Monialustaisten ohjelmien tekeminen C++-kielellä

Diplomityö
TURUN YLIOPISTO
Informaatioteknologian laitos
Ohjelmistotekniikka
Toukokuu 2013
Lauri Nurmi

TURUN YLIOPISTO
Informaatioteknologian laitos

LAURI NURMI: Monialustaisten ohjelmien tekeminen C++-kielellä

Diplomityö, 71 s.
Ohjelmistotekniikka
Toukokuu 2013

Käyttöjärjestelmiä ja prosessoriarkkitehtuureita on olemassa useita erilaisia. Näiden kahden yhdistelmää yhdessä käyttöjärjestelmän tarjoamien kirjastojen kanssa kutsutaan tässä diplomityössä alustaksi. Tietylle alustalle tarkoitettuja konekielisiä natiiviohjelmia voidaan yleensä suorittaa vain kyseistä alustaa käyttävissä tietokoneissa. Natiiviohjelmien tekemiseen käytetään tavallisesti C- ja C++-ohjelmointikieliä.

Monesti ohjelmia on voitava käyttää useammalla kuin yhdellä alustalla. Koska kehitystyö vie aikaa ja resursseja, on toivottavaa, että ohjelma toimisi uudella alustalla joko ilman muutoksia tai mahdollisimman paljon olemassa olevaa lähdekoodia hyödyntäen. Tässä diplomityössä esitellään erilaisia monialustaisten ohjelmien toteutustapoja. Myöhemmin arvioidaan niiden soveltuvuutta kaupallisen kameravalvontaohjelmiston toteutukseen. Keskeisimmät esiteltävät toteutustavat ovat natiiviohjelmat, tulkattavat kielet, virtualisointi sekä web-selain, joka on osittainen toteutustapa monialustaisille ohjelmille.

Tämä diplomityö käsittelee monialustaisten ohjelmien kehitystä erityisesti C++-kielellä, joka on yksi maailman eniten käytetyistä ohjelmointikielistä, ja myös tarkasteltavana olevan kameravalvontaohjelmiston nykyinen toteutuskieli. Kielen syntaksin ja C++-standardikirjaston määrittelevä C++-standardi on alusta-agnostinen. Tästä johtuen C++-lähdekoodi voi olla täysin alustariippumatonta vaikka konekielelle käännetyt ohjelmat ovatkin sidottuja tiettyyn alustaan. Useimmat käytännön C++-ohjelmat käyttävät standardikirjaston lisäksi alustan tarjoamia lisäkirjastoja.

Toteutusratkaisuiden arvioinnin ja vertailun perusteella voidaan todeta, että C++-kielellä tehty natiiviohjelma on monialustaisen kameravalvontaohjelmiston toteutukseen hyvin soveltuva ratkaisu, mutta myös muiden esiteltyjen tapojen käyttöä kannattaa harkita joissakin ohjelmiston osissa käytettäväksi. Nykyisen natiivitoteutuksen monialustaista kehitystä voitaisiin mahdollisesti tehostaa vaihtamalla käytettävä ohjelmistokehys toiseen sekä tekemällä käännösympäristöstä monialustaisempi ja automatisoidumpi. Johtopäätökset ovat sovellettavissa myös muihin samankaltaisiin ohjelmistoihin.

Asiasanat: monialustaisuus, alustariippumattomuus, C++, alusta, käyttöjärjestelmä

UNIVERSITY OF TURKU
Department of Information Technology

LAURI NURMI: Creating Cross-Platform Software Using C++

Master of Science Thesis, 71 p.
Software Engineering
May 2013

There are various kinds of operating systems and processor architectures. The combination of these two, together with libraries provided by the operating system, is called platform in this thesis. Programs targeted for a particular platform, compiled into native machine code, can usually only be executed on computers running that particular platform. Such native programs are typically written using the C and C++ programming languages.

It is often a requirement that programs can be used on more than one platform. Due to the time and resources needed for software development, it is desirable for a program to function on a new platform either without modifications, or for it to exploit as much existing source code as possible. Several methods for implementing cross-platform software are presented in this thesis. Later, their suitability for implementing a commercial video surveillance application is evaluated. The key implementation methods presented are native programs, interpreted languages, virtualization, and the web browser, which is a partial method for implementing cross-platform application programs.

This thesis discusses cross-platform software development especially using C++, which is one of the most widely used programming languages in the world, and also the current implementation language of the video surveillance application being analyzed. The C++ standard, defining the language syntax and the C++ standard library, is platform agnostic. Therefore C++ source code can be completely platform independent even though programs compiled into machine code are bound to a specific platform. Most real world C++ programs utilize additional libraries provided by the platform in addition to using the standard library.

Based on the evaluation and comparison of the different implementation methods it can be concluded that a native program written in C++ is a well-suited solution for the implementation of the video surveillance application, but other methods presented are worth considering for some components of the application. The cross-platform development process of the current native implementation could possibly be enhanced by substituting another framework for the one being used now, as well as making the build environment more cross-platform and more automated. The conclusions are also applicable to other similar software.

Keywords: cross-platform, platform independence, C++, platform, operating system

UNIVERSITÄT TURKU
Fachbereich Informatik

LAURI NURMI: Entwicklung plattformunabhängiger Software in C++

Masterarbeit, 71 S.
Software-Engineering
Mai 2013

Es gibt viele verschiedene Betriebssysteme und Prozessorarchitekturen. Die Gesamtheit aus diesen zwei, zusammen mit Bibliotheken des Betriebssystems, wird in dieser Masterarbeit Plattform genannt. In Maschinensprache übersetzte Programme können meistens nur auf Computern mit gleicher Plattform ausgeführt werden. Häufig sind plattformspezifische Programme in den Programmiersprachen C oder C++ geschrieben.

Oftmals müssen Programme auf mehr als einer Plattform ausgeführt werden können. Weil Entwicklungsarbeit Zeit und Ressourcen erfordert, ist es wünschenswert, dass das Programm auf einer neuen Plattform entweder ohne Änderungen funktioniert, oder den existierenden Quellcode so weit wie möglich wiederverwendet. Mehrere Implementierungsmittel für plattformunabhängige Programme werden in dieser Masterarbeit vorgestellt. Später wird bewertet, inwiefern sie auf die Implementierung einer kommerziellen Videoüberwachungssoftware angewendet werden können. Die wichtigsten Mittel die vorgestellt werden sind übersetzte Programmen, interpretierte Sprachen, Virtualisierung und der Webbrowser, der teilweise ein Mittel zur Implementierung von plattformunabhängigen Anwendungsprogrammen ist.

Diese Masterarbeit behandelt Entwicklung von plattformunabhängigen Programmen insbesondere auf C++, da sie eine der meistgebrauchten Programmiersprachen der Welt ist, und auch die derzeitige Implementierungssprache der Videoüberwachungssoftware, die hier untersucht wird. Der C++-Standard, der die Syntax und die C++-Standardbibliothek definiert, ist plattformagnostisch. Deswegen kann C++-Quellcode völlig plattformunabhängig sein, obwohl auf Maschinensprache übersetzte Programme an eine spezifische Plattform gebunden sind. Die meisten tatsächlichen C++-Programme nutzen allerdings plattformspezifische Bibliotheken zusätzlich zur Standardbibliothek.

Basierend auf der Bewertung und dem Vergleich der verschiedenen Implementierungsmittel kann festgestellt werden, dass ein übersetztes Programm in C++ eine gut anwendbare Lösung für die Implementierung einer Videoüberwachungssoftware ist, obgleich auch andere vorgestellte Mittel für einige Komponenten der Software in Betracht kommen. Die plattformunabhängige Entwicklung der derzeitigen Implementierung könnte womöglich verbessert werden, indem das derzeit verwendete Framework durch ein anderes ersetzt wird. Die Schlussfolgerungen können auch auf andere ähnliche Software angewendet werden.

Schlagwörter: Plattformunabhängigkeit, C++, Plattform, Betriebssystem

Sisältö

1 Johdanto

Aikojen alussa ohjelmistot tehtiin tiettyä tietokonetta ja tiettyä prosessoria varten suoraan prosessorin käyttämällä konekielellä. Ohjelman suorittaminen toisenlaisella tietokoneella ja prosessorilla edellytti koko ohjelman kirjoittamista uudelleen.

Monestakin syystä myöhemmin kehitettiin korkeamman tason ohjelmointikieliä, joiden kanssa ohjelmakoodi kirjoitettiin korkeammalla abstraktiotasolla, ja tällöin ihmisen kirjoittaman ohjelmakoodin muuntaminen konekielelle jäi kääntäjän tehtäväksi. Nyt sama ohjelmakoodi voitiin periaatteessa saada toimimaan eri prosessoreilla kääntäjää vaihtamalla.

Korkean tason kielten ja niihin liittyvien kääntäjien lisäksi fyysistä laitteistoa abstrahoi käyttöjärjestelmä. Käyttöjärjestelmän tehtävä on tarjota tunnettu, muuttumaton rajapinta fyysiseen laitteistoon ja tietokoneen resursseihin kuten kiintolevyyn, muistiin ja oheislaitteisiin.

Ohjelmiston käyttötarkoituksesta ja kohdeyleisöstä riippuen saattaa olla taloudellisesti ja teknisesti järkevää tehdä ohjelmisto tasan yhdelle alustalle. Jos on odotettavissa, että ohjelmisto halutaan saattaa myöhemmin toimimaan muilla alustoilla, kannattaa jo toteutuskielen valinta tehdä ottaen huomioon kielen käyttökelpoisuus muilla alustoilla ja käyttöjärjestelmillä.

C++ on yksi maailman eniten käytetyistä ohjelmointikielistä. Se on käytettävissä sovellusten tekemiseen supertietokoneissa, pöytäkoneissa, pelikonsoleissa sekä monissa puhelimissa ja taulutietokoneissa (tableteissa). Myös monet näistä alustoista tai käyttöjärjestelmistä on toteutettu C++-kielellä. Vaikka C++:aa ei yleisesti mielletä kovinkaan alustariippumattomaksi kieleksi, oikein käytettynä sillä voi kirjoittaa ohjelmakoodia, joka on hyödynnettävissä kymmenillä erilaisilla alustoilla. Esimerkiksi yleisimmin käytetty Java-virtuaalikone, Oraclen HotSpot, on kirjoitettu C++:lla.

Tässä diplomityössä kerrotaan monella alustalla toimiviksi tarkoitettujen ohjelmien tekemisestä C++-kielellä sekä arvioidaan C++:n soveltuvuutta tähän tarkoitukseen verrattuna muihin mahdollisiin tapoihin toteuttaa monella alustalla toimivia ohjelmia. Konkreettisena tarkastelun kohteena on C++-kielellä toteutettu ja monella alustalla

toimiva Ksenos-kameravalvontaohjelmisto, jonka kehityksen parissa työskentely on ollut suurimpana motivaationa koko tämän diplomityön aiheelle. Työssä tutkitaan myös sitä, kannattaisiko olemassa oleva Ksenos-ohjelmisto tai muu samankaltainen tuote toteuttaa C++:n sijaan jollakin vaihtoehtoisella tavalla, tai voidaanko C++:ssa pysyen monialustaisuutta jotenkin toteuttaa nykyistä paremmin tai helpommin.

Monella alustalla toimivien ohjelmien tekemiseen liittyy toisaalta alustojen eroja abstrahoivien kirjastojen ja/tai ohjelmistokehysten käyttäminen ja toisaalta ohjelmakoodin kirjoittaminen mahdollisimman suurelta osin siirrettäväksi. Siirrettävyydellä tarkoitetaan lyhyesti sanottuna lähdekoodin kirjoittamista siten, että sama lähdekoodi voidaan ilman muutoksia kääntää kaikilla halutuilla kohdealustoilla.

C++-standardi määrittelee kielen syntaksin, kokoelman käytettävissä olevia funktioita ja säiliöluokkia, yms. Pitäytymällä pelkästään standardin määrittelemissä funktiokutsuissa ja käytännöissä ohjelma pysyy siirrettävänä, sillä C++-standardi on tarkoituksella hyvin alusta-agnostinen. Toisaalta tästä seuraa, ettei standardin rajoissa pysymällä saada tehtyä monia sellaisia asioita, joita todellisissa ohjelmissa oletetaan voitavan tehdä. Standardi ei esimerkiksi ota kantaa näppäimistön lukemiseen tai grafiikan piirtämiseen näytölle. C++-kieltä kuitenkin voidaan käyttää ja käytetään lukemattomiin asioihin, joita standardi ei käsittele. Standardi määrittelee pakolliset luokat ja funktiot, mutta ei kiellä kehitysympäristöä tarjoamasta muitakin. Niinpä jokainen kääntäjä tai kehitysympäristö määrittelee myös standardiin kuulumattomia ohjelmointirajapintoja, joiden kautta onnistuu esimerkiksi syötteen lukeminen oheislaitteilta.

Luku 2 kertoo monelle alustalle kehitettävistä ohjelmista yleisesti – siitä, mitä monialustaisilla ohjelmilla halutaan saavuttaa, ja millaisia ratkaisuja näiden tavoitteiden saavuttamiseen on olemassa. Käännetyt kielet kuten C, C++ ja Objective-C käännetään kääntäjällä tietyn alustan natiiville konekielelle. Tulkattavia kieliä ei käännetä varsinaiselle prosessorin konekielelle, mutta usein jollekin välitason kielelle. Toinen samantyyppinen ja nykyään suosittu tekniikka on virtualisointi, jossa ohjelmat suoritetaan todellista tai kuvitteellista tietokonetta jäljittelevässä suoritusympäristössä. Myös web-selaimet ovat eräänlainen monialustainen suoritusympäristö, jota voidaan käyttää hyväksi monella alustalla toimivien sovellusten toteuttamisessa. Tässä luvussa

esitellään myös joukko kriteereitä, joiden perusteella monialustaisten ratkaisujen käyttökelpoisuutta voidaan arvioida.

Luku 3 esittelee ensin lyhyesti C++-kieltä, keskittyen sitten monella alustalla toimivien ohjelmien tekemiseen. C++-standardi määrittelee kielen syntaksin sekä C++-standardikirjaston, jota käyttäen voidaan kirjoittaa kaikilla alustoilla kääntyvää ja toimivaa C++-lähdekoodia. Standardikirjasto tarjoaa tarkoituksellisesti melko rajallisen valikoiman tapoja järjestelmän hyödyntämiseen, ja monessa käytännön ohjelmassa tarvitaan C++-standardikirjaston lisäksi alustakohtaisia ja kolmannen osapuolen kirjastoja. Lopuksi käsitellään prosessoriarkkitehtuurien ja kääntäjien eroja, jotka monialustaiseksi tarkoitetun C++-ohjelman kirjoittamisessa on syytä ottaa huomioon oikean toiminnan ja monella alustalla käännettävyyden varmistamiseksi.

Luku 4 kuvailee tarkastelun kohteena olevan Ksenos-kameravalvontaohjelmiston käyttötarkoitusta sekä esittelee ohjelmistoon liittyvät monialustaiset vaatimukset. Nykyisten toteutusratkaisuiden esittelyn yhteydessä arvioidaan alustakohtaisen lähdekoodin määrää suhteessa koko projektin koodimäärään.

Luvussa 5 arvioidaan alaluvussa 2.5 esiteltävien monialustaisuuden toteutusratkaisuiden etuja ja haittoja, ja niiden käyttökelpoisuutta tarkasteltavana olevaan Ksenos-ohjelmistoon. Nykyisellään kameravalvontaohjelmisto on toteutettu C++-kielellä, ja se on siten natiivisovellus kaikilla alustoillaan. Arvioinnin perustana käytetään alaluvussa 2.4 esiteltäviä monialustaisuuden kriteereitä ja toisaalta alaluvussa 4.2 esiteltäviä tähän ohjelmistoon liittyviä monialustaisia vaatimuksia. Arvioitavana on myös, kannattaisiko nykyinen C++-natiivitoteutus korvata jollakin toisella monialustaisella ratkaisulla.

Luvussa 6 pohditaan keinoja, joilla olemassa olevaa C++-natiivitoteutusta voitaisiin saada parannettua, ja erityisesti sen kehitystä monella alustalla tehostettua. Tehostumisen karkeana mittarina voitaneen pitää alustakohtaisen lähdekoodin määrän vähenemistä. Myös sovelluksen kääntämiseen tarvittavan käännösympäristön luominen ja varsinainen sovelluksen ja riippuvuuksina olevien kirjastojen kääntäminen vaativat alustakohtaisia ratkaisuja. Siksi käännösympäristön ja käännösprosessin monialustaisuuden parantaminen on tärkeä osa monialustaisen kehityksen tehostamista koko projektin mittakaavassa.

Monialustaisten toteutusratkaisuiden vertailussa johtopäätökseksi saadaan, että C++ on monialustaisten ohjelmien toteutukseen hyvin soveltuva ratkaisu, mutta myös erityisesti tulkattavat kielet voisivat olla käyttökelpoisia joidenkin ohjelmiston osien toteutukseen. Nykyisen C++-natiivitoteutuksen monialustaista kehitystä voitaisiin mahdollisesti tehostaa vaihtamalla käytettävä ohjelmistokehys toiseen sekä tekemällä käännösympäristöstä monialustaisempi ja automatisoidumpi. Myös uusi C++11-standardi tuo saataville joitakin alustakohtaisen lähdekoodin määrää vähentäviä keinoja.

2 Monen alustan ohjelmat

Tietokoneita on vuosikymmenten aikana valmistettu erilaisia ja eri käyttötarkoituksiin. Pienimmät ovat pieniä sulautettuja järjestelmiä, suurimmat suurtietokoneita (mainframe). Näiden ääripäiden väliin jäävät muun muassa henkilökohtaiset tietokoneet, pelikonsolit, palvelimet ja monet muut. Usein samat ohjelmistot tai ohjelmiston osat halutaan saada toimimaan erilaisilla tietokoneilla. Koska ohjelmistojen kehittäminen on suhteellisen hidasta työtä, on toivottavaa, että ohjelmistot toimisivat uudella alustalla ilman muutoksia tai edes mahdollisimman paljon olemassa olevaa lähdekoodia hyväksikäyttäen.

Goodwin [1, s. 1] määrittelee projektin olevan monialustainen silloin, kun projektia pysyvästi kehitetään yhtäaikaisesti vähintään kahdella alustalla, samaa päivittyvää lähdekoodia käyttäen. Tämä on olennaista vikojen nopeaa havaitsemista varten ja ohjelmakoodin pitämiseksi aidosti toimivana useammalla kuin yhdellä alustalla.

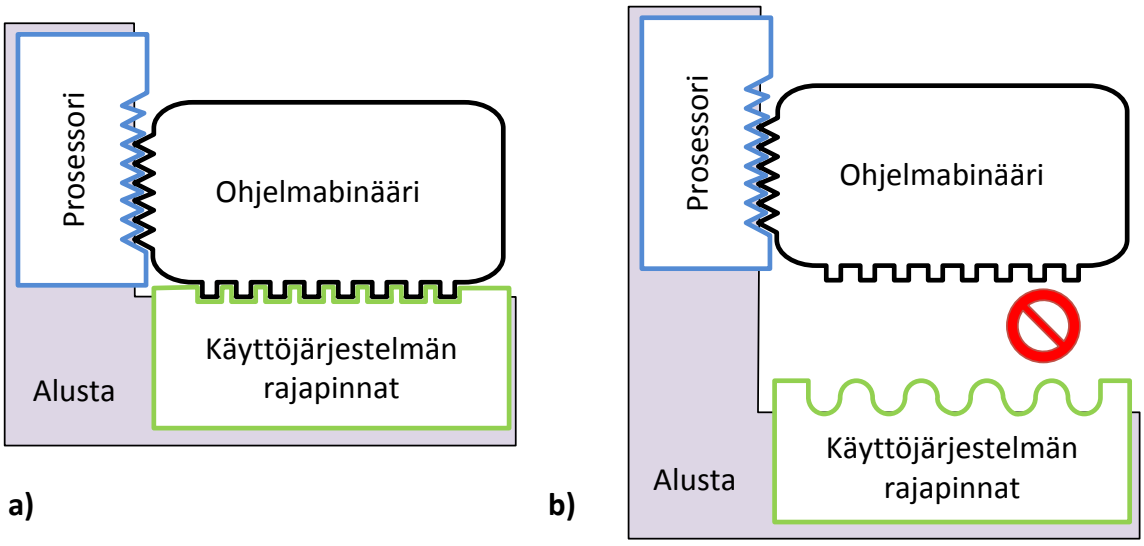
Tässä luvussa määritellään, mitä alustalla tässä työssä tarkoitetaan. Alaluku 2.2 kertoo hieman siitä, millä tavalla erilaisia fyysiset alustat teknisesti ovat; toisin sanoen mistä johtuu se, etteivät kaikki tietokoneohjelmat suoraan toimi kaikilla mahdollisilla tietokoneilla. Seuraava alaluku 2.3 käsittelee käyttöjärjestelmän tehtäviä. Käyttöjärjestelmä on niin yksi- kuin monialustaistenkin ohjelmien tekemistä merkittävästi helpottava abstraktiokerros laitteiston ja ohjelmiston välissä.

Alaluku 2.4 esittelee monialustaisille ohjelmille kriteereitä, joilla alaluvussa 2.5 esiteltäviä monella alustalla toimivien ohjelmien tekemiseen käytettävissä olevia menetelmiä voidaan arvioida.

Viimeinen alaluku 2.6 käsittelee alustariippumattomuuden ja monialustaisuuden rajoja, eli sitä, millaisia asioita ei voi tai kannata tehdä monialustaisiksi, ja millaiset seikat estävät täydellisen alustariippumattomien ohjelmien tekemisen.

2.1 Alustan määritelmä ja rajaus

Sana *alusta* (*platform*) viittaa toisaalta fyysiseen laiteympäristöön eli tietokoneeseen, jossa ohjelmaa suoritetaan, ja toisaalta ohjelmistorajapintoihin, joita ohjelma käyttää ja tarvitsee toimiakseen. Tässä työssä *(suoritus)alustalla* tarkoitetaan tietokoneen, siinä toimivan käyttöjärjestelmän ja käyttöjärjestelmän tarjoamien kirjastojen muodostaman suoritusympäristön yhdistelmää [2, s. 52]. Esimerkiksi i386-prosessoria käyttävä Windows 7 on yksi alusta, i386-prosessoria käyttävä Linux toinen alusta, ja ARMv6-prosessoria käyttävä Linux puolestaan on kolmas alusta. Kahden ensimmäisen kohdalla prosessori on sama, kahden viimeisen kohdalla taas käyttöjärjestelmä on sama. Kaikki kolme ovat silti erillisiä alustoja, joiden *ohjelmabinäärit* eivät ole keskenään yhteensopivia, eli mitään niistä ei voida ajaa toisella alustalla (Kuva 2-1).



Kuva 2-1: a) Ohjelmabinääri käyttää yhteensopivaa prosessoria ja käyttöjärjestelmän rajapintoja. b) Ohjelmabinääriä ei voida suorittaa, sillä käyttöjärjestelmä tarjoaa ohjelmabinäärille sopimattomia rajapintoja.

Myös kääntäjän voidaan eräässä mielessä katsoa kuuluvan osaksi alustan käsitettä, sillä tietyn kääntäjän tuottamilla ohjelmabinääreillä on yleensä riippuvuus kyseiseen kääntäjään liittyviin ajonaikaisiin kirjastoihin (run-time).

Virtualisointia käyttävissä ratkaisuissa suoritusalustan tietokoneena on virtuaalikone fyysisen tietokoneen sijaan, ja fyysinen laitteisto on vähemmän tärkeässä osassa.

