Dizertație Master

Plug and Play

Universal Description, Discovery and Integration

pentru susținerea tezei de Master în Sisteme Avansate Aplicații Internet,

în cadrul Facultații de Automatică si Calculatoare
a Universității Politehnica București de

ing. Mihai Matei

Februarie 2010

Coordonator: Conf. Dr. ing. Răzvan Rughiniș

Rezumat

Conceptul PnP UDDI. Lucrarea își dorește să prezinte o nouă modlitate de descoperire a serviciilor web prin intermediul unui device USB, denumit **PnP UDDI**, ce poate fi conectat la o mașină, fizică sau virtuală. Se vor prezenta unele dintre limitările actuale ale tehnologiei UDDI de descoperire a serviciilor web, noua modalitate de access la structura UDDI din punctul de vedere al abstracțiilor sistemor de operare *NIX* pentru device-uri, avantajele si dezavantajele utilizării tehnologiei USB pentru un astfel de access, inclusiv interogarea unei metode a serviciilor web prin acest dispozitiv. Pentru implementarea conceptului se vor da exemple de astfel de dispozitive USB, cu caracteristicile acestora din punctul de vedere al realizării firmware-ului sau al implementării lor într-un sistem de operare.

Dispozitiv PnP UDDI Simulat. Pentru susținerea conceptului se va explica implementarea dispozitivului USB intr-un mediu de Windows NT cu Windows Driver Kit instalat împreună cu Device Simulation Framework, tehnologia Microsoft pentru a simula un host controller EHCI pe masina curenta. În acest mod va fi prezentat un mediu de dezvoltare propice pentru implementatorii de PnP UDDI, ce permite îmbunătățirea ulterioara a dispozitivului.

Dispozitiv PnP UDDI Emulat. Pentru prezentarea unui mediu de operare în care acest dispozitiv are o semnificație practica, din punctul de vedere al utilizării, se va alege mediu virtualizat folsind hypervizorul XEN, peste care vor funcționa mai multe mașini virtuale pe aceeași mașină fizica, partajând resursele sistemului hardware. Se va arata adăugarea dispozitivului USB la runtime la o mașină virtuală Windows Vista, și vor fi explicate, în linii mari, elementele funcționale acestui mediu: tehnologia de virtualizare Xen, emulatorul Qemu pentru request-urile I/O ale device-urilor mapate in Windows-ul Vista, posibilitatea de adăugare a unui dispozitiv de conversie USB-Char Device in Qemu, conectarea acestuia, pe de o parte, la un controller UHCI în masina guest Vista și la o conexiune tcp (char device-ul) la un server Apache Axis.

Nu în ultimul rând se vor prezenta elementele generice pentru implementarea unui astfel de sistem, și anume o aplicație consumator de servicii web (.Net Box2Virt), driverul de sistem de operare (usb2virt.sys) și interacțiunea acestora cu firmware-ul dispozitivului, fie el simulat sau emulat.

Conținut

1 Servicii Web, UDDI și USB

Rezumat. În acest capitiol se vor prezenta, în linii marim, conceptele din spatele Serviciilor Web și a descrierii UDDI precum și avantajele si dezavantajele acestora.

1.1 Servicii Web

Serviciile web reprezintă o colecție de standarde și protocoale deschise folosite pentru a schimba date între aplicații și sisteme conectate prin diferite rețele, cea mai cunoscuta fiind cea Ethernet peste care este construit Internetul. Ele pot fi privite și ca aplicații de sine stătătoare, modulare, dinamice si distribuite ce pot fi descrise, publicate sau invocate la o anumită locatie în cadrul rețelei. În acest mod este asigurată comunicația între diverse sisteme eterogene, dar și interoperabilitatea la nivel de implementare in diverse limbaje de programare (Java, C# - .Net, C, Cobol). Acest lucru este realizat prin folosirea standardului XML la nivel de prezentare (descrierea tipurilor, operatiilor, flow-urilor) dar și HTTP peste TCP/IP la nivel de rețea.

