Ilmo Euro

PELIPALVELINRATKAISUT

Opinnäytetyö
Tietojenkäsittelyn koulutusohjelma

Marraskuu 2014

KUVAILULEHTI

MAMK University of Applied Sciences	**Opinnäytetyön päivämäärä** **28.11.2014**
Tekijä(t) Ilmo Euro	**Koulutusohjelma ja suuntautuminen** **Tietojenkäsittelyn koulutusohjelma**
Nimeke Pelipalvelinratkaisut	

Tiivistelmä

Tarkastelen ja vertailen opinnäytetyössäni valmiita pelipalvelinratkaisuja sekä selvitän niiden soveltuvuutta peliohjelmoinnin opintojaksolle, jossa käytetään Unity-pelimoottoria. Työhön valitut pelipalvelinratkaisut ovat Pomelo, Cubeia FireBase, SmartFox Server, Photon, BigWorld sekä Unity Master Server. Ratkaisuiden vertailukriteereinä ovat niiden suorituskyky, käyttöönoton helppous, ohjelmistokehityksen vaivattomuus, monipuolisuus, avoimuus, sopimusehdot sekä yhteensopivien ohjelmointikielien sekä ohjelmistoalustojen määrä. Lisäksi käyn läpi ratkaisuiden taustalla olevia toteutusteknologioita sekä yleisesti tietoliikenneverkon yli pelattavien moninpelien esiin tuomia toteutusteknisiä haasteita.

Vertailun ohessa opinnäytetyö sisältää esimerkkitoteutuksen käyttäen yhtä pelipalvelinteknologiaa (SmartFox Server 2X), johon kuuluu asiakasohjelmisto, palvelimen laajennusohjelmisto sekä ohjeet pelipalvelimen asennukseen ja käyttöönottoon.

Opinnäytetyön motivaationa on madaltaa opiskelijoiden kynnystä tietoverkon yli pelattavien moninpelien ohjelmoimiseen sekä selvittää, mitä mahdollisia teknologioita voitaisiin ottaa mukaan tulevien opintojaksojen opetukseen.

Lopputulemana on, että valmiit pelipalvelinratkaisut soveltuvat hyvin hidastempoisiin ja vuoropohjaisiin peleihin, ja niitä voi käyttää varauksin myös reaaliaikaisissa peleissä. Ne eivät tee kehityksestä helpompaa, mutta vähentävät kehittäjien työmäärää ja kehittämiseen kuluvaa aikaa. Erityisen paljon pelipalvelimet helpottavat pelien hallinnointia.

Asiasanat (avainsanat)

Pelipalvelin, SmartFox Server, Cubeia FireBase, Photon, Moninpeli, Unity

Sivumäärä 30	**Kieli** Suomi
Huomautus (huomautukset liitteistä)	
Ohjaavan opettajan nimi Jukka Selin	**Opinnäytetyön toimeksiantaja** Jukka Selin

DESCRIPTION

MAMK University of Applied Sciences	**Date of the bachelor's thesis** November 28, 2014
Author(s) Ilmo Euro	**Degree programme and option** Business Information Technology
Name of the bachelor's thesis Game server solutions	

Abstract

I compared and described ready-made game server solutions, and assessed their suitability for a programming course where the Unity game engine was used. The motivation for the thesis was lowering the threshold for developing online multiplayer games, and assessing the possible technologies for future programming courses. The solutions I picked were Pomelo, Cubeia FireBase, SmartFox Server, Photon, BigWorld and Unity Master Server. The assessment criteria were performance, ease of deployment, ease of development, full-featuredness, openness, terms of use and the number of compatible programming languages and platforms. I also reviewed the technologies used in game server solutions and multiplayer network programming in general. The thesis contained an example game implementation with one of the game server solutions (SmartFox Server 2X), consisting of client software, a server extension and documentation for installing and deploying the game server.

The outcome of the thesis was that ready-made game server solutions were suitable for slow and turn-based games, and some real-time games could benefit from them, too. They did not make development easier, but reduced the amount of work and the time needed for development. Ready-made game servers excel at making the administration of online multiplayer games easier.

Subject headings, (keywords)

Game server, SmartFox Server, Cubeia FireBase, Photon, Online, Multiplayer, Unity

Pages 30	**Language** **Finnish**
Remarks, notes on appendices	
Tutor Jukka Selin	**Bachelor's thesis assigned by** Jukka Selin

SISÄLTÖ

1 JOHDATUS MONINPELIEN TEKNOLOGIAAN

Verkkoyhteyden yli pelattava moninpeli on klassinen esimerkki hajautetusta järjestelmästä: siinä on useita komponentteja, jotka viestivät keskenään verkon yli saavuttaakseen yhteisen päämäärän, eli saumattoman moninpelikokemuksen (Haddad, ym. 2011). Järjestelmän osina ovat tietokonelaitteisto (yleiskäyttöiset tietokoneet tai pelikonsolit), verkkoyhteyslaitteisto, peliohjelmat sekä itse pelaajat. Koska viestintäkanavien määrä komponenttien välillä kasvaa polynomisesti suhteessa komponenttien määrään (Brooks 1995, 18), tulee viestien määrästä helposti suuri.

Verkkomoninpelejä on kahdenlaisia: keskitettyyn palvelimeen perustuvia client-server -tyyppisiä pelejä sekä pelaajaporukan välisiä, joko ad-hoc -palvelimeen (pelin isäntä) tai palvelimettomaan arkkitehtuuriin perustuvia pelejä. Nykyään client-server-tyyppiset pelit ovat suosittuja, koska niiden pelaaminen on vaivatonta, ja niihin tarvittavat laajakaistaiset Internet-yhteydet ovat yleistyneet (ITU WTID 2014). Tässä työssä keskityn client-server -tyyppiin, koska pelipalvelinratkaisut pohjautuvat siihen.

Verkkomoninpelit voivat olla joko vuoropohjaisia tai reaaliaikaisia. Vuoropohjaisia pelejä toteutettaessa voidaan yleensä käyttää samoja tekniikoita kuin esim. tyypillisten Web-sovellusten tapauksessa, sillä keskipitkätkään viiveet eivät yleensä vaikuta pelikokemukseen. Sen sijaan reaaliaikaiset verkkomoninpelit ovat huomattavasti monimutkaisempia toteuttaa, koska yli 200 ms viiveet voivat jo pilata pelikokemuksen (Leadbetter 2009). Tämän monimutkaisuuden helpottamiseksi ja hallitsemiseksi on kehitetty valmiita pelipalvelinratkaisuja.

Luvussa 2 tarkastelen muutamaa yleisesti saatavilla olevaa pelipalvelinratkaisua. Koska pelipalvelinratkaisuiden toiminnallisuudet muistuttavat toisiaan, niin yhden pelipalvelinteknologian osaaminen helpottaa muiden teknologioiden oppimista. Siksi valitsen tässä opinnäytetyössä yhden ratkaisun, jota käyttäen toteutan pienen pelin sekä esittelen sen tekniikkaa ja käyttöönottoa. Painotan palvelinratkaisun valinnassa opittavuutta, monipuolisia monitorointiominaisuuksia sekä helppoa asennusta ja käyttöönottoa, jotta sen kokeileminen onnistuisi myös pelipalvelimiin vähemmin perehtyneeltä.

Luvussa 3 käsittelen verkkomoninpeleissä eteen tulevia teknisiä haasteita. Sulavan moninpelikokemuksen ylläpitäminen tulee sitä vaativammaksi, mitä enemmän on pe-

laajia, ja mitä nopeatempoisempi peli on. Tekninen arkkitehtuuri ja osaaminen muuttuvat siis hyödykkeen sijasta kilpailueduksi, joten niiden jakaminen kilpailijoille ei kannata. Nopeatempoiset ja massiiviset pelit vaativat myös nimenomaan kyseiselle pelille optimoituja ohjelmistoja, joiden yleiskäyttöiseksi muuttaminen heikentäisi niiden suorituskykyä. Useimmat pelit eivät kuitenkaan saavuta miljoonia pelaajia, ja monet suositut pelit ovat hidastempoisia. Pelipalvelinratkaisut soveltuvat niihin vähentäen kehittäjien työmäärää ja siten nopeuttaen kehitystä. Kaikki ratkaisut tarjoavat saman perustoiminnallisuuden, eli peleille optimoidun abstraktiotason kuljetustason protokollien (TCP, UDP) päälle, mutta eroavat tekniseltä toteutukseltaan, lisensointiehdoiltaan, lisätoiminnallisuuksiltaan (kuten valmiit chat-, pelihuone- ja pistetaulukkopalvelut) sekä sopivuudeltaan eri peleihin, kuten massiiviset moninpelit ja actionpelit.

Luvussa 4 käsitellään esimerkkitoteutusta pelikokonaisuudesta käyttäen SmartFox Server-pelipalvelinratkaisua. Toteutuksen asiakasohjelma on toteutettu Unity-pelimoottoria ja C# -kieltä käyttäen, mutta samankaltaista ohjelmakoodia voi käyttää myös muunlaisiin asiakasohjelmiin, kuten JavaScript- tai Flash -sovelluksiin. Palvelinlaajennus on kehitetty käyttäen Java -ohjelmointikieltä. Luvussa kuvataan itse ohjelmistojen toteutuksen lisäksi pelipalvelimen asennus ja käyttöönotto sekä pelin asennus pelipalvelimelle.

2 PELIPALVELINRATKAISUT

Vertailen tässä luvussa valmiita pelipalvelinratkaisuja. Otan mukaan pelipalvelinkirjastot sekä ohjelmistokehykset, jotka voi asentaa omalle palvelintietokoneelle. Kokonaisia pelimoottoreita tai ohjelmistopalveluita (Software as a Service) en käsittele, sillä ne sitovat pelintekijät yhden toimittajan teknologioihin.

2.1 Pomelo

Pomelo on kiinalaisen NetEase -yrityksen kehittämä nopea, skaalautuva pelipalvelinkehys (*framework*) Node.js:lle. Se on julkaistu MIT -lisenssillä, joten sitä voi käyttää huoletta myös kaupallisissa peleissä. Esittelyyn käytetty peli on massiivinen online-moninpeli, joten suuret pelaajamäärät ja reaaliaikaisuus ovat Pomelon prioriteetteja. (Pomelo 2014.)

NetEase on Internet -teknologiayritys, jonka ydintoimintana ovat massiiviset online-roolipelit (MMORPG). Se toimii yhteistyössä Blizzard Entertainmentin kanssa, joka on maailman suosituimman massiivisen online -roolipelin, World of Warcraftin, kehittäjä. Lisäksi NetEase kehittää itse MMORPG:eja, kuten Westward Journey. NetEase on myös Kiinan suurin sähköpostipalvelujen tarjoaja sekä ylläpitää yhtä Kiinan suurimmista web-portaaleista. (NTES Fact Sheet August 2014.)

Pomelossa asiakasohjelmiston ja palvelinohjelmiston välinen kommunikaatio perustuu avoimeen protokollaan, ja protokollan toteuttavia kirjastoja on useille ohjelmistoalustoille, mm. JavaScriptille, Unitylle ja Androidille. (Crane 2013.)

Koska Pomelo on vapaata ohjelmistoa, on se monella tavalla houkutteleva: siitä ei tarvitse maksaa mitään ja sitä voi tarvittaessa muokata omiin tarpeisiin sopivaksi. Palvelinohjelmointi pitää kuitenkin tehdä JavaScriptillä, projekti on melko uusi eikä sille tarjota kaupallista tukea. Pomelo ei myöskään tarjoa valmiita ratkaisuja, vaan työkalut ratkaisuiden kehittämiseen. Esimerkkejä Pomelon avulla kehitetyistä peleistä ovat mm. online-korttipeli 全民AAA sekä strategiapeli 二战前线 (Pomelo成功案例 2014).

2.2 Cubeia FireBase

Cubeia Firebase on Cubeian kehittämä Java-pohjainen pelipalvelinkehys (*framework*). Se on suunniteltu korttipelien tarpeisiin, koska se toimii pohjana saman yrityksen *Poker* -pokeripelipalvelimelle.

Cubeia on ruotsalainen ohjelmistoyritys, jonka ydinliiketoimintaa ovat pokeripelit. Yrityksen tuotteet ovat Poker, joka on online-pokeripeliohjelmisto, Social, joka on palvelu omien pokeripelisivustojen luomiseen ja Firebase, joka on muiden tuotteiden pohjalla toimiva verkkomoninpeliteknologia. (Cubeia 2013.)

Firebasen palvelintoiminnot ohjelmoidaan Javalla, mutta asiakasohjelmia varten on olemassa Java-, JavaScript- sekä ActionScript-rajapinnat. Lisäksi protokolla on dokumentoitu, joten uusien rajapintojen laatiminen onnistuu myös kolmansilta osapuolilta. (Cubeia Firebase 2013.)

Firebase on suunniteltu palvelemaan hyvin suurta määrää yhtäaikaisia pelaajia. Neljällä palvelimella, jossa on 2.4-gigahertsinen Intel Xeon suorittimena, ja joiden yhteishinta on alle 3000 € (vuonna 2008), voi palvella yli 20 000 pelaajaa, jotka ovat jakautuneet suurimmaksi osaksi 10 pelaajaan huoneisiin. (Johansson 2008.)