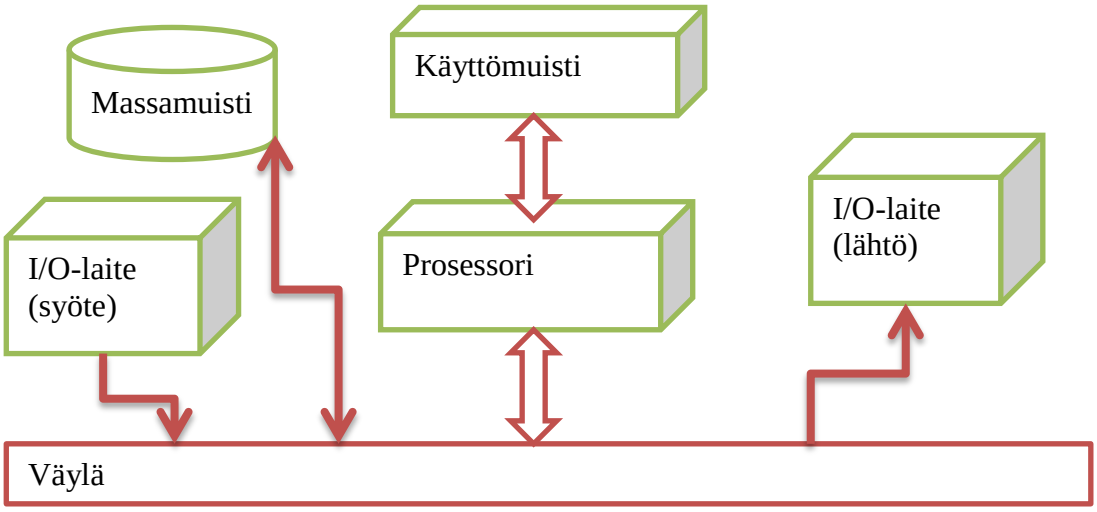
Tässä työssä oletetaan, että ohjelmaa suorittavassa tietokoneessa on käyttöjärjestelmä, eikä ohjelmaa suoriteta suoraan laitteiston päällä, eikä ohjelma myöskään itse ole käyttöjärjestelmä. Käyttöjärjestelmän tehtävistä kerrotaan enemmän alaluvussa 2.3. *Tietokoneella* tarkoitetaan tässä työssä kaikenlaisia laitteita, myös sulautettuja, joissa voidaan suorittaa ohjelmia, ja joissa muistia on vähintään useita megatavuja ja prosessorilla kellotaajuutta vähintään megahertsejä.

2.2 Alustan fyysinen osuus – laitteisto

Samoin kuin tietokoneita, myös prosessoreita ovat vuosikymmenten aikana useat eri valmistajat suunnitelleet ja valmistaneet, osin samoihin ja osin erilaisiin käyttötarkoituksiin. Prosessorin suunnittelussa huomioon otettavia rajoitteita ovat erityisesti valmistuskustannukset, nopeus ja virrankulutus [3, s. 8]. Näiden rajoitteiden keskinäinen tärkeysjärjestys vaihtelee prosessorin käyttötarkoituksen mukaan, sillä esimerkiksi matkapuhelimen prosessorille matala virrankulutus on paljon oleellisempi tekijä kuin aina sähköverkkoon kytketyn palvelintietokoneen prosessorille.

Prosessoreilla on niille ominainen *konekielinen käskykanta* (*instruction set architecture, ISA*). Tämä käskykanta on muuttumattomana pysyvä rajapinta prosessorin laitteistototeutuksen ja prosessoria käyttävien ohjelmistojen välillä [4, s. 7]. Toisin ilmaistuna käskykannan käskyt ovat kieli, jota kyseinen prosessori ymmärtää. Samaa käskykantaa käyttävillä prosessoreilla sanotaan olevan sama *prosessoriarkkitehtuuri*, eli prosessorit ovat yhteensopivia ja voivat suorittaa samoja konekielisiä ohjelmia [4, s. 2]. Toisaalta samaa käskykantaa käyttävien prosessoreiden laitteistototeutus voi olla hyvinkin erilainen. Käskykantaan voidaan lisätä laajennoksia eli uusia käskyjä, mutta vanhojen käskyjen merkitys säilyy ennallaan. Intelin 80386-prosessorissa ensimmäistä kertaa käytetty *IA-32*-käskykanta, joka yleisesti tunnetaan myös nimillä *x86* ja *i386*, on varmasti tunnetuin esimerkki hyvin pitkäikäisestä ja laajasti käytetystä käskykannasta, jota käyttäviä prosessoreita on valmistanut useampi kuin yksi valmistaja. IA-32-käskykannan olemassaolon aikana prosessorien laitteistototeutus on muuttunut täysin ja nopeudet ovat kasvaneet monisatakertaisiksi alkuperäiseen 80386-prosessoriin verrattuna.

Prosessorin lisäksi jokaisessa tietokoneessa on myös muita melko välttämättömiä komponentteja: on oltava jonkinlainen massamuisti tai muu muistilaite, jolta suoritettava ohjelmakoodi ladataan. Jotta ohjelma voisi tehdä mitään hyödyllistä, on oltava käytössä jonkinlaisia I/O-laitteita, joiden kautta voidaan vaikuttaa laitteiston muihin osiin ja ympäröivään maailmaan sekä saada syötettä ohjelmalle.



Kuva 2-2: Tietokoneen välttämättömät komponentit ja tietokoneen yleinen rakenne.

Vaikka prosessori ja sen käskykanta olisi tunnettu, I/O-laitteet ja menetelmät niiden käyttämiseen eivät välttämättä ole vakioituja ja tunnettuja. I/O-laitteiden ja muun laitteiston eroavaisuuksien piilottamiseen käytetään kuitenkin käyttöjärjestelmää, jonka tärkeä tehtävä on abstrahoida laitteiston osat tunnetun rajapinnan kautta käytettäviksi.

2.3 Alustan ohjelmistollinen osuus – käyttöjärjestelmä

Käyttöjärjestelmä on ohjelmistokerros, jota käytetään lähes kaikissa tietokoneissa aivan pienimpiä sulautettuja järjestelmiä lukuun ottamatta. Käyttöjärjestelmän päällä ajetaan sovellusohjelmia. Tanenbaum määrittelee [5, s. 3] käyttöjärjestelmään kuuluvaksi vain niin sanotussa *kernel-tilassa* toimivan kaikkein matalimman tason ohjelmistokerroksen, johon kuuluvat muun muassa laiteajurit, mutta ei esimerkiksi olennaisia palveluita kuten käyttöliittymä tai lähes kaikkien ohjelmien tarvitsema järjestelmän C-kirjasto.

Tässä diplomityössä käyttöjärjestelmä-termiä käytetään Tanenbaumin määrittelemää laajemmassa merkityksessä tarkoittamaan koko sitä ohjelmistokokonaisuutta, jonka tarjoamia rajapintoja ja palveluita käyttäen omat sovellukset tehdään, ja jonka voidaan olettaa ilman lisäosien asennuksia olevan käytettävissä tiettyä alustaa edustavassa järjestelmässä.

2.3.1 Matalan tason tehtävät

Tanenbaumin mukaan [5, s. 4–6] käyttöjärjestelmän tehtäviä voi katsoa kahdesta näkökulmasta: Ensimmäisestä näkökulmasta käyttöjärjestelmän tehtävä on toimia korkean tason abstraktiona tietokoneen fyysiselle laitteistolle ja siten tarjota sovellusohjelmille laitteiston suoraa käsittelyä yksinkertaisempi rajapinta tietokoneen ja sen komponenttien käyttämiseen. Osa käyttöjärjestelmän laiteabstraktiotehtävää on erilaisten mutta samaan käyttötarkoitukseen tarkoitettujen fyysisten laitteiden abstrahointi saman rajapinnan taakse – esimerkiksi SATA-kiintolevy ja USB-muistitikku näyttävät sovellusohjelman ja käyttäjän näkökulmasta samanlaiselta massamuistilta, vaikka ne ovat laitteistollisesti ja liitännällisesti aivan erilaiset laitteet.

Käyttöjärjestelmän tehtävä toisesta näkökulmasta katsottuna on hallita tietokoneen resurssien eli laitteiden jakamista niitä käyttävien tahojen kesken. Jakamista tehdään kahdella tasolla: resursseihin jaetaan käyttöaikaa ja toisaalta osuuksia resurssien sisältämään tilaan.

Ajallisesti jaettavia resursseja ovat muun muassa tietokoneen prosessoriytimet ja tulostimet. Yhtä prosessoriydintä voi käyttää tasan yksi prosessi – eli ohjelma – kerrallaan. Koska moniajoon kykenevässä käyttöjärjestelmässä prosesseja on käynnissä useita samanaikaisesti, käyttöjärjestelmä jakaa prosesseille käyttöaikaa yksittäiseen prosessoriytimeen. Tämä tapahtuu vuorottelemalla käynnissä olevien prosessien välillä, vaihtaen suoritusvuorossa olevaa prosessia satoja tai tuhansia kertoja sekunnissa. Tulostin on esimerkki laitteesta, jossa resurssin ajallinen jakaminen on ehkä konkreettisemmin nähtävissä. Useat ohjelmat voivat käyttää yhtä tulostinta, mutta vain yksi kerrallaan, sillä tulostin voi piirtää vain yhdelle paperiarkille kerrallaan, eikä eri lähteistä tulevien tulostustöiden päätyminen samalle paperiarkille ole myöskään

toivottavaa. Ongelmien estämiseksi käyttöjärjestelmä asettaa samaan aikaan aloitetut tulostustyöt jonoon, josta työt lähetetään tulostimelle yksi kerrallaan.

Tilallisesti jaettavia resursseja puolestaan ovat esimerkiksi käyttömuisti ja kiintolevytila. Kun käyttömuistia on riittävästi, se on jaettu usean käynnissä olevan prosessin kesken prosessikohtaisiin muistialueisiin sen sijaan, että yksi prosessi saisi koko muistin käyttöönsä. Kiintolevytilaa jaetaan samaan tapaan käyttäjien ja ohjelmien kesken. Kiintolevyn sisältö on käyttöjärjestelmän kannalta pohjimmiltaan miljardien alkioiden pituinen sarja tavuja, jotka saadaan jaettua suuremmiksi loogisiksi ja toisistaan erillisiksi kokonaisuuksiksi – kiintolevyn tallennustila jaetaan normaalisti ensin *osioiksi* ja osion sisältämän tiedostojärjestelmän sisällä *tiedostoiksi* ja vapaaksi tilaksi.

Käyttöjärjestelmä edistää siis omalta osaltaan monialustaisuutta toteuttamalla hyvinkin erilaisiin laitteistoihin samat rajapinnat, joita sovellusohjelmat voivat käyttää. Laajemmassa merkityksessä tällaisia rajapintoja ovat myös tietyt peruskäsitteet, kuten tiedostojärjestelmän puumainen rakenne eli tiedostot ja hakemistot, jotka kaikki nykyaikaiset käyttöjärjestelmät toteuttavat [5, s. 38].

2.3.2 Rajapinnat

Käyttöjärjestelmät ja niiden rajapinnat ovat keskenään suurelta osin erilaisia, ja tästä erilaisuudesta seuraa itse asiassa sovellusohjelmien tasolla paljon enemmän monialustaisuuden ongelmia ratkaistavaksi kuin erilaisista prosessoriarkkitehtuureista.

2.3.2.1 Binääriset rajapinnat

Käyttöjärjestelmä tarjoaa sovelluksille binääritason rajapinnan, *ABIn* (*Application Binary Interface*). ABI on matalan tason rajapinta, joka määrittelee miten käyttöjärjestelmässä suoritettavat sovellukset ovat vuorovaikutuksessa käyttöjärjestelmän, kirjastojen ja prosessorin kanssa. ABI on samantapainen käsite kuin *API* (*Application Programming Interface*), kuitenkin sillä erolla, että API määrittelee rajapinnan lähdekoodin tasolla, ABI puolestaan käännettyjen binäärien tasolla.

ABIin kuuluu se osa prosessorin käskykannasta, jota sovellusohjelmat käyttävät. Lisäksi ABI määrittelee, millaisia järjestelmäkutsuja sovellukset voivat tehdä käyttääkseen käyttöjärjestelmän laiteabstraktioita. [4, s. 8] Järjestelmäkutsuja tarvitaan lähes kaikkeen

toimintaan – tiedostojen avaamiseen, lukemiseen, ja niin edelleen. ABI määrittelee myös sen, miten funktiokutsujen parametrit välitetään funktioille ja minkä kokoisia näissä parametreissa käytettävät tietotyypit ovat.

2.3.2.2 Lähdekooditason rajapinnat

Binäärisen rajapinnan lisäksi käyttöjärjestelmä tai kääntäjän valmistaja tarjoaa käyttöjärjestelmän binäärirajapintoja vastaavan APIn uusien sovellusten kääntämistä varten. Tämä API koostuu yleensä C-kielen otsikkotiedostoista (header). Uusien sovellusten tekemiseen käytettävissä on vähintään C-kielen *standardikirjasto*, ja usein myös C++-kielen standardikirjasto. Standardikirjastoista kerrotaan enemmän kohdassa 3.1.

Unix-tyyppiset käyttöjärjestelmät kuten Linux, BSD, Mac OS X ja Solaris pyrkivät seuraamaan *POSIX*-standardia (*Portable Operating System Interface*), joka standardoi tietyt APIt, tiettyjen komentorivityökalujen toimintaa ja muita asioita. POSIXin rajapinnat määrittelevät muun muassa prosessien, sokettien, tiedostojärjestelmän hakemistorakenteen ja säikeiden käsittelyyn tarvittavia funktioita, toisin sanoen sellaisia yleisesti tarvittavia työkaluja, jotka eivät C- tai C++-kielen standardikirjastoon kuulu. POSIX tuo siis Unix-tyyppisten järjestelmien sovelluskehitykseen tietynlaista yhtenäisyyttä tarjoamalla moniin tavallisiin ongelmiin monella alustalla toimivan ratkaisun. Todellisuudessa käyttöjärjestelmien POSIXiin kuuluvissa rajapinnoissakin on pieniä eroja, jotka on alustakohtaisesti huomioitava.

Windows-ympäristössä C- ja C++-standardikirjastojen ohella on käytettävissä Win32-API, jolla tehdään osin samoja asioita kuin POSIXin APIlla, mutta erinimisiä funktioita käyttäen ja eri tavalla. Win32-API ei ole monialustainen muuten kuin Windowsin eri versioiden välillä.

C- ja C++-standardikirjastot, POSIX-rajapinnat ja Win32-API ovat matalimman tason julkisia ohjelmointirajapintoja. Näiden lisäksi käyttöjärjestelmässä on käytettävissä vaihteleva määrä asennukseen kuuluvia tai kolmannen osapuolen korkeamman tason kirjastoja, joiden rajapintoja sovellusohjelmat voivat hyödyntää. On tosin vaikeata määritellä mitä käyttöjärjestelmän asennukseen kuuluva kirjasto tarkalleen tarkoittaa, sillä etenkin Linux-jakeluiden asennuksissa asennuksen sisältö vaihtelee minimaalisesta

laajaan, ja asennusta voidaan täydentää lataamalla ja asentamalla lisäkirjastoja verkosta käyttöjärjestelmän omilla työkaluilla.

2.4 Monialustaisuuden kriteerit

Monialustaisten ohjelmistojen tekemisellä tavoitellaan ja saavutetaan useita erilaisia hyötyjä. Markkinoinnillisesta näkökulmasta katsoen ohjelmistoista halutaan tehdä monialustaisia siksi, että mahdollisten käyttäjien tai asiakkaiden joukkoa voidaan laajentaa tarjoamalla sama ohjelmistotuote monelle eri alustalle [1, s. 4]. Ohjelmistotuotannollisesta näkökulmasta taas monialustaisuus on houkuttelevaa siksi, että hyödyntämällä samaa ohjelmakoodia monella alustalla vältetään saman toiminnallisuuden toteuttaminen moneen kertaan. Tämä puolestaan vähentää toteutukseen käytettyä työmäärää sekä helpottaa testausta ja vikojen korjaamista, kun yksi korjaus korjaa vian hyvässä tapauksessa kaikilla alustoilla.

Monialustaisten ohjelmien tekeminen perustuu abstraktioihin ja alustakohtaisen ohjelmakoodin eristämiseen mahdollisimman pieneksi itsenäiseksi yksiköksi. Näin monialustainen ohjelmistokehitys edistää ohjelmakoodin modulaarisuutta, ja siten parantaa koodin laatua. Natiiviohjelmien tapauksessa erilaisten kääntäjien käyttäminen eri alustoilla antaa mahdollisuuden nähdä usean kääntäjän antamat varoitukset ja virheet, ja näin löytää uusia ongelmakohtia ohjelmakoodista [1, s. 5].

Seuraavassa alaluvussa esitettävät ratkaisut saavuttavat monialustaisuuden piilottamalla alustojen eroja erilaisten abstraktiokerrosten taakse. Monialustaisuuden tavoite ei kuitenkaan ole abstrahoida taustalla olevaa alustaa näkymättömiin, vaan integroitua siihen alustalle ominaisella tavalla.

Heitkötter et al. [6] esittelevät seitsemän kriteeriä monialustaisten mobiilisovellusten kehityksen ratkaisuiden arviointiin sovelluksen toiminnallisuuden ja elinkaaren näkökulmasta. Nämä seitsemän kriteeriä ovat, uudelleen numeroituina: 0. *License and costs,* 1. *Supported platforms,* 2. *Access to advanced device-specific features,* 3. *Long-term feasibility,* 4. *Look and feel,* 5. *Application speed,* 6. *Distribution*. Kriteerit sopivat melko hyvin myös muiden kuin mobiilisovellusten kehitysratkaisuiden arviointiin, joten valitaan kriteerit 1 – 6 hieman mukailtuina tässä tutkielmassa käytettävien kriteereiden

pohjaksi. Nollas kriteeri jätetään pois, sillä lisenssikysymykset ja kehitysympäristön hinta eivät mobiilisovellusten ulkopuolella ole merkittävä tekijä; lähes kaikille alustoille voi kehittää ohjelmia sekä ilmaisilla että maksullisilla työkaluilla, haluamallaan lisenssillä. Kuudes kriteeri muutetaan käsittelemään asennuspaketin tekemisen eikä niinkään jakelun helppoutta. Lisäksi otetaan käyttöön seitsemäs kriteeri, jolla arvioidaan monialustaisuudesta saatavaa hyötyä.

1. Tuetut alustat: Miten laajasti ja erilaisilla alustoilla kyseinen ratkaisu on käytettävissä.
2. Pääsy laitekohtaisiin lisäominaisuuksiin: Voidaanko kyseistä ratkaisua hyödynnettäessä käyttää alustan tai laitteen epätyypillisiä lisälaitteita.
3. Pitkän aikavälin käyttökelpoisuus: Onko odotettavissa, että valittu ratkaisu on käyttökelpoinen myös tulevaisuudessa.
4. Ulkoasu, integroituminen alustaan: Ohjelman tulee toimia alustan käytäntöjen mukaisesti. Sen on noudatettava alustalle ominaisia tiedostopolkuja ja käyttöliittymän ulkoasua, ja ohjelmasta ei saa ilmiselvästi näkyä, että se on alun perin kehitetty jotakin toista alustaa varten. Sen on toimittava yhteen muiden sovellusten kanssa.
5. Nopeus, tehokkuus: Ohjelman tulee toimia käyttötarkoitukseen riittävällä nopeudella, niin vasteaikojen kuin raa'an laskentatehon osalta.
6. Julkaisun helppous: Miten yksinkertaista on kääntää tuote ja tuottaa siitä asennuspaketti eri alustoja varten.
7. Hyöty: Monialustaisen toteutuksen on oltava vähemmällä työllä kehitettävissä ja ylläpidettävissä kuin erillisten alustakohtaisten toteutusten.

Näiden seitsemän kriteerin lisäksi ehkä ilmiselvä, mutta erittäin tärkeä vaatimus monialustaisille ohjelmille on oikeellisuus. Ohjelman tulee tuottaa samasta syötteestä samat tulokset kaikilla alustoilla, ellei muuta tietoisesti haluta. Oikeellisuuden toteutumista ei ole kuitenkaan järkevää arvioida numeroasteikolla, sillä kysymys on jonkinlaisesta perusoletuksesta, jonka tulee aina olla kunnossa. Odottamattomat erot tuloksissa alustojen välillä ovat yksiselitteisesti vikoja joko ohjelmassa tai ympäristössä, jossa sitä suoritetaan.

Heitkötter et al. arvioivat ratkaisuita sanallisesti sekä numeroasteikolla 1 – 6, jolla 1 tarkoittaa erittäin hyvää ja 6 erittäin huonoa. Tässä tutkielmassa käytetään samaa asteikkoa, kun seuraavaksi esitettäviä ratkaisuita myöhemmin arvioidaan näiden kriteerien perusteella luvussa 5.

2.5 Ratkaisuja monialustaisuuden toteuttamiseen

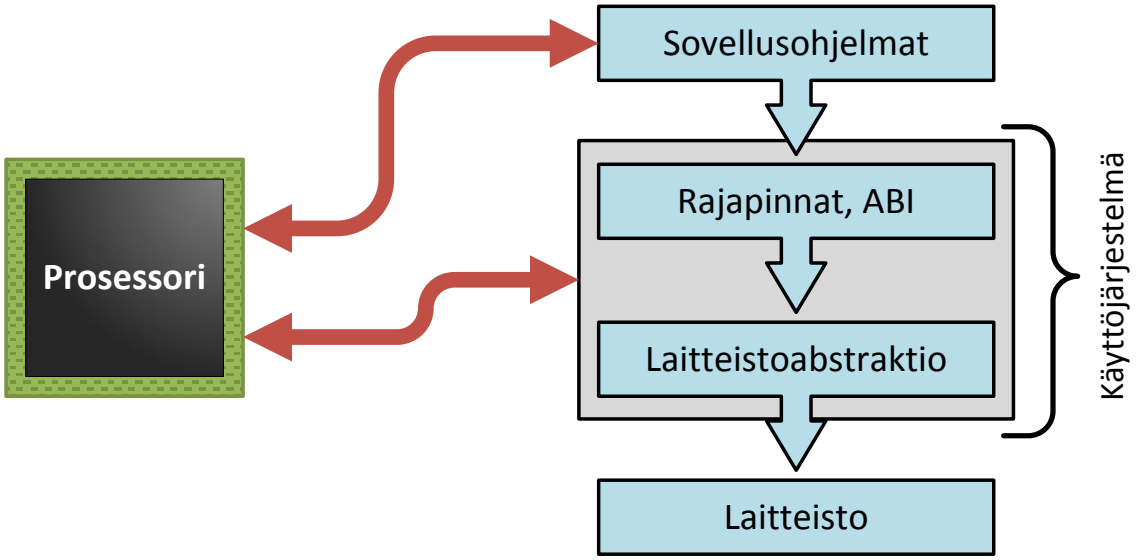
Monella alustalla toimivien ohjelmien tekemiseen on useita erilaisia lähestymistapoja. Seuraavassa esitellään mahdollisia ratkaisuja. Monet esitetyt ratkaisut ovat itse asiassa useamman ratkaisuvaihtoehdon yhdistelmiä.

2.5.1 Natiiviohjelmat

Natiiveja eli alustalle luontaisia, järjestelmän prosessorin käyttämästä konekielestä koostuvia ohjelmia tuotetaan kääntämällä lähdekoodia, joka on yleensä kirjoitettu C-, C++- tai Objective-C-kielellä. Kääntäjä siis muuntaa korkean tason ohjelmointikielen prosessorin käyttämän konekielisen käskykannan käskyiksi. Käännöksen tuottamaa suoritettavaa ohjelmatiedostoa kutsutaan tässä *ohjelmabinääriksi*.

Natiivin ohjelman ohjelmabinääri on vähintään kahdella tasolla riippuvainen siitä alustasta, jolla sitä ajetaan (Kuva 2-3). Ensinnäkin koska ohjelmabinääri on käännetty tietyn prosessorin tuntemalle konekielelle, se voidaan suorittaa vain juuri tällaisella tai yhteensopivalla prosessorilla. Toiseksi pelkkä oikeanlainen prosessori ei riitä ohjelman onnistuneeseen suoritukseen, sillä ohjelma on riippuvainen myös käyttöjärjestelmän tarjoamista rajapinnoista, ABIsta. Lisäksi ohjelmabinäärillä voi olla – ja usein on – riippuvuuksia myös muihin kuin käyttöjärjestelmän mukana toimitettaviin dynaamisiin kirjastoihin.

Natiiviohjelmat ovat siis hyvin heikosti monialustaisia käännettyinä, binäärimuodossa. Käännettyä ohjelmabinääriä ei ilman emulointia tai virtualisointia voi suorittaa muulla alustalla. Natiivit ohjelmat voivat kuitenkin olla hyvin monialustaisia *lähdekoodin tasolla*. Lähdekooditason monialustaisuus tarkoittaa lähdekoodia, joka ei sisällä viittauksia alustakohtaisiin funktiokutsuihin tai muihin alustakohtaisiin symboleihin, ja on sen vuoksi käännettävissä mille tahansa alustalle ilman muutoksia.