Pentru a rezuma, un serviciu web complet este acea aplicatie care:

- este disponibilă pe Internet sau într-un intranet
- folosește standardul XML pentru împachetarea de mesaje
- nu este legată de un sistem de operare sau de un limbaj de programare
- poate fi descrisă printr-o gramatică comună XML
- poate fi localizată printr-un mecanism de căutare simplu

Platforma de bază a unui serviciu web este XML + HTTP. Toate serviciile web standard folosesc următoarele componente:

- SOAP (Simple Object Access Protocol)
- UDDI (Universal Description, Discovery and Integration)
- WSDL (Web Services Description Language)

1.1.1 Stiva de protocoale

Pentru o mai buna întelegere a integrării dispozitivului USB PnP UDDI, trebuie văzut nivelul de integrare al acestuia în cadrul stivei de protocoale a serviciilor web, compusă din patru nivele principale:

1. **Nivelul transport** – răspunzător pentru transportarea mesajelor între aplicatii, realizat prin intermediul protocoalelor HTTP, SMTP, FTP sau BEEP.
2. **Encapsularea XML** – encodarea mesajelor în format XML pentru portabilitate. Acest nivel include protocoalele XML-RPC si SOAP.
3. **Descrierea serviciului** – într-un format interpretabil de limbajele de programare existente a interfeței publice a serviciului. Această descriere este realizată prin intermediul Web Service Description Language (WSDL).
4. **Descoperirea serviciului** – pentru publicarea/centralizarea serviciilor într-un registru comun, descoperirea efectivă realizându-se prin Universal Description, Discovery, and Integration (UDDI). Acesta este nivelul la care se va afla, din punct de vedere conceptual, dispozitivul USB PnP UDDI prezentat.

1.1.2 Securitatea serviciilor web

Din punctul de vedere al securității întâlnim trei aspecte:

Confidentialitatea

Dacă un client trimite un request XML unui server cum putem fi siguri că comunicația este confidențială? Modalitatea cea mai frecvent folosita este folosirea Secure Sockets Layer (SSL) ca modalitate de encriptare a mesajelor la nivel de rețea. Totuși această soluție nu este una pretabilă pentru o aplicație complexă, care nu poate fi sigură că serviciul apelat nu face apeluri ne-encriptate către alte entități, luând în considerare natura loose-coupled a serviciilor web.

Autentificarea

Aceast aspect conține identificarea unui client ce se conectează la un serviciu web, precum și a drepturilor sale de utilizare. Acest lucru este momentan rezolvat atât la nivel HTTP, folosindu-se suportul de Basic and Digest authentification, sau prin extensii la protocolul SOAP, cum ar fi SOAP-DSIG (SOAP Digital Signature).

Securitatea rețelei

Se pune probleme cum să se filtreze traficul serviciilor web intr-o rețea. Pentru filtrarea mesajelor SOAP sau XML-RPC o posibilitate ar fi identificarea request-urilor HTTP POST care au headerul *Content-Type: text/xml*, sau prin identificarea header-ului *SOAPAction*.

1.2 Universal Description, Discovery, and Integration

UDDI este un standard bazat pe XML pentru descrierea publicarea și găsirea serviciilor web. Acesta conține un registru al tuturor serviciilor web, împreuna cu WSDL-ul corespunzător descrierii fiecarui serviciu, precum și o serie de porturi pentru manipularea și căutarea în acest registru. Pentru accesul la acest registru standardul UDDI, versiunea 2.0, specifică doua interfețe pentru consumatorii și implementatorii de servicii, respectiv interfețele *Inquiry* si *Publisher*. Aceste interfețe sunt descrise printr-o Schema XML. Un exemplu de access la registrul UDDI este dat mai jos:

- pentru publicare

```
POST /save_business HTTP/1.1
Host: www.XYZ.com
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn
SOAPAction: "save_business"
<?xml version="1.0" encoding="UTF-8" ?>
<Envelope xmlns="http://schemas/xmlsoap.org/soap/envelope/">
      <Body>
         <save_business generic="2.0" xmlns="urn:uddi-org:api_v2">
         <businessKey="">
         </businessKey>
         <name>
         www.2virt.com
         </name>
         <description>
             Adresa site-ului corespunzător serviciului
         </description>
         <identifierBag> ... </identifierBag>
         ...
         </save_business>
      </Body>
</Envelope>
```