Firebasesta on saatavilla sekä AGPL-lisensoitu *Community*-versio että kaupallisesti lisensoidut *Standard*- ja *Enterprise*-versiot. Jos kehittää pelin, jossa käytetään *Community* -lisensoitua versiota palvelimesta, tulee peli julkaista AGPL -lisenssillä lain mukaan, joten kaupallista pelinkehitystä varten tarvitsee kaupallisen lisenssin.

Firebase on aktiivisessa käytössä, ja se sisältää paljon ominaisuuksia sekä saatavana vapaana ohjelmistona. Se on kuitenkin suunniteltu hidastempoisiin peleihin, joissa on paljon käyttäjiä, nopeatempoisten, pienissä ryhmissä pelattavien pelien sijasta (Nilsson 2010). Myöskään Unity/C# -asiakaskirjastoa ei ole, joten se ei sovellu esimerkkipelin palvelinalustaksi.

2.3 Photon Server

Photon on Exit Gamesin kehittämä arkkitehtuuri moninpelien verkkoliikennettä varten. Se on suosittu Unity-kehittäjien keskuudessa, sillä *Photon Unity Networking* (*Photon*in Unity-asiakaskirjasto) on saatavana helposti Unity Asset Storesta. Photon toimii ensisijaisesti ohjelmistopalveluna (*Photon Cloud*), mutta on myös asennettavissa omalle tietokoneelle (*Photon Server*).

Exit Games on ohjelmistoyritys, joka on erikoistunut ainoastaan Photon-tuoteperheeseen. Siihen kuuluu Photon Realtime reaaliaikaisia moninpelejä varten, Photon Chat peleissä tapahtuvaa keskustelua varten, Photon Turnbased vuoropohjaisia moninpelejä varten, Photon PUN (*Photon Unity Networking*) Unityä varten sekä Photon Server verkkoarkkitehtuurin *on-site* asennuksia varten. (Exit Games 2014.)

Photonin pilviversiota Photon Cloudia käytettäessä tulee kaikki pelilogiikka olla asiakasohjelmistossa, sillä pilvipalvelussa ei voida tietoturvasyistä ajaa mitä tahansa ohjelmakoodia. Photon Serveriä käytettäessä voi palvelinlaajennuksia ohjelmoida Mic-

rosoftin Dot NET -ympäristöä käyttäen Asiakaskirjastoja on useille alustoille, kuten Webiin, Unitylle sekä kaikille mobiilialustoille.

Photon, kuten Pomelo, tarjoaa työkalut reaaliaikaisten verkkopelien kehittämistä varten, mutta useimmat lisäpalvelut (kuten keskustelu, laaja monitorointi yms) siihen joudutaan hankkimaan tai kehittämään erikseen (Exit Games 2014). Siihen on saatavilla eri lisenssivaihtoehtoja, mutta enintään 100 yhtäaikaisen käyttäjän (CCU) lisenssi on ilmainen. En valinnut Photon Serveriä esimerkkipelin palvelinratkaisuksi, koska asiakaskirjaston lataaminen Exit Gamesin kotisivuilta vaatii rekisteröitymisen, joka voi olla esimerkiksi peliohjelmoinnin luokkahuoneopetuksessa opiskelijoiden sekä opettajan kannalta hankalaa.

2.4 SmartFox Server

SmartFox Server on tunnettu, kaupallinen pelipalvelinohjelmisto. Siitä on 3 versiota: Basic, Pro ja 2X. Basic-versio on ilmainen, mutta sitä käytettäessä kaikki pelilogiikka tapahtuu **asiakasohjelmistossa**, ja pelipalvelinta käytetään vain asiakasohjelmien väliseen kommunikaatioon. Basic -versio tukee vain Flash-pohjaisia asiakasohjelmia. Pro -versiossa palvelimeen voi lisätä toiminnallisuutta ActionScript-, Java- ja Python-laajennuksilla, ja asiakasrajapintoja on myös Unity- ja Dot NET-alustoille. Sitä voi käyttää ilmaiseksi enintään 20 pelaajan peleissä, ja yli 20 pelaajan peleissä lisenssillä (alkaen 500 €). 2X-versio on kehittynein SmartFox-pelipalvelimista. Se on suunniteltu alusta alkaen suorituskykyisemmäksi ja rajapinnaltaan yksinkertaisemmaksi kuin muut perheen palvelimet, mutta samalla palvelinpään ActionScript sekä asiakasohjelmiston ActionScript 2 -tuki on poistettu. Lisäksi 2X-versioon on saatavana monipuoliset analytiikkaominaisuudet. 100 pelaajan lisenssi maksaa 350 €, mutta palvelin on saatavana myös ilmaiseksi 100 pelaajalle (*Community Edition*), jos näyttää *”Powered by SmartFoxServer”* -ilmoituksen pelissä.

Kaikki palvelinversiot sisältävät *BlueBox* -nimisen komponentin, jota käyttämällä liikenne voidaan tunneloida HTTP:n läpi, jolloin pelit toimivat myös useimpien palomuurien läpi. BlueBoxia käytettäessä menetetään kaikki UDP:n tuomat edut, joten sitä kannattaa käyttää vain viimeisenä hätäkeinona tai hidastempoisiin peleihin.

SmartFox Server on ollut käytössä vuodesta 2004, ja sitä käyttäviä pelejä ovat mm.

Worms, Settlers MyCity sekä Drift Mania: Street Outlaws. Se on myös seikkaperäisesti dokumentoitu ja sisältää monipuoliset hallinnointi- ja monitorointiominaisuudet. Muista tekniikoista poiketen siinä on myös sisäänrakennettu tietokantatuki. Lisäksi siihen on saatavilla Dot NET/Mono-asiakaskirjasto, joten valitsin sen esimerkkipelin palvelinratkaisuksi.

2.5 Muut

Unity sisältää oletuksena teknologian nimeltä *Unity Master Server*. Se on Unity Technologiesin kehittämä pelinvälityspalvelin, joka toimii yhteistyössä muun Unity -verkkoteknologian kanssa. Unity Master Server ei kuitenkaan ole pelipalvelin samassa merkityksessä kuin muut kuvatut pelipalvelimet, sillä se on vastuussa vain pelaajien yhdistämisestä varsinaisiin pelipalvelimiin, ei varsinaisesta verkkoliikenteestä. Unity Master Serveristä on olemassa Unity Technologiesin hallinnoima esimerkkiasennus, johon otetaan oletuksena yhteyttä (Unity Manual - Master Server 2014). Lisäksi pelinkehittäjät voivat asentaa ja hallinnoida Master Server -asennuksia itse. Unity Master Serverin lisäksi Unityssä on vahvasti pelimoottoriin integroidut verkko-ominaisuudet, kuten peliobjektien synkronointi, joiden avulla suuren osan verkkomoninpelien kehityksestä voi automatisoida. Laajat hallinnointi- ja monitorointitoiminnot eivät sisälly pakettiin, joten ne on kehitettävä itse, jos haluaa toteuttaa keskitettyyn palvelimeen perustuvan moninpelin. *Ad-hoc* -tyyppiseen peliin sen sijaan Unityn verkko-ominaisuudet sopivat mainiosti, joskin niitä täydentämään saatetaan tarvita räätälöityä ja pitkälle optimoitua verkkokoodia.

BigWorld on BigWorld Technology -nimisen yrityksen kehittämä massiivisiin moninpeleihin suunniteltu pelimoottori, jota käytetään mm. suositussa World of Tanks -pelissä. Se sisältää pelipalvelimen, mutta sen lisäksi mm. fysiikkamallinnuksen, grafiikan, skriptausominaisuudet sekä muita pelimoottoriin kuuluvia toimintoja, eikä sitä voi mielekkäästi käyttää esim. Unityn kanssa. BigWorldia on saatavana kahdella eri lisenssillä: 300 dollarin hintainen Indie -lisenssi enintään 25 kehittäjälle, sekä tarjouspyynnöllä hinnoiteltu kaupallinen lisenssi rajoittamattomalle määrälle kehittäjiä. BigWorldin asiakasohjelmistoon voi Indie -lisenssillä kehittää Python-laajennuksia, ja kaupallisella lisenssillä saa lisäksi täyden C++-lähdekoodin, jonka pohjalta voi laajentaa sekä asiakas- että palvelinohjelmistoja. (BigWorld Technology 2014.)

OpenSpace on SmartFox Serverin päälle rakennettu pelimoottori, joka helpottaa isometristen moninpelien kehittämistä. Se soveltuu sekä *Habbo Hotel*-tyyppisten virtuaalimaailmojen että massiivisten moninpelien kehittämiseen. *OpenSpace*lla on luotu mm. *The Settlers – My City* -Facebook-peli sekä virtuaalinen Mingoville-oppimisympäristö. Toiminnaltaan OpenSpace on *point and click* -tyyppinen, eli se sisältää kenttäeditorin, jolla voi toteuttaa hyvin suuren osan pelistä. Ohjelmakoodin määrä on siis pyritty pelimoottorin suunnittelussa minimoimaan. OpenSpace-asiakasohjelma toimii käyttäen Adobe Flash -teknologiaa.

3 PELIPALVELINTEKNOLOGIA

Kun kehitetään peliä jonka tarpeisiin valmiit pelipalvelinratkaisut eivät sovellu, joudutaan itse peliin kehittämään enemmän tai vähemmän valmiiden ratkaisuiden sisältämiä toiminnallisuuksia. Valmiita ratkaisuja käyttäessäkin tulee hallita niiden pohjalla olevat tekniikat, jotta mahdollisissa ongelmatilanteissa voi ymmärtää ongelmien syyt ja vaikutukset. Käsittelen tässä luvussa muutamia haasteita, joita pelipalvelinta kehittäessä voi tulla eteen.

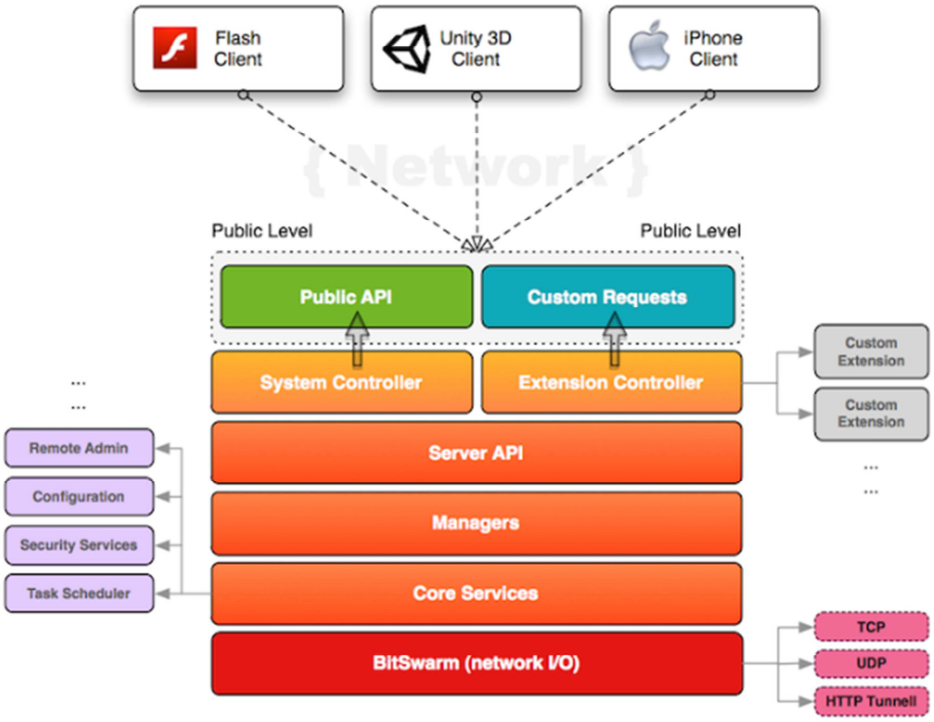
3.1 Verkkoprotokollat

Tunnetuin kuljetustason protokolla Internetissä on TCP eli *Transmission Control Protocol*. Se on ohjelmoijan kannalta hyvin mukava: se tarjoaa yksinkertaisen ”bittiputken”, johon lähetetty data saapuu vastaanottajalleen varmasti ja oikeassa järjestyksessä (Information Sciences Institute 1981). Kadonneista ja väärässä järjestyksessä saapuneista paketeista huolehditaan kuljetustasolla. Kääntöpuolena tälle mukavuudelle on suorituskyky: jos verkkoajuri joutuu lähettämään saman paketin useita kertoja, syntyy väistämättä viivettä (Fiedler 2008). Tästä syystä verkkomoninpeleissä käytetään yleensä UDP-protokollaa (*User Datagram Protocol*). UDP on *yhteydetön protokolla*, joka ei takaa pakettien perillemenoa eikä järjestystä. Ohjelmoija joutuu siis joko rakentamaan oman, erikoistuneen varmistusjärjestelmän UDP:n päälle tai suunnittelemaan pelinsä sietämään kadonneita ja väärään aikaan saapuvia paketteja. (Postel 1980.)