Kuva 2-3: Sovellusohjelma käyttää sekä prosessoria että käyttöjärjestelmän rajapintoja.

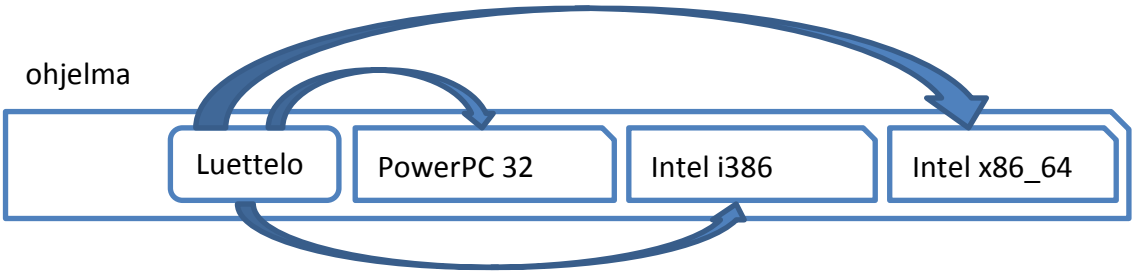
Monialustaisiksi soveltuvien natiiviohjelmien tietoinen suunnittelu perustuu tietämykseen siitä, mitkä osat lähdekoodista ovat käyttökelpoisia kaikilla alustoilla ja mitkä osat on toteutettava erikseen eri alustoille.

C- ja C++-kielten kanssa esikääntäjän makroja ja ehtolauseita käyttäen voidaan alustakohtaiset osat ohjelmakoodista kääntää ehdollisesti mukaan ohjelmaan tarvittavilla alustoilla. Esikääntäjästä kerrotaan lisää alaluvussa 3.3. Alustakohtainen lähdekoodi voidaan eriyttää vaihtoehtoisiksi toteutuksiksi samalle rajapinnalle ja sijoittaa erillisiin lähdekooditiedostoihin, joista valitaan käännettäväksi kullekin alustalle sopiva.

2.5.2 Moniarkkitehtuuriset binäärit

Moniarkkitehtuuriset binäärit (*fat binary*, *multiarchitecture binary*) ovat eräs tapa saada sama ohjelmabinääri toimimaan usealla alustalla, ennen kaikkea samassa käyttöjärjestelmässä mutta eri prosessoriarkkitehtuureilla. Ohjelmasta käännetään tai ristiinkäännetään ensin natiiviohjelmabinääri jokaiselle halutulle prosessoriarkkitehtuureille erikseen; lopuksi käännösten tuloksena saadut ohjelmabinäärit – tai yhtä lailla kirjastot – yhdistetään apuohjelmaa käyttäen yhdeksi

tiedostoksi. Ohjelmaa käynnistäessään käyttöjärjestelmä etsii tiedostosta sen binäärin, joka sopii käytössä olevalle prosessorille.



Kuva 2-4: Universaalibinäärin rakenne yksinkertaistettuna. Yhteen tiedostoon on tässä koottu binäärit kolmelle eri arkkitehtuurille. Käyttöjärjestelmä valitsee sopivimman ohjelmaa käynnistäessään.

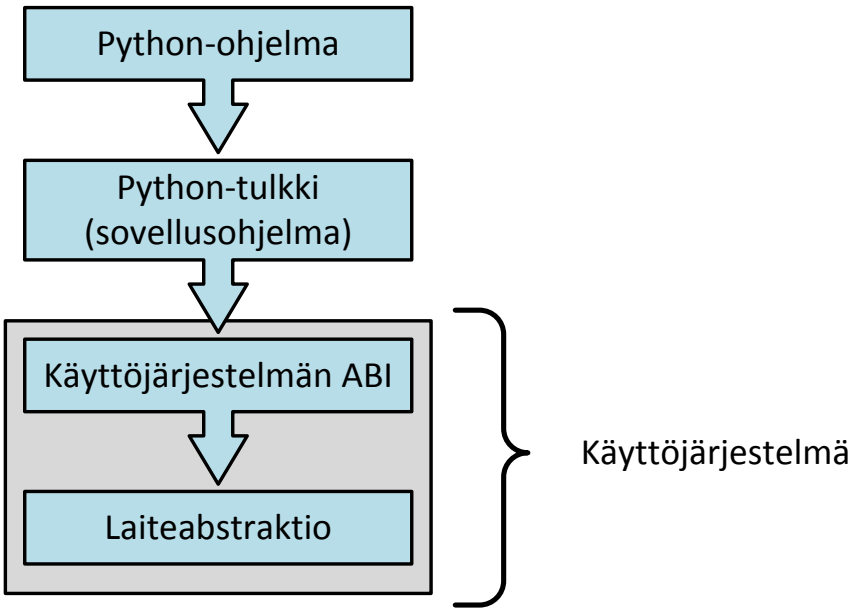
Moniarkkitehtuuriset binäärit ovat siis yhteen koottuja natiiviohjelmien binääreitä. Ne eivät näin ollen ole uusi, erillinen ratkaisutapa monialustaisuuteen. Moniarkkitehtuuriset binäärit ratkaisevat monialustaisuuden ongelmaa ohjelman paketoinnin ja levityksen tasolla, mutta varsinaisen suoritettavan ohjelmakoodin tasolla ratkaisu on täysin sama kuin kohdassa 2.5.1 kuvatuissa natiiviohjelmissa.

Yksinkertaisesta ideastaan huolimatta moniarkkitehtuurisia binääreitä eivät ole käyttäneet juuri muut kuin NeXT ja Apple, joka kutsuu niitä *universaalibinääreiksi* (*Universal Binary*). Universaalibinääreiden suurin etu on, että ne ovat loppukäyttäjälle täysin läpinäkyviä, eikä loppukäyttäjän tarvitse tuntea tietokoneensa prosessorin arkkitehtuuria. Universaalibinäärit esiteltiin suurelle yleisölle Applen luopuessa PowerPC-prosessoreista vuonna 2006, mutta niillä on käyttöä myös Intel-siirtymän ulkopuolella: sama universaalibinääri voi sisältää 32-bittisen että 64-bittisen version ohjelmasta sekä PowerPC:lle että Intelille, ja tietyille prosessorityypeille optimoituja versioita.

Universaalibinäärien haittapuoli on ohjelmabinäärien ja kirjastojen koon kasvaminen noin kaksinkertaiseksi yksiarkkitehtuurisiin verrattuna. Yleensä kuitenkin ohjelman arkkitehtuurista riippumattomien resurssitiedostojen kuten kuvien, äänten ja käyttöohjeitten yhteiskoko on suurempi kuin suoritettavan ohjelmakoodin. Näin ollen kaksinkertaisen suureksi kasvanut ohjelmabinääri useimmissa käytännön tapauksissa kasvattaa ohjelman kokonaiskokoa korkeintaan kymmenillä prosenteilla.

2.5.3 Tulkattavat kielet

Ohjelmointikielet jaetaan käännettäviin ja tulkattaviin kieliin. Edellisessä kohdassa mainitut C, C++ ja Objective-C ovat käännettäviä kieliä, jotka käännetään konekieliseen muotoon ennen suoritusta. Tulkattavat kielet kuten Python, Perl ja PHP tulkataan jokaisella suorituskerralla uudelleen. Monet tulkattavat kielet tosin käännetään suorituksen nopeuttamiseksi jollekin välitason kielelle.



Kuva 2-5: Tulkatun Python-ohjelman riippuvuussuhde tulkkiin ja käyttöjärjestelmään.

Koska tulkattavalla kielellä tehty sovellus on aina lähdekoodimuodossa, sovellus ei ole sidottu tiettyyn fyysiseen alustaan. Ensinnäkään ohjelmaa suorittavan koneen prosessorilla ei ole merkitystä, koska ohjelma ei ole konekielisessä muodossa. Toiseksi tulkattava ohjelma ei ole natiivisovellus, joten se ei kutsu käyttöjärjestelmän rajapintoja – ainakaan suoraan. Tulkattavalla kielellä tehdyllä sovelluksella on sen sijaan riippuvuus omaan alustaansa, eli sovelluksen ja käyttöjärjestelmän välissä olevaan tulkkiin tai ajoympäristöön (Kuva 2-5).

Tulkattavalla kielellä tehty ohjelma käyttää alla olevaa käyttöjärjestelmää oman suoritusympäristönsä tarjoaman abstraktion kautta. Koska tähän abstraktiorajapintaan on valittu kaikilla tai useilla käyttöjärjestelmillä saatavissa olevia palveluita, seuraa tästä myös, että suoritusympäristön kautta on mahdollista käyttää vain rajoitettua osajoukkoa

jonkin tietyn käyttöjärjestelmän kaikista rajapinnoista. Tulkatulla ohjelmalla on toisin sanoen natiiviohjelmaa suppeampi, kapeampi näkymä käyttöjärjestelmän tarjoamiin rajapintoihin. Toisaalta tulkatulla ohjelmalla voi olla käytettävissään muita hyödyllisiä korkean tason abstraktioita, joita käyttöjärjestelmä ei tarjoa.

Tulkatulla kielellä tehtyä ohjelmaa on mahdollista laajentaa ja suoritusympäristön rajoituksia kiertää tekemällä ohjelman osalle natiivitoteutus esimerkiksi C-kieltä käyttäen ja kutsumalla tätä tulkattavasta ohjelmasta. Natiivitoteutus voidaan tehdä eri alustoja varten erikseen.

Tulkattujen kielten perusidea ei ole kaukana seuraavaksi esiteltävästä tekniikasta, virtualisoinnista.

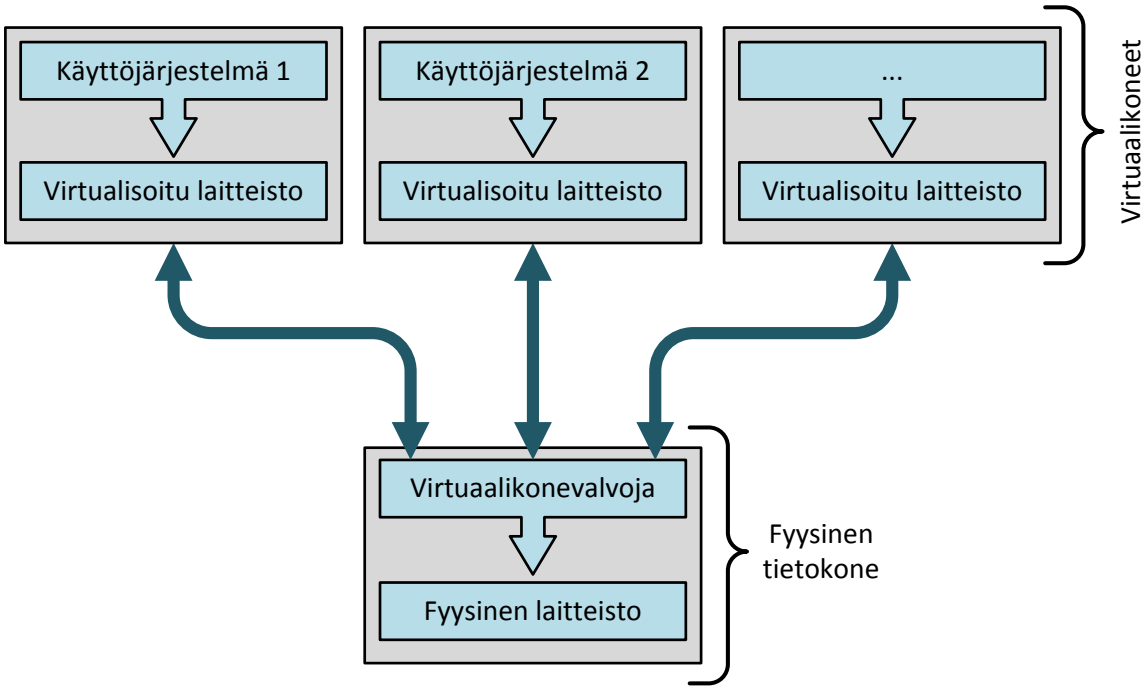
2.5.4 Virtualisointi

Virtualisointi (*virtualization*) on jo 1960-luvulla suurtietokoneissa käytetty tekniikka, joka on 2000-luvulla saavuttanut suuren suosion myös PC-pohjaisissa palvelimissa ja työpöytäkäytössä. Pilvipalveluitten tarjoamiseen keskittyvät palvelinkeskukset ovat nykyään yksi tärkeä virtualisoinnin käyttökohde.

Virtualisoinnin idea on, että suoritettavia ohjelmia ei suoriteta suoraan fyysisellä laitteistolla, vaan virtuaalikoneessa, joka ohjelman kannalta toimii juuri samoin kuin fyysinen laitteisto. Tämä ylimääräinen abstraktiokerros mahdollistaa muun muassa toisistaan täysin eristettyjen ohjelmien suorittamisen samalla fyysisellä koneella.

Virtuaalikoneita on kahden tyyppisiä: *järjestelmävirtuaalikoneita* (*system virtual machine*) ja *prosessivirtuaalikoneita* (*process virtual machine*) [3, s. 10–11]. Järjestelmävirtuaalikoneet ovat virtualisoinnin alkuperäinen, perinteinen muoto.

Järjestelmävirtuaalikoneet tarjoavat suoritusympäristöksi kokonaisen virtuaalisen tietokoneen, jossa suoritetaan käyttöjärjestelmää samaan tapaan kuin fyysisessä tietokoneessa. Järjestelmävirtuaalikoneita käytetään erityisesti yhden fyysisen koneen resurssien jakamiseen turvallisesti usean virtuaalikoneen kesken. Virtuaalikoneet ovat toisistaan eristettyjä, joten koneilla ei ole pääsyä esimerkiksi toistensa tiedostoihin. Niissä voidaan myös suorittaa eri käyttöjärjestelmiä tai käyttöjärjestelmäversioita toisistaan riippumatta. [3, s. 18]



Kuva 2-6: Fyysinen laitteisto, jossa suoritettava *virtuaalikonevalvoja* (*virtual machine monitor*) jakaa resursseja virtualisoiduissa laitteistoissa (virtuaalikoneissa) toimivien käyttöjärjestelmien kesken.

Yleensä järjestelmävirtuaalikoneet tarjoavat suoritusympäristöksi alla olevan fyysisen koneen kaltaisen virtuaalisen koneen, joka käyttää samaa konekielistä käskykantaa kuin fyysinen prosessori [3, s. 19]. Tästä on suorituskyvyn kannalta suurta etua, sillä osa virtuaaliselle prosessorille annetuista käskyistä voidaan suorittaa sellaisinaan fyysisellä prosessorilla. Jos taas fyysinen prosessori käyttää erilaista, epäyhteensopivaa käskykantaa, virtuaalisen prosessorin toiminta joudutaan kokonaan *emuloimaan*.

Prosessivirtuaalikoneet tarjoavat yksittäiselle prosessille eli yksittäiselle sovellusohjelmalle virtualisoidun ympäristön, jossa ohjelmalla on käytettävissään tunnettu rajapinta [3, s. 13]. Toisin kuin järjestelmävirtuaalikoneet, prosessivirtuaalikoneet suorittavat yleensä ohjelmia, jotka käyttävät jotakin muuta kuin suorittavan fyysisen laitteiston käyttämää konekielistä käskykantaa. Ohjelmien käyttämä käskykanta voi olla joko jonkin todellisen, olemassa olevan prosessorin käskykanta tai varta vasten virtualisointiin suunniteltu käskykanta.

Kun Apple vuonna 2006 luopui IBM:n PowerPC-prosessorien käytöstä Macintosh-tietokoneissa ja alkoi käyttää Intelin i386-käskykantaa käyttäviä prosessoreita, vanhojen PowerPC:lle tehtyjen sovellusten toiminta uusissa Intel-koneissa turvattiin suorittamalla PowerPC-ohjelmabinäärit eräänlaisessa prosessivirtuaalikoneessa. Tämä loppukäyttäjälle näkymätön virtuaalikone on nimeltään Rosetta, ja se toimii muuntamalla (translate) PowerPC:n konekieltä i386:lla suoritettavaan muotoon. [7, s. 65]

Varta vasten virtualisointiin suunniteltua käskykantaa käyttävät *korkean tason kielen virtuaalikoneet* (*high-level language virtual machine*), jotka ovat prosessivirtuaalikoneiden osajoukko. Nykyaikana tunnettuja ja laajasti käytettyjä korkean tason kielen virtuaalikoneita ovat Sun Microsystemsin/Oraclen Java-virtuaalikone (JVM) sekä Microsoftin .NET-ohjelmistokehykseen liittyvä Common Language Runtime (CLR), joka toteuttaa Common Language Infrastructure -spesifikaation. [3, s. 17]

Molemmat edellä mainitut korkean tason kielen virtuaalikoneet suorittavat omanlaistaan *tavukoodia*, joka on olemassa olevien prosessorien käskykantoja yksinkertaisempaa konekieltä. Tämä tavukoodi on alustasta riippumatonta, joten samaa tavukoodia voidaan suorittaa kaikilla alustoilla, joille tarvittava virtuaalikone – JVM tai CLR – on olemassa [3, s. 17]. On syytä korostaa, että JVM ja CLR eivät käytä samaa, yhteensopivaa tavukoodin käskykantaa, vaikka käyttötarkoitus onkin samankaltainen.

2.5.5 Web

Web-pohjaiset palvelut ovat eräänlainen tapa toteuttaa monialustaisuutta. Web-selain toimii alustariippumattomana työkaluna graafisten käyttöliittymien esittämiseen sekä eräänlaisena virtuaalikoneena JavaScript-kielelle. Toisin kuin edellä esitetyt ratkaisut, web-selain ei ole kokonainen ratkaisu monialustaisten ohjelmien toteuttamiseen, sillä selaimessa suoritettavan käyttöliittymäosan lisäksi tarvitaan palvelimessa suoritettava taustaohjelma (back end).

Erilaisten web-selainten suuren määrän ja niiden välisten pienten toiminnallisuuserojen vuoksi web-selaimen tarjoama ympäristö ei ole yksi yhtenäinen ja ongelmaton alusta käyttöliittymien toteuttamiseen. Selainten toiminnallisuuserojen vuoksi kohdataan

samoja ongelmia ja ratkaisuja kuin monialustaisten natiiviohjelmien kanssa, eli alustojen eroja piilotetaan käyttäen kirjastojen tarjoamaa abstraktiota yksityiskohdille, jotka muuten täytyisi toteuttaa erikseen eri selainversioita varten. Joissakin käyttötarkoituksissa voidaan tosin selaimen ja selainversion olettaa pysyvän aina samana, jolloin toiminnallisuuserot voidaan jättää huomioimatta, mutta ratkaisu on vähemmän monialustainen.

Web-selaimella toteutettu käyttöliittymä saa jätettyä graafisen käyttöliittymän ikkunoiden ja komponenttien luomisen ja näyttämisen kokonaan selaimen tehtäväksi. Tällä tasolla tarkasteltuna web-selain on toimiva ja alustariippumaton tapa toteuttaa tietyntyyppisiä sovelluksia, siitäkin huolimatta että matalammalla tasolla joudutaan ratkomaan selainten välisiä yhteensopivuusongelmia.

Kuten edellä todettiin, web-selain ei ole kokonainen ratkaisu monialustaisten ohjelmien toteuttamiseen. Web-palvelimella suoritettava taustaohjelma suorittaa toteutuksesta riippuen pienen tai suuren osan käyttöliittymän ulkopuolisesta toiminnallisuudesta, mutta ilman taustaohjelmaa web-palvelu ei voi toimia. Web-selaimen rajapinta toiminnan kannalta välttämättömään taustaohjelmaan on verkko, käytännössä HTTP-protokolla. Web-selaimessa toimiva ohjelman käyttöliittymäosa saa tämän yhden rajapinnan kautta kaikki tarvitsemansa palvelut web-palvelimella toimivalta taustaohjelmalta. Samalla monialustaisuuden ongelmat siirretään ratkaistavaksi palvelimen päässä, sillä taustaohjelma itse on tavallinen ohjelma, jossa mahdolliset monialustaisuuden tarpeet on ratkaistava samoin kuin ohjelmissa yleensä, esimerkiksi jollakin aiemmissa alaluvuissa esitetyllä tavalla.

Toisenlaisesta näkökulmasta katsottuna web-pohjainen käyttöliittymä ei ole monialustainen, sillä se on vahvasti sidottu juuri web-alustaan. HTML:ää ja JavaScriptiä käyttäen toteutettu web-käyttöliittymä ei ole helposti muunnettavissa esimerkiksi tavalliseksi graafisen natiiviohjelman käyttöliittymäksi.

2.6 Monialustaisuuden rajat

Edellä esiteltiin ratkaisuja monialustaisten ohjelmien tekemiseen. Vaikka kaikissa ratkaisumalleissa yritetään alustojen eroja piilottaa erilaisten abstraktiokerrosten taakse,

loppujen lopuksi alustoissa kuitenkin on eroja, joita ei voi täysin piilottaa. Täydelliseen alustariippumattomuuteen pyrkiminen ei siis monesti ole realistista. Tässä alaluvussa luetellaan joitakin ohjelmistojen osa-alueita, joilla alustariippumattomuuteen pyrkiminen ei ole mahdollista tai kannattavaa.

2.6.1 Asennuspaketit

Ohjelmistoja jaellaan loppukäyttäjille yleensä yhdeksi tiedostoksi paketoituna. Tällainen paketoitu tiedosto sisältää ohjelmabinäärin ja sen tarvitsemat resurssitiedostot kuten kuvat, fontit ja mahdolliset dynaamiset kirjastot. Paketointi voi sisältää skriptejä, jotka suoritetaan asennusvaiheessa tai asennusta poistettaessa. Windows-alustalla tällaista paketoitua tiedostoa sanotaan asennusohjelmaksi. Linux-jakeluissa puolestaan saman asian ajavat tyypillisesti – ja jakelusta riippuen – rpm- tai deb-paketit. Mac OS X:ssä käytetään dmg-tiedostoja (disk image), joissa voi olla mukana asennusohjelma.

Paketointitapoja on siis lähestulkoon yhtä monta kuin käyttöjärjestelmiäkin. Tästä seuraa, että sinänsä alustariippumatonkin ohjelma on paketoitava erikseen jokaiselle loppukäyttäjien käyttämälle käyttöjärjestelmälle. Käyttöjärjestelmäkohtainen paketointi on lähes välttämätöntä, jotta ohjelman asentaminen ja käyttöönotto onnistuisi riittävän helposti, kyseiselle käyttöjärjestelmälle ominaisella ja loppukäyttäjien tuntemalla tavalla. Paketointi huolehtii yleensä myös pikakuvakkeen tai vastaavan käynnistystavan lisäämisestä asennettujen ohjelmien luetteloon.

2.6.2 Polut

Ohjelmat, lukuun ottamatta hiekkalaatikkoympäristössä suoritettavia ohjelmia, toimivat käyttäen ympäristönään käyttöjärjestelmän tarjoamaa tiedostojärjestelmää ja sen hakemistorakennetta. Käyttöjärjestelmissä noudatetaan tiedostojen sijainteihin liittyviä määräyksiä, suosituksia ja vakiintuneita käytäntöjä. Nämä määräävät sen, minne ohjelman kuuluu tallentaa esimerkiksi oma asetustiedostonsa tai käyttäjän luomat dokumentit.

Polkukäytäntö vaihtelee käyttöjärjestelmien välillä, ja joissakin vanhoissa järjestelmissä erityistä käytäntöä ei ole. Kun käytäntö on olemassa, tiedostojen tallentaminen vääriin

sijainteihin tekee ohjelmasta hankalan käyttää yhdessä järjestelmän muiden osien kanssa. Tiedostojen tallentaminen mielivaltaisiin, vääriin polkuihin on usein myös mahdotonta puuttuvien kirjoitusoikeuksien vuoksi.