- pentru căutare

```
POST /get_businessDetail HTTP/1.1
Host: www.2virt.com
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn
SOAPAction: "get_businessDetail"
<?xml version="1.0" encoding="UTF-8" ?>
<Envelope xmlns="http://schemas/xmlsoap.org/soap/envelope/">
   <Body>
      <get_businessDetail generic="2.0" xmlns="urn:uddi-org:api_v2">
          <businessKey="C90D731D-772HSH-4130-9DE3-5303371170C2">
      </businessKey>
```

```
      </get_businessDetail>
   </Body>
</Envelope>
```

Acest capitol nu se doreste a fi o descriere a conceptele UDDI, doar să evidențieze, în mare, modul de organizare al acestuia, faptul că accesul se realizează tot prin binding-uri SOAP, dar, în primul rând, o mare limitare. Să presupunem că o aplicație trebuie să caute un anumit serviciu. Pentru a îl identifica aplicația are nevoie de doua elemente:

1. o descriere unica a modalității de interogare a UDDI pentru respectivul serviciu (ex: *businessKey* de mai sus)
2. o modalitate de a identifica locația UDDI-ului potrivit (***domain***, ***ip:port*** pentru trimiterea mesajului SOAP)

Aceasta încalcă arhitectura loose-coupled a serviciilor web prin legarea statică a unei aplicații de un anumit UDDI, și, mai ales, de o anumită implementare de UDDI. În arhitecturile mai noi de cloud acest lucru se complică și mai mult prin migrarea acestuia între diferitele mașini.

1.2.1 Limitările arhitecturii

Voi enumera câteva dintre dezavantajele arhitecturii curente de access la servicii web:

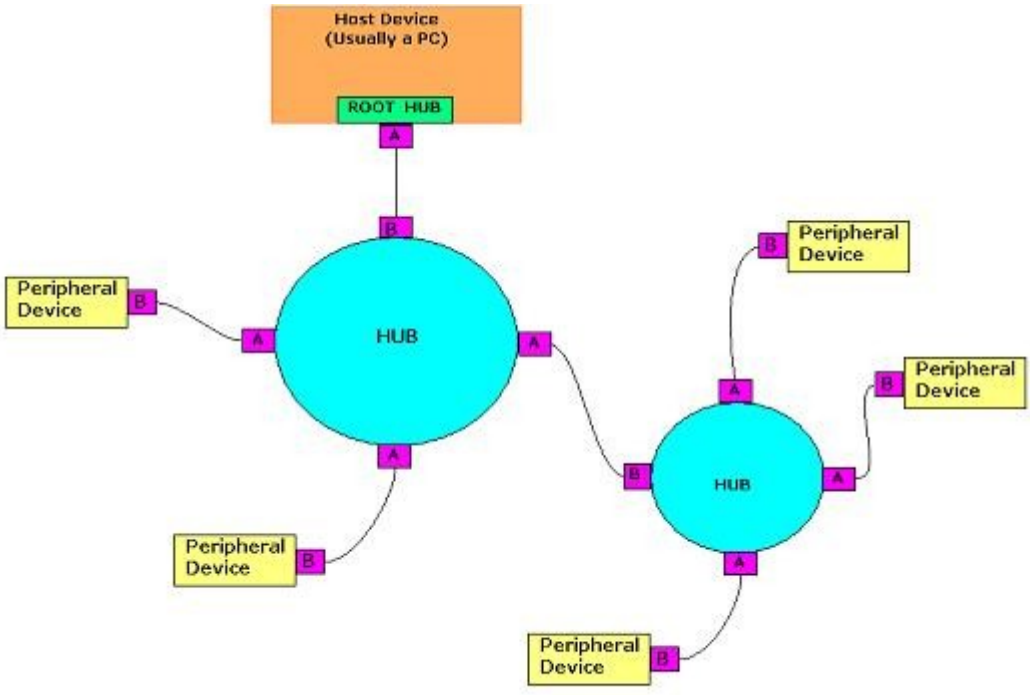
- securitatea comunicației precum și drepturile de access sunt, de cele mai multe ori, realizate fie de extensii la protocoloale folosite, ce pot influența interoperabilitatea serviciilor web, fie nu poate fi garantată datorită necunoasterii implementării din spatele acesteia.
- căutarea unui anumit serviciu în registrul UDDI necesită cunoașterea dinainte a modului de identificare al acestuia, precum și modalitatea de access și locația însuși a registrului.
- Nu există o posbilitate de actualizare către un nou registru UDDI în cazul în care acesta nu mai este disponibil (exemplu mașina a picat).
- De asemenea un administrator trebuie să aibă în vedere conectivitatea dintre mașinile implicate in rețea, ceea ce pe un caz de implementare complex înseamnă multe reguri de filtrare greu de întreținut.

