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=950428.

Sheikh I. Ahamed, Alex Pezewski, ja Al Pezewski. 2008. "Towards Framework Selection Criteria and Suitability for an Application Framework". http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1286492.

Sommerville, Ian. 2010. *Software Engineering 9th edition.* Pearson.

Southern California, University of. 1981. "DARPA INTERNET PROGRAM, PROTOCOL SPECIFICATION". https://tools.ietf.org/html/rfc791.

Steve, Burbeck. 1992. "Applications Programming in Smalltalk-80(TM): How to use Model-View-Controller (MVC)". http://st-www.cs.illinois.edu/users/smarch/st-docs/mvc.html.

Web Application Development, Some Trends in. 2007. “Some Trends in Web Application Development”. https://www.researchgate.net/profile/Mehdi_Jazayeri/publication/4250861_Some_Trends_in_Web_Application_Development/links/57d02f5b08ae0c0081dea3bd.pdf.

Liitteet