2.6.3 Näyttö ja syöttölaitteet

Työpöytätietokoneiden näytöt ovat keskimäärin huomattavasti suurikokoisempia kuin mobiililaitteiden kuten matkapuhelinten näytöt. Koska näytöt on tarkoitettu ihmissilmin luettaviksi, näytöllä esitettävän tekstin fonttikokoa ei voida rajattomasti pienentää vaikka resoluutio sen sallisikin. Näin ollen suurelle näytölle mahtuu enemmän sisältöä kuin pienelle. Sopivia käyttöliittymäkirjastoja käytettäessä saman ohjelmakoodin hyödyntäminen niin työpöytäkoneessa kuin mobiililaitteessa saattaa olla teknisesti mahdollista. On kuitenkin kyseenalaista, onko suurikokoista näyttöä varten tehty käyttöliittymä kovinkaan usein suoraan käyttökelpoinen paljon pienemmällä näytöllä, tai päinvastoin.

Työpöytätietokoneita käytetään normaalisti hiirellä ja näppäimistöllä, mobiililaitteita taas yhä enenevässä määrin kosketusnäytön avulla. Erityisesti pienelle kosketusnäytölle siirrettävä työpöytäsovellus vaatii käyttöliittymän uudelleensuunnittelua, sillä painikkeiden ja kaikkien muiden käyttöliittymän komponenttien on oltava suurempia ja näkyvämpiä kuin hiirellä käytettävässä työpöytäversiossa. Lisäksi virtuaalinen näppäimistö voi tekstiä syötettäessä peittää puolet näytön pinta-alasta.

2.6.4 Muisti ja teho

Monialustaisen ohjelman muisti- ja tehovaatimukset voivat rajata sitä koneiden joukkoa, millä ohjelman todellisuudessa voi suorittaa. Vaikka jopa älypuhelinten ja taulutietokoneitten kaltaisissa sulautetuissa järjestelmissä on nykyaikana muistia jo satoja megatavuja, on edelleen olemassa myös paljon pienemmällä muistimäärällä varustettuja tietokoneita. Muistin riittävyys on johtanut sovelluksien kehittämiseen sen oletuksen varaan, että muistia voi aina onnistuneesti varata tarpeellisen määrän, eikä muistin loppumiseen tarvitse varautua. Tällä periaatteella kehitetyt ohjelmat toimivat huonosti tai eivät ollenkaan silloin, kun suorittavassa tietokoneessa muistia on vain muutamia kymmeniä megatavuja tai vähemmän.

Myös prosessorien tehoissa on jopa monen kertaluokan nopeuseroja erilaisten tietokoneiden välillä. Nopeus ei sinänsä vaikuta ohjelman suorituksen tulokseen ja sen oikeellisuuteen, mutta moniin interaktiivisiin sovelluksiin liittyy jonkinlainen vaatimus reaaliaikaisuudesta. Käyttäjälle esitettävää ääntä ja videota on esimerkiksi pystyttävä toistamaan reaaliajassa, ilman taukoja. Käyttöliittymissä nopeudelle ei ole näin ehdotonta minimiä, mutta liian hidas käyttöliittymä on epämukava käyttää.

3 Monen alustan ohjelmat C++-kielellä

Tämä luku kertoo ensin C++-kielen taustasta ja kielen suunnittelussa käytetyistä periaatteista. C++ ja C ovat suoritusympäristönsä ja kielen toiminnan kannalta monessa suhteessa niin samanlaisia, että suurin osa käsiteltävistä asioista pätee C++:n lisäksi myös C-kieleen vaikkei tätä erikseen mainittaisi.

Luvun varsinainen tarkoitus on kuvailla millainen kieli C++ on monialustaisten ohjelmien tekemisen kannalta. C++ sisältää piirteitä, jotka edesauttavat monialustaisten ohjelmien tekemistä, mutta toisaalta oikean toiminnan varmistaminen kaikilla alustoilla edellyttää tietyissä asioissa suurempaa huolellisuutta kuin muiden ohjelmointikielien kanssa vaaditaan.

3.1 Standardi C++

C++ on Bjarne Stroustrupin kehittämä yleiskäyttöinen ohjelmointikieli, joka on saanut merkittäviä vaikutteita Simula-nimisestä olio-ohjelmointikielestä [8]. Teknisesti ja syntaksin osalta C++ perustuu ennen kaikkea Brian W. Kernighanin ja Dennis M. Ritchien kehittämään C-kieleen [8]. C-kieli on kehittäjiensä mukaan ”riippumaton mistään tietystä tietokonearkkitehtuurista” [9, s. 3]. Samaa suunnittelun periaatetta on noudatettu myös C++-kielessä.

C++-standardi määrittelee kielen syntaksin ja toiminnan. Standardi määrittelee myös useita tilanteita, joissa ohjelman toiminta on joko kääntäjän toteutuksen määrittelemää (*implementation-defined behavior*) tai määrittelemätöntä (*undefined behavior*) [10, s. 8–9]. Itse kielen lisäksi standardin toinen hyvin tärkeä tehtävä on määritellä kokoelma käytettävissä olevia funktioita ja muita työkaluja. *Standardi mallikirjasto* (*standard template library*, *STL*) sisältää säiliöluokkia ja algoritmeja. Koska C-kieli on lähes C++:n osajoukko, *C-kirjasto* kuuluu C++:aan, sisältäen C-kielen funktiot ja muut määrittelyt. Nämä kaikki yhdessä muodostavat *C++:n standardikirjaston*.

C++:aa koskevasta ISO-standardista ISO/IEC 14882 on kolme versiota: alkuperäinen standardi vuodelta 1998, pieniä korjauksia ja täsmennyksiä sisältävä versio vuodelta 2003 ja merkittävästi laajennettu versio vuodelta 2011. Uusin versio tunnetaan yleisesti

nimellä *C++0x*, johtuen siitä, että standardin toivottiin valmistuvan jo 2000-luvun ensimmäisen vuosikymmenen lopulla. Koska tämä nykyisin C++11:nä tunnettu standardi hyväksyttiin vasta loppuvuodesta 2011, uusimmatkaan kääntäjät eivät vielä tällä hetkellä (2013) toteuta kaikkia C++11:n uusia ominaisuuksia. Usealla erilaisella ja eri-ikäisellä kääntäjällä käännettäväksi tarkoitetussa C++-ohjelmakoodissa onkin toistaiseksi syytä käyttää vasta C++98:n ominaisuuksia.

Edellä mainittu Kernighanin ja Ritchien esittämä periaate tietystä tietokonearkkitehtuurista riippumattomuudesta ei koske ajettavaksi ohjelmabinääriksi käännettyä ohjelmaa. Ne ovat täysin riippuvaisia siitä alustasta, toisin sanoen siitä ABIsta, jolle ne on käännetty. Sen sijaan lähdekoodin muodossa olevalle, standardia noudattaen kirjoitetulle ohjelmalle riippumattomuus tietystä tietokonearkkitehtuurista merkitsee sitä, että ohjelma voidaan kääntää ja suorittaa millä tahansa kääntäjällä ja alustalla siten, että samalla syötteellä käytettynä ohjelma toimii kaikilla alustoilla samalla tavalla.

Standardikirjastoa käyttävällä C++-ohjelmalla on varsin rajallisesti keinoja olla vuorovaikutuksessa ympäristönsä kanssa. Ohjelmalla on vakiosyötevirta (*standard input*), jota se voi lukea, ja kaksi vakiotulostevirtaa (*standard output* ja *standard error*), joihin se voi kirjoittaa. Tyypillisessä interaktiivisessa käyttötapauksessa syöte tulee käyttäjän kirjoittamana näppäimistöltä, ja tuloste näytetään tekstinä kuvaruudulla. Standardi ei kuitenkaan määrittele sitä, mihin fyysisiin laitteisiin syöte- ja tulostevirrat liittyvät – jos mihinkään. Syöte voi esimerkiksi tulla suoraan toiselta ohjelmalta, ja tuloste voi päätyä sarjaporttiin.

Syöte- ja tulostevirtojen lisäksi C++-ohjelma voi vaikuttaa ympäristöönsä tiedostoja lukemalla ja kirjoittamalla sekä niitä luomalla ja poistamalla. Ohjelma voi myös selvittää järjestelmän kellonajan ja päiväyksen, lukea ympäristömuuttujien arvoja tai suorittaa järjestelmässä komennon. Tämän läheisempää pääsyä ohjelmalla ei standardikirjastoa käyttäen käyttöjärjestelmän palveluihin tai laitteistoon ole.

Voidaan siis todeta, että C++ sinänsä on alusta-agnostinen ohjelmointikieli. Tämä ilmenee myös siinä, että standardi ei juuri aseta rajoja ja vaatimuksia laitteistolle. Esimerkiksi muistin määrälle tai prosessorin suorituskyvylle ei ole määriteltyä alarajaa,

eikä myöskään ylärajaa. Minkään tietyn laitteiston osan tai oheislaitteen olemassaoloa ei oleteta eikä vaadita. Käyttöjärjestelmästä oletetaan vain sen verran, että se jollain tavalla toteuttaa tiedostojen käsitteen.

Käytännössä todellisten ohjelmien usein halutaan tekevän sellaisia asioita, jotka standardikirjastoa käyttäen eivät ole mahdollisia. Niinkään tavanomainen toiminto kuin hakemiston sisältämien tiedostonimien selvittäminen ei pelkkää standardikirjastoa käyttäen ja alustariippumattomasti onnistu lainkaan. Muita hyvin tavallisia ja tarpeellisia, mutta standardikirjastolla mahdottomia toimenpiteitä ovat muun muassa kaikenlainen grafiikan piirtäminen kuvaruudulle, äänentoisto, verkkoliikenne ja syötteen lukeminen oheislaitteilta kuten hiireltä tai peliohjaimelta. Mahdottomasta tulee mahdollista, kun standardikirjaston lisäksi otetaan käyttöön muita, standardiin kuulumattomia implementaation tai kolmannen osapuolen tarjoamia kirjastoja [11, s. 45].

3.2 Kirjastot

Kirjasto on kokoelma muuttumattomana pysyvää ohjelmakoodia, jota kutsutaan tunnetun *APIn* (*Application Programming Interface*) kautta. Jokainen kääntäjätoteutus tarjoaa C++-standardikirjaston lisäksi myös alusta- ja käyttöjärjestelmäkohtaisia järjestelmäkirjastoja sekä laajennoksia standardikirjastoon. Näiden yhtenä tärkeänä tarkoituksena on tarjota dokumentoitu rajapinta ja abstraktio käyttöjärjestelmän palveluille. Win32-API on esimerkki hyvin laajasta lisäkirjastosta, jonka kautta Windows-alustalla hoidetaan niin graafinen käyttöliittymä, äänentoisto kuin verkkoliikennekin – ja monta muuta asiaa.

Lähes kaikkien kolmannen osapuolen (eli muiden kuin alustan mukana tulevien) kirjastojen olemassaolon tarkoituksena on koota samaan aiheeseen liittyvää yleiskäyttöistä koodia yhdeksi moduuliksi, ja näin edesauttaa sitä, ettei jokaisen ohjelmistoprojektin tarvitse toteuttaa samoja asioita uudelleen. Kirjastot voivat olla suljettua tai avointa lähdekoodia, maksullisia tai maksuttomia.

Ohjelmien siirrettävyyden kannalta kiinnostavia ovat kirjastot, joiden tarkoituksena on abstrahoida jokin käyttöjärjestelmäkohtaisen kirjaston tarjoama palvelu ja tarjota

toteutus useille eri käyttöjärjestelmille yhden ja saman APIn kautta. Tällaisesta kirjastosta voi olla hyötyä myös yhdelle alustalle kehitettäessä, jos kirjaston tarjoama uusi rajapinta on yksinkertaisempi ja helppokäyttöisempi kuin alkuperäinen.

Monialustaisuuden edistäminen voi olla kirjaston tarkoitus, mutta usein ei ole. Kirjasto voi ilman erityistä tavoitetta olla alustariippumaton myös siksi, että se käyttää vain standardikirjastoa, tai on täysin itsenäinen eikä siten hyödynnä edes standardikirjastoa.

3.3 Esikääntäjä

C- ja C++-kieliset lähdekooditiedostot käsitellään käännettäessä kahdessa vaiheessa, ja lähdekooditiedostot koostuvat kahta erilaista syntaksia noudattavista osista. Nämä kaksi vaihetta on tosin kääntäjän tasolla automatisoitu niin, ettei kääntäjän käyttäjä normaalisti näitä erillisiä vaiheita havaitse.

Ensimmäisessä vaiheessa lähdekoodi käsitellään *C-esikääntäjällä* (*C preprocessor*), joka käsittelee ja poistaa #-merkillä alkavat rivit. Tällaisia rivejä sanotaan *esikääntäjän direktiiveiksi* (*preprocessor directive*). Direktiivejä käyttäen voidaan liittää muiden tiedostojen sisältöä nyt käännettävään lähdekoodiin, määritellä *makroja* sekä ohjata kääntäjän toimintaa. Esikäännöksen yhteydessä poistetaan myös kommentit.

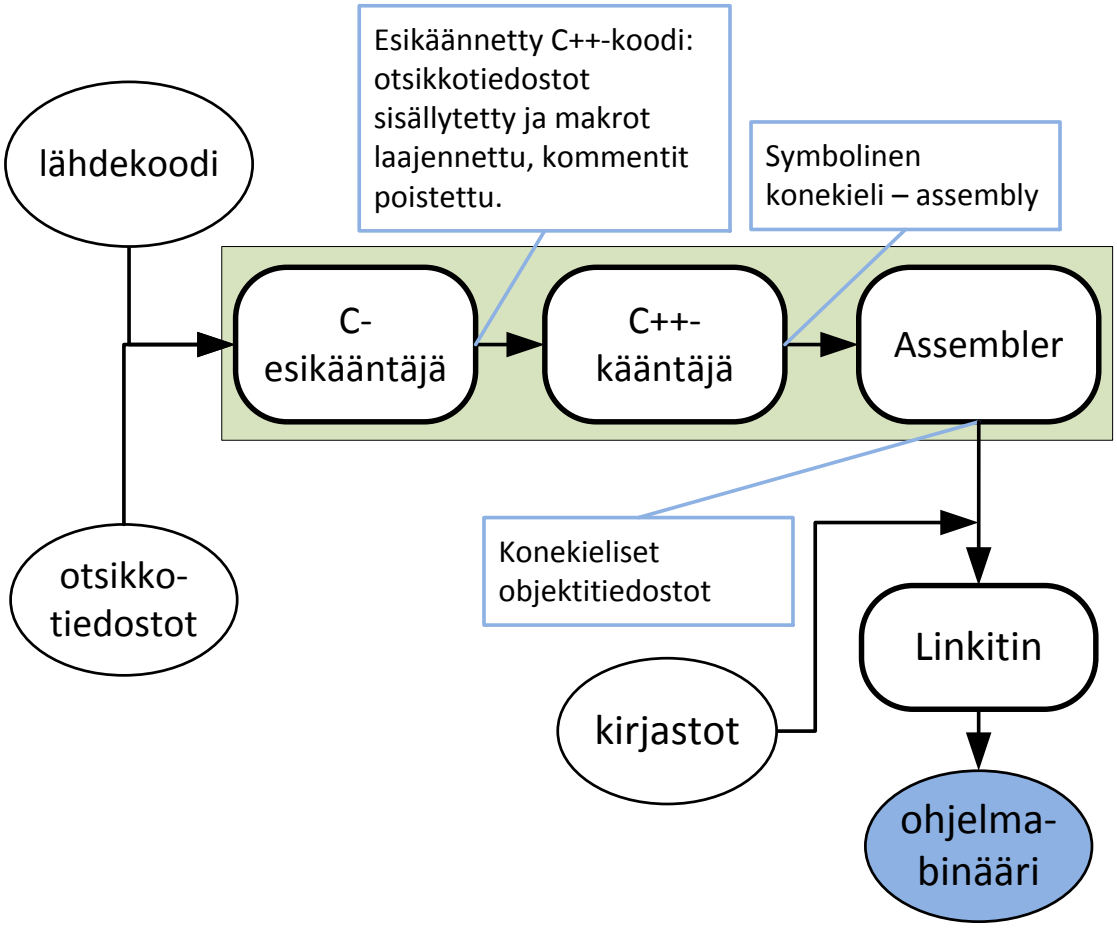
Esikääntäjän direktiivejä hyödynnetään lähes jokaisessa C++-ohjelmassa ainakin hieman. Esimerkiksi rivi

```
#include <ctime>
```

korvataan esikäännösvaiheessa esikääntäjän tuntemassa polussa sijaitsevan ctime-nimisen otsikkotiedoston sisällöllä. Jos ctime-otsikkotiedosto itse sisältää lisää `#include`-direktiivejä, ne käsitellään ja korvataan samalla tavalla. Lopulta lähdekoodi on esikäännetty muotoon, jossa ei enää esiinny #-alkuisia rivejä ja direktiivejä. Tämä esikäännetty lähdekoodi välittyy varsinaiselle C++-kääntäjälle (Kuva 3-1).

Esikääntäjä käsittelee #-alkuiset direktiivinsä, ja vain ne. Esikääntäjä ei jäsennä varsinaista C++-lähdekoodia, joten se ei myöskään havaitse siinä mahdollisesti esiintyviä syntaksivirheitä tai tuntemattomia symboleita. Toisaalta varsinainen C++-

kääntäjä katsoisi #-alkuiset rivit C++:n näkökulmasta syntaksivirheiksi, jos esikäännös jätettäisiin alussa tekemättä.



Kuva 3-1: C++:n käännösprosessi lähdekoodista ohjelmabinääriksi.

Esikääntäjä tekee direktiiveissä määriteltyjen *makrojen* perusteella korvauksia myös tiedoston muihin osiin. Yksinkertaisimmillaan makrolla määritellään nimetty vakio, jonka esiintymät lähdekoodissa korvataan:

```
#define PI 3.1415926
double a = PI * r * r;
```

Tässä tapauksessa makro `PI` korvataan sille määritellyllä arvolla, ja varsinainen C++-kääntäjä tulee näkemään viimeisen rivin muodossa:

```
double a = 3.1415926 * r * r;
```

Makrolle voidaan antaa myös parametreja, ja makron määrittelemä arvo voi koostua myös ohjelmakoodista:

```
#define H(X) do { for(int i = 0; i < X(); i++) h(); } while(0)
```

Esikääntäjän toimintaa voidaan ohjata ehtolauseilla käyttäen direktiivejä `#ifdef`, `#if`, `#elif` ja `#endif`. Kääntäjät määrittelevät sisäisesti makroja, joiden olemassaolosta tai puuttumisesta on pääteltävissä alusta, jolle ollaan kääntämässä. Voidaan määritellä esimerkiksi funktion toteutus kokonaan tai osittain ehdollisesti:

```
void delay(int ms) {
    if (ms <= 0)
        return;
#ifdef _WIN32
    Sleep(ms);
#elif defined(__linux__)
    usleep(1000 * ms);
#endif
}
```

Toteutuvaa ehtolausetta vastaava haara otetaan mukaan esikäännettyyn lähdekoodiin, ei-toteutuvat haarat taas jätetään kokonaan pois. Varsinainen kääntäjä ei siis näe muille alustoille tarkoitettua koodia, joten ehtolauseiden suojaamina esiintyvät tuntemattomat symbolit eivät häiritse kääntämistä. Juuri tämä tekee esikääntäjän direktiiveistä ja makroista käyttökelpoisia monialustaisuuden kannalta, sillä se mahdollistaa eri alustoille tarkoitettujen lähdekoodin osien esiintymisen samassa tiedostossa.

Kuitenkin jo yllä olevan lyhyen esimerkin perusteella on selvää, että ohjelmakoodin luettavuus ja ymmärrettävyys kärsii, jos esikääntäjän ehtolauseita käytetään lähdekoodissa tällä tavalla paljon. Luettavampi ratkaisu onkin määritellä kokonaisia funktioita ensin yhdelle alustalle, sitten toisille. Jos näissä funktioissa esiintyy merkittävästi koodin toistoa, on ylläpidettävyyden kannalta järkevää refaktoroida funktioita niin, että kaikille alustoille yhteinen toiminnallisuus siirretään alustasta riippumattomaan funktioon, joka kutsuu alustakohtaiset osat toteuttavia apufunktioita.

Esikääntäjän ehtolauseiden käyttö voidaan monessa tilanteessa välttää jopa kokonaan hyödyntämällä aliluvussa 2.5.1 mainittua käännettävien tiedostojen valintaa ehdollisesti alustan mukaan. Tällöin otsikkotiedostoon sijoitetaan luokan tai funktioiden määrittelyt

ilman alustakohtaisia riippuvuuksia. Alustakohtainen lähdekoodi sijoitetaan erillisiin toteutustiedostoihin, joita on tarvittaessa yksi jokaista alustaa kohden.

Kun standardin määrittelemän esikääntäjän ominaisuudet eivät riitä, voidaan lähdekooditiedostoissa käyttää monipuolisempaa makrokieltä käsittelemällä lähdekooditiedostot ylimääräisellä ulkoisella esikääntäjällä. Tämä tekniikka ei tosin ole laajasti käytössä muualla kuin Qt-ohjelmistokehyksessä. Qt:ta käyttävä lähdekoodi ei ole tarkalleen ottaen ole C++-kieltä, vaan vaatii ennen kääntämistä käsittelyn Qt:n moc-kääntäjällä (Meta Object Compiler), joka generoi makrojen perusteella C++-koodia.

3.4 Prosessoriarkkitehtuurien erojen huomioiminen

Konekieliset ja symbolisella konekielellä kirjoitetut ohjelmat ovat tiettyyn prosessorin käskykantaan sidottuja. Korkeamman tason kielet kuten C ja C++ piilottavat lähes kaikki prosessoriarkkitehtuurien erot kielen käyttäjän näkyviltä. Toisaalta C ja C++ ovat kuitenkin laitteistoläheisiä kieliä, joilla voi käsitellä kaikkea ohjelman käytössä olevaa muistia suoraan, tavu tavulta. On muutamia juuri muistinkäsittelyyn liittyviä prosessoriarkkitehtuurikohtaisia ominaisuuksia, joita kielen ja kääntäjän tuoma abstraktio ei täysin piilota.

Kaikki seuraavaksi esiteltävät ominaisuudet ovat esimerkkejä C++-standardin sallimasta kääntäjän toteutuksen määrittelemästä toiminnasta. Näiden ominaisuuksien huomiotta jättäminen voi aiheuttaa ohjelmakoodiin piileviä potentiaalisia vikoja, jotka aiheuttavat virheellistä toimintaa vasta kun ohjelma käännetään ja suoritetaan toisenlaisella prosessorilla. Erityisesti ensimmäisenä esiteltävä tavujärjestys on seikka, joka koskettaa hyvin monia käytännön ohjelmia.

3.4.1 Tavujärjestys eli endianness

Tietokoneen muistin pienin atominen yksikkö on *bitti* (*bit*), jolla on kaksi mahdollista arvoa: 0 ja 1. Prosessorin ja ohjelmointikielten kannalta kuitenkin muistinkäsittelyn pienin yksikkö on *tavu* (*byte*), joka koostuu useimmiten kahdeksasta bitistä, ja näin ollen tavuun voidaan tallentaa 2^8 eri arvoa, toisin sanoen lukuarvot 0x00 – 0xFF. Muistiosoitteet ja muistiosoitteita säilövä tietotyyppi *osoittimet* ovat kokonaislukuja,

joiden arvo kuvaa tiettyä kohtaa tietokoneen muistissa. Muistiosoitteet viittaavat yhden tavun pituisiin alueisiin muistissa.

Prosessorit käsittelevät kuitenkin kokonaislukuja suurempina kuin yhden tavun kokonaisuuksina, ja esimerkiksi 32-bittisen kokonaisluvun tallentamiseen käytetään neljä peräkkäistä tavua muistissa. *Tavujärjestys* eli *endianness* on prosessoriarkkitehtuurin ominaisuus, joka määrää missä järjestyksessä kokonaisluku näihin neljään tavuun kirjoitetaan ja vastaavasti missä järjestyksessä luetut tavut tulkitaan.

Nykyisin [1] käytössä on kaksi erilaista tavujärjestystä: *big endian* ja *little endian*. Little endian -tavujärjestystä käytetään muun muassa kaikissa Intel x86:n kanssa yhteensopivissa prosessoreissa, mistä johtuen little endian on hyvin yleinen tavujärjestys. Big endian -tavujärjestystä käyttäviä prosessoriperheitä puolestaan ovat esimerkiksi PowerPC ja muut POWER-arkkitehtuurin prosessorit sekä SPARC, ARM ja MIPS.