1.3 Universal Serial Bus

Usb reprezintă un set de specificații de interfațare a comunicației de periferice legate prin magistrala USB cu sau fară un computer prezent. USB în primul rând își dorește să înlocuiasca mutimea de conectori prezenți pentru device-urile periferice, ceea ce la ora actuală este un scop în mare parte îndeplinit. Principalul avantaj al magistralei USB este facilitatea de Plug And Play a dispozitivelor, astfel că adresele și întreruperile necesare funcționarii dispozitivului sunt ascunse de arhitectura magistralei. De asemenea un dispozitiv este recunoscut de sistem doar prin combinația unică de ProductID/VendorID, singura necesară pentru încărcarea driver-ului corect în sistem.

1.3.1 Topologia USB

Un sistem USB este alcătuit dintr-un controller host și mai multe porturi USB la care sunt conectate device-urile intr-o topologie de stea. Pentru a multiplica numărul de porturi hub-uri USB pot fi incluse în topologie, creând structuri arborescente de maxim cinci nivele. Această topologie are avantajul separării domeniilor de tensiune între device-uri precum și capacitatea de a conecta diferite resurse cu viteze de transfer variabile.



Topologia USB

Usb-ul este defapt o magistrala adresabilă, cu adrese de 7 biți, permițând conectarea a maxim 127 de diferite device-uri. Arhitectura de comunicație este una master-slave, cu master-ul implementat de host controller ce controlează activitatea de comunicație pe

magistrală, iar device-urile conectate sunt slave-urile. Deși magistrala este una singura fiecare dispozitiv conectat poate avea maxim 16 canale logice unidirecționale (16 de INPUT, 16 de OUTPUT) – pipe-uri conectate la host controller pentru multiplexarea comunicație.

1.3.2 Convenții de comunicație

Pachetele USB sunt trimise în mod serial, LSB (least significant bit). Acestea au ca sursă, sau destinație, hubul root al controller-ului host și trec prin topologia rețelei, prin posibil mai multe hub-uri, ajungând la device. Un transfer USB este alcătuit din:

- Token Packet (un header al tipului de mesaj/transfer)
- Optional Data Packet (conținutul de date al mesajului)
- Status Packet (pentru verificarea transacțiilor și corectia erorilor)

Pentru a ilustra cel mai bine structura comunicării putem analiza câmpurile unui astfel de pachet USB:

- SYNC – 8/32 biți pentru transferuri low/high pentru a sincroniza ceasul recever-ului și al transmiter-ului
- PID – packet ID – 8 biți dintre care primii 4 reprezintă tipul de pachet, iar ceilalți 4 complementul pe biți al primilor 4
- ADDR – destinația pachetului corespunzătoare unuia dintre cele 127 de device-uri
- ENDP – specifică canalul logic de transfer, pe 4 biți permițând selectarea a uneia din cele 16 posibile endpoint-uri
- CRC – Cyclic Redundancy Checks – pe 5 sau 16 biți în funție de tipul de pachet
- EOP – end of pachet – encodare electronică a sfârșitului pachetului

Din punctul de vedere al tipului de transfer, cele patru modalități de trimitere a datelor, caracterizate prin tipul de endpoint sunt descrise mai jos, conform specificației USB 2.0:

Tip Transfer	Low-speed	Full-speed	High-speed
Control	24 Kbytes/sec	832 Kbytes/sec	15.5 Mbytes/sec
Interrupt	0.8 Kbytes/sec	64 Kbytes/sec	24 Mbytes/sec
Bulk	-	1216 Kbytes/sec	52 Mbytes/sec
Isochronous	-	1023 Kbytes/sec	24 Mbytes/sec

Endpointurile de control schimbă informații de configurare, precum și setarea sau obținerea diferiților parametrii. Cele Isonchronous sunt folosite de dispozitive real-time,

cum ar fi o cameră video. Acestora li se asigură o lățime de bandă garantată din cea pe care o are la dispoziție magistrala USB. Transferurile Bulk sunt folosite de dispozitive care doresc să transfere un chunk mare de date odată. Cele Interrupt sunt folosite pentru a se asigura procesarea unei cantități mică de informație, cum ar fi un eveniment, într-un timp relativ scurt.