Yksi ongelma UDP-liikenteen kanssa on palomuurit: monissa yrityksissä on palomuuri, joka estää suurimman osan UDP-liikenteestä. Tätä varten tarjotaan HTTP tai

HTTPS-tunnelointiratkaisuja, kuten *BlueBox*, joiden avulla pelit voivat käyttää tavallista TCP 80-porttia, joka on useimmissa palomuureissa auki. Kaikki ylimääräiset käsittelyvaiheet kuitenkin pudottavat verkkoliikenteen suorituskykyä, jolloin UDP-protokollan edut vähenevät tai katoavat kokonaan.

Varsinaisen peliliikenteen lisäksi peleissä tarvitaan usein teksti-, ääni- ja videokeskustelua pelaajien välillä. Sitä varten käytetään yleensä valmiita, yleiskäyttöisiä ratkaisuja, kuten XMPP tai Mumble. SmartFox Serverissä on tuki vapaata lähdekoodia olevalle Red5-videokeskustelujärjestelmälle, joka tarjotaan *RedBox* -nimisen komponentin kanssa. Komponentti sisältää mediapalvelimen, asiakaskirjastot sekä palvelinlaajennokset, joiden avulla peliin on vaivatonta lisätä video- ja äänikeskusteluyhteys.

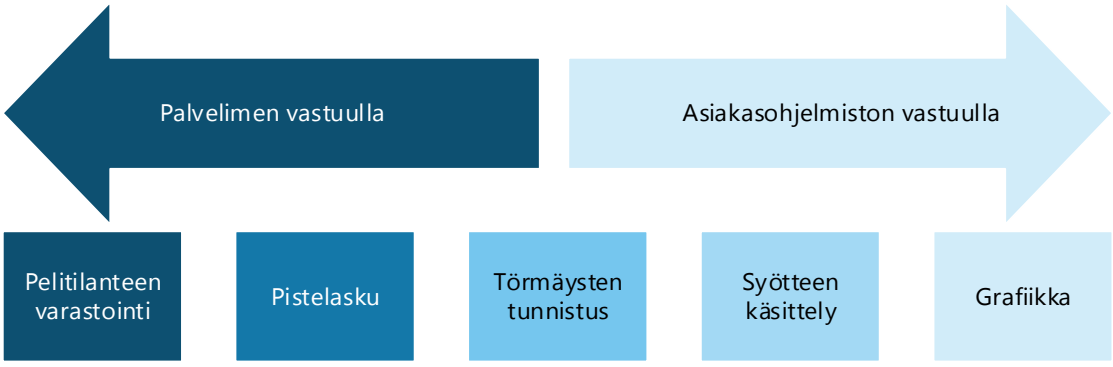


KUVA 1. SmartFox Serverin arkkitehtuuri (Lapi 2012)

Valmiit pelipalvelinratkaisut sisältävät yleensä UDP:n tai TCP:n päälle rakennetun verkkoprotokollakerroksen. SmartFox Serverin tapauksessa se on nimeltään BitSwarm (Lapi 2012). BitSwarm sisältää TCP/UDP-yhteyden lisäksi istuntotoiminnallisuuden, tietoturvaominaisuuksia, yhteyksien vikasietoisuustoiminnot, HTTP-tunnelointirajapinnan (johon BlueBox yhdistetään) sekä monitoroinnin (kuva 1). Pomelossa on myös oma räätälöity yhteysprotokollansa, joka sisältää mm. kättely- ja *heartbeat*-määrittelyt.

3.2 Pelipalvelimen rooli

Pelipalvelin voi olla joko *autoritaarinen* tai *ei-autoritaarinen* (Unity Technologies 2014). **Autoritaarinen** pelipalvelin käsittelee kaikki pelaajien syötteet ja laskee pelimaailman muutokset syötteiden perusteella, eli suorittaa *simulaatio*ta. Autoritaarista palvelinta käytettäessä on kaikkien pelaajien pelitilanne aina identtinen, joten mm. viiveestä tapahtuvia heittoja pistelaskussa ei voi tapahtua. Huijaaminen on myös paljon vaikeampaa autoritaarista palvelinta käytettäessä, koska asiakasohjelmalle ei välttämättä lähetetä kaikkea tietoa esim. muiden pelaajien toimista. Autoritaarisen palvelimen suurin ongelma on kuormitus, sillä kaikkien pelaajien kaikkien syötteiden käsittely vaatii paljon suoritintehoa. Toinen ongelma on viive: koska syötteet joudutaan siirtämään verkkoyhteyden yli ennen käsittelyä, saattaa aika syötteen antamisen ja pelitilanteen päivittämisen välillä kasvaa niin suureksi, että viivekompensointi on välttämätöntä. **Ei-autoritaarinen** palvelin ei käsittele pelimaailman simulointia, vaan ainoastaan välittää viestejä asiakasohjelmilta toisille. Se on siis yksinkertaisimmillaan vain viestirele. Ei-autoritaarisen palvelimen tapauksessa on pelitilanteen pitäminen samanlaisena asiakasohjelmien välillä haastavaa, sillä muilta asiakasohjelmilta saadut viestit koskevat pelitilannetta viestin lähetyshetkellä, ei sen saapumishetkellä. Myös viive asiakasohjelmien välillä saattaa vaihdella, joten pelitilanteen samanlaisena pitämisen vaatima ennakointi vaatii paljon ohjelmointityötä. Ei-autoritaarisen palvelimen kuormitus ei kuitenkaan kasva huomattavasti pelaajamäärän kasvaessa, joten se saattaa olla ainoa vaihtoehto joidenkin nopeatempoisten pelien tapauksessa. (Bernier 2001.)



KUVA 2. Vastuunjako palvelimen ja asiakasohjelmiston välillä

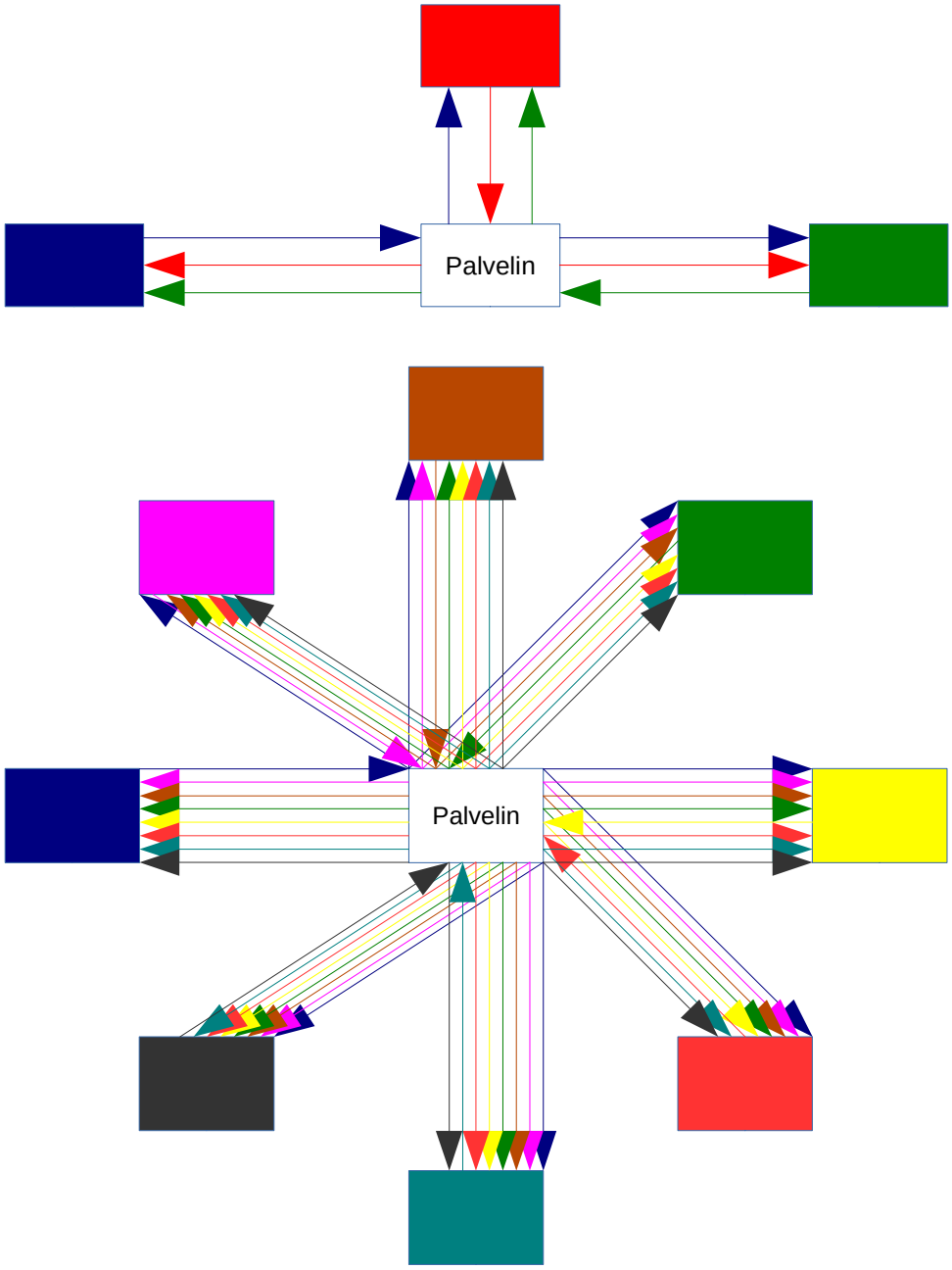
Kolmas vaihtoehto on jakaa eri toiminnallisuudet palvelimen ja asiakasohjelmiston välillä (kuvio 2). Yleisesti ottaen jako on mielivaltainen, mutta muutamat seikat tulee ottaa huomioon sitä laatiessa: jos esimerkiksi pelitilanteen varastoi asiakasohjelmaan, tulee synkronoinnista kaikkien asiakasohjelmien välillä laskennallisesti ja tiedonsiirrollisesti intensiivistä. Pelitilanne sopii siis paremmin palvelimella varastoitavaksi. Asiakasohjelmistossa suoritettava pistelasku tekee pelissä huijaamisesta helppoa ns. Man-In-The-Middle-tekniikan tai ns. trainereiden avulla. Tämän voi ratkaista joko huijauksenesto-ohjelmistoilla tai suorittamalla pistelaskun palvelimella. Jos törmäyksen tunnistuksen tai fysiikkamallinnuksen suorittaa asiakasohjelmassa, riskinä ovat synkronointiongelmat: mitä, jos yhdessä asiakasohjelmassa tapahtuu törmäys, joka toisessa asiakasohjelmassa jää verkkoviiveen takia tapahtumatta? Toisaalta kaikkien törmäysten tunnistaminen palvelimella kuormittaisi palvelinta raskaasti. Syötteen käsittelyn voi suorittaa joko asiakasohjelmassa, jolloin pelitilannetta käsitellään syötteen mukaan, tai syötteet voi lähettää palvelimelle – syötteitä lähettäessä voi verkkoviive olla ongelma. Grafiikan piirtäminen palvelimella kuormittaisi palvelinta erittäin raskaasti, ellei peli ole hyvin hidastempoinen. Se on siis yleensä jätetty asiakasohjelmalle.

3.3 Skaalautuvuus

Pelipalvelimien käyttäjämäärän kasvaessa syntyy nopeasti skaalautuvuusongelmia: oletetaan esimerkiksi yksinkertainen pelipalvelin, joka lähettää kunkin saamansa viestin sellaisenaan edelleen kaikkiin siihen liittyneisiin asiakasohjelmiin. Tähän pelipalvelimeen on liittynyt N asiakasohjelmaa, ja kukin asiakasohjelma lähettää M viestiä sekunnissa. Tällöin pelipalvelimeen saapuu NM viestiä sekunnissa, ja kunkin näistä viesteistä se joutuu lähettämään jokaiselle N asiakasohjelmalle, jolloin lähetettyjen viestien lukumäärä on N^2M viestiä sekunnissa (kuva 3).