Big endian -prosessorit pystyvät usein toimimaan myös little endian -tilassa, joskin tavujärjestys on muun laitteiston tasolla kiinnitetty jompaankumpaan tilaan, eikä prosessori pysty suorittamaan yhtä tavujärjestystä varten käännettyjä binääreitä toisessa tilassa ollessaan. Etenkin ARMia ja MIPSiä käytetään yleisesti myös little endian -tilassa.

Big endian -tavujärjestys on työpöytätietokoneissa nykyään hyvin harvinainen, mutta hyvin yleinen muun muassa pelikonsoleissa. Nintendo Wii, Xbox 360 ja Sony PlayStation 3 käyttävät kaikki POWER-arkkitehtuuriin perustuvaa prosessoria big endian -tilassa.

Big endian -tavujärjestyksessä eniten merkitsevä tavu on muistissa ensimmäisenä, sen jälkeen tulee toiseksi merkitsevin tavu, ja niin edelleen (Kuva 3-2). Järjestys on siis samanlainen kuin mihin arkielämässä kymmenjärjestelmän kanssa on totuttu. Little endian -järjestys on tasan päinvastainen; vähiten merkitsevä tavu on muistissa

[1] Lisäksi on olemassa *middle endian* -tavujärjestys, jota käyttäviä prosessoreita ei ole valmistettu vuosikymmeniin.

ensin, eniten merkitsevä viimeisenä (Kuva 3-3). Tavujärjestyskäytäntö koskee 16-, 32- ja 64-bittisiä kokonaislukutyyppejä; sen sijaan yksittäisellä tavulla eli 8-bittisellä kokonaisluvulla on vain yksi mahdollinen tulkinta.

Muistiosoite	**1000**	**1001**	**1002**	**1003**
Sisältö	0x12	0x34	0x56	0x78

Kuva 3-2: Esimerkki: 32-bittinen eli 4-tavuinen kokonaisluku 0x12345678 tallennettuna *big endian* -tavujärjestyksessä alkaen muistiosoitteesta 1000.

Muistiosoite	**1000**	**1001**	**1002**	**1003**
Sisältö	0x78	0x56	0x34	0x12

Kuva 3-3: Esimerkki: 32-bittinen eli 4-tavuinen kokonaisluku 0x12345678 tallennettuna *little endian* -tavujärjestyksessä alkaen muistiosoitteesta 1000.

Ohjelma kirjoittaa kokonaislukuarvot muistiin arkkitehtuurille ominaisessa muodossa ja lukee ne samassa muodossa. Tavujärjestys on tällöin toteutuksen yksityiskohta, josta ei tarvitse olla tietoinen. Tavujärjestyksellä kuitenkin on paljon merkitystä silloin, kun dataa käsitellään tavujen tasolla tai data tulee ulkoisesta lähteestä, esimerkiksi tiedostosta tai verkosta. Tavujärjestys vaikuttaa silloin ratkaisevasti datan tulkintaan. [1, s. 82]

Seuraava ohjelmakoodiesimerkki lukee kuvitteellisessa formaatissa olevan kuvatiedoston alusta binäärimuodossa tallennetun kuvan resoluution (eli leveyden ja korkeuden) niitä vastaaviin etumerkittömiin 32-bittisiin kokonaislukumuuttujiin:

```
uint32_t width, height;
FILE *file = fopen("kuva.img", "rb");
fread(&width, sizeof(width), 1, file);
fread(&height, sizeof(height), 1, file);
```

Luettavan kuva.img-tiedoston alkuosan sisältö on kuvailtu oheisessa taulukossa (Kuva 3-4). Taulukko voisi yhtälailla esittää verkkosoketista tai sarjaportista luettua tavujonoa; tiedostosta lukeminen ei ole tässä yhteydessä merkitsevä tekijä.

Sijainti tiedostossa	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Tavun arvo	0x00	**0x00**	0x07	**0x80**	0x00	**0x00**	0x04	**0xB0**
Muuttuja	width	width	width	width	height	height	height	height

Kuva 3-4: Tiedoston ensimmäiset 8 tavua. Muuttuja-rivi kertoo, kumman muuttujan osaksi kyseinen tavu kopioidaan.

Kumpikin neljän tavun ryhmä kopioidaan muistiin sellaisenaan, ja prosessori tulkitsee lukuarvon oman tavujärjestyksensä mukaisesti. Jos ohjelma suoritetaan little endian -tavujärjestystä käyttävällä prosessorilla, width-muuttujan arvoksi tulkitaan silloin 0x**80**07**00**00. Vastaavasti height-muuttujan arvoksi tulee 0x**B0**04**00**00. Näin kuvan resoluutioksi kymmenjärjestelmässä ilmaistuna saadaan 2 147 942 400 × 2 953 052 160, joka on epäilyttävän suuri resoluutio. Big endian -prosessorilla suoritettuna muuttujat `width` ja `height` saavat arvoikseen 0x00**00**07**80** ja 0x00**00**04**B0**, jolloin resoluutio kymmenjärjestelmässä ilmaistuna on 1920 × 1200.

Erilaisten tavujärjestysten vuoksi alustojen välillä siirrettäväksi tarkoitetut binääriset kommunikaatioprotokollat ja tiedostoformaatit normaalisti määrittelevätkin, kumpaa tavujärjestystä käyttäen lukuarvot ilmaistaan. Verkkoprotokollien yhteydessä puhutaan *verkon tavujärjestyksestä* (*network byte order*), johon ja josta kaikki tiettyjä protokollia käyttävät ohjelmat tarvittaessa muuntavat lähettämänsä ja vastaanottamansa datan. RFC 1700 [12] määrittelee verkon tavujärjestykseksi big endianin.

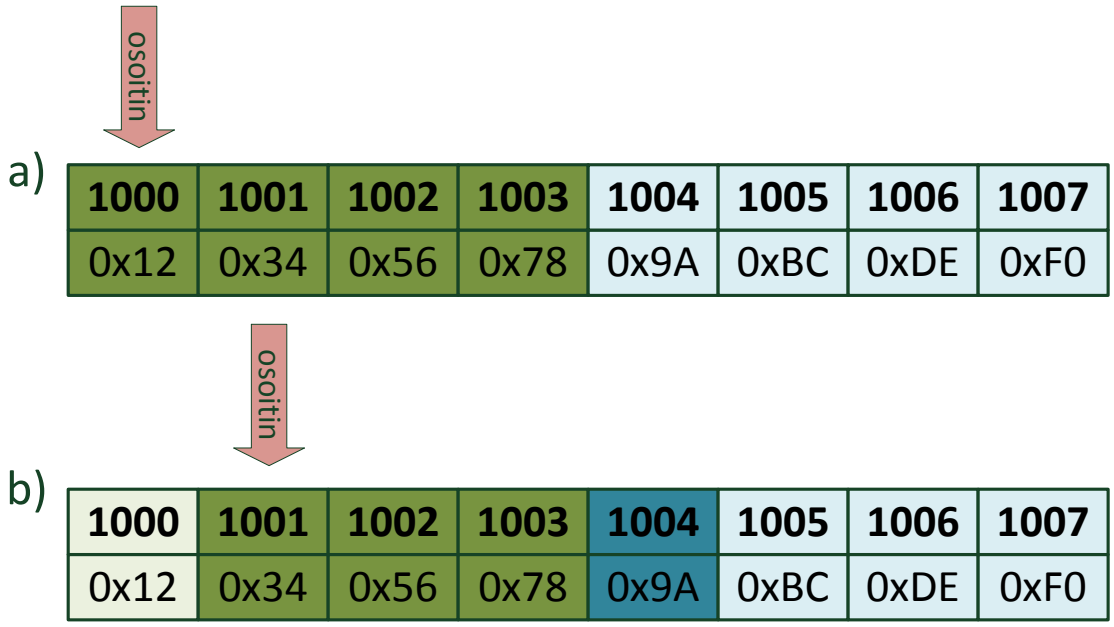
Kun verkkoprotokollan tai tiedostoformaatin käyttämä tavujärjestys ei vastaa ohjelmaa suorittavan prosessorin tavujärjestystä, ongelma voidaan ratkaista kahdella tavalla. Tavujen järjestys voidaan triviaalisti yhden rivin pituisella apufunktiolla kääntää päinvastaiseksi verkosta tai tiedostosta lukemisen jälkeen ja ennen niihin kirjoittamista. Tällainen funktio on toteutettava esikääntäjän makrojen tai ajonaikaisen tarkastuksen avulla siten, että tavujärjestyksen kääntämistä ei tehdä sen ollessa jo valmiiksi oikea. Toinen, periaatteessa hidas mutta prosessorin tavujärjestyksestä täysin riippumaton tapa on tulkita syötettä tavu kerrallaan ja suorittaa lukuarvon tulkintaan tarvittavat kertolaskut (toisin sanoen bittien siirrot) ohjelmakoodissa:

```
char buf[8];
fread(buf, sizeof(buf), 1, file);
uint32_t height =
    buf[0] << 24 | buf[1] << 16 | buf[2] << 8 | buf[3];
```

3.4.2 Kohdistus eli alignment

C++-kielessä käytetään osoittimia osoittamaan varsinaisen datan sisältävään muistiosoitteeseen. 32-bittisellä alustalla osoittimen sanotaan olevan *kohdistettu* (*aligned*), kun se osoittaa neljällä jaolliseen muistiosoitteeseen. Vastaavasti 64-bittisellä alustalla kohdistetun muistiosoitteen on oltava jaollinen kahdeksalla. Käskykantalaajennokset kuten SSE ja AltiVec vaativat käsiteltävälle datalle jopa 16 tavun eli 128 bitin kohdistusta.

Muistin lukeminen ja kirjoittaminen on tehokkainta silloin, kun luettava tai kirjoitettava data sijaitsee kohdistetussa osoitteessa. Tämä johtuu siitä, että prosessori käsittelee muistia yleensä neljän tai kahdeksan tavun kokoisina lohkoina. Jos luettava arvo on kohdistamattomassa osoitteessa (Kuva 3-5), on luettava kaksi lohkoa ja tehtävä tarpeelliset laskutoimitukset varsinaisen arvon selvittämiseksi.



Kuva 3-5: a) Kohdistettu osoitin 32-bittiseen kokonaislukuun. b) Kohdistamaton osoitin 32-bittiseen kokonaislukuun.

Eräillä prosessoriarkkitehtuureilla muistin lukeminen tai kirjoittaminen kohdistamattoman osoitteen kohdalta ei ole vain tehokkuusongelma, vaan kielletty operaatio. Näiden prosessorien suorittamana kohdistamattoman osoitteen käsittely keskeyttää ohjelman suorituksen samaan tapaan kuin nollalla jakaminen tai sallitun muistialueen ylittäminen keskeyttäisi.

3.4.3 Tietotyyppien koot

C++:n ja C:n skalaaristen perustietotyyppien – kuten `char`, `short`, `int` ja `long` – arvoalueille on C-standardissa [13, s. 22] määritelty vain vähimmäiskoot: esimerkiksi `int`-tyyppisen muuttujan on voitava sisältää vähintään lukuarvot [-32767,+32767]. Näiden tietotyyppien todelliset arvoalueet eli bittimäärät ovat kääntäjän ja alustan määriteltävissä. Tästä johtuen `int` on eri alustoilla erikokoinen, joskin hyvin yleisesti 32 bittiä nykyaikaisissa käyttöjärjestelmissä.

Kohdassa 3.4.1 käsiteltiin binäärimuotoisen kokonaislukudatan alustojen välisen siirtämisen ongelmia tavujärjestyksen kannalta. Samankaltaisia datan tulkintaongelmia seuraa myös silloin, kun lähettäjä ja vastaanottaja käyttävät erikokoisia tietotyyppejä eikä tätä ole otettu huomioon.

C++:n rakenteet (*struct*), jotka ovat teknisesti lähes sama asia kuin luokat (*class*), ovat käytännöllinen tapa ryhmitellä toisiinsa liittyviä muuttujia yhdeksi kokonaisuudeksi. Esimerkki, jonka kommentteihin on merkitty kenttien mahdolliset koot tavuina:

```
struct NamedCircle {
    short radius;  // 2 tavua
    int x;         // 4 tavua
    int y;         // 4 tavua
    char name[5];  // 5 tavua
};
NamedCircle c = { 1, 3, 3, "ABC" };
```

Kun rakenne sisältää pelkkiä kokonais- tai liukulukuja kuten yllä, rakennetta vastaava olion sisältö on helppo serialisoida tiedostoon tai verkkoon:

```
write(fd, &c, sizeof(c));
```

Tässä tavassa on kuitenkin monia ongelmia, jotka tekevät serialisoinnin tuloksesta alustariippuvaisen ja siksi kelvottoman alustojen välillä siirrettäväksi. Ensimmäinen ongelma ovat jäsenten tietotyyppien koot, jotka siis voivat vaihdella alustasta toiseen. Toinen ongelma on kääntäjän suorittama kohdistus, jonka kääntäjä saa tehdä valitsemallaan tavalla, tai olla tekemättä. Kohdistuksen tarkoitus on saattaa jokainen rakenteen jäsen kohdistettuun osoitteeseen. Näin $2 + 4 + 4 + 5 = 15$ tavua dataa sisältävä rakenne saattaa olla 15 tavua pitkä, mutta 32-bittisellä alustalla todennäköisemmin 20 tavua, sillä kohdistuksen vuoksi muistiin jätetään käyttämättömiä tavuja ennen x:ää ja olion loppuun, jotta pituus saadaan neljällä jaolliseksi (Kuva 3-6).

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
00 –	radius	radius			x	x	x	x	y	y
10 –	y	y	name	name	name	name	name			

Kuva 3-6: NamedCircle-olion jäsenten mahdollinen sijoittuminen muistiin. Tyhjät ruudut ovat käyttämättömiä täytetavuja.

Serialisointi muistia suoraan kopioimalla on muutenkin huono tapa, sillä se ei toimi oikein jos luokka tai rakenne sisältää osoittimia, virtuaalifunktioita tai muita kuin perustietotyyppejä edustavia jäseniä.

3.5 Kääntäjien erot

C++-kääntäjiä on olemassa useita erilaisia, eri-ikäisiä ja eri valmistajien tekemiä. Vaikka standardia noudattaen kirjoitetun C++-ohjelmakoodin pitäisi ideaalisesti olla käännettävissä kaikilla kääntäjillä, käytännössä näin ei aina ole. Ensimmäinen C++-standardi hyväksyttiin vuonna 1998, mutta vasta kymmenen vuotta myöhemmin uusien kääntäjäversioiden saattoi olettaa ymmärtävän esimerkiksi malleihin (templates) liittyvää syntaksia kunnolla. Mallit otettiin mukaan standardiluonnokseen suhteellisen myöhäisessä vaiheessa standardointiprosessia [8, s. 337].

Erityisesti vanhojen kääntäjäversioiden vuoksi laajaa alustavalikoimaa tavoittelevassa monialustaisessa projektissa voidaan joutua tyytymään C++98:aa suppeampaan standardin C++:n osajoukkoon ohjelmakoodin käännettävyyden ja oikean toiminnan

takaamiseksi. Kielen uudet ominaisuudet saattavat puuttua kääntäjästä, olla toteutettuna vain osittain, toteutus voi sisältää vikoja, tai toteutusta käytetään väärin. [14, s. 54] Uuden C++11-standardin myötä uusien ominaisuuksien rajoitettu saatavuus on jälleen ajankohtainen kysymys myös uusien kääntäjäversioiden kanssa.

C++-kääntäjät toteuttavat yleisesti omia epästandardeja laajennoksiaan C++-kieleen. Laajennokset on tarkoitettu helpottamaan ohjelmakoodin kirjoittamista, ja ne ovat normaalisti käytössä, ellei kääntäjää erikseen kehoteta standardin tarkkaan noudattamiseen. Laajennosten käyttäminen on kuitenkin monialustaisuuden kannalta ongelmallista juuri siksi, että ne ovat standardiin kuulumattomia, eivätkä yleensä toimi kuin yhdellä tietyllä kääntäjällä.

Esimerkkinä hyödyllisestä mutta epästandardista kääntäjän laajennoksesta voidaan käyttää seuraavaa funktiota, joka käyttää GCC:n laajennosta [15]:

```
int f(const char *input) {
    const size_t len = strlen(input);
    int buffer[len];
    // ...
}
```

Yllä oleva esimerkki ei noudata C++-standardia, sillä standardin mukaan `buffer`-taulukolle annettavan pituuden tulee olla käännösaikainen vakio, eli taulukon pituus on tiedettävä jo ohjelmaa käännettäessä. Se ei ole tässä tapauksessa mahdollista, koska muuttujan `len` arvo riippuu funktion saamasta syötteestä.

Kääntäjien välillä on eroja myös siinä, miten tarkasti ne edellyttävät standardia noudatettavan. Yksi kääntäjä saattaa ongelmitta kääntää lähdekoodin osan, jonka toinen havaitsee standardin vastaiseksi ja siten virheelliseksi tai vähintään varoituksen arvoiseksi. Koska kääntäjä siis voi hyväksyä standardia noudattamatonta lähdekoodia joko laajennosten vuoksi tai muuten, yhden kääntäjän kanssa työskentelevä kehittäjä voi huomaamattaan käyttää kirjoittamassaan lähdekoodissa epästandardeja ominaisuuksia.

Kun monialustaista projektia käännetään erilaisilla kääntäjillä säännöllisesti, osana kehitysprosessia, tietyn kääntäjän epästandardeihin laajennoksiin perustuva ohjelmakoodi tulee nopeasti huomatuksi ja korjatuksi. Logan [14, s. 53] luettelee tämän ja muita heterogeenisten kääntäjien käytöstä saatavia hyötyjä: Toinen hyöty ovat

kääntäjien tuottamat erilaiset käännösvaroitukset ja -virheet. Kaikki kääntäjät varoittavat osin samoista mahdollisista ongelmista, mutta jäljelle jää monia ongelmia, jotka yksi kääntäjä löytää ja toinen ei. Usean kääntäjän varoituksista yhdessä saadaan kattavampi kuva ohjelmakoodin mahdollisista ongelmakohdista. Kolmas hyöty puolestaan on, että kääntäjät generoivat erilaista suoritettavaa koodia. Näin esimerkiksi suorituskyvyn pullonkaulat ja kääntäjän toteutuksen määrittelemään toimintaan nojaavan lähdekoodin toiminnallisuuserot tulevat todennäköisesti esille.

4 Case: Monialustainen kameravalvontaohjelmisto

Tämän diplomityön käytäntöä esittelevä osuus käsittelee C++-kielellä toteutettua ja monella alustalla toimivaa Ksenos-kameravalvontaohjelmistoa. Idea aiheelle on lähtöisin pitkäaikaisesta työskentelystä tämän ohjelmiston kehityksen parissa.

4.1 Käyttötarkoitus ja -ympäristö

Kameravalvontaohjelmiston pääasiallinen käyttötarkoitus on videokuvan kaappaaminen valvontakameroilta ja tämän videomateriaalin tallentaminen kiintolevylle myöhempää selailua varten. Ohjelmistoa käytetään digitaalisen kameravalvontatallentimen osana muun muassa kaupoissa, teollisuudessa, matkustajalaivoissa ja muissa ympäristöissä, joissa kameravalvontaa käytetään. Kameravalvontatallentimet ovat normaaleja PC-tietokoneita, joissa voi olla erityisesti valvontakäyttöön tarkoitettuja laajennuskortteja. Yhteen tallentimeen on tyypillisesti kytketty noin 4 – 40 kameraa.

Ohjelmisto on alun perin tehty Windows-alustalle, joskin tuleva tarve Linux-versiolle on jo projektin alkaessa ollut tiedossa. Myöhemmin on toteutettu Mac OS X -versio, jota ei juuri käytetä tallentimissa, mutta sen sijaan tallenninten etäkäyttöön verkon kautta.

4.2 Nykytila ja vaatimukset

Ksenos on kaupallinen ohjelmisto, joka on ollut olemassa vuodesta 2004 lähtien. Sitä on kehitetty useiden vuosien ajan, pienehkön kehitystiimin voimin, ja kehitys jatkuu edelleen aktiivisesti. Ohjelmistossa on kuitenkin jo kauan ollut lähes kaikki ne ominaisuudet, joita suurin osa käyttäjistä tarvitsee ja osaa kaivata. Ohjelmisto on siis sikäli valmis tuote, että se on käyttötarkoitukseensa käyttökelpoinen ja se on tuotantokäytössä sadoissa kohteissa. Siitä ei myöskään puutu olennaisia, kaikkien tarvitsemia ominaisuuksia. Näin ollen kyseessä on todennäköisesti melko tyypillinen pienen yrityksen tekemä ohjelmistotuote.

Ohjelmisto on toteutettu C++-kielellä ja sen laajuus on noin 80 000 riviä yrityksen sisällä tuotettua lähdekoodia. Lisäksi toteutus hyödyntää useita avoimen lähdekoodin kirjastoja sekä joitakin ei-avoimia kirjastoja.

Seuraavalle listalle on valittu seitsemän olennaista kameravalvontaohjelmistoon liittyvää monialustaista tarvetta, joita ei joko voida toteuttaa pelkkää C++:n standardikirjastoa hyödyntäen tai toteutus on niin laaja ja monimutkainen, että ulkoisen kirjaston käyttäminen on välttämätöntä. Tätä tarpeista seuraavaa vaatimuslistaa käytetään apuna seuraavassa luvussa monialustaisia toteutusratkaisuita arvioitaessa. Kaikkia vaatimuksia ei voida pitää tasan yhtä tärkeinä, sillä esimerkiksi äänentoisto on toistaiseksi hyvin vähän käytetty ominaisuus, ja analogisen videokuvan kaappauksen tärkeys on vähenemässä verkkokameroiden yleistyessä.

- **Graafinen käyttöliittymä**
- **Liikkuvan kuvan piirto tehokkaasti**
- **Analogisen videokuvan kaappaus**
- **Videon pakkaus ja purku**
- **Säikeet**
- **Verkkoliikenne**[2]
- **Äänentoisto**

Jokaiselle alustalle ei ole tarpeellista toteuttaa aivan kaikkea toiminnallisuutta. Esimerkiksi videonkaappauskorteille ei ole Mac OS X -ajureita eikä useimmissa Macintosh-tietokoneissa edes ole laajennuskorttipaikkoja, joten analogisen videokuvan kaappaukselle ei Mac OS X -versiossa ole mahdollisuuksia eikä tarvetta.

Listalla olevia tarpeita täytetään kahdella tavalla. Osa listan kohdista on ratkaistu tekemällä kameravalvontaohjelmiston tasolla erilaiset toteutukset eri alustoja varten. Toisissa kohdissa kolmannen osapuolen kirjasto piilottaa kaikki alustojen erot, eikä eri alustoja tarvitse tämän ohjelmiston tasolla erityisesti ottaa huomioon. Jälkimmäisessä tapauksessa kutsut kolmannen osapuolen kirjastoihin on usein vielä piilotettu yhden rajapinnan taakse, jolloin kirjastot ovat ainakin periaatteessa korvattavissa toisilla ilman

[2] Tarvittavia verkkoprotokollia ovat muun muassa TCP, UDP, HTTP, RTSP ja MODBUS.

laajoja muutoksia kaikkeen lähdekoodiin. Tämä ylimääräinen abstraktiotaso auttaa myös tilanteissa, joissa muuten alustojen erot piilottava kirjasto tarvitsee muutaman rivin alustakohtaista lähdekoodia esimerkiksi kirjaston initialisointiin.

Toistaiseksi ohjelmiston käyttöliittymä ja varsinainen tallennustoiminnallisuus – esimerkiksi videon kaappaus, liikkeentunnistus ja tallennus levylle – ovat sidoksissa toisiinsa, ja sovellusta ei voi käynnistää ilman käyttöliittymää edes tallentimissa, joita ei koskaan käytetä paikallisesti. Lähitulevaisuudessa tarjolle tulee nykyversion rinnalle palvelinversio, niin sanottu *demoni* (*daemon*), joka toimii taustaprosessina, ilman käyttöliittymää. Tällaisella versiolla monialustaisia vaatimuksia on vähemmän, koska graafista käyttöliittymää, kuvan piirtoa ja äänentoistoa ei tarvita.