1.3.3 Host Controller

În acest paragraf vom descrie această componentă din arhitectura USB, deoarece pe prezentările practice, de firmware simulat și emulat, aceasta este singura componentă virtuală care se adaugă unei mașini. Practic emulatoarele și simulatoarele nu fac decât să emuleze funcționalitatea acestei componente, care reprezintă de fapt emularea întregii magistrale USB, deoarece aceasta este singura componentă prin care se asigură accesul unui sistem de operare la dispozitivele USB aflate pe magistrală.

Interfața de programare dintre o mașina, OS, și controller-ul host poartă numele de HCD – Host Controller Device, definită de implementatorul hardware. Pentru standardul USB 1.x existau două implementări de HCD: OHCI (Open Host Controller Interface) și UHCI. Trecerea la USB 2.0 a însemnat implementarea EHCI (Enhanced Host Controller Interface) ce poate suporta transferuri de 480 Mbit/sec. Majoritatea implementarilor de controalere EHCI legate la magistrala PCI a platformei hardware suportă și transferuri Full Speed (12 Mbit/sec). Ce trebuie reținut este că un sistem de operare ar trebui să conțină drivere pentru configurarea și comunicarea cu oricare dintre aceste trei standarde. Peste nivelul arhitectural dat de aceste drivere de controller se situează driverele fiecarui tip de dispozitiv USB adăugat, ce asigură transferul de date către device-ul respectiv folosind endpointurile cunoscute de device.