Autoritaarisilla palvelimilla tilanne on vaikeampi, sillä niiden pitää viestien lähettämisen lisäksi simuloida pelitilanne, joka saattaa sisältää monimutkaisia fysiikkamallinnuksia. Tästä syystä toiminnallisissa ja fysiikkapainotteisissa peleissä asiakasohjelmien lukumäärä palvelinta kohti on yleensä melko pieni ja massiivisia kilpa-ajopelejä ja räiskintöjä on vähän. Jos halutaan tehdä suurilla pelaajajoukoilla jouhevasti toimivia toimintapelejä, on autoritaarisen palvelimen ohjelmiston kehittäminen hyvin haastavaa, ellei mahdotonta. Sen sijaan asynkronisten pelien,

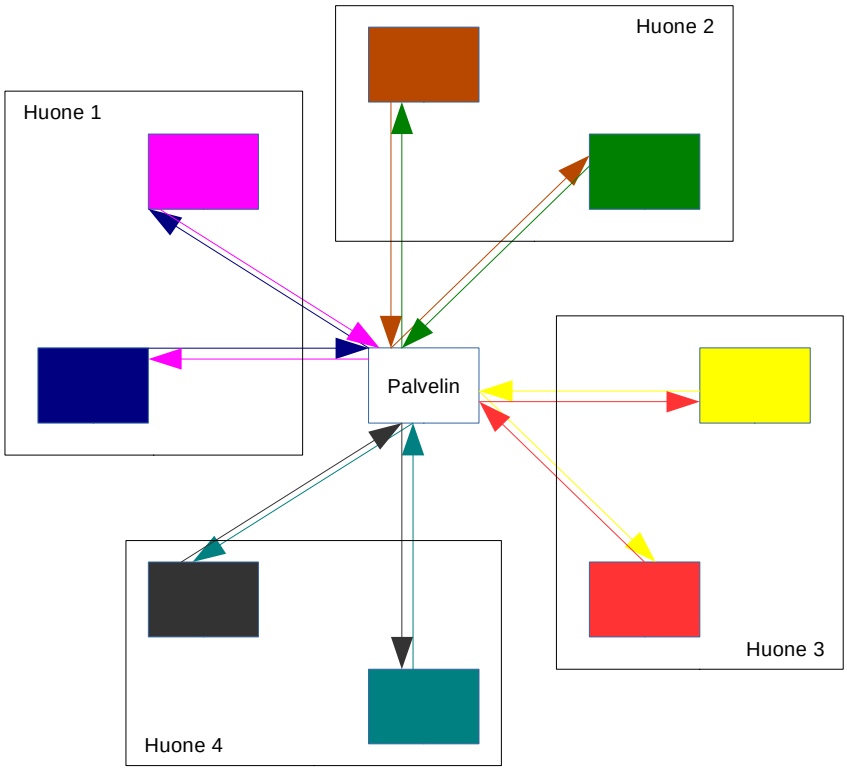
kuten vuoropohjaisten pelien ja hidastempoisten simulaatioiden, toteutus on helpompaa, sillä viestien tiheys on paljon pienempi.



KUVA 3. Viestien määrä suhteessa asiakasohjelmien määrään

Yksi tapa rajoittaa kuormaa on kunkin viestin vastaanottavien asiakkaiden lukumäärän vähentäminen. Sitä varten palvelin voidaan jakaa huoneisiin, joissa kukin viesti lähetetään vain samassa huoneessa oleville asiakasohjelmille. Jos asiakasohjelmien lukumäärä on *N*, mutta ne on jaettu *L* samansuuruiseen huoneeseen, on saapuvien viestien lukumäärä vielä *NM*, mutta lähtevien viestien määrä tippuu arvoon $(N/L)^2M$ eli $(N^2M)/L^2$. Lähetettävien viestien määrä on siis kääntäen verrannollinen huoneiden

lukumäärän neliöön (katso kuva 4). Pienet huoneet voivat olla joissain peleissä epäkäytännöllisiä, ja saattavat latistaa pelikokemuksen. Massiivisissa moninpeleissä käytetään erilaista, koordinaatteihin perustuvaa jaottelua, jota mm. SmartFox Server tukee (SmartFox Server 2X Documentation Central - MMO Rooms 2014).



KUVA 4. Asiakasohjelmien jako huoneisiin

Yhteyksien lukumäärän kasvaessa voi perinteinen säikeistetty ohjelmointimalli tuottaa myös ongelmia: jos kullekin pelipalvelimeen yhdistetylle pelaajalle varataan oma säie, saattaa palvelin jumiutua nopeasti (Kegel 2006). Siksi esimerkiksi SmartFox Serverin BitSwarm -verkkoajurissa käytetään mallia, jossa pelaajayhteydet eivät varaa (*block*) säiettä (Lapi 2012). Pomelo on kehitetty käyttäen säikeetöntä Node.js-teknologiaa, joten siinä säikeiden varaaminen on mahdotonta.

3.4 Verkkoviiveen kompensointi

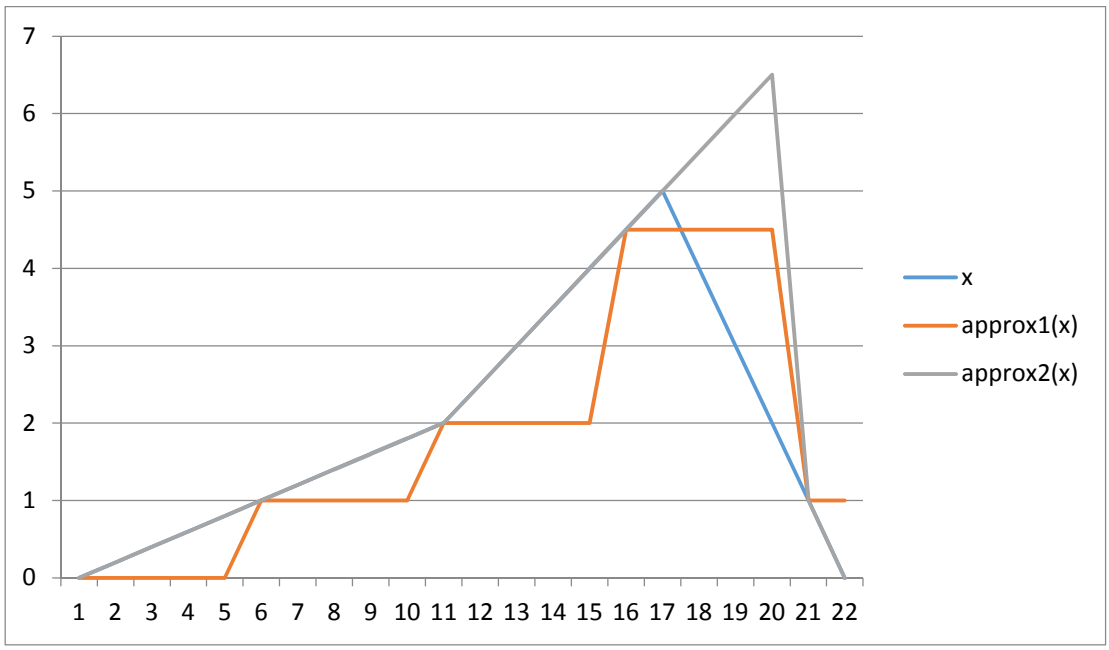
Koska verkon yli tapahtuvan liikenteen viive saattaa kasvaa huomattavasti yli 100 millisekunnin, tulee viivettä kompensoida asiakasohjelmassa erilaisin menetelmin sulavan moninpelikokemuksen ylläpitämiseksi. Vaikka palvelin siis olisi autoritaarinen, joudutaan simulaatiota tekemään ainakin jossain määrin myös asiakasohjelmassa. Näin pelaaja saa välittömän palautteen toimistaan, vaikka viive viestin lähetyksen ja vastauksen saapumisen välillä olisi pitkäkin. Koska asiakasohjelma ei pysty luotettavasti ennustamaan muiden pelaajien liikkeitä, ovat kaikki kompensaatiomenetelmien tulokset vain likiarvoja, mutta niiden avulla on mahdollista luoda illuusio maailmasta, jossa kaikki pelaajat toimivat yhtäaikaisesti.

TAULUKKO 1. Tuntemattoman suureen approksimointi ekstrapoloimalla.

x	näyte(Δx)	näyte(x)	approx1(x)	approx2(x)
0	0,2	0	0	0
0,2			0	0,2
0,4			0	0,4
0,6			0	0,6
0,8			0	0,8
1	0,2	1	1	1
1,2			1	1,2
1,4			1	1,4
1,6			1	1,6
1,8			1	1,8
2	0,5	2	2	2
2,5			2	2,5
3			2	3
3,5			2	3,5
4			2	4
4,5	0,5	4,5	4,5	4,5
5			4,5	5
4			4,5	5,5

3			4,5	6
2			4,5	6,5
1	-1	1	1	1
0			1	0

Yksinkertaisimmat kompensaatiomenetelmät ovat interpolaatio ja ekstrapolaatio. Ekstrapolaatiossa lasketaan suureen kulloinenkin arvo sen aikaisemmasta arvosta ja muuttumisnopeudesta (*delta*). Esimerkkejä suureista ovat sijainti ja katsomissuunta, joita käytämme myös myöhemmin esiteltävässä esimerkkipelissä. Taulukossa 1 kuvataan esimerkkisuureen muutosta askeleittain simulaation mukaan (*x*), asiakasohjelmalle säännöllisin väliajoin saapuvaa näytettä arvosta (*näyte(x)*) sekä sen muuttumisnopeudesta (*näyte(Δx)*) sekä kahdesta niiden pohjalta lasketusta likiarvosta (*approx1(x)* ja *approx2(x)*). Ensimmäinen likiarvo lasketaan *Sample & Hold* -menetelmällä, eli pidetään aina viimeisimmän saapuneen näytteen arvo. Toinen likiarvo lasketaan ekstrapoloimalla aiemmin saatua näytettä sen muuttumisnopeuden mukaan. Niin kauan muuttumisnopeudessa ei tapahdu äkkinäisiä muutoksia, kuvaa toinen likiarvo alkuperäistä näytteen *x* arvoa tarkasti. Äkkinäiset muutokset kuitenkin aiheuttavat likiarvossa heittelehtimistä (kuva 5), joka näkyy asiakasohjelmassa pelihahmojen tökkimisenä. (Halmetoja 2014.)



KUVA 5. Tuntemattoman suureen approksimointi ekstrapoloimalla.

Interpolaatiossa ei tapahdu hyppimistä, mutta se lisää asiakasohjelmassa tapahtuvaa viivettä. Interpolaatiossa määritetään suureen kukin arvo simulaation *edellisen* sekä *toiseksi edellisen* arvon välille. Näin asiakasohjelmassa oleva arvo on aina vähintään yhden simulaatioaskelen verran jäljessä palvelimella olevaa arvoa, joka tulee ottaa huomioon joko asiakasohjelman tai palvelinohjelman laskelmissa. Jos tätä *interpolaatioviivettä* ei ota huomioon, joutuu pelaaja ennakoimaan sen itse pelissä. Näin esim. ammuntapelissä kohteeseen osuakseen ei voi ampua sinne, missä kohde on, vaan sinne, mihin kohde on menossa.

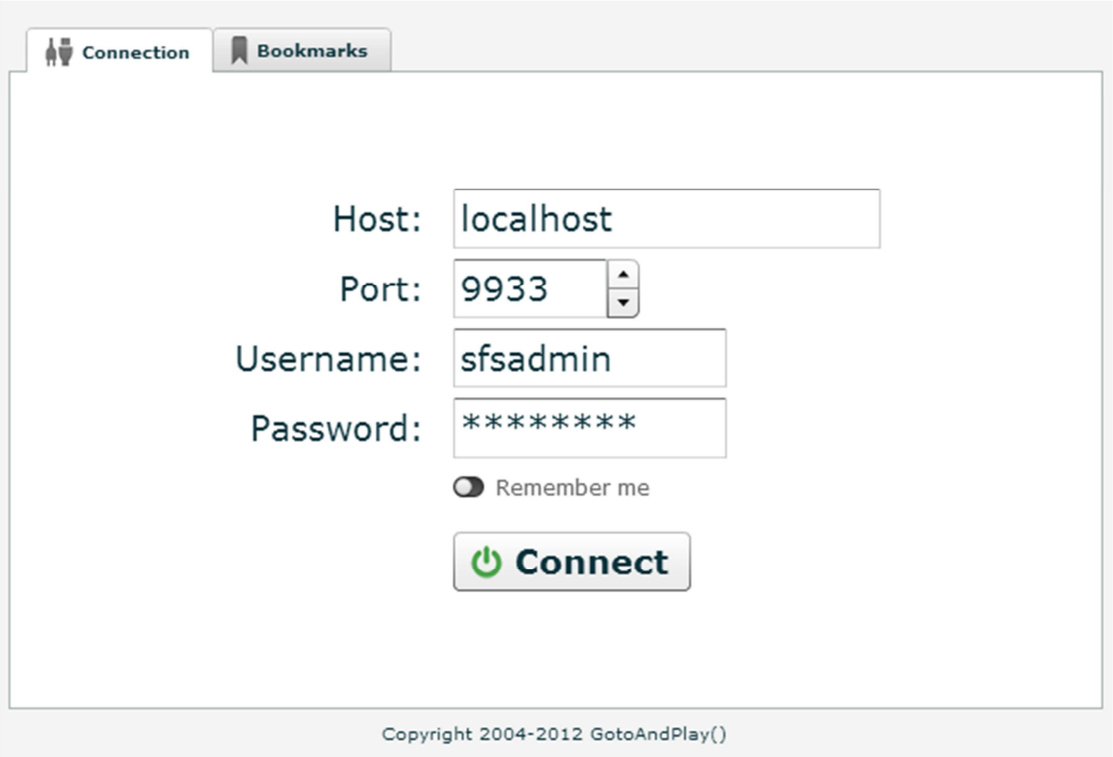
4 VERKKOMONINPELIN TOTEUTUS JA KÄYTTÖÖNOTTO

Esimerkkipeliksi valitsin virtuaalisen hipan, koska se on yksinkertainen selittää ja toteuttaa, mutta vaatii suuren määrän asiakasohjelmien välistä viestinvälitystä. Kuvaan tässä luvussa pelipalvelimen asennuksen sekä konfiguroinnin sekä itse pelin toteutuksen. Tämän tehtävän kannalta olennaiset Web -osoitteet löytyvät liitteestä 1.