4.3 Toteutusratkaisut

Ohjelmiston nykyisenä toteutuskielenä on C++, tarkemmin sanoen C++98-standardin mukainen C++. Kuten useimmissa käytännön ohjelmistoissa, tässäkin ohjelmistossa on paljon ominaisuuksia, joita ei voi toteuttaa pelkän C++:n standardikirjaston avulla. Näihin ominaisuusvaatimuksiin on vastattu kahdella menetelmällä: osittain kolmannen osapuolen kirjastoja käyttäen, ja osittain tekemällä itse alustakohtaisia toteutuksia ohjelman komponenteille. Periaatteena on ollut hyödyntää samaa ohjelmakoodia kaikilla alustoilla siinä määrin kuin se on mahdollista.

Varsin keskeisessä asemassa nykyisessä toteutuksessa on wxWidgets-ohjelmistokehys, joka on Qt:n ohella toinen vanhoista ja tunnetuista graafisten käyttöliittymien tekoon soveltuvista C++-ohjelmistokehyksistä. wxWidgetsiä käytetään paitsi käyttöliittymän luomiseen myös muuhun. Käyttöliittymien lisäksi wxWidgets sisältää muutakin monialustaisten sovellusten tekemisessä käyttökelpoista toiminnallisuutta, kuten alustariippumattoman abstraktion säikeille.

Toinen tärkeä vaatimus on mahdollisuus piirtää videokuvaa yhtä aikaa useaan ikkunaan riittävän tehokkaasti. Tähän on käytetty OpenGL-grafiikkakirjastoa, jolle saadaan lähes kaikilla näytönohjaimilla laitteistokiihdytys, ja näin muun muassa näytettävien kuvien oikeaan kokoon skaalaamiseen tarvittava laskenta saadaan siirrettyä näytönohjaimen tehtäväksi.

Kaikkein tärkein vaatimus tallenninohjelmistolle on kuitenkin kyky kaapata videokuvaa kameroilta. Valvontakameroita on olemassa kolmenlaisia:

1. Perinteisistä *analogisista kameroista* kuva kulkee PAL- tai NTSC-muotoisena komposiittisignaalina koaksiaalikaapelia pitkin tallentimelle. Analogisia kameroita varten tallentimeen tarvitaan erityisesti kameravalvontakäyttöön tarkoitettu *videonkaappauskortti*, johon voidaan liittää korttimallista riippuen enintään 4, 8, 16 tai 32 kameraa.
2. Yhä enemmän käytetään Ethernet-liitäntäisiä *verkkokameroita* (IP-kameroita). Nämä eivät vaadi tallentimeen fyysisiä lisäosia, mutta käyttöönotto edellyttää yleensä IP-osoitteiden manuaalista asettamista.
3. Uusin tyyppi ovat niin sanotut *HD-SDI-kamerat*, joissa digitaalinen, korkearesoluutioinen kuvasignaali siirretään koaksiaalikaapelia pitkin erityiselle HD-SDI-videonkaappauskortille. HD-SDI:n etu verkkokameroihin nähden on mahdollisuus hyödyntää vanhaa kaapelointia sekä verkkokameroita helpompi käyttöönotto.

Verkkokamerat ovat yleensä monialustaisuuden kannalta helpoimmin käsiteltävä kameratyyppi, koska kaappaaminen ei edellytä lisälaitteita, eikä niin ollen laiteajureita lisälaitteille. Toisaalta muun ohjelmistokehityksen kannalta verkkokamerat ovat työläämpiä, sille ne eivät käytä yhtä tiettyä protokollaa. Nykyisissä verkkokameroissa käytetään enimmäkseen RFC 2326:een perustuvaa RTSP-protokollaa, mutta senkin toteutuksissa esiintyy paljon valmistajakohtaisia pieniä eroja ja epäyhteensopivuuksia.

Analogiset kamerat ja HD-SDI-kamerat ovat monialustaisuuden näkökulmasta samassa asemassa: Molemmat tarvitsevat videonkaappauskortin, joka tarvitsee valmistajan toimittaman laiteajurin ja joskus kirjaston. Jos valmistaja on tehnyt kortilleen vain Windows-ajurin, ja ehkä vain 32-bittisen Windows-ajurin, korttia ei muilla alustoilla voi käytännössä käyttää. Silloinkin kun valmistaja tukee sekä Windows- että Linux-kehitystä, samaa korttia käytetään aivan erilaisen APIn kautta eri alustoilla. Toisinaan tämä johtuu siitä, että valmistaja pyrkii tarjoamaan videonkaappauksen käyttöjärjestelmän oman videonkaappaus-APIn kautta. Joka tapauksessa valmistajan API harvoin tarjoaa käyttöjärjestelmäriippumatonta abstraktiota laitteeseen.

Videonkaappauskortit ovatkin eniten alustakohtaista ohjelmakoodia vaativa osa koko sovelluksessa.

Videon purkaminen ja pakkaaminen on välttämätön osa tallenninta, sillä verkkokameroiden kuva vastaanotetaan valmiiksi pakattuna MPEG-4-, H.264- tai joskus MJPEG-muodossa, ja se on purettava näyttämistä ja liikkeentunnistusta varten. Videonkaappauskortilla kaapattu kuva taas on ennen kiintolevylle tallennusta pakattava, jotta kiintolevytilan käyttö pysyy kohtuullisena. Molempiin tarkoituksiin käytetään avoimen lähdekoodin FFmpeg-projektiin kuuluvaa libavcodec-kirjastoa.

Säikeisiin käytetään wxWidgetsin tarjoamaa wxThread-luokkaa, joka on alustasta riippumaton rajapinta säikeiden käynnistykseen ja suoritukseen. wxWidgets tarjoaa myös säikeiden synkronointiin tarvittavat työkalut.

Verkkoliikennettä käytetään sekä videokuvan vastaanottamiseen verkkokameroilta että tallentimen etäkäyttöön toiselta tietokoneelta. HTTP-protokollalla otettavia yhteyksiä varten käytössä on cURL-kirjasto, RTSP-protokolla puolestaan on toteutettu ilman ulkoisia kirjastoja. Eräisiin muihin tarkoituksiin, esimerkiksi kameroiden automaattiseen hakuun lähiverkosta, käytetään BSD-soketteja ja lähes vastaavia Winsock-soketteja suoraan.

Joissakin käyttötapauksissa tallentimen halutaan tallentavan myös ääntä. Äänen toistamiseen käyttöliittymässä käytetään OpenAL-kirjastoa. OpenAL:n API jäljittelee OpenGL:ää ja sen varsinainen käyttötarkoitus on toistaa kuvitteellisessa kolmiulotteisessa avaruudessa liikkuvien äänilähteiden tuottamaa ääntä tietyssä avaruuden kohdassa olevan kuulijan kannalta oikein. OpenAL on kuitenkin kolmiulotteisten ominaisuuksiensa ohella myös helposti saatavilla oleva ja sopivasti lisensoitu monialustainen kirjasto äänen toistamiseen ilman kolmiulotteisuusefektejä.

Taulukko 4-1 esittää yhteenvedon kameravalvontaohjelmiston lähdekoodissa esiintyvien alustakohtaisten lähdekoodirivien määrästä aiemmin listattujen vaatimusten mukaan luokiteltuna. Näiden luokkien ulkopuolelle jäävät lähdekoodirivit koostuvat sekalaisista pienistä lohkoista eri puolilla lähdekoodia, ei yhdestä suuresta kokonaisuudesta.

Vaatimus	Alustakohtaisia rivejä (noin)
Graafinen käyttöliittymä	200 – 300
Liikkuvan kuvan piirto tehokkaasti	20 – 50
Analogisen videokuvan kaappaus	2 100
Videon pakkaus ja purku	20
Säikeet	0
Verkkoliikenne	50
Äänentoisto	4
Yllä oleviin kuulumattomat	300 – 400

Taulukko 4-1: Alustakohtaisten lähdekoodirivien määrä kameravalvontaohjelmistossa.

Alustakohtaisen lähdekoodin osuus ohjelmiston 80 000 rivin kokonaiskoodimäärästä on siis melko pieni, vain noin 2,5 prosenttia. Merkittäviä määriä alustakohtaista koodia on kolmannen osapuolen kirjastoissa, jotka tarjoamillaan abstraktioilla mahdollistavat yllä esitetyt pienet rivimäärät.

4.4 Käännösympäristö

Kuten aiemmin mainittiin, kameravalvontaohjelmistoa kehitetään kolmelle alustalle. Kullakin alustalla käytetään projektin kääntämiseen alustalle ominaisimpia työkaluja, jotka on lueteltu alla (Taulukko 4-2).

Alusta	Käännettävien tiedostojen hallinta	Kääntäjä
Windows	Microsoft Visual Studio	Visual C++
Linux	käsin kirjoitettu Makefile	GCC
Mac OS X	Apple Xcode	Applen GCC

Taulukko 4-2: Kääntämiseen käytettävät työkalut.

Kun projektiin lisätään tai siitä poistetaan tiedosto, vastaava muutos on tehtävä jokaisen kolmen alustan kohdalla erikseen. Tästä aiheutuu hieman ylimääräistä vaivaa, mutta toimenpide on sinänsä yksinkertainen.

Lisäksi on käännettävä kolmannen osapuolen avointa lähdekoodia olevat riippuvuuskirjastot. Libavcodecin kääntäminen Windowsille on erityisen hankalaa, sillä libavcodec käyttää C99-standardiin kuuluvia C-kielen ominaisuuksia, joita Visual C++:n kääntäjä ei tunne. Siksi libavcodec on myös Windows-alustaa varten käännettävä GCC:llä.

4.5 Nykytoteutuksen vaihtoehdot

Seuraavassa luvussa on tarkoitus arvioida, miten hyvin tai huonosti alaluvussa 2.5 esitetyt ratkaisut soveltuisivat Ksenos-ohjelmiston toteuttamiseen. Nykyisin ohjelmisto on toteutettu C++-kielellä, ja se on siten natiivisovellus kaikilla alustoillaan. Arvioitavana on myös, kannattaisiko nykyinen C++-natiivitoteutus korvata jollakin toisella monialustaisella ratkaisulla. Nykyisen toteutuksen hylkääminen kokonaan ei ole realistinen vaihtoehto, sillä olemassa olevaa ohjelmakoodia on jo paljon, ja kaiken muuntaminen toiselle ohjelmointikielelle sekä testaus vaatisi vähintään kymmeniä henkilötyökuukausia. Saavutettavien hyötyjen pitäisi pystyä kompensoimaan muunnokseen käytetyt resurssit kohtalaisen lyhyellä aikavälillä. Jos natiivitoteutus päätettäisiin korvata jollakin muulla, siirtymän olisi välttämätöntä tapahtua vaiheittain, kirjoittamalla toisistaan erillisiä osia uudelleen kohtuullisen kokoisina, koko sovellusta helpommin testattavina kokonaisuuksina.

Vaihtoehtoisena tapana monialustaisen ohjelmoinnin tehostamiseksi luvussa 6 pohditaan keinoja, joilla olemassa olevaa C++-toteutusta voitaisiin saada parannettua. Monialustaisuuden tehostumisen karkeana mittarina voinee pitää alustakohtaisen ohjelmakoodin määrän vähenemistä. Sovelluksen kääntämiseen tarvittavan ympäristön luominen ja varsinainen kääntäminen vaativat alustakohtaisia skriptejä ja konfiguraatiotiedostoja, joten myös alustakohtaisten ratkaisuiden vähentäminen käännösympäristössä edesauttaisi monialustaista kehitystä.

5 Monialustaisuuden toteutustapojen arviointi

Tässä luvussa käydään läpi alaluvussa 2.5 esitettyjä monella alustalla toimivien ohjelmien toteutustapoja ja arvioidaan tapojen soveltuvuutta tarkasteltavana olevan Ksenos-ohjelmiston toteuttamiseen. Viimeisessä alaluvussa esitetään kootusti arvioinnin tulokset ja pyritään vielä lisää perustelemaan tuloksia oletettujen vasta-argumenttien varalta.

Kameravalvontaohjelmistolla on ehkä osin epätavallisia vaatimuksia toteutettavanaan. Siitä huolimatta erilaisten monialustaisuusratkaisuiden vertailun tulokset ovat todennäköisesti monilta osin yleistettävissä muihinkin samankaltaisia tekniikoita käyttäviin ohjelmistotuotteisiin.

5.1 Arvioinnin perusteet

Alaluvussa 4.2 esiteltiin ohjelmiston monialustaiset vaatimukset: *graafinen käyttöliittymä, liikkuvan kuvan piirto tehokkaasti, analogisen videokuvan kaappaus, videon pakkaus ja purku, säikeet, verkkoliikenne* ja *äänentoisto*. Jokaisessa alaluvun 5.2 kohdassa käsitellään näistä tarpeista seuraavien vaatimusten toteutettavuutta kyseisellä ratkaisutavalla.

Jokaisen kohdan lopussa arvostellaan hieman yleisemmällä tasolla kyseistä ratkaisutapaa perustuen alaluvussa 2.4 esiteltyihin kriteereihin, jotka on myös lueteltu alla (Taulukko 5-1). Nämä kriteerit ja edellä mainitut monialustaiset vaatimukset käsittelevät monialustaisuutta eri kannoilta: kriteerit enemmän ohjelmistotuotannollisia yleisiä seikkoja, vaatimukset juuri tähän ohjelmistoon liittyviä teknisiä seikkoja. Joillakin kriteereistä on silti melko suora vastaavuus monialustaisten vaatimusten listan kohtiin.

Viimeisessä alaluvussa verrataan ratkaisutapoja toisiinsa arvosanojen painotettujen keskiarvojen perustella. Keskiarvon painotuskertoimina käytetään oheisessa taulukossa annettuja arvoja, joilla arvioidaan kyseisen kriteerin olennaisuutta Ksenos-kameravalvontaohjelmiston kannalta.

Viimeisessä alaluvussa näitä painotuskertoimia käytetään painottamaan toteutusratkaisuille kriteerien perusteella annettuja arvosanoja. Näin saadaan toteutusratkaisujen soveltuvuutta tähän käyttötarkoitukseen kuvaavat lukuarvot.

Kriteeri	**Kerroin [0, 1]**	**Perustelu kertoimelle < 1**
1. Tuetut alustat	0,7	Nykyiset alustat riittävät, uusia ei lähitulevaisuudessa tarvita.
2. Pääsy laitekohtaisiin lisäominaisuuksiin	0,6	Verkkokamerat ovat vähentäneet videonkaappauskorttien tarvetta.
3. Pitkän aikavälin käyttökelpoisuus	1,0	
4. Ulkoasu, integroituminen alustaan	0,9	Natiivin näköistä ulkoasua ei tavoitella ohjelmiston kaikkiin osiin.
5. Nopeus, tehokkuus	1,0	
6. Julkaisun helppous	0,5	Julkaisuja tehdään vain 4 – 5 vuodessa, julkaisujen tekoon ei kulu paljon aikaa vuositasolla.
7. Hyöty	1,0	

Taulukko 5-1: Monialustaisuuden kriteerit ja niiden painotuskertoimet perusteluineen Ksenos-ohjelmiston kannalta.

Kriteerien numeeristen arvosanojen asteikko on laskeva 1 – 6, jossa arvosana 1 on paras, 6 huonoin. Heitkötter et al. [6] ei määrittele arvosana-asteikon arvoille mitään tiettyjä edellytyksiä tai arvosteluperusteita. Määritellään siksi tässä kohdassa (Taulukko 5-2)

tarkemmat arvosteluperusteet. Vaikka arvosteluperusteet on määritelty, arvosanat ovat väistämättä aina hieman subjektiivisia.

Arvosana	Arvosanan edellytykset
1	Ratkaisu täyttää kriteerin täydellisesti.
2	Ratkaisu täyttää kriteerin harvoja poikkeustapauksia lukuun ottamatta.
3	Ratkaisu täyttää kriteerin suurelta osin.
4	Ratkaisu täyttää kriteerin pieneltä osin.
5	Ratkaisu ei täytä kriteeriä kuin harvoissa poikkeustapauksissa.
6	Ratkaisu ei täytä kriteeriä lainkaan.

Taulukko 5-2: Kriteerien arvosana-asteikon arvosteluperusteet.

5.2 Monialustaisuuden toteutustavat

5.2.1 Natiiviohjelmat

Kameravalvontaohjelmisto on toteutettu C++-kielisenä natiiviohjelmana, joten natiiviohjelmatoteutustavan soveltuvuudesta on jo empiiristä näyttöä.

Nykyinen toteutus käyttää graafisen käyttöliittymän ja eräiden muiden osien toteuttamiseen wxWidgets-ohjelmistokehystä, joka on käytettävissä Windows-, Linux- ja Mac OS X -alustoilla. wxWidgets olisi käytettävissä myös muissa Unix-tyyppisissä käyttöjärjestelmissä kuten Solaris ja FreeBSD. wxWidgetsiä käyttävän ohjelmakoodin on tarkoitus toimia samalla tavalla kaikilla wxWidgetsin tukemilla alustoilla. Suurin osa wxWidgetsin käyttöliittymäkomponenteista on toteutettu kunkin tuetun alustan omilla, natiiveilla käyttöliittymäkomponenteilla, millä saavutetaan kaikilla alustoilla alustalle ominainen ulkoasu. Tästä johtuen eri alustoilla käytetään eri toteutuksia wxWidgetsin käyttöliittymäkomponenteista, ja todennäköisesti tämän vuoksi muun muassa erilaiset fonttikoot, käyttöliittymäkomponenttien luontijärjestys ja erilainen tapahtumankäsittely ovat seikkoja, jotka joskus aiheuttavat odottamatonta tai virheellistä toimintaa jollakin alustalla. Yhdellä alustalla kehitetyn ja testatun käyttöliittymän ongelmattomaan toimintaan jollakin toisella alustalla ei voi täydellisesti luottaa; testaus kaikilla alustoilla on tärkeää. Ilmeneviä alustakohtaisia eroja voidaan kompensoida käyttämällä

ehdollisesti käännettävää, C-esikääntäjän `#ifdef`- ja `#if`-direktiiveillä erotettua ohjelmakoodia, joka ottaa huomioon yhtenäiseen toimintaan alustakohtaisesti tarvittavat pienet muutokset.

Liikkuvan videokuvan piirtämiseen käytetään OpenGL-rajapintaa. OpenGL on ensisijaisesti tarkoitettu kolmiulotteisen grafiikan piirtämiseen, mutta sitä käytetään tässä ohjelmistossa hyväksi lähinnä kaksiulotteisesti. Piirrettävät kuvat muunnetaan OpenGL:n tekstuureiksi, ja näytönohjain skaalaa kuvan ikkunaan sopivaksi kuormittamatta tietokoneen omaa prosessoria. OpenGL lienee monialustaisin ratkaisu laitteistokiihdytettyyn piirtoon. On kuitenkin mahdollista, ettei se ole lopulta tehokkain ratkaisu.

Analogista videokuvaa kaapataan videonkaappauskorteilta käyttäen korttivalmistajien tarjoamia kirjastoja, joiden rajapinnat ovat yleensä C-kieltä. Muita vaihtoehtoja näiden korttien käyttämiseen ei ole.

Kameravalvontaohjelmistossa eniten prosessoriaikaa kuluu videon purkuun ja pakkaamiseen. Tarkoitukseen käytettävä libavcodec on avointa lähdekoodia ja se on melko hyvin optimoitu useille erilaisille prosessoriarkkitehtuureille. Yhtä monialustaisia vaihtoehtoja ei ole olemassa edes kaupallisina kirjastoina. Jonkinlaista tehokkuushyötyä voitaisiin saada käyttämällä alustakohtaisesti tarjolla olevaa muuta koodekkikirjastoa, esimerkiksi Applen QuickTimeä. Tämä kuitenkin edellyttäisi ylimääräisen alustakohtaisen lähdekoodin kirjoittamista.

Säikeisiin käytetään wxWidgetsin tarjoamaa wxThread-luokkaa. Jonkin muun säieabstraktion valitseminen käyttöön ei juuri toisi hyötyjä, joskaan ei erityisiä haittojakaan. Tulevaisuudessa, kun kääntäjät sen mahdollistavat, voi olla perusteltua siirtyä käyttämään uuteen C++11-standardiin kuuluvaa säierajapintaa.

Verkkoliikenteeseen ja äänentoistoon liittyvät ratkaisut on kuvattu alaluvussa 4.3, ja voidaan todeta, että ratkaisut ovat toimivia ja riittäviä.

Kriteerit:

1. Natiiviohjelmia voi teknisesti tehdä kaikille alustoille, sekä uusille että vanhoille. Eräillä alustoilla (muun muassa Series 40 ja Windows Phone 7) kolmannen

osapuolen tekemien natiiviohjelmien ja -kirjastojen suorittaminen on enimmäkseen ei-teknisistä syistä estetty. **Arvosana: 2.**

2. Natiiviohjelmilla on paras mahdollinen pääsy laitekohtaisiin lisäominaisuuksiin, koska niillä on käytettävissään kaikki käyttöjärjestelmän laitteistorajapinnat. **Arvosana: 1.**
3. Natiiviohjelmat ovat käyttökelpoisia myös tulevaisuudessa. Vaikka muut ratkaisut ovat yleistyneet, ei ole odotettavissa, että natiiviohjelmien suoritusta alettaisiin laajamittaisesti työpöytäkäyttöjärjestelmissä estää. Toisaalta mobiililaitteissa natiiviohjelmien ajoa voidaan enenevässä määrin haluta estää. **Arvosana: 2.**
4. Natiiviohjelmalla on mahdollisuus integroitua alustaan erittäin hyvin, mutta natiivius sinänsä ei takaa että ohjelma noudattaisi esimerkiksi alustalle ominaisia tiedostopolkuja tai käyttöliittymän suunnitteluperiaatteita. Arvosanaa huonontaa se, että alustaan integroitumisen tekeminen oikein vaatii jonkin verran työtä ja alustakohtaisia toteutuksia, eikä kääntäjä tai suoritusympäristö pakota tai ohjaa oikeisiin ratkaisuihin. **Arvosana: 3.**
5. Natiiviohjelma käyttää prosessoria suoraan, ilman ylimääräisiä abstraktiotasoja, ja sillä on mahdollisuus käyttää prosessorille parhaiten sopivia optimointeja. **Arvosana: 1.**
6. Natiiviohjelmat on käännettävä erikseen jokaista alustaa varten. Kääntäminen on yksinkertaisinta tehdä samaa alustaa edustavassa järjestelmässä. Ohjelmabinäärien ja asennuspakettien luominen yhdellä alustalla toista alustaa varten on varsin hankalaa. Tästä seuraa, että jokaista tuettua alustaa kohden on yksinkertaisinta olla joko fyysinen tai virtuaalikone, jolla kääntäminen ja pakettien luominen suoritetaan. **Arvosana: 4.**
7. Ohjelmien toiminnallisuudesta suurin osa on yleensä alustasta riippumatonta, jolloin alustakohtaiset lähdekoodin osat voidaan eristää omiksi kokonaisuuksikseen. Siksi monialustaisen ohjelman kehittäminen ja ylläpito on vähemmän aikaa vievää kuin erillisten alustakohtaisten toteutusten. Joissakin poikkeustapauksissa ohjelma voi koostua lähes pelkästään alustakohtaisesta lähdekoodista, jolloin monialustaisuudesta ei ole hyötyä. **Arvosana: 2.**

5.2.2 Moniarkkitehtuuriset binäärit

Moniarkkitehtuuriset binäärit ratkaisevat monialustaisuuden ongelmia ohjelmiston paketoinnin ja levityksen tasolla, ollen muuten sama ratkaisu kuin natiiviohjelmat. Kameravalvontaohjelmisto hyödyntääkin moniarkkitehtuurisia eli universaalibinääreitä Mac OS X -versiossaan, tarjoten natiivisti toimivan ohjelman sekä PowerPC- että Intel-prosessoreilla varustetuille koneille. Jos 64-bittinen Mac OS X -versio myöhemmin julkaistaan, se voidaan liittää universaalibinääriin vielä yhdeksi arkkitehtuuriksi.