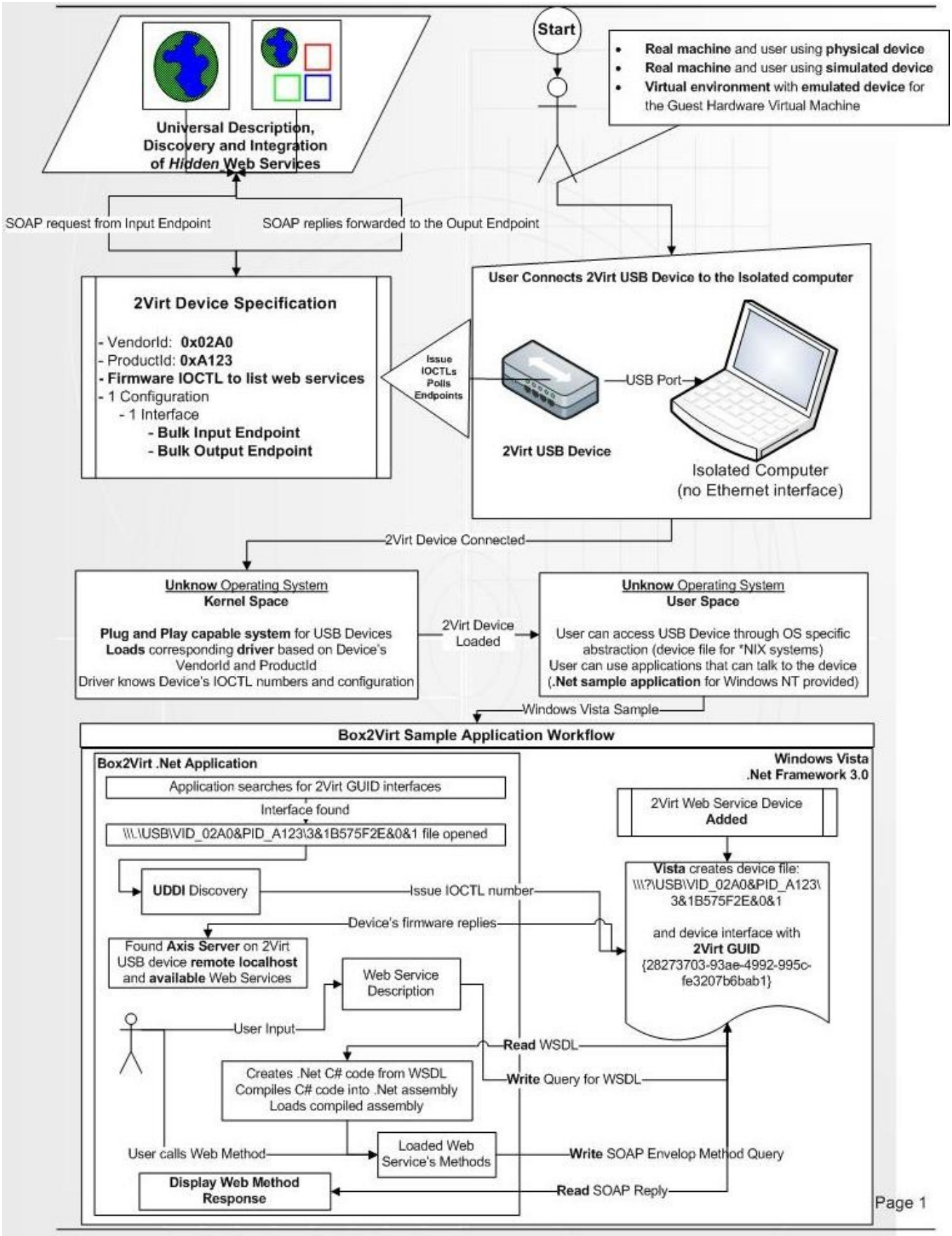
2 Conceptul PnP UDDI

Pentru a soluționa limitările enunțate în paragraful 1.3 s-a dorit ca accesul și enumerarea serviciilor web să se realizeze, sau să fie abstractizată, prin intermediul unui **dispozitiv** adăugat la masina curentă. În felul acesta se decuplează comunicația, descoperirea serviciilor web, de folosirea acestora de către un consumator. Practic aplicația ce rulează în sistemul de operare curent trebuie doar să știe să interogheze device-ul adaugat, lucru ce, deși e strâns legat de arhitectura sistemului de operare pentru prezentarea și accesul la device, ofera o mult mai mare transparență în consumarea functionalităților serviciilor web. În plus alegerea tehnologiei USB pentru implementarea device-ului asigura capacitatea Plug And Play, atât la nivel de descoperire cât și la accesul la serviciile expuse de device.

Ca o scurtă paralelă la ideea din spatele conceptului, acesta poate fi comparat cu folosirea unei imprimante. Pentru ca un utilizator să aiba access la un serviciu de tiparire trebuie să folosească acest dispozitiv, și anume imprimanta. Ca să obtină access trebuie fie sa folosească o imprimantă conectată la un alt calculator, fie să conecteze o imprimantă calculatorului curent. În primul caz, asemanător cu implementarea actuală a serviciilor web, trebuie să stie unde să caute imprimanta, să o poata accesa, să aiba permisiunea de a o folosi si să astepte un slot pentru folosirea serviciului de tipărire. Modalitatea cea mai transparentă este conectarea directă a imprimantei, tipărirea urmată de deconectarea la alegere a acesteia. În ambele cazuri utilizatorul trebuie să aibă instalat pe mașina sa un driver pentru device-ul de tipărire corespunzător sistemului de operare folosit și o aplicație userspace pentru comunicația cu driver, respectiv device pentru tipărirea efectivă.

Practic dispozitivul USB 2Virt nu se dorește a fi doar un proxy pentru accesul la serviciile web expuse de acesta (functionalitatea UDDI) ci dorește migrarea arhitecturii de comunicație și descoperire a serviciilor web ce momentan este strâns legata de structura rețelei în care se găsește mașina curentă în funcționalitatea acestui device. În acest mod un dezvoltator nu mai trebuie să țină cont de eterogenitatea mediului de rulare a aplicațiilor ci doar să dezvolte o aplicație ce comunica cu acest device, ce apare ca adăugat fizic la mașina curentă. Implementarea efectivă a device-ului, fie ca dispozitiv fizic sau virtual, simulat sau emulat de platforma sub care rulează sistemul de operare rămane în responsabilitatea implementatorului de servicii web ce dorește să le facă accesibile unui anumit tip de consumator.