4.1 Pelipalvelimen asennus ja konfigurointi

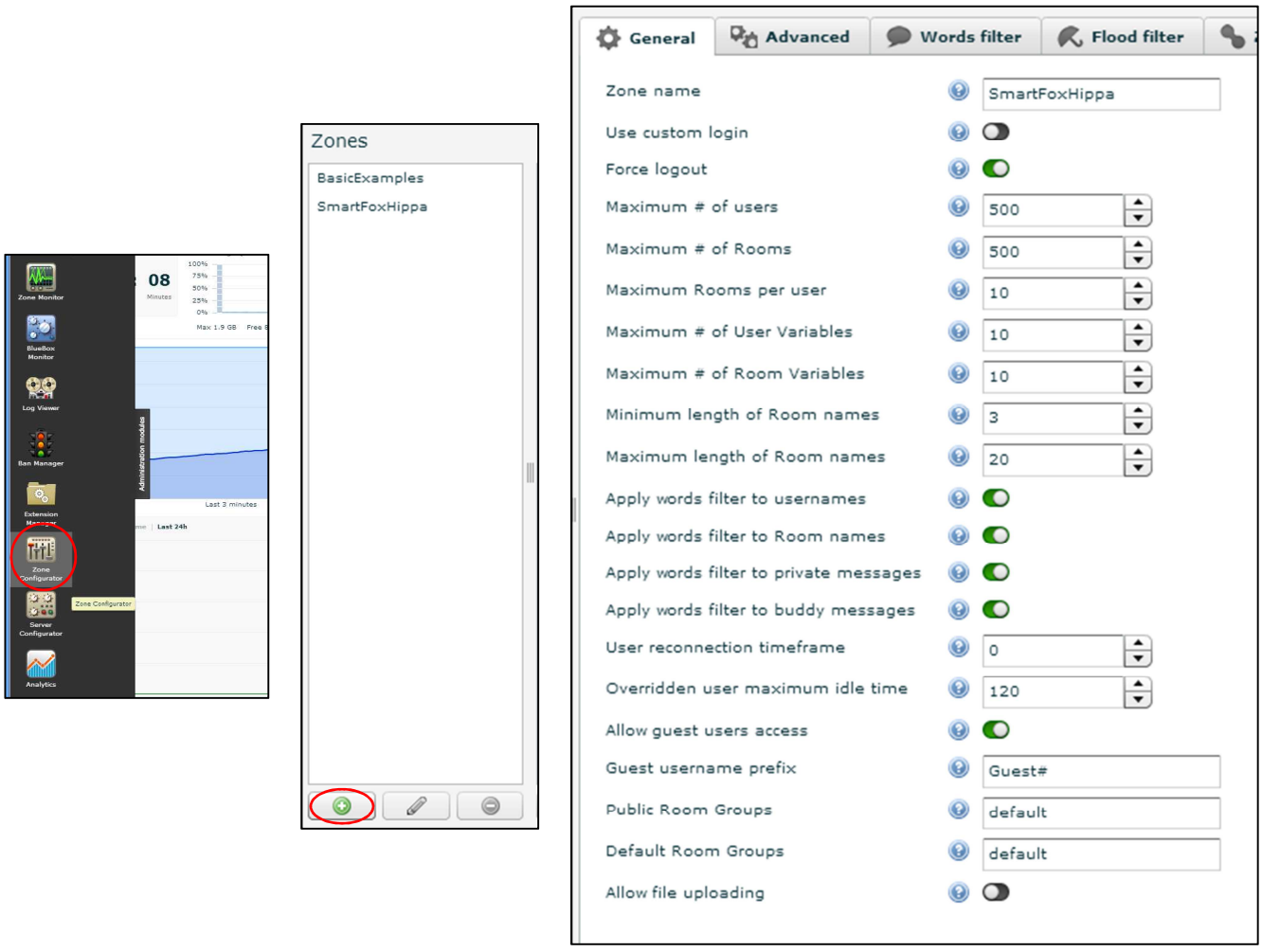
Pelipalvelimena toimii SmartFox Server 2X. Sen voi ladata ilmaiseksi käyttöönsä osoitteesta *http://smartfoxserver.com/download/sfs2*. Samalla sivulla on myös asennusohjeet. Huomaa asentaessasi, että **uusilla Windows-versioilla** (Vistasta eteenpäin) ohjelma tulee asentaa käyttäjän **kotikansion alaisuuteen**, jotta se toimisi oikein (SmartFox Server 2X - Installing under Windows 2014). Ohjelman asentaminen vaatii pääkäyttäjän oikeudet, eikä kotikansioon asennettu ohjelma toimi muilla käyttäjillä. Asennusohjelma kysyy, asennetaanko ohjelmisto palveluna (*service*) vai itsenäisesti käynnistettävänä sovelluksena. Kehityskäytössä itsenäisesti käynnistyvä sovellus on helpompi, koska sen tulostetta on helpompi seurata. Varmista asentaessasi ohjelmaa, että palomuuri ei estä sen käyttöä.

Kun ohjelmisto on asennettu ja käynnistetty, siirry osoitteeseen *http://localhost:8080/admin/* avataksesi hallinnointinäkymän. Sen tulisi näyttää kuvan 6 kaltaiselta. Syötä kuvan mukaiset tiedot, salasana on **sfsadmin**.



KUVA 6. SmartFox Serverin kirjautumisnäkymä

Hallintanäkymän **vasemmassa laidassa** on teksti *Administration modules*. Kun sen päälle vie hiiren, avautuu hallinnointivalikko (kuva 7a). Tässä valikossa on useita toimintoja (*moduuleja*) pelipalvelimen hallinnointiin. *Dashboard* -moduuli on yleisnäkymä, josta näkee palvelimen kuormituksen sekä muita järjestelmän valvomisen kannalta tärkeitä tietoja. *Zone Monitor* -työkalulla voi seurata pelialueita, huoneita ja yksittäisiä pelaajia. Se on arvokas virheenjäljityksessä: sen avulla voi mm. varmistaa, että käyttäjien tiedot ovat oikeasti sitä miltä ne asiakasohjelman mukaan näyttävät. *Analytics*- ja *Log Viewer* -moduulien avulla voi seurata palvelimen toimintaa yksityiskohtaisesti. *Server Configurator* -moduulissa säädetään palvelimen yleiset asetukset.



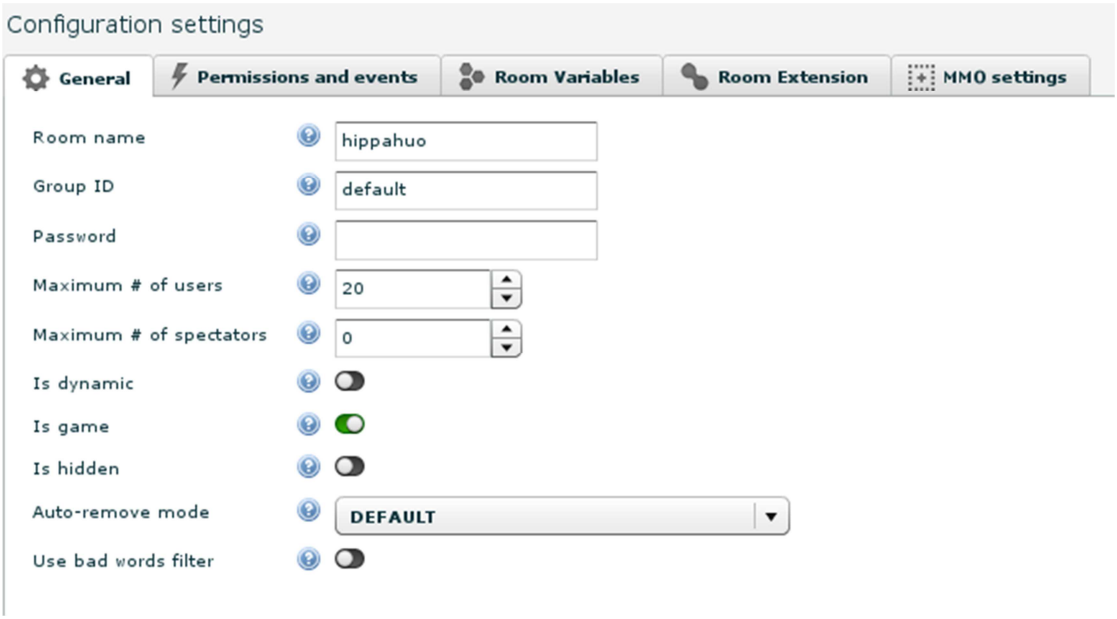
a **b** **c**

KUVA 7. Pelialueen asetukset

SmartFox Serverissä kullekin pelille tulee luoda oma pelialueensa (*zone*). Näin samalla palvelimella voi ajaa useita eli pelejä ilman, että ne häiritsisivät toisiaan. Pelialueen luonti tapahtuu seuraavasti:

1. Valitse hallinnointivalikosta vaihtoehto *Zone Configurator*.
2. Klikkaa Zone Configurator -näkymässä Zones -listan alapuolella olevaa, pientä vihreää plus-ikonia lisätäksesi uuden pelialueen (kuva 7b).
3. Syötä kuvan 7c mukaiset tiedot lomakkeelle ja klikkaa *Submit*.

Useimmat lomakkeen kentistä ovat melko yksiselitteisiä, mutta muutamat vaativat selvennystä: *Use custom login* mahdollistaa kirjautumisen esim. Active Directory -järjestelmää käyttämällä, jolloin peli on helpompi integroida muihin järjestelmiin. Integraatiokomponentit on kuitenkin kehitettävä itse. *Force logout* taas kirjaa automaattisesti ulos pelaajat, joihin ei saada yhteyttä verkon yli. *Allow guest users access* sallii pelaajien liittymisen huoneeseen kirjautumatta, jolloin ne merkitään *vieraiksi*. Koska esimerkkipelissä ei ole käyttöliittymää kirjautumiselle, tulee tämä asetus olla päällä.



KUVA 8. Huoneen asetukset

Pelialueen lisäksi tarvitaan huone (*room*). Pelialue jaetaan huoneeksi aiemmassa kohdassa *”Skaalautuvuus”* kuvatuista syistä. Huoneen luominen muistuttaa pelialueen luomista, ja se tapahtuu seuraavasti:

1. Valitse hallinnointivalikosta vaihtoehto *Zone Configurator*.
2. Klikkaa Zone Configurator -näkymässä Zones -listassa näkyvää *SmartFox-Hippa* -tekstiä.
3. Klikkaa ilmestyvän Rooms -listan alapuolella olevaa, vihreää plus-ikonia.
4. Syötä kuvan 8 mukaiset tiedot lomakkeelle ja klikkaa *Submit*.

Kun pelipalvelin on asennettu, konfiguroitu ja käynnissä, siihen voi ottaa yhteyden asiakasohjelmalla. Asiakasohjelmaan pitää syöttää vain pelipalvelimen IP-osoite sekä

portti (oletuksena 9933) ja muodostaa yhteys. Pelien kehittämiseen ei välttämättä tarvita siis palvelinlaajennusta, ellei haluta pelipalvelimesta autoritaarista.

4.2 Unity-asiakasohjelma

Esimerkkipelin pääasiakasohjelma on toteutettu Unityn avulla. Pelin uusin versio on saatavilla osoitteesta https://bitbucket.org/ilmo_euro/smartfoxhippa/. Tässä luvussa olevat **koodiesimerkit ovat *ServerConnection.cs* -tiedostosta**, johon kaikki SmartFox -spesifinen koodi on kapseloitu. Muu koodi käyttää tiedostossa määriteltyä *ServerConnection* -luokkaa viestinvälitykseen ja synkronointiin.

SmartFox Serverin kotisivuilta on saatavana Dot NET/Mono -kirjasto asiakasohjelmien käyttöön. Koska Unity sisältää Mono-ympäristön, on kirjaston käyttöönotto helppoa: kopioidaan vain DLL -tiedosto projektin Assets -kansioon, jolloin se on käytettävissä C# -skripteistä käsin. Koska Unityn JavaScript -murre (UnityScript) ei tue nimiavaruuksia, sitä ei voi käyttää SmartFox Server -luokkien kanssa. Kaikki kirjaston luokat sijaitsevat *Sfs2X* -nimiavaruuden alla (kuva 9).

```
using Sfs2X;
using Sfs2X.Core;
using Sfs2X.Controllers;
using Sfs2X.Requests;
using Sfs2X.Entities;
using Sfs2X.Entities.Variables;
using Sfs2X.Util;
```

KUVA 9. Asiakaskirjaston nimiavaruudet

Kaikki viestintä pelipalvelimen ja asiakasohjelman välillä tapahtuu *SmartFox* -oliota käyttämällä (kuva 9). Olio luodaan *new*-operaattorilla ilman parametreja, ja sille välitetään yhdistämistä varten tarvittavat tiedot *ConfigData* -olion avulla. *Host* -kentässä on pelipalvelimen osoite, *Port* -kentässä UDP-portin osoite (jos palomuuri estää sen, yritetään HTTP-porttia), *Zone* -kentässä pelialue siltä varalta, että palvelimella pyörii useita eri pelejä, sekä *UseBlueBox* -kentässä tieto siitä, yritetäänkö UDP-yhteyden epäonnistuessa HTTP-yhteyttä.