Koska moniarkkitehtuuriset binäärit tarvitsevat toimiakseen käyttöjärjestelmän tukea, eikä tätä tukea ole muilla käyttöjärjestelmillä olemassa, ratkaisun soveltaminen rajoittuu nyt ja todennäköisesti myös tulevaisuudessa Mac OS X:ään. Tietyllä tavalla samantapaista ratkaisua tosin hyödynnetään Windows-version asennusohjelmassa, joka sisältää sekä 32- että 64-bittisen ohjelmabinäärin ja lisäkirjastot. Asennusohjelma asentaa automaattisesti käyttöjärjestelmän bittisyyttä vastaavan version.

Kriteerit:

1. Moniarkkitehtuuriset binäärit vaativat tukea käyttöjärjestelmältä, ja tukea olemassa vain Mac OS X:ssä. **Arvosana: 4.**
2. Sama kuin natiiviohjelmilla (5.2.1).
3. Universaalibinäärien tuki ei tule katoamaan tulevaisuudessa, mutta niiden hyödyllisyys on pitkään ollut laskussa. PowerPC-koneet ovat nykyään harvinaisia ja moniin tehtäviin liian tehottomia. Lähes kaikki Intel-prosessoria käyttävät Mac-koneet pystyvät suorittamaan 64-bittisiä ohjelmia. **Arvosana: 5.**
4. Sama kuin natiiviohjelmilla (5.2.1).
5. Sama kuin natiiviohjelmilla (5.2.1).
6. Universaalibinäärin tuottaminen on jonkin verran vaikeampaa kuin tavallisen natiivibinäärin. Vaikeuksia ja työtä aiheuttaa erityisesti mahdollisten riippuvuuksiksi tarvittavien kolmannen osapuolen kirjastojen ristiinkääntäminen toiselle arkkitehtuurille. **Arvosana: 4.**
7. Universaalibinäärit käännetään normaalisti automaattisesti molemmille (tai kaikille) arkkitehtuureille. Tämä on käytännöllisempää ja nopeampaa kuin binäärien kääntäminen erikseen, mahdollisesti kahdella eri tietokoneella;

universaalibinääreihin liittyvät käännöstyökalut ovat siten ainakin jossain määrin hyödyllisiä myös kehityksen ja ylläpidon kannalta. **Arvosana: 2.**

5.2.3 Tulkattavat kielet

Yleisesti tunnettuja tulkattavia kieliä on olemassa kymmenittäin, ja monia niistä ei ole edes tarkoitettu laajojen sovellusten tai graafisten käyttöliittymien tekoon. Tämän vuoksi ei ole kannattavaa yrittää arvioida tulkattavien kielten soveltuvuutta kameravalvontaohjelmiston toteutukseen yleisellä tasolla. Sen sijaan valitaan hypoteettiseksi toteutuskieleksi suosittu korkean tason kieli Python, ja arvioidaan sen soveltuvuutta.

Python on laajasti käytetty ohjelmointikieli, jonka ajoympäristö on saatavilla hyvin monelle alustalle. Ajoympäristö on jopa valmiiksi asennettuna Mac OS X:ssä, kaikissa Linux-jakeluissa ja monissa muissa Unix-tyyppisissä järjestelmissä.

Python-asennus sisältää Tkinter-nimisen rajapinnan graafisten käyttöliittymien tekemiseen. Lisäksi on olemassa muita Python-rajapintoja samaan tarkoitukseen, tärkeimpinä wxWidgets-ohjelmistokehystä käyttävä wxPython ja Qt:ta käyttävä PyQt [16, s. 496]. Kaikki kolme ovat alustariippumattomia tapoja graafisten käyttöliittymien toteuttamiseen. Koska kameravalvontaohjelmisto hyödyntää laajasti wxWidgetsiä, wxPython olisi todennäköisesti luontevin vaihtoehto käyttöliittymän toteuttamiseen. wxPythonin API noudattelee hyvinkin tarkasti wxWidgetsin C++-APIa luokkien ja funktioiden nimeämisen suhteen sekä käsitteiden osalta. Näin ollen C++-lähdekoodin muuntaminen Pythoniksi onnistuisi melko systemaattisesti. wxPythonin API olisi myös samasta syystä C++-APIn tunteville kehittäjille valmiiksi tuttu.

Python-ohjelma voi käyttää videokuvan piirtämiseen OpenGL:ää PyOpenGL-*sidonnan* (*binding*) avulla. OpenGL on myös Pythonin tapauksessa alustariippumaton tapa hyödyntää näytönohjaimen laitteistokiihdytystä kuvan piirrossa, ja olemassa oleva OpenGL:ää käyttävä C++-lähdekoodi on suoraviivaisesti muunnettavissa Pythoniksi.

Analogisen videokuvan kaappaukseen Python itse ei tarjoa ratkaisua. Videon kaappaukseen tarkoitetut kolmannen osapuolen kirjastotkin ovat alustakohtaisia, eivätkä ne tue kameravalvontakäytössä tarvittavia videonkaappauskortteja. Pythonille on

kuitenkin mahdollista Python/C-APIn avulla kirjoittaa *laajennosmoduuleita* (*extension module*) C- ja C++-kielillä [17] ja näin päästä käyttämään laitteistoresursseja, joihin Pythonin ajoympäristöstä suoraan ei ole pääsyä. Pythonin laajennosmoduuleita käyttäen videonkaappauskorttien tuki pystytään siis teknisesti toteuttamaan, mutta tällaiset moduulit ovat natiiveja dynaamisia kirjastoja, jotka vaativat toteutuksen jokaiselle tarvittavalla alustalle erikseen. Laajennosmoduuleiden käyttäminen ei näin ollen tuota sellaista monialustaisuushyötyä, että se vähentäisi kehitystyön määrää.

Natiiviversiossa käytettävälle libavcodec-kirjastolle on olemassa Python-sidonta, joten natiiviversiossakin käytettäviä videokoodekkeja voidaan käyttää. Pythonilla toteutettuja videokoodekkeja ei liene olemassa, sillä tulkatut kielet eivät ole paras vaihtoehto hyvin prosessori-intensiivisiin tehtäviin.

Pythonin standardikirjasto sisältää tuen säikeille. Samoin standardikirjasto mahdollistaa sokettien käytön, HTTP-URLien noutamisen ja muita verkkoliikenteeseen liittyviä toimintoja. RTSP-protokollalle ei ole standardikirjaston tasolla tukea, mutta kolmannen osapuolen kirjastoja on olemassa. Pythonin standardikirjastossa ei ole alustariippumatonta tapaa äänen toistamiseen. Useita kolmannen osapuolen ratkaisuita on kuitenkin tähänkin tarkoitukseen olemassa.

Python-toteutus on teknisesti mahdollinen, mutta se tulisi tarvitsemaan avukseen natiivitoteutuksia tietyille ohjelman osille. Näitä osia ovat videonkaappauskorttien käyttäminen, videon pakkaus ja purku sekä muutamat muut paljon prosessoritehoa käyttävät ja siksi suorituskykykriittiset toiminnot.

Kriteerit:

1. Python on saatavissa hyvin monelle alustalle, ja se on valmiiksi asennettuna useimmissa Unix-tyyppisissä käyttöjärjestelmissä. Python ei ole käytettävissä niillä harvoilla alustoilla, joilla Python-tulkkia ei valmiiksi ole ja kolmannen osapuolen natiivikoodin suorittaminen on estetty. **Arvosana: 2.**
2. Python-ohjelmalla on hyvin rajallinen pääsy laitekohtaisiin lisäominaisuuksiin, erikoisten lisälaitteiden käyttäminen tarvitsee avuksi natiivitoteutusta laajennosmoduulin avulla. **Arvosana: 4.**

3. Python on ollut olemassa yli 20 vuotta, ja sen käyttö lisääntyy edelleen. Python alustana ei tule katoamaan tulevaisuudessa. **Arvosana: 1.**
4. Python-ohjelmalla on sopivaa käyttöliittymäkirjastoa käyttäessä mahdollisuus noudattaa natiivikäyttöliittymien ulkoasua. Samoin tiedostopolkuja on mahdollisuus käyttää oikein, mutta tähän ei pakoteta. **Arvosana: 3.**
5. Python on riittävän nopea useimpiin asioihin, mutta kaikkein eniten prosessoritehoa tarvitsevat ohjelman osat kannattanee toteuttaa natiivikirjastoja käyttävinä laajennosmoduuleina. **Arvosana: 3.**
6. Tulkattavia kieliä ei tarvitse kääntää, mutta asennuspakettien teossa on samat ongelmat kuin natiiviohjelmilla. Voi olla tarpeellista asentaa myös Pythonin ajoympäristö tuotteen mukana. **Arvosana: 3.**
7. Pythonin standardikirjasto on laajempi kuin C++:n, ja sillä pystyy tekemään monia asioita, jotka C++:n kanssa tarvitsevat standardikirjaston ulkopuolisia kirjastoja. **Arvosana: 1.**

5.2.4 Virtualisointi

Virtuaalikoneet jaetaan järjestelmävirtuaalikoneisiin ja prosessivirtuaalikoneisiin. Kokonaisia käyttöjärjestelmiä suorittavat järjestelmävirtuaalikoneet ovat varteenotettava vaihtoehto kameravalvontaohjelmiston suorittamiseen palvelinkäytössä ilman paikallista käyttöliittymää silloin kun käytössä ei ole videonkaappauskortteja. Järjestelmävirtuaalikoneissa suoritetaan jonkin alustan natiiviohjelmia fyysisen koneen käyttöjärjestelmästä eristettynä.

Koska eristystä ja hiekkalaatikkoympäristöä ei kameravalvonnan käyttötarkoituksessa kaivata, prosessivirtuaalikoneet ovat muuhun järjestelmään integroituvaksi tarkoitetulle sovellukselle relevantimpi vaihtoehto. Arvioitavaksi tekniikaksi valitaan Java, jonka suoritusympäristö JVM on Oraclen toimittamana saatavilla useille eri käyttöjärjestelmille. Myös Microsoftin .NET-ohjelmistokehykseen liittyvä CLR on samantapainen prosessivirtuaalikone, mutta se on Microsoftin tukemana monialustainen vain erilaisten Windows-versioiden välillä. Avoimen lähdekoodin Mono-projekti on tehnyt muilla alustoilla toimivan vaihtoehtoisen toteutuksen CLR:lle, mutta tällä kolmannen osapuolen toteutuksella ei ole samanlaista virallista asemaa kuin Javan tapauksessa Oraclen JVM:llä on muilla alustoilla.

Liikkuvan kuvan piirtämiseen voidaan käyttää OpenGL:ää myös Javan kanssa. Java-ohjelma voi JNI:n (Java Native Interface) kautta kutsua natiivikoodia. Videonkaappauskorttien käyttö olisi toteutettava tällä tavalla, sillä videonkaappauskorttien APIt ovat C-kieltä.

Javalla toteutettuja videokoodekkeja on olemassa. Ennestään toimivaksi tunnetun libavcodecin käyttäminen JNI:n kautta voi silti olla varmempi vaihtoehto kuin koodekin vaihtaminen uuteen. Javan standardikirjasto on paljon esimerkiksi C++:n standardikirjastoa laajempi, ja sisältää rajapinnat niin säikeiden ja verkon käyttämiseen kuin äänentoistoonkin.

Kameravalvontaohjelmiston toteuttaminen Javan avulla on mahdollista, mutta samoin kuin Pythonin kanssa, Javankin kohdalla joudutaan kirjoittamaan joitakin osia natiivikoodina.

Kriteerit:

1. Oraclen Java on saatavissa monille käyttöjärjestelmille. Kuitenkin vanhoille tai harvinaisille käyttöjärjestelmille on heikosti saatavissa Javan ajoympäristön uusia versioita. Tällä hetkellä (2013) alle neljä vuotta vanha Mac OS X:n versio 10.6 on jo liian vanha Javan uusimmalle versiolle. **Arvosana: 4.**
2. Javalla on rajallinen pääsy laitekohtaisiin lisäominaisuuksiin, ja se tarvitsee lisäkirjastoja tai natiivitoteutusta avuksi. **Arvosana: 4.**
3. Javaa käytetään laajasti yritysmaailmassa (palvelimissa), mutta työpöytäkäytössä Java on muuttumassa harvinaiseksi vähäisten käyttötarkoitusten ja monista tietoturva-aukoista seuranneiden poistosuositusten vuoksi. **Arvosana: 3.**
4. Javassa on vain rajallisesti mahdollisuuksia selvittää oikeat tiedostopolut. SWT:llä tehdyt käyttöliittymät käyttävät natiiveja käyttöliittymäkomponentteja, mikä mahdollistaa ohjelmalle natiivin näköisen ulkoasun. **Arvosana: 4.**
5. Javan nopeus tai hitaus on asia, josta on esitetty paljon vastakkaisia mielipiteitä. Javan väitetään olevan yhtä nopea kuin natiiviohjelmat. Javan suoritusympäristön muistivaatimukset ovat varsin suuret ja JVM:n

käynnistämiseen kuluva aika voi vain hetkellisesti suoritettavien sovellusten kanssa olla suhteettoman suuri. **Arvosana: 2.**

6. Sama ohjelmabinääri toimii kaikilla alustoilla, mutta asennusohjelman paketoinnissa ratkaistavana samat ongelmat kuin muutenkin. Voi olla tarpeellista asentaa Javan ajoympäristö tuotteen mukana. **Arvosana: 2.**
7. Javaa käyttämällä saadaan kolmannen osapuolen kirjastojen määrää hieman vähennettyä verrattuna natiiviohjelmiin. Alustakohtaisen lähdekoodin määrä kameravalvontaohjelmiston tasolla pysyisi likimain samana. **Arvosana: 2.**

5.2.5 Web

Kameravalvontaohjelmistolla on web-käyttöliittymä, mutta se on rajoittuneempi kuin natiivisovellus. Vaikka web-käyttöliittymää kehitettäisiin täydellisemmäksi, suurin osa laskennasta olisi tehtävä palvelinpäässä, sillä ohjelman on toimittava ja tallennettava videota riippumatta siitä käytetäänkö sitä selaimella.

Monipuolisemmalla web-käyttöliittymällä olisi ehkä kysyntää satunnaisessa käytössä. Web-käyttöliittymän toimintaa kuitenkin rajoittaisi se, mitä web-selaimella ylipäätään voi tehdä. Kamerakuvia halutaan isoissa ympäristöissä katsella 2 – 4 fyysisellä näytöllä ja kääntyviä kameroita ohjata ja zoomata ohjaussauvaa (joystickiä) käyttäen. Kumpikaan näistä ei ole web-selaimella mahdollista ilman alustakohtaista selaimen liitännäistä (add-on, plug-in). Web-selain ei ole optimaalinen ratkaisu kymmenien videokuvien yhtäaikaiseen näyttämiseen.

Analogisen videokuvan kaappaus, videokoodekit ja kaikki tallennustoiminnallisuus jää web-palvelun taustaprosessin tehtäväksi. Taustaprosessi voidaan toteuttaa edelleen natiivisovelluksena, tai jollakin muulla toteutustavalla, joista on edellä arvioitu.

Kriteerit:

1. Selaimia on hyvin monelle käyttöjärjestelmälle, mutta uusimpia standardeja kuten HTML5:tä tukevien selainten saatavuus vanhoille ja harvinaisille käyttöjärjestelmille on rajallinen. **Arvosana: 2.**
2. Web-selaimessa toimivalla ohjelmalla ei ole pääsyä laitekohtaisiin lisäominaisuuksiin kuten edellä mainittuun ohjaussauvaan. **Arvosana: 6.**

3. Odotettavissa on, että web-pohjaiset ratkaisut toimivat pitkälle tulevaisuuteen, mutta riskinä että seuraava selainsukupolvi tai mobiililaitteiden selaimet edellyttävät ylimääräisiä muutoksia. **Arvosana: 2.**
4. Ulkoasu on selainikkunan sisällä muokattavissa vapaasti, eikä ole mitään tiettyä ulkoasua, johon pitäisi sopeutua. Web-pohjainen käyttöliittymä ei integroidu ollenkaan hyvin järjestelmän polkujen kanssa, koska web-selain ei voi vaikuttaa esimerkiksi ladattavien tiedostojen tallennuspolkuun. **Arvosana: 5.**
5. Suoritettavan JavaScriptin tehokkuus on riittävä. Verkon aiheuttama viive voi aiheuttaa joissakin tilanteissa haittaavaa hitautta. **Arvosana: 3.**
6. Paketointi on erittäin helppoa, sillä loppukäyttäjää varten asennuspakettia ei tarvitse tehdä lainkaan. **Arvosana: 1.**
7. Web-käyttöliittymällä vältetään alustakohtaisia eroja käyttöliittymän teossa. Kuitenkin selainten erojen huomioiminen mahdollisesti tuo samankaltaisia lisäongelmia kuin natiiviohjelmilla on eri alustojen välillä. **Arvosana: 3.**

5.3 Yhteenveto ja johtopäätös

Seuraavaan taulukkoon (Taulukko 5-3) on kerätty yhteenveto edellä annetuista arvosanoista. Näkyvissä ovat myös alaluvussa 5.1 annetut kertoimet, jotka kuvaavat kriteerin merkittävyyttä Ksenos-ohjelmiston kannalta.

Taulukon alimmalla rivillä lasketut painotetut keskiarvot kuvaavat kyseisen monialustaisen toteutusratkaisun sopivuutta Ksenos-ohjelmiston toteutukseen, ja ne on saatu kertomalla kriteerikohtaiset pisteet kriteerikohtaisilla painotuskertoimilla.

$$sopivuus = \frac{1}{N} \sum_{i=1..N}^{N} (painotuskerroin_i \times arvosana_i)$$

Sopivuuden asteikko on edelleen sama 1 – 6 kuin kriteereillä, joten pienin sopivuusarvo merkitsee parasta. Parhaan tuloksen – 1,67 – saavat natiiviohjelmat. Tämän tuloksen perusteella natiiviohjelma on hyvä ratkaisu kameravalvontaohjelmiston toteutukseen, ja sen kehittämistä kannattaa jatkaa. Natiiviohjelmien hyvään sijoitukseen vaikutti tässä tapauksessa suurelta osin se, että muilla tavoilla toteutettuna kameravalvontaohjelmisto

joka tapauksessa tarvitsisi toteutuksensa avuksi natiivitoteutuksia tietyille komponenteille, kun taas natiiviohjelmien tapauksessa kolmannen osapuolen rajapintoja voidaan käyttää suoraan. Jossakin toisessa ja toisen tyyppisessä, vähemmän natiivitoteutuksia vaativassa sovelluksessa natiiviohjelmat eivät olisi olleet näin vahvoilla.

	Kerroin	**Natiivi-ohjelmat**	**Moniarkk. binäärit**	**Tulkattavat kielet**	**Virtualisointi**	**Web**
Kriteeri 1	0,7	2	4	2	4	2
Kriteeri 2	0,6	1	1	4	4	6
Kriteeri 3	1,0	2	5	1	3	2
Kriteeri 4	0,9	3	3	3	4	5
Kriteeri 5	1,0	1	1	3	2	3
Kriteeri 6	0,5	4	4	3	2	1
Kriteeri 7	1,0	2	2	1	2	3
Sopivuus		1,67	2,30	1,86	2,40	2,57

Taulukko 5-3: Kriteerien perusteella annetut arvosanat sekä Ksenos-ohjelmistoon liittyvä painotuskerroin. Alimmalle riville on näitä tietoja käyttäen laskettu painotettu keskiarvo kullekin monialustaiselle toteutustavalle.

Tulkattavat kielet eli tässä tapauksessa Python ei jää pisteissä kauas natiiviohjelmien taakse. Siksi Pythonin tai muun vastaavan tulkattavan kielen käyttäminen sovelluksen joidenkin komponenttien toteutukseen voisi olla harkitsemisen arvoista. Uuden ohjelmointikielen käyttöönotolle ja olemassa olevien C++-kielellä toteutettujen komponenttien mahdolliselle uudelleentoteutukselle on kuitenkin löydettävä jokin syy, jolla voidaan perustella siihen kuluva aika. Tällaisia syitä voisivat olla muun muassa kehitystyön nopeutuminen korkeamman tason kielen ansiosta tai mahdollisuus muuttaa toteutusta ilman pääsyä käännösympäristöön ja kaikkiin lähdekoodeihin.

Erityisen monialustaisena toteutusratkaisuna yleisesti pidetty Java sijoittui tässä vertailussa huonosti. Tähän olivat syinä se, että sen monialustaisuus perustuu virtuaalikoneeseen, jota harvoissa järjestelmissä on esiasennettuna sekä se, ettei Java-virtuaalikoneen uusia versioita ole saatavissa vanhoille käyttöjärjestelmäversioille.

Uusia Java-versioita varten käännettyjä ohjelmia ei voi suorittaa vanhoissa virtuaalikoneissa, sillä Javan tavukoodi ei ole binääritasolla yhteensopivaa kaikkien vanhempien virtuaalikoneiden kanssa.

Yksinomaan tiettyyn käyttöön tarkoitetussa palvelintietokoneessa Javan ajoympäristön asennus ei ole suuri ongelma, ja käyttöjärjestelmä voidaan valita niin, että riittävän uusi virtuaalikoneversio on sille saatavissa. Suhteellisen suurikokoisen ja paljon vakavia tietoturvaongelmia äskettäin sisältäneen ajoympäristön asentaminen yleiskäyttöiselle työpöytätietokoneelle vain yhden sovellusohjelman vuoksi on vaikeammin perusteltavissa. Kameravalvontakäytössä varsinaista tallennintietokonetta etäkäytetään usein toisella tietokoneella, jolla on muitakin käyttötarkoituksia.

Javan lisäksi toinen yleisesti käytetty prosessivirtuaalikoneratkaisu on .NET-ohjelmistokehykseen liittyvä CLR, jolle kirjoitetaan ohjelmia etenkin C#-kielellä. CLR ei ollut tässä vertailussa mukana muun muassa siksi, ettei se ole varsinainen monialustainen toteutusratkaisu kuin erilaisten Microsoftin alustojen välillä. Kuten kohdassa 5.2.4 todettiin, avoimen lähdekoodin Mono-projektilla ei ole virallista asemaa samalla tavalla kuin Javan kohdalla Oraclen JVM:llä on. Jos CLR:ää ja sen vaihtoehtototeutusta Monoa kaikesta huolimatta arvioitaisiin monialustaisena ratkaisuna, se saisi jonkin verran Javaa paremman tuloksen vähintään kriteereissä 1 ja 3. Suoraan tuettuja alustoja ovat kaikki modernit Windows-versiot, ja Mono mahdollistaa vielä lisää alustoja. Pitkän aikavälin käyttökelpoisuuden tilanne näyttää Javan vastaavaa paremmalta.

Vertailussa huonoimmin sijoittui web-pohjainen ratkaisu, mutta tämä ei tarkoita että web-selain olisi kaikin puolin huono ratkaisu. Satunnaiseen käyttöön ja mobiililaitteilla käytettäväksi kameravalvontaohjelmiston web-käyttöliittymä olisi käytännöllinen. Kameravalvontaohjelmiston käyttöliittymän epäilemättä voi toteuttaa web-pohjaisena, mutta web-selain asettaa tiettyjä rajoituksia sille, mitä web-pohjaiset palvelut voivat tehdä. Web-selaimen sisältö ei varsinaisesti integroidu muuhun alustaan, koska osittain turvallisuussyistä web-selaimessa suoritettavalla sivulla ei ole pääsyä esimerkiksi selainta suorittavan käyttöjärjestelmän tiedostojärjestelmään.

Viimeisen kymmenen vuoden aikana trendinä on ollut toteuttaa perinteisten natiivien työpöytäsovellusten vastineita web-pohjaisina palveluina. Älypuhelinten yleistyttyä trendinä on ollut toteuttaa web-pohjaisten palveluiden vastineita natiiveina mobiilisovelluksina, koska kaivataan parempaa alustaan integroitumista ja parempia vasteaikoja kuin web-selain pystyy tarjoamaan. Myöskään työpöytäsovellusten kohdalla web-selain ei ole automaattisesti paikallisesti suoritettavaa ohjelmaa parempi ratkaisu.