2.1 Arhitectura PnP UDDI



2 Conceptul PnP UDDI

Pentru exemplificarea conceptului s-a considerat functionarea dispozitivului la un sistem de operare oarecare, pentru a putea urmării pașii necesari în accesarea serviciilor web, în cazul unui implementator de servicii web care vrea să le publice/facă accesibile unui utilizator de pe o masină izolată (nu are conexiune directă la Internet):

1. Implementatorul conectează dispozitivul PnP UDDI USB (fizic sau virtual) la mașina izolată.
2. Kernelul sistemului de operare necunoscut descoperă un device nou, cu un anumit VendorId si ProductId, elemente unice în identificarea tipului device-ului conform tehnologiei USB.
3. Sistemul de operare va încărca un device driver pentru dispozitivul curent.
4. Sistemul de operare va expune aplicațiilor user-space un device file pentru accesul la dispozitiv.
5. Conform abstractizării UNIX, o aplicație userspace poate efectua system call-uri pe acest device file – IOCTL-uri pentru control, read/write generice (sincron și asincron).
6. O aplicație userspace, consumatorul de servicii web, poate fi pornită în acest moment de către utilizator.
7. Aceasta va efectua un IOCTL corespunzător device-ului adăugat (această legătură este străns legată de implementarea device-ului) pentru enumerarea serviciilor web.
8. Aplicația va putea efectua un alt IOCTL pentru obținerea WSDL-ului unui anumit serviciu web.
9. Odata ce un serviciu este selectat și cunoscut (WSDL primit) aplicația va putea efectua un apel de write pentru a trimite un query SOAP pentru apelarea unei metode din servicul respectiv, efectuând apoi un apel de read pentru primirea raspunsului SOAP.

În toate cazurile de mai sus va rămâne la latitudinea implementării device-ului respectiv, ca funcționalitate a firmware-ului, fie el fizic, emulat sau simulat, pentru a trimite mai departe mesajele SOAP sau IOCTL-ului unei anumite entități (registru UDDI remote pentru IOCTL, sau Web Server remote pentru apelul metodei cu drepturile de access și autentificare ale device-ului conectat, și nu ale utilizatorului).

2.1.1 Avantajele dispozitivului PnP UDDI

Pentru a întelege mai bine la ce folosește migrarea funcționalităților implementate de anumite elemente din tehnologia de Servicii Web într-un device USB trebuie să enumeram beneficiile folosirii acestuia:

- tehnologia USB asigură facilitatea de Plug and Play, capacitatea de a adăuga și înlătura un dispozitiv. Această faciliate este implementată de mai toate sistemele de operare. În acest mod un consumator de servicii web poate cere să i se fie adăugat un astfel de dispozitiv sau poate fi activat când acest dispozitiv devine accesibil pe mașina curentă.
- toate sistmele de operare asigură, atât la nivel de kernel, cât si la nivel de user-space, un API pentru implementarea și folosirea componentelor necesare comunicării cu device-ul respectiv. Astfel, la nivel de kernel, există API-uri pentru implementarea de drivere USB arhi-cunoscute și făcute publice pentru orice sistem, iar la nivel de aplicații userspace cea mai des întâlnită abstractizare este cea de device file – fișier corespunzător unui anumit tip de sistem de fișiere ce expune un API comun – IOCTL, read/write, seek
- specificația USB permite identificarea sigură și unică a unui anumit tip de dispozitiv conectat la mașină prin intermediul identificatorilor *ProductID* și *VendorID*, hardcodați în firmware-ul dispozitivului. De asemenea se asigură și o mare elasticitate a implementarii, un dispozitiv putând avea mai multe configurații și mai multe interfețe corespunzătoare funcționalităților sale. De exemplu, pentru implementarea dispozitivului PnP UDDI, s-a ales o configurație cu un IOCTL pentru enumerarea serviciilor web curente și o interfață cu doua endpointuri – pentru trimiterea și recepționarea mesajelor SOAP.
- Dispozitivul PnP UDDI asigură transparență în ceea ce privește comunicarea cu serviciile web, de exemplu nu necesită ca mașina fizică să fie conectată la o rețea. Pentru exemplificare acestui scenariu s-a implementat un dispozitiv PnP UDDI emulat într-un mediu virtualizat în care o masină virtuală guest, ce nu conține o interfață ethernet, se conectează la un serviciu web al unui server Axis ce ascultă pe interfața de localhost al altei mașini virtuale.
- Securitatea comunicarii nu mai trebuie să fie asigurată de consumatorii de servicii web, ci este realizată de dispozitiv. Acelaș lucru este adevărat și pentru celelalte elemente care țin de securitate, ca autentificarea și drepturile de utilizare.

- Implementatorii de servicii web și de clienți ai acestora trebuie doar să se asigure de interfațarea cu dispozitivul USB. De exemplu o aplicație client nu trebuie să caute într-un registru UDDI serviciul respectiv, trebuie doar sa urmărească apariția device-ului PnP UDDI pe masina locală, și să comunice cu acesta. Implementările la nivel de API de sistem de operare fac acest lucru foarte usor, o aplicație putând urmării apariția unei anumit tip de dispozitive conectate la mașina curentă.

2.1.2 Dezavantajele folosirii unui dispozitiv USB

Ca la orice concept există si dezavantaje, inclusiv la nivel de arhitectură, pe lângă cele legate de implementare efectivă:

- folosirea tehnologiei USB presupune familiarizarea atât a implementatorilor de dispozitive cât și a utilizatorilor acestuia cu particularitățile specificației în fiecare din sistemele de operare folosite.
- Există o limitare a technologiei din punctul de vedere a performanțelor comunicării prin endpoint-urile expuse de dispozitiv.
- Numarul endpointurilor reprezintă practic numărul de conexiuni fizice care se poate realiza cu serviciile la care sunt legate aceste conexiuni. De exemplu, pentru PnP UDDI aceasta reprezintă numărul de servicii web care pot fi interogate simultan de unul sau mai multi clienți. Dacă se dorește mărirea acestui număr o logică și diferite nivele de multiplexare trebuie realizată ca în cazul tehnologiei Ethernet.
- Conform specificației USB la un moment dat dispozitivul poate avea o singură configurație activă cu mai multe interfețe, fiecare interfață punând la dispoziție mai multe conexiuni (endpoint-uri). Acest lucru poate fi o limitare teoretică pentru un dispozitiv ce expune mai multe configurații (cum ar fi cea de low-power).
- Momentan nu există o modalitate de a grupa dispozitivele împreună, ceea ce inseamna că dacă o aplicație complexă foloseste mai mulți implementatori de dispozitive PnP UDDI, va însemna că masina locală va avea conectate mai multe dispozitive USB, ce vor fi mai greu de gestionat. Există și o limitare a numărului de dispozitive ce pot fi conectate la un controller USB (EHCI sau UHCI), fiind necesar adaugarea de hub-uri la porturile expuse de controller, ceea ce limitează performanța.
- Nu în ultimul rând există mai multe implementări a standardului USB (1.1, 2.0, 3.0)

2.2 Componente implementării

După cum spuneam în capitolele precedente, ceea ce se își propune lucrarea este separarea mult mai clară între dezvoltatorii de platforme de servicii web și consumatorii acestora. De aceea voi prezenta seturile de componente din perspectiva celor două categorii de dezvoltatori, enumerând și cazurile speciale implementate deja pentru susținerea lucrării.

2.2.1 Clienții dispozitivelor

Astfel un utilizator al unui anumit tip de serviciu va trebui să implementeze doar o aplicație care stie să comunice cu dispozitivul PnP UDDI dorit. Pentru această situație s-a implementat o aplicație demonstrativă C# .Net ce comunică cu dispozitivul USB. O prezentare mai amplă a acestei aplicații se va realiza în capitolul următor. Practic un client ce dorește să comunice cu dispozitivul PnP UDDI trebuie doar să cunoasca:

- modalitatea specifică sistemului de operare curent de a descoperi si identifica dispozitivul. De exemplu pe platformele Windows NT se poate cauta o anumită interfața de dispozitiv folosind un GUID. În felul acesta se identifică ***device file***-ul ce va fi folosit pentru operații-le de read-write.
- descrierea apelurilor IOCTL corespunzătoare device-ului PnP UDDI. În momentul de față este disponibil unul singur pentru enumerarea de servere de servicii web.
- Modelul de threading asociat dispozitivului pentru o interfață. Acest lucru se traduce în câte conexiuni concomitente dispozitivul poate abstractiza prin operații de read/write asociate device file-ului.
- Modalitatea de comunicație. Pentru dispozitivul PnP UDDI pentru apelarea SOAP a unei metode dintr-un serviciu web, este necesară blocarea canalului de write până când un răspuns complet s-a primit.

Aplicația .Net implementată este comună pentru toate cazurile în care un dispozitiv PnP UDDI este conectat la masina rulând Windows NT. Practic aplicația interoghează dispozitivul pentru a obține serviciile web, WSDL-