Toinen, dokumentaation paremmin käsittelemä tapa asetusten säätämiseksi on *sfsconfig.xml* -niminen tiedosto. Asiakasohjelmisto etsii sitä oletuksena työhakemistosta, mutta Unityn tapauksessa työhakemiston kanssa työskentely, kuten tiedoston etsimi-

nen mielivaltaisista paikoista, voi olla hankalaa. Kun asetukset säädetään ohjelmallisesti, voi ne helposti kysyä käyttöliitymän avulla tai asettaa Unityn kenttäeditorista käsin.

```
sfs = new SmartFox();

…

ConfigData cfg = new ConfigData();
cfg.Host = host;
cfg.Port = port;
cfg.Zone = zone;
cfg.UseBlueBox = true;
sfs.Connect(cfg);
```

KUVA 10. Yhteyden muodostaminen palvelimelle

Viestit pelipalvelimelle lähetetään *Send* -metodilla, ja pelipalvelimelta asiakasohjelmalle tulevat viestit käsitellään tapahtumina (*event*). Viestejä voi kuunnella *SmartFox* -luokan *AddEventListener* -metodin avulla. Kuvassa 10 mainitut *On* -alkuiset metodit ovat kaikki *ServerConnection* -luokan jäseniä. En käsittele suurinta osaa niitä tässä, mutta ne voi löytää BitBucketissa olevasta lähdekoodista.

SmartFox Server ilmaisee virhetilanteet kahdella tavalla. Virhetilanne voi olla joko osa positiivista tapahtumaa, kuten *CONNECTION* tai *LOGIN*, jolloin *Params* -kokoelman *success* -alkio on epätosi, tai oma virhetapahtumansa, kuten *CONNECTION_LOST* tai *ROOM_JOIN_ERROR*. Tavallista poikkeusten käsittelyä ei voida käyttää SmartFoxin asynkronisesta luonteesta johtuen. Kuhunkin virhetilanteeseen liittyy lisätietoja, joita tarvitaan virheistä toipumiseen sekä virheanalytiikkaan; tiedot löytyvät *Params* -kokoelmasta eri avainten alta. Tarkista kuhunkin kokoelmaan liittyvät tiedot SmartFoxin asiakaskirjaston API-dokumentaatiosta (liite 1.)

```
sfs.AddEventListener (SFSEvent.CONNECTION, OnConnect);
sfs.AddEventListener (SFSEvent.CONNECTION_LOST, OnConnectionLost);
sfs.AddEventListener (SFSEvent.LOGIN, OnLogin);
sfs.AddEventListener (SFSEvent.ROOM_JOIN, OnJoinRoom);
sfs.AddEventListener (SFSEvent.ROOM_JOIN_ERROR, OnJoinRoomError);
sfs.AddEventListener (SFSEvent.USER_VARIABLES_UPDATE,
OnUserVariablesUpdate);
sfs.AddEventListener (SFSEvent.USER_ENTER_ROOM, OnUserEnterRoom);
sfs.AddEventListener (SFSEvent.USER_EXIT_ROOM, OnUserExitRoom);
```

KUVA 11. Tapahtumakäsittelijöiden kytkeminen

Koska SmartFox Server on alun perin dynaamisesti tyypitettyjä ohjelmointikieliä (JavaScript, ActionScript) varten luotu, siinä käytetään *Dictionary* -tyyppisiä kokoelmia tapahtumien sisältämien tietojen välittämiseen luokkahierarkian sijasta. Sen sijaan että esim. tieto palvelimeen yhdistämisen onnistumisesta olisi *ConnectEvent* -olion *Success* -jäsenenä, on se *BaseEvent* -olion *Params* -kokoelman alkio, jonka avain on *”success”* (ks. kuva 11).

Eri tapahtumien *Params* -kokoelmat poikkeavat huomattavasti toisistaan. Esimerkiksi *CONNECT* -tapahtuman yhteydessä tieto onnistumisesta on *success* -alkiossa, mutta *ROOM_JOIN* -tapahtuman epäonnistuminen ilmaistaan erillisellä *ROOM_JOIN_ERROR* -tapahtumalla; sen sijaan *Params*issa on vain huoneen nimi *room* -alkiona. *Params* -kokoelman parametrit löytyvät API-dokumentaatiosta kunkin SFSEvent -luokan kentän yhteydestä.

```
private void OnConnect(BaseEvent evt) {
     if ((bool)evt.Params[''success'']) {
       Debug.Log (''Logging in...'');
       sfs.Send(new LoginRequest(''''));
     } else {
       Debug.Log (''Connection failed: ''
         + (string)evt.Params[''errorMessage'']);
     }
}
```

KUVA 12. Esimerkki tapahtumakäsittelijästä

Koska kokoelman alkiot voivat olla mitä tyyppiä tahansa, joudutaan tulos tyyppimuuntamaan (*cast*) oikean tyyppiseksi. Mahdollisuuksia virheille, joita kääntäjä ei löydä, on siis kaksi: merkkijonotyyppisen avaimen voi kirjoittaa väärin, tai tyyppimuunnosta yritetään väärään tyyppiin. Koin tämän itse kantapään kautta: oletin, että muuttujien päivitystapahtuman sisältämä *”changedVars”* -kenttä olisi tyyppiä *List<UserVariable>*. Debuggerin avulla kuitenkin selvisi, että SmartFox käytti vanhempia, tyypittämättömiä kokoelmatyyppejä, tässä tapauksessa ArrayListiä, joka toteuttaa IList -rajapinnan (kuva 13). Joudun siis käyttämään kahdentyyppisiä kokoelmia: vanhoja tyypittämättömiä sekä uusia geneerisiä. Annoin vanhalle *System.Collections* -nimiavaruudelle aliaksen *OldCollections*.

Koska *ServerConnection* -luokka vain välittää viestejä pelimaailman ja SmartFoxin välillä, reagoidaan muuttujien muutoksiin kutsumalla luokan omia *userVariableHandlers* -kokoelman jäseniä. Tähän kokoelmaan on aiemmin liitetty pelimaailmassa olevia hahmoja, jotka päivittävät omaa sijaintiansa saapuvien viestien perusteella.

```
private void OnUserVariablesUpdate(BaseEvent evt) {
   User user = (User)evt.Params ["user"];
   OldCollections.ArrayList changedVars =
      (OldCollections.ArrayList) evt.Params ["changedVars"];
   foreach (string varName in changedVars) {
      foreach (var entry in userVariableHandlers) {
         if (entry.userId == user.Id && entry.variable == varName) {
            entry.handler (user.GetVariable (varName).Value);
         }
      }
   }
}
```

KUVA 13. Viestien lähettäminen pelipalvelimelle

Viestien lähettäminen pelipalvelimelle ja sitä kautta muille asiakasohjelmille tapahtuu *SmartFox* -olion *Send* -metodia käyttämällä. Toisin kuin tapahtumakuuntelijoissa, *Send* -metodissa käytetään polymorfismia: sille annetaan *IRequest* -rajapinnan toteuttava olio, joka sisältää viestin tiedot. Esimerkiksi palvelimelle kirjautuminen tapahtuu *LoginRequest* -olion avulla, jonka konstruktori ottaa parametrikseen käyttäjänimen ja mahdollisesti salasanan (kuva 14). Huomaa että *Send* -komento palaa heti eikä anna palautusarvoa, joten viestien tuloksia joudutaan odottamaan tapahtumakuuntelijoilla.

Koska asiakasohjelman käynnistämisen ja pelin aloittamisen välillä on monta vaihetta, joudutaan tapahtumakuuntelijoita ketjuttamaan. Yhteyden muodostamisen jälkeen ammutaan *CONNECT* -tapahtuma, jonka käsittelijä lähettää sisäänkirjautumispyynnön. Sisään kirjautuminen taas ampuu *LOGIN* -tapahtuman, jonka käsittelijä lähettää huoneeseenliittymispyynnön. Näin vaikka ohjelma näyttää ulkoisesti vastaavan tapahtumiin mielivaltaisessa järjestyksessä, on sen käynnistysvaihe luonteeltaan perättäinen.

```
private void OnConnect(BaseEvent evt) {
   if ((bool)evt.Params["success"]) {
       Debug.Log ("Logging in...");
       sfs.Send(new LoginRequest(""));
```

```
    } else {
        Debug.Log ("Connection failed: "
          + (string)evt.Params["errorMessage"]);
    }
}

private void OnLogin(BaseEvent evt) {
    Debug.Log ("Joining room...");
    sfs.Send(new JoinRoomRequest(room));
}

private void OnJoinRoom(BaseEvent evt) {
…
```

KUVA 14. Asiakasohjelman käynnistysvaiheen tapahtumakäsittelijät

Tässä tapauksessa asiakasohjelma on kehitetty käyttäen Unityä, mutta sen voisi toteuttaa myös muita teknologioita käyttäen samankaltaisella ohjelmakoodilla, esimerkiksi Flash -liitännäisen tai HTML5/JavaScript -tekniikoiden avulla. Asiakaskirjasto on Dot NET -moduuli (*assembly*), jossa ei ole mitään Unity -spesifiä, joten sitä voi käyttää sellaisenaan esim. MonoGame-, Windows Store- tai Windows Phone-sovelluksissa. Jos käytetään erilaisia asiakasohjelmia, pitää varmistaa että simulaatio toimii niissä samalla tavalla, jotta peli toimisi oikein.

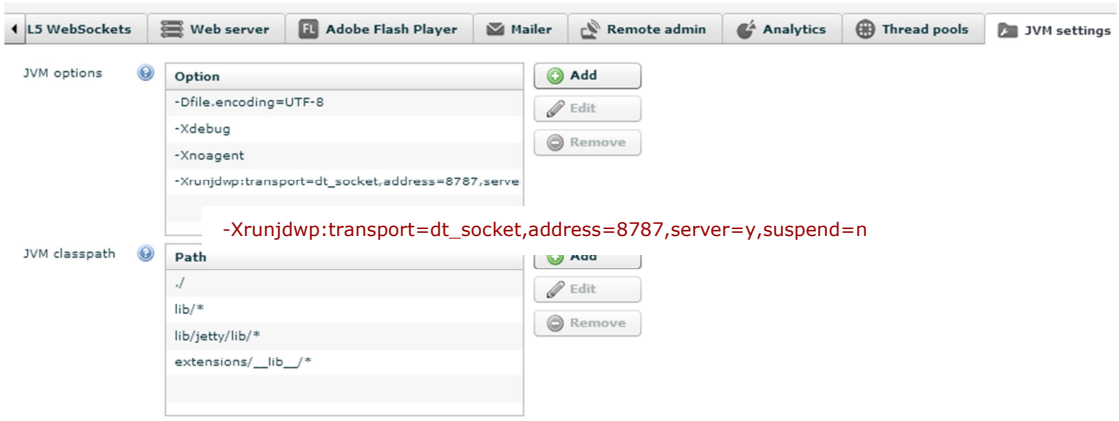
4.3 Palvelinlaajennus

Hippapeli toimii myös täysin ilman palvelinohjelmointia, mutta esimerkiksi kulloisenkin hipan määrittäminen sekä huijauksen esto tulevat yksinkertaisemmiksi palvelinlaajennuksella. SmartFox Serverin laajennuksia voi tehdä Javan tai Pythonin avulla; käytän tässä esimerkissä Javaa sen suosion takia.

Palvelinlaajennusta kehitettäessä virheenetsintä (*debugging*) on välttämätöntä. Se onnistuu tavallisella Java -debuggerilla, joka on sisäänrakennettuna useimpiin kehitysympäristöihin, kuten NetBeansiin. Virheenetsintää varten tulee SmartFox Server konfiguroida kuvan 15 mukaisesti. Asetukset löytyvät *Administration Modules* -valikon *Server Administration* -moduulista. Kun asetukset ovat kunnossa, tulee SmartFox Server käynnistää uudelleen.

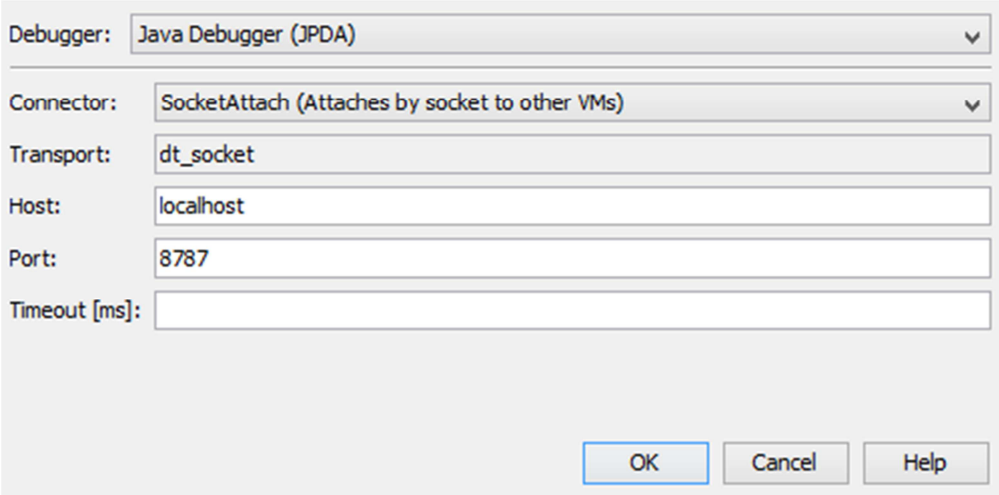
Muita tapoja virheenetsintään on lokiin kirjoittaminen, johon voi käyttää esimerkiksi tunnettua *log4j*-kirjastoa, sekä tietojen tarkasteleminen *Zone Monitor* -

hallinnointimoduulin avulla. *Zone Monitor* -moduulissa voi tarkastella mm. pelaajien lukumäärää eri huoneissa, huoneiden muuttujia (huoneen asetusnäkymän *Room Variables* -välilehdellä) sekä käyttäjämuuttujia (pelaajan asetusnäkymän *User Variables* -välilehdellä).



KUVA 15. Virheenetsintätilan asetus SmartFox Serverissä

Itse virheenetsintä käynnistetään NetBeansissa *Debug*-valikon *Attach Debugger*-toiminnolla, kun palvelinlaajennusprojekti on aktiivinen. Toiminto avaa kyselyikkunan, johon syötetään kuvan 16 mukaiset tiedot. Virheenetsinnän ollessa käynnissä voidaan käyttää samoja toimintoja kuin *Debug* -komennolla käynnistetyn ohjelman kanssa: asettaa pysähdyspisteitä (*breakpoint*), tutkia muuttujien arvoja sekä askeltaa ohjelmaa.



KUVA 16. Virheenetsintätilan asetus NetBeansissa

Palvelinlaajennusten laatimiseen tarvittavat luokat sijaitsevat *sfs2x.jar* sekä *sfs2x-core.jar* -nimisissä tiedostoissa SmartFox Server -kansion *SFS2X/lib/* -alikansiossa. Nämä tiedostot pitää linkittää Java-projektiin, jotta laajennuksen voi kääntää. Luokat ovat *com.smartfoxserver.v2* -paketin alaisuudessa (kuva 17). Palvelinluokat ja asiakasohjelmiston luokat ovat melko samankaltaisia nimeämiseltään ja tarkoitukseltaan, mutta niiden tarkoitus on pyörittää pelimaailman simulaatiota, ei vastata verkkoliikenteestä.

Jos *sfs2x.jar* ja *sfs2x-core.jar* -luokat on linkitetty projektiin, on palvelinlaajennuksen paketointi ja asentaminen helppoa: käännetään projekti JAR-tiedostoksi ja pudotetaan se pelipalvelimen *extensions* -kansion *SmartFoxHippaExtension* -alikansioon (alikansio on luotava tarvittaessa). Huomaa, että kansion nimessä oleva Extension on **pakollinen**, muuten palvelin ei tunnista laajennusta. JAR-tiedoston lisäksi kansioon kannattaa lisätä tyhjä *config.properties* -niminen tiedosto, johon voi myöhemmin laittaa laajennuksen tarvitsemat asetukset. Oletuksena laajennus ladataan uudestaan aina, kun se muuttuu, mutta tuotantokäytössä tämä toiminto kannattaa kytkeä pois päältä Zone Configuratoria käyttäen, pelialueen (*zone*) *Zone Extension* -välilehden *Reload mode* -valitsimesta.

```
import com.smartfoxserver.v2.SmartFoxServer;
import com.smartfoxserver.v2.core.ISFSEvent;
import com.smartfoxserver.v2.core.ISFSEventListener;
import com.smartfoxserver.v2.core.SFSEventParam;
import com.smartfoxserver.v2.core.SFSEventType;
import com.smartfoxserver.v2.entities.Room;
import com.smartfoxserver.v2.entities.User;
import com.smartfoxserver.v2.entities.variables.RoomVariable;
import com.smartfoxserver.v2.entities.variables.SFSRoomVariable;
import com.smartfoxserver.v2.exceptions.SFSException;
import com.smartfoxserver.v2.extensions.SFSExtension;
```

KUVA 17. Palvelinlaajennuksen tarvitsemat luokkaviittaukset

Kuhunkin pelihuoneeseen voi liittää yhden laajennusolion, jonka tulee olla *SFSExtension* -luokan aliluokan ilmentymä. Tämä olio voi tietenkin sisältää myös muita olioita; se on hieman kuin pääluokka (*main class*) Java -komentoriviohjelmissa (kuva 18). Muista että **luokan nimen tulee päättyä Extension**, muuten SmartFox Server ei löydä sitä. Tässä esimerkissä laajennusolion tarkoitus on päivittää tietoa siitä, mikä pelaaja on kulloinkin hippa. Tieto pidetään kahdessa huonemuuttujassa (*room variable*):

tagTimer ja *tagHolder*. *TagTimer* -muuttuja on laskuri, joka laskee ajan viimeisimmästä hipan vaihtumisesta. Sitä tarvitaan, jotta hippa ei vaihtuisi liian usein. *TagHolder* pitää sisällään nykyisen hipan *id*-numeron, tai numeron -1 jos kukaan ei ole hippa.

Kun laajennus käynnistetään (heti pelialueen käynnistämisen jälkeen), kutsutaan sen *init*-metodia. Metodissa liitetään laajennuksen tapahtumakuuntelijat ja -käsittelijät SmartFoxin tapahtumiin ja suoritetaan muut alustustoimenpiteet. Esimerkkipelissä tarkistetaan säännöllisin väliajoin, mikä pelaajista on hippa, ja asetetaan huonemuuttujat vastaavasti. Käytän kyseistä tehtävää varten SmartFoxin ajoitustoimintoa (*task scheduler*). Lisäksi tarvitaan tapahtumakuuntelija pelaajan huoneeseen saapumiselle, jotta ensimmäinen huoneeseen saapuva pelaaja asetetaan hipaksi.

```
public class SmartFoxHippaExtension extends SFSExtension {
…
@Override
public void init() {
     SmartFoxServer
            .getInstance()
            .getTaskScheduler()
            .scheduleAtFixedRate(
                new UpdateTagTask(),
                100,
                100,
                TimeUnit.MILLISECONDS);
    addEventListener(SFSEventType.PLAYER_JOIN_ROOM,
                     new PlayerJoinListener());
}
```

KUVA 18. Palvelinlaajennuksen alustaminen

Itse tapahtumakuuntelija toimii samoin kuin asiakasohjelman tapahtumakuuntelijat: sillä on yksi metodi, *handleServerEvent*, joka ottaa *ISFSEvent* -tyyppisen olion parametrikseen. Koska kaikki tapahtumakuuntelijat ovat samaa tyyppiä, joudutaan tapahtuman lisätiedot pyytämään *getParameter* -metodilla tapahtumaoliolta. Toisin kuin C# -kirjaston tapauksessa, metodin parametriksi voidaan antaa jokin *SFSEventParam* -luokan symbolisista vakioista, jolloin kääntäjä hylkää kirjoitusvirheet todennäköisemmin. Koska metodin palautustyyppi on *Object*, joudutaan sen paluuarvot kuitenkin tyyppimuuntamaan (*cast*) halutun tyyppiseksi, jolloin pitää olla tarkkana tyyppivirheiden varalta.

Kun asiakasohjelmistossa viestintä pelipalvelimelle tapahtuu *SmartFox* -tyyppisen olion avulla, niin palvelinlaajennuksissa suositeltu tapa pelipalvelimen käsittelemiseksi on *SFSExtension* -kantaluokan *getApi* -metodi. Esimerkissä sitä käytetään huonemuuttujien (*room variables*) asettamiseksi. Huomaa, että määrittelemämme *PlayerJoinListener* -luokka on ylemmän *SmartFoxHippaExtension* -luokan **sisäluokka,** joten siitä voidaan kutsua ulomman luokan metodeja, tässä tapauksessa *getApi* -metodia (kuva 19).

```
private class PlayerJoinListener implements ISFSEventListener {

    @Override
    public void handleServerEvent(ISFSEvent event) throws SFSException {
        Room room = (Room)event.getParameter(SFSEventParam.ROOM);
        User user = (User)event.getParameter(SFSEventParam.USER);
        if (room.getUserList().size() == 1) {
            getApi().setRoomVariables(null, room, Arrays.asList(
                new SFSRoomVariable("tagHolder", user.getId()),
                new SFSRoomVariable("tagTimer", 0)));
        }
    }
}
```

KUVA 19. Palvelinpään tapahtumakäsittelijä

Muuttujien päivittäminen tapahtuu toisessa sisäluokassa. Se toteuttaa Javan standardin *Runnable* -rajapinnan, joten sen voisi ajaa myös käyttäen Javan säikeistystoimintoja. Jos näin tehdään, pitää kuitenkin pitää huolta, ettei säikeiden välille muodostu yhtäaikaisuusongelmia. Itse työ suoritetaan *run* -metodissa. Koska kyseessä on sisäluokka, voidaan taas kutsua ulomman luokan eli *SmartFoxHippaExtension* -luokan metodeja, kuten *getParentZone* ja *getApi* (kuva 20).

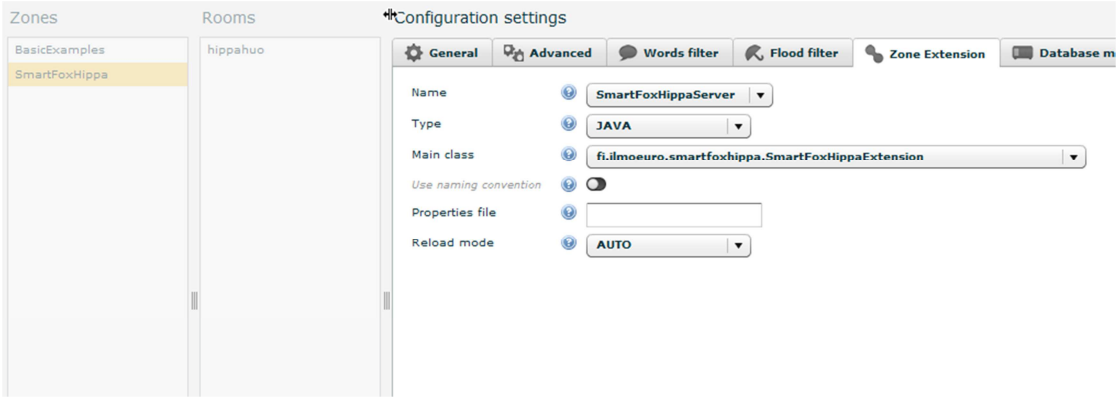
Koska laajennukset ovat alue- eivätkä huonekohtaisia, hippamuuttujien päivittäminen vaatii kaksi sisäkkäistä silmukkaa: ulommassa käydään läpi huoneet ja sisemmässä kaikki huoneessa olevat pelaajat. Tämä on suorituskyvyltään huono keino, ja se toimii vain esitysmielessä; parempi tapa olisi käyttää SmartFoxin kyselylausekkeita (*match expressions*). Kyselylausekkeilla voidaan muodostaa ehtoja, jonka mukaan käyttäjiä tai huoneita rajataan. Esimerkkejä näistä ehdoista ovat mm. ”kaikki pelaajat, jotka ovat liittyneinä vähintään 3 huoneeseen, ja joiden *age* -muuttuja on pienempi kuin 20”.

```
private class UpdateTagTask implements Runnable {
    @Override
    public void run() {
        List<Room> rooms = getParentZone().getRoomList();
        for (Room room : rooms) {
            for (User user : room.getPlayersList()) {
                if (hasTag(room, user)) {
                    transferTag(room, user);
                } else {
                    idle(room);
                }
            }
        }
    }
    …
}
```

KUVA 20. Ajastettu tehtävä

Tässä palvelinlaajennuksessa on muitakin puutteita: siinä oletetaan, että *run* -metodia ei koskaan ajeta kahta kertaa samaan aikaan, ja että muuttujia ei muuteta kesken pelin ajon. Näiden puutteiden korjaamiseen voi käyttää esim. Javan yhtäaikaisuustyökaluja (atomiset *AtomicLong*, *AtomicInteger* yms. tyypit sekä lukot).

Palvelinlaajennukset ovat tavallisia Java-kirjastotiedostoja (JAR), joten niiden kehityksessä voi käyttää kaikkia Java-kehitystekniikoita ja -apuvälineitä, kuten Maven ja Ant. Laajennusta kehitettäessä tulee ottaa huomioon, että SmartFox Server käyttää asennuksen mukana tulevaa, **omaa Java-virtuaalikonettaan** järjestelmän virtuaalikoneen sijasta. Sen vuoksi laajennusta kehitettäessä tulee JAR-tiedoston olla **yhteensopiva JDK 6:n kanssa**, eikä Java 7:n tai Java 8:n toimintoja voi käyttää. Yhteensopivuustason voi asettaa NetBeansin tapauksessa projektin asetuksista (*File → Project Properties → Sources → Source/Binary Format → JDK 6*).



KUVA 21. Palvelinlaajennuksen käyttöönotto

Palvelinlaajennoksen voi ottaa käyttöön SmartFox Serverin hallinnointinäkymän Zone Configurator -moduulissa: valitse Zones -listasta SmartFoxHippa ja klikkaa kynäikonia. Valitse aukeavasta *Configuration Settings* -näkymästä *Zone Extension* -välilehti (kuva 21). Jos JAR-tiedosto on kopioitu oikeaan kansioon (*SmartFox_2X/SFS2X/extensions/SmartFoxHippaServerExtension*), niin *Name*-pudotusvalikossa pitäisi näkyä vaihtoehto *SmartFoxHippaServer*. *Type*-pudotusvalikossa valitaan, onko laajennus Java- vai Python-ohjelmointikielellä laadittu; tässä esimerkissä käytämme Java-ohjelmointikieltä. *Main class* -pudotusvalikossa valitaan laajennuksen pääluokka, joka on siis *SFSExtension* -luokan alaluokka. Normaalisti pudotusvalikossa listataan kaikki luokat, mutta jos niitä on monta, voi *Use naming convention* -kytkimen kytkeä päälle, jolloin listataan vain ne luokat, joiden nimi loppuu tekstiin *Extension*. Oletuksena laajennuksen tarvitsema konfiguraatio ladataan *config.properties* -nimisestä tiedostosta, mutta sen voi vaihtaa toiseen *Properties file* -kentän avulla. *Reload mode*n avulla valitaan, käynnistetäänkö laajennus uudestaan JAR-tiedoston muuttuessa; AUTO-vaihtoehto käynnistää laajennuksen uudestaan aina, kun tiedostoa muutetaan, ja MANUAL-vaihtoehtoa käytettäessä laajennus tulee käynnistää uudelleen manuaalisesti. Laajennusta kehitettäessä AUTO on kätevä, koska muutosten jälkeen ei aina tarvitse käynnistää palvelinta uudestaan tai käyttää uudelleenlataustoimintoa. Tuotantokäytössä MANUAL tai NONE on parempi, sillä jos pelipalvelin on käynnissä, niin laajennuksen uudelleenkäynnistys saattaa aiheuttaa ylimääräisiä viiveitä pelaajille tai johtaa pelien katkeamiseen, joten uudelleenkäynnistykset on parempi tehdä esimerkiksi huoltokatkon aikana. Kun asetukset ovat kunnossa, klikkaa näkymän *Submit*-painiketta.

5 PÄÄTÄNTÖ

Reaaliaikaisen verkkomoninpelin tekninen toteuttaminen on haastavaa, ja sitä varten tulee hallita sekä matalan tason tekniikoita (IP, TCP, UDP) että laajoja järjestelmäarkkitehtonisia kokonaisuuksia. Valmiin pelipalvelinratkaisun käyttäminen **ei poista sitä vaatimusta**: esimerkiksi mahdollisten virhetilanteiden ja suorituskykyongelmien ilmetessä tulee pelin kehittäjien ymmärtää koko järjestelmän toiminta. Verkkomoninpeleihin tarvitaan yleensä toiminnot esimerkiksi pelien etsimistä ja aloittamista, pelaaji-

en välistä keskustelua, pistetaulukoita ja pelitilanteen synkronointia varten, ja valmiit pelipalvelut vähentävät työmäärää näiden toimintojen toteuttamisessa. Pelipalvelimet sitovat kuitenkin ohjelmistokehittäjät käyttämään palvelimen kehittäjän hyväksi katsomia ratkaisuja, ja toisinaan palvelimen tuoma abstraktiotaso saattaa tehdä joistain ongelmista hyvin hankalia, mitkä olisivat matalan tason tekniikalla toteutettuna helppoja.

Tärkein tätä opinnäytetyötä tehdessäni oppimani asia oli, että verkkomoninpelit ovat erittäin monimutkaisia, hajautettuja reaaliaikaisia järjestelmiä, ja on ihmeellistä, kuinka saumattoman immersiokokemuksen niillä voi saavuttaa. Kun aletaan rakentaa peliä puhtaalta pöydältä, ovat valmiit komponentit erittäin arvokkaita työkaluja kokemattomalle kehittäjälle: vaikka niitä ei käyttäisi valmiissa pelissä, antavat ne kuitenkin osviittaa tehokkaan ja laajennettavissa olevan sovellusarkkitehtuurin kehittämistä varten. Pelipalvelinratkaisut sisältävät myös paljon toimintoja, jotka eivät ole välttämättömiä itse pelin toiminnan kannalta, mutta helpottavat pelien hallinnointia ja ylläpitoa huomattavasti; esimerkiksi monipuolisia varmuuskopiointi-, monitorointi- ja tietokantatoimintoja. Koska näitä toimintoja on kehitetty vuosia, ja ne ovat useiden eri osapuolten käytössä, on niiden käytettävyys saatu hiottua sille tasolle, ettei pelin hallinnointi tai ylläpito vaadi erityisiä teknisiä taitoja tai kalliita koulutuksia.

Valimiin pelipalvelinratkaisun tuomat hyödyt riippuvat pelistä, johon sitä sovelletaan: vuoropohjaisissa ja hidastempoisissa peleissä valmista pelipalvelinta on melkein aina järkevää käyttää, sillä sen tuoma ylimääräinen kuormitus (*overhead*) on huomaamaton, ja ”ilmaiseksi” saadut ominaisuudet säästävät huomattavasti ohjelmointiaikaa ja kustannuksia. Lisäksi usealle yhtäaikaiselle käyttäjälle optimoitu pelipalvelinohjelmisto mahdollistaa useampien pelaajien palvelemisen yhtäaikaisesti pienemmillä laitteistoresursseilla. Nopeatempoisissa moninpeleissä saatetaan kuitenkin tarvita paljon verkkokerroksessa olevan ohjelmakoodin pelikohtaista optimointia, joten pakettiratkaisujen hyödyt muuttuvat kyseenalaisiksi. Ylimääräistä kuormitusta voidaan kompensoida lisäämällä laitteiston suorituskykyä, joten loppujen lopuksi kyse on tasapainottelusta pelinkehitykseen käytettyjen resurssien ja laiteresurssien välillä.

LÄHTEET

Bernier, Yahn W 2001. Latency Compensating Methods in Client/Server In-game Protocol Design and Optimization. https://developer.valvesoftware.com/wiki/Latency_Compensating_Methods_in_Client/Server_In-game_Protocol_Design_and_Optimization. Päivitetty 2001. Luettu 5.11.2014.

BigWorld Technology 2014. http://bigworldtech.com/en/. Päivitetty 2014. Luettu 7.11.2014.

Brooks, Frederick P 1995. The Mythical Man-Month. Boston: Addison Wesley Longman, Inc, 1995.

Crane, Charlie 2013. Client platform supported. https://github.com/NetEase/pomelo/wiki/client-platform-supported. Päivitetty 29.9.2013. Luettu 5.11. 2014.

Cubeia. 2013. http://www.cubeia.com/. Päivitetty 2013. Luettu 5.11.2014.

Cubeia Firebase. 2013. http://www.cubeia.com/components/cubeia-firebase/. Päivitetty 2013. Luettu 5.11.2014.

Exit Games. 2014. http://www.exitgames.com/. Päivitetty 2014. Luettu 5.11.2014.

Fiedler, Glenn 2008. UDP vs. TCP. http://gafferongames.com/networking-for-game-programmers/udp-vs-tcp/. Päivitetty 2008. Luettu 5.11.2014.

SmartFox Server 2X - Installing under Windows 2014. Goto And Play. http://docs2x.smartfoxserver.com/GettingStarted/install-windows. Päivitetty 2014. Luettu 5.11.2014.

SmartFox Server 2X Documentation Central - MMO Rooms 2014. Goto And Play. http://docs2x.smartfoxserver.com/AdvancedTopics/mmo-rooms. Päivitetty 2014. Luettu 7.11.2014.

Haddad, Serge, Fabrice Kordon, Laurent Pautet, ja Laure Petrucci 2011. Distributed Systems. London: ISTE Ltd, John Wiley & Sons, Inc, 2011.

Halmetoja, Heikki 2014. Unity-pelialustan moninpeliominaisuudet. Opinnäytetyö, Oulu: Oulun seudun ammattikorkeakoulu, 2014.

Information Sciences Institute 1981. RFC 793: Transmission Control Protocol. Request For Comments-asiakirja, Arlington: DARPA, 1981.

ITU WTID. 2014. http://www.itu.int/en/ITU-D/Statistics/Pages/stat/default.aspx. Päivitetty 2014. Luettu 5.11.2014.

Johansson, Fredrik 2008. Firebase - Performance Analysis. White paper, Tukholma: Cubeia Ltd, 2008.

Kegel, Dan 2006. The C10K problem. http://www.kegel.com/c10k.html. Päivitetty 2.9.2006. Luettu 7.11.2014.

Lapi, Marco 2012. SmartFoxServer 2X Performance And Scalability White Paper. White paper, Cuneo: Goto And Play, 2012.

Leadbetter, Richard 2009. EuroGamer.net. http://www.eurogamer.net/articles/digitalfoundry-lag-factor-article?page=2. Päivitetty 9.5.2009. Luettu 7.11.2014.

NTES Fact Sheet August 2014. Peking: NetEase, 2014.

Pomelo. 2014. http://pomelo.netease.com/. Päivitetty 2014. Luettu 5.11.2014.

Pomelo成功案例. 2014. https://github.com/NetEase/pomelo/wiki/Pomelo%E6%88%90%E5%8A%9F%E6%A1%88%E4%BE%8B. Päivitetty 2014. Luettu 5.11.2014.

Nilsson, Lars J 2010. Asynchronous Games in Firebase; pt I. http://www.cubeia.com/2010/03/asynchronous-games-in-firebase-pt-i/. Päivitetty 11.3.2010. Luettu 5.11.2014.

Postel, J 1980. RFC 768: User Datagram Protocol. Request For Comments-asiakirja, Arlington: DARPA, 1980.

Unity Manual - Master Server. 2014. http://docs.unity3d.com/Manual/net-MasterServer.html. Päivitetty 2014. Luettu 7.11.2014.

LIITE 1.

Olennaiset WWW-osoitteet

Tähän on listattu opinnäytetyön ja esimerkkiohjelman kannalta tarpeellisiä WWW-osoitteita.

SmartFoxServerin lataussivu:

http://smartfoxserver.com/download/sfs2x#p=installer

SmartFoxServerin Client API:

http://docs2x.smartfoxserver.com/api-docs/csharp-doc/Index.html

SmartFoxServerin Server API:

http://docs2x.smartfoxserver.com/api-docs/javadoc/server/

Esimerkkipelin asiakasohjelma (SmartFoxHippa):

https://bitbucket.org/ilmo_euro/smartfoxhippa

Esimerkkipelin palvelinlaajennus:

https://bitbucket.org/ilmo_euro/smartfoxhippa-server

LIITE 1(1).

Pelipalvelinten vertailutaulukko

Ohjelmisto	**Kaupalliset**		**Asiakaskirjastot**				
	Vapaa?	Hinta (100 pelaajaa)	JavaScript (Web)	ActionScript (Flash)	Unity	Java	Python
Pomelo	x	0	x	x	x		
Photon		95	x	x	x		
Cubeia Firebase	x *1	0 *1				x	
SmartFox Server Pro		500		x	x	x	
SmartFox Server 2X		350 *2	x	x	x	x	
BigWorld		300 *3					x
Unity Master Server	x	0			x		
**1 Community Edition*							
**2 Logon kanssa ilmainen*							
**3 10 000 pelaajaa*							
**4 varauksin*							

LIITE 1(2).

Pelipalvelinten vertailutaulukko

Ohjelmisto	**Palvelinkielet**					**Toteutusteknologia**		
	JavaScript	ActionScript	C#	Java	Python	NodeJS	Java	Muu
Pomelo	x					x		
Photon			x					x
Cubeia Firebase				x			x	
SmartFox Server Pro		x					x	
SmartFox Server 2X				x	x		x	
BigWorld					x			x
Unity Master Server			x					x
**1 Community Edition*								
**2 Logon kanssa ilmainen*								
**3 10 000 pelaajaa*								
**4 varauksin*								

Pelipalvelinten vertailutaulukko

Ohjelmisto	**Pelaajia/huone**			**Pelityypit**		
	1–10	11–100	101+	Lauta/kortti	Action	MMO
Pomelo			x		x	x
Photon			x	x	x	
Cubeia Firebase	x			x	x *4	
SmartFox Server Pro		x		x	x	
SmartFox Server 2X			x	x	x	
BigWorld			x			x
Unity Master Server	x			x	x	
**1 Community Edition*						
**2 Logon kanssa ilmainen*						
**3 10 000 pelaajaa*						
**4 varauksin*						