Web-selain alustana on toisaalta yksi alusta, toisaalta monta kymmentä alustaa. Erilaisia selaimia ja selainversioita on maailmassa käytössä suuri määrä. Vanhalle selaimelle tehty palvelu ei välttämättä toimi uudella, ja uudelle tehty ei välttämättä toimi vanhalla. Erojen piilottamiseen sovelluskehittäjän kannalta voidaan käyttää jQueryä tai muita JavaScript-kirjastoja, mutta on vaikeata ennakoida toimiiko nyt tehty toteutus vielä viiden vuoden kuluttua julkaistavissa selaimissa. Jos selainversiota ei voida kiinnittää, on selain alustana paljon epävakaampi eli toiminnallisesti muutosherkempi kuin työpöytäsovellusten käyttämät ohjelmointirajapinnat.

C++ on edelleen käyttökelpoinen toteutuskieli myös uusille ohjelmistoille. Oikein käytettynä sillä voidaan kirjoittaa monialustaista ohjelmakoodia. Toisin kuin esimerkiksi Javan kanssa, monialustaisuus ei koske käännettyä binäärimuotoa. C++-kääntäjiä on kuitenkin saatavissa hyvin monenlaisille alustoille.

6 C++-natiivitoteutuksen parantamismahdollisuudet

Edellisessä luvussa päädyttiin siihen johtopäätökseen, että nykyisen C++-natiivitoteutuksen kehitystä kannattaa jatkaa. Tässä luvussa pohditaan, mitä nykytoteutuksen kanssa voitaisiin tehdä paremmin erityisesti monialustaisuuden kannalta. Samassa yhteydessä käsitellään lyhyesti parantamistapojen merkitystä alaluvussa 4.2 esiteltyjen monialustaisten vaatimusten suhteen.

6.1 Kolmannen osapuolen kirjastot

Ksenos-ohjelmiston graafisessa käyttöliittymässä ja muissakin osissa keskeisesti käytetty wxWidgets on vanha ja tunnettu ohjelmistokehys, mutta sen kehitystyö on täysin vapaaehtoisten varassa. Kukaan ei kehitä wxWidgetsiä täysipäiväisesti työkseen. Näin ollen sen on vaikea kilpailla esimerkiksi uusien ominaisuuksien, uusien alustojen ja dokumentaation määrässä Qt-ohjelmistokehyksen kanssa, jota kehittää suuri yritys yli sadan palkatun työntekijän voimin.

wxWidgetsin valinta kameravalvontaohjelmiston graafisen käyttöliittymän toteutustavaksi oli aikoinaan perusteltua, sillä se oli toiminnallisuudeltaan riittävä, saatavilla tarvittaville alustoille ja ilmaisuudestaan johtuen merkittävästi edullisempi kuin Qt tuolloin. Vuosien saatossa Qt on kuitenkin kehittynyt paljon enemmän kuin wxWidgets, ja lisäksi lisensointi on muuttunut sallivammaksi, mahdollistaen Qt:n maksuttoman käytön lähes kaikkiin käyttötarkoituksiin. Nykyinen kehitystiimi tuntee wxWidgetsin Qt:ta paremmin, mutta yleisesti Qt-taitoisia mahdollisia uusia kehittäjiä on todennäköisesti olemassa enemmän.

wxWidgetsin korvaaminen Qt:lla on siis harkitsemisen arvoista ja siihen on enemmän perusteita kuin aiemmin. Koko ohjelmistokehyksen vaihtaminen toiseen on kuitenkin suuri hanke, joka edellyttää muutoksia lähes jokaiseen projektin lähdekooditiedostoon, koska myös käyttöliittymään liittymättömässä ohjelmakoodissa on monin paikoin käytetty muun muassa wxWidgetsin wxString-merkkijonotietotyyppiä. Muutostyö olisi todennäköisesti tehtävä yhdellä kerralla, sillä wxWidgetsin ja Qt:n luomia käyttöliittymän osia ei voi käyttää samassa sovelluksessa ainakaan kohtuullisella

vaivalla, erityisesti sen vuoksi että molemmat ohjelmistokehykset tarvitsevat ohjelman pääsilmukan haltuunsa tapahtumienkäsittelyä varten.

Qt:n käyttö vaikuttaisi luonnollisesti vaatimuslistalla olevan graafisen käyttöliittymän toteutukseen ja jossain määrin ulkonäköön. Samoin wxWidgetsin säieabstraktio vaihtuisi Qt:n vastaavaan. Qt on wxWidgetsiä laaja-alaisempi ohjelmistokehys, ja kameravalvontaohjelmiston vaatimuslistalla olevista kohdista verkkoliikenteen ja äänentoiston nykyiset toteutukset olisivat korvattavissa Qt-toteutuksella. Tämä lisäisi riippuvuutta yhden tietyn valmistajan ohjelmistokehyksestä, mutta toisaalta vähentäisi alustakohtaisen ohjelmakoodin sekä tarvittavien ylimääräisten kirjastojen määrää, ja siten yksinkertaistaisi ohjelmiston käännösympäristöä hieman.

Videonkaappauskorttien eli analogisen videokuvan kaappauksen tuki on tällä hetkellä eniten alustakohtaista lähdekoodia vaativa komponentti koko ohjelmistossa. Kolmannen osapuolen abstraktiokirjastoa videon kaappaamiseen erilaisilta korteilta ei ole olemassa, ja siksi videonkaappaukseen liittyvästä alustakohtaisesta lähdekoodista ei voida päästä kokonaan eroon. Eräs mahdollisuus on siirtää kaikki videonkaappauskoodi erilliseen taustalla toimivaan ohjelmaan, joka syöttää videokuvaa varsinaiselle kameravalvontaohjelmistolle jollakin prosessien välisen kommunikaation (*IPC, inter-process communication*) tekniikalla; esimerkiksi soketteja käyttäen.

Tällaisella videonkaappausohjelmalla olisi vain yksi tehtävä, ja voidaan perustellusti olettaa, että kaappausohjelman ja kameravalvontaohjelmiston rajapinta pysyy vakaana, eikä kaappausohjelmaa siten tarvitse muuttaa ja päivittää muuta ohjelmistoa päivitettäessä. Tälläkin tavalla alustakohtaisten osien määrä varsinaisessa kameravalvontaohjelmiston lähdekoodissa vähenee ja kameravalvontaohjelmiston kääntämiseen tarvittavaan ympäristöön riittää aiempaa pienempi määrä kirjastoja ja niihin liittyviä otsikkotiedostoja.

6.2 C++11

Uusi C++11-standardi sisältää kymmeniä merkittäviä uudistuksia [18], jotka muun muassa lisäävät C++-kielen ilmaisuvoimaa; sama asia saadaan tehtyä pienemmällä määrällä ohjelmakoodia kuin standardin vanhemman version, C++98:n, ominaisuuksia

käyttämällä. Ilmaisuvoimaa lisäävistä uudistuksista ehkä tärkeimmät ovat `auto`-avainsanan ottaminen käyttöön uudessa merkityksessä sekä niin sanotut *lambda-lausekkeet*.

Muuttujan esittelyssä `auto`-avainsanalla voidaan korvata muuttujan eksplisiittisesti määritelty tietotyyppi, jolloin kääntäjä päättelee tietotyypin muuttujan alustavan lausekkeen perusteella. Tässä esimerkissä `x` on `int`-tyyppinen:

```
auto x = 7;
```

Suurempaa hyötyä `auto`-avainsanasta saadaan monimutkaisten tietotyyppien kanssa, joita esiintyy varsinkin STL:n säiliöluokkien iteraattoreita käytettäessä. C++98:ssa järjestettyjä pareja sisältävän säiliön alkuun viittaavan iteraattorin sijoittaminen muuttujaan tehdään seuraavasti:

```
std::map<std::string, std::pair<std::string, int> >::iterator
    it = employees.begin();
```

C++11:ssä `auto`-avainsanan käyttäminen lyhentää ja selkeyttää edellistä koodiriviä paljon:

```
auto it = employees.begin();
```

Lambda-lausekkeilla määritellään funktio-objekteja. Esimerkiksi vektorin lajittelussa käytettävä kahden alkion vertailufunktio voidaan antaa lambda-lausekkeena funktiokutsun parametrina:

```
std::sort(v.begin(), v.end(),
    [](int a, int b) { return abs(a) < abs(b); }
);
```

Koska C++-standardi on alusta-agnostinen eikä siksi ota kantaa alustoihin ja niiden eroihin, suoraan monialustaista kehitystä koskevia uudistuksia on vähän, mutta on kuitenkin muutamia standardirajapinnan saaneita toimintoja, jotka aikaisemmin edellyttivät alustakohtaista toteutusta. Yksi uudistus on tuki säikeille ja niiden synkronoinnille sekä muut rinnakkaiseen ohjelmointiin käytettävät uudet rajapinnat. C++98-standardi ei ottanut lainkaan kantaa säikeiden olemassaoloon, joten kaikki säikeisiin liittyvät operaatiot on ollut ennen C++11:tä välttämätöntä toteuttaa

alustakohtaisella ohjelmakoodilla tai kolmannen osapuolen kirjastoa hyödyntäen. Toinen uudistus on osin säikeisiinkin liittyvä korkearesoluutioinen ajan mittaus [19], joka mahdollistaa ajan mittaamisen erilaisia tarkoituksia varten jopa nanosekuntia paremmalla tarkkuudella, jos käyttöjärjestelmä ja laitteisto tällaista tarkkuutta pystyvät tarjoamaan. C++98:ssa aikaa ei voi standardikirjaston tarjoamin välinein mitata tarkemmin kuin yhden sekunnin resoluutiolla.

C++11:n uusien ominaisuuksien käyttöönottoa rajoittaa toistaiseksi nyt käytössä olevien kääntäjäversioiden vajavainen tai puuttuva tuki C++11:n ominaisuuksille. Tilanne tulee todennäköisesti paranemaan muutaman vuoden sisällä. C++11 ei säikeitä lukuun ottamatta tuo vaatimuslistan kohtiin parannuksia monialustaisuuden kannalta.

6.3 Projektien ja käännösympäristön hallinta

Käännösympäristö ja projektitiedostot ovat C++-lähdekoodiin suoraan liittymätön mutta muuten olennainen tekijä monialustaisen projektin kehityksen sujuvuuden kannalta. Projektitiedostot määrittelevät listan niistä lähdekooditiedostoista, jotka käännetään ja linkitetään mukaan ohjelmabinääriin. Lisäksi projektitiedostot määrittelevät linkitettävät ulkoiset kirjastot, kääntäjän varoitusten tason ja muita kääntäjän asetuksia. Käännösympäristöön kuuluvat kääntäjän lisäksi kaikki ne lisäkirjastot ja mahdolliset työkalut, joita kameravalvontaohjelmiston kääntäminen edellyttää.

Kameravalvontaohjelmiston käännöstä hallitaan nykyään ylläpitämällä kolmea erillistä projektitiedostoa. Näitä ovat Visual Studio -projekti, Xcode-projekti sekä käsin kirjoitettu Makefile. Suurin osa lähdekooditiedostoista on samoja kaikilla alustoilla, mutta tietyt alustakohtaisia toteutuksia sisältävät lähdekooditiedostot kuuluvat projektiin vain kyseisellä alustalla. Kolmen projektitiedoston manuaalinen päivittäminen aina lähdekooditiedostoja lisättäessä, poistettaessa tai uudelleen nimettäessä aiheuttaa jonkin verran ylimääräistä työtä, mutta on sinänsä yksinkertainen toimenpide.

Tavoiteltava tilanne olisi ylläpitää vain yhtä käännettävien tiedostojen listaa, josta voidaan automaattisesti generoida alustakohtaiset kääntämiseen vaadittavat tiedostot. Tällaisia monialustaisia käännöksenhallintajärjestelmiä on olemassa, mutta ne eivät ole isojen projektien kanssa erityisen helppokäyttöisiä, ja käyttöohjeita tai muuta aiheeseen

liittyvää kirjallisuutta on olemassa vain rajallisesti. Jos monialustaisen käännöksenhallintajärjestelmän käyttöönottoon ja hienosäätöön kuluu useita viikkoja työaikaa, on vaikea perustella sen paremmuutta verrattuna manuaalisesti mutta yksinkertaisesti ylläpidettäviin kolmeen projektitiedostoon.

Kameravalvontaohjelmisto käyttää hyväksi useita kolmannen osapuolen avoimen lähdekoodin kirjastoja. Pääsääntöisesti kirjastosta käytetään käyttöjärjestelmän tarjoamaa versiota, jos sellainen on saatavilla; muussa tapauksessa kirjasto on käännettävä osana ohjelmiston käännösympäristön käyttöönottoa. Linux-jakeluissa suurin osa tarvittavista kirjastoista saadaan helposti paketinhallinnan kautta. Mac OS X sisältää ennalta tunnetun joukon kirjastoja, myös monia avoimen lähdekoodin kirjastoja. Windows puolestaan ei sisällä mitään avoimen lähdekoodin kirjastoja, joten kaikki ylimääräiset kirjastot on hankittava ja käännettävä itse. Valmiiksi käännettyjä kirjastoja ei yleensä ole saatavilla.

Uuden kirjaston ottaminen mukaan projektiin vaatii manuaalisia muutoksia käännösympäristön luoviin skripteihin kaikilla alustoilla. Ideaalisessa tapauksessa uuden kirjaston lähdekoodipaketti lisättäisiin listalle, ja käännösympäristöä luova skripti hoitaisi kirjaston kääntämisen automaattisesti. Näin pitkälle menevä automatisointi tuskin on käytännössä mahdollista. Uusien kirjastojen käyttöönottoon tarvittavia toimenpiteitä voisi silti jossain määrin yhdenmukaistaa ja automatisoida. Toisaalta uusia kirjastoja ei kannata ottaa käyttöön harkitsemattomasti, sillä jokainen kirjasto lisää lopullisen sovelluksen asennuspaketin kokoa.

Koska yksittäinen kehittäjä kääntää ohjelmistoa kehityksen yhteydessä yleensä vain yhdellä alustalla kerrallaan, on mahdollista että versionhallintajärjestelmään asti päätyy koodia, joka ei käänny tai toimii väärin toisella alustalla. Vaikka tämä tuleekin ennen pitkää havaituksi ja korjatuksi, olisi regressioiden välttämiseksi hyödyllistä kääntää ohjelmisto ajastetusti kaikille alustoille vähintään kerran vuorokaudessa, ja suorittaa samalla yksikkötestit ja mahdolliset muut testit.

7 Yhteenveto

Tässä työssä on käsitelty erilaisia tapoja, joilla sama ohjelma saadaan toimimaan monella alustalla. Natiiviohjelmat ovat käännettyinä vahvasti sidottuja juuri siihen alustaan, jolle ne on tarkoitettu. Kuitenkin lähdekoodin muodossa natiiviohjelmat voivat olla kokonaan tai osittain alustasta riippumattomia. Moniarkkitehtuuriset binäärit yhdistävät useampia eri prosessoreille tarkoitettuja natiivibinääreitä samaan tiedostoon, josta käyttöjärjestelmä valitsee suoritettavaksi sopivimman. Tulkattujen kielten, virtualisoinnin ja web-selaimen tapauksessa sovellusohjelman kannalta alustana toimii tulkki, virtuaalikone tai selain, joka piilottaa abstraktiokerroksen taakse todellisen prosessorin ja käyttöjärjestelmän. Web-pohjainen palvelu tarvitsee lisäksi toimiakseen taustaohjelman, joka voi olla esimerkiksi natiiviohjelma tai virtualisoitu ohjelma.

Natiiviohjelmia kirjoitetaan yleensä C- ja C++-kielillä, joskin Applen alustoilla muuten harvinainen Objective-C on laajasti käytössä. Tässä työssä käsiteltiin monialustaisten ohjelmien tekemistä erityisesti C++-kieltä käyttäen. ISO C++ -standardi määrittelee C++-kielen alusta-agnostisesti: ohjelmaa suorittavalle laitteistolle ei ole paljonkaan vaatimuksia eikä rajoituksia. Lisälaitteiden olemassaoloa ei oleteta eikä vaadita, eikä lisälaitteiden suoraan käyttöön ole standardin määrittelemiä keinoja. Tämän vuoksi C++-standardikirjaston käyttöön rajoittuva ohjelmakoodi on siirrettävissä ilman muutoksia hyvin erilaisten alustojen välillä. Toisaalta tämän hyvin laajan monialustaisuuden kääntöpuoli on se, että C++:n standardikirjasto on paljon suppeampi kuin monien muiden ohjelmointikielten standardikirjastot. On kuitenkin C++:n tietoinen suunnitteluperiaate, että alustalta enemmän ominaisuuksia vaativiin toimintoihin käytetään alustakohtaisia, standardiin kuulumattomia kirjastorajapintoja.

Monialustaisten C++-ohjelmien monialustaisuus perustuu alustakohtaisten lähdekoodin osien eristämiseen niistä osista, jotka eivät käytä alustakohtaisia rajapintoja ja ovat siten alustariippumatonta lähdekoodia. Alustakohtaisia osia voidaan eristää usealla tavalla. Yksi tapa on käyttää C-esikääntäjän direktiivejä, jolloin lähdekoodista saadaan ehdollisesti jätettyä pois tai otettua mukaan tiettyjä rivejä alustan mukaisesti. Toinen tapa on eriyttää saman rajapinnan toteutus erillisiin tiedostoihin eri alustoja varten. Kolmas tapa on käyttää kolmannen osapuolen kirjastoja, jotka on suunniteltu

tarjoamaan abstraktio jollekin alustakohtaiselle rajapinnalle, esimerkiksi grafiikan piirtämiselle. Kolmannen osapuolen kirjastoja käyttämällä voidaan alustakohtaisen lähdekoodin olemassaolo kehitettävän sovelluksen tasolla välttää jopa kokonaan.

Monialustaisten ohjelmien toteutukseen esiteltiin viisi erilaista tapaa. Näiden toteutustapojen vertailua ja arviointia varten esitettiin seitsemän kriteeriä, jotka mittaavat toteutustavan hyvyyttä eri kannoilta. Myöhemmin näiden kriteereiden perusteella arvioitiin eri toteutustapojen soveltuvuutta olemassa olevan kaupallisen Ksenos-kameravalvontaohjelmiston toteutukseen. Kriteereiden lisäksi rinnakkaisena mutta toisentyyppisenä arvioinnin lähtökohtana käytettiin tähän ohjelmistoon liittyviä monialustaisia vaatimuksia.

Arvioinnin tulokseksi saatiin, että toteutus C++-natiiviohjelmana soveltuu esitetyistä toteutustavoista parhaiten tarkastelun kohteena olleen kameravalvontaohjelmiston toteutukseen. Lähes yhtä hyväksi tavaksi todettiin tulkatun kielen kuten Pythonin käyttäminen. Myös arvioinnissa huonoimman arvosanan saanut web-selainpohjainen toteutus on tiettyihin käyttötarkoituksiin käytännöllinen, vaikkakin ainoaksi toteutukseksi joiltakin osin tarpeettoman rajoittunut. Arvioinnissa painotettiin juuri tälle kameravalvontaohjelmistolle olennaisia kriteereitä. Toisentyyppinen ohjelmisto, joka tarvitsee vähemmän komponenttien natiivitoteutuksia avukseen, muuttaisi arvioinnin painotuksia ja tuloksia kaikkien toteutustapojen osalta.

Lopuksi pohdittiin tapoja, joilla nykyisen C++-natiivitoteutuksen kehitystä voitaisiin tehostaa erityisesti monialustaisuuden kannalta, mutta myös muuten. Nyt käytössä olevan wxWidgets-ohjelmistokehyksen korvaaminen Qt:lla olisi suuritöinen toimenpide, mutta tarjoaisi paremmin dokumentoidun ja aktiivisemmin kehitetyn ohjelmistokehyksen. Lisäksi Qt korvaisi joitakin muita nykyisen alustakohtaisia toteutuksia, ja siten vähentäisi alustakohtaisen lähdekoodin määrää varsinaisessa kameravalvontaohjelmistossa jonkin verran. Uusi C++-standardi C++11 ei niinkään tuo monialustaisia hyötyjä, mutta sen sijaan se tuo kielen syntaksiin monia uudistuksia, jotka lisäävät kielen ilmaisuvoimaa ja mahdollistavat entistä lyhyempiä tapoja tehdä tiettyjä asioita ohjelmakoodissa. C++-kieleen suoraan liittymätön tapa ohjelmiston monialustaisen kehityksen tehostamiseen olisi käännösympäristön, testien ja projektin sisällön hallinnan automatisoiminen siten, että projektissa voitaisiin ottaa käyttöön uusia

tiedostoja ja kirjastoja tekemättä joka kerta niiden kääntämiseksi alustakohtaisia toimenpiteitä.

C++-kielellä toteutetut natiiviohjelmat ovat edelleen hyvä ja toimiva tapa toteuttaa myös uusia ohjelmistoja. Virtualisoidut toteutustavat eivät ole syrjäyttäneet natiiviohjelmia. Uusi C++11-standardi ja sen esittelemät kymmenet hyödylliset uudistukset osoittavat, että C++ on edelleen aktiivisesti kehittyvä ohjelmointikieli.

Lähdeluettelo

[1] S. Goodwin, Cross-Platform Game Programming, Hingham, MA: Charles River Media, Inc., 2005.

[2] G. Coulouris, J. Dollimore, T. Kindberg ja G. Blair, Distributed Systems: Concepts and Design, 5th Ed., Boston, MA: Addison-Wesley, 2012.

[3] J. L. Hennessy ja D. A. Patterson, Computer Architecture: A Quantitative Approach, 4th Ed., San Francisco, CA: Elsevier, 2007.

[4] J. E. Smith ja R. Nair, Virtual Machines: Versatile Platforms for Systems and Processes, San Francisco, CA: Elsevier, 2005.

[5] A. S. Tanenbaum, Modern Operating Systems, 2nd Ed., Upper Saddle River, NJ: Prentice Hall, 2001.

[6] H. Heitkötter, S. Hanschke ja T. A. Majchrzak, ”Comparing Cross-Platform Development Approaches for Mobile Applications,” [Online]. http://www.wi1.uni-muenster.de/pi/veroeff/heitkoetter/Comparing-Cross-Platform-Development-Approaches-for-Mobile-Applications.pdf. [Haettu 19. 3. 2013].

[7] Apple Inc., ”Universal Binary Programming Guidelines, Second Edition,” 2005. [Online]. http://developer.apple.com/legacy/mac/library/documentation/MacOSX/Conceptual/universal_binary/universal_binary.pdf. [Haettu 27. 1. 2013].

[8] B. Stroustrup, The Design and Evolution of C++, Reading, MA: Addison-Wesley, 1994.

[9] B. W. Kernighan and D. M. Ritchie, The C Programming Language, 2nd Ed., Englewood Cliffs, NJ: Prentice Hall, 1988.

[10] ISO, ISO/IEC 14882:2011: Information technology — Programming languages — C++, 2011.

[11] B. Stroustrup, The C++ Programming Language, 3rd Ed., Boston, MA: Addison-Wesley, 1997.

[12] J. Reynolds ja J. Postel, ”RFC 1700,” 1994. [Online]. http://tools.ietf.org/pdf/rfc1700.pdf. [Haettu 16. 3. 2013].

[13] ISO, ISO/IEC 9899: Programming languages — C, New York, NY: American National Standards Institute, 1999.

[14] S. Logan, Cross-Platform Development in C++: Building Mac OS X, Linux, and Windows Applications, Upper Saddle River, NJ: Addison-Wesley, 2008.

[15] The Free Software Foundation, ”Arrays of Variable Length,” [Online]. http://gcc.gnu.org/onlinedocs/gcc/Variable-Length.html. [Haettu 9. 4. 2013].

[16] M. Lutz ja D. Ascher, Learning Python, 2nd Ed., Sebastopol, CA: O’Reilly, 2004.

[17] Python Software Foundation, ”Python/C API Reference Manual,” [Online]. http://docs.python.org/3.3/c-api/intro.html. [Haettu 25. 4. 2013].

[18] B. Stroustrup, ”C++11 FAQ,” [Online]. http://www.stroustrup.com/C++11FAQ.html. [Haettu 13. 5. 2013].

[19] H. E. Hinnant, W. E. Brown, J. Garland ja M. Paterno, ”N2661=08-0171: A Foundation to Sleep On,” [Online]. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2661.htm. [Haettu 21. 5. 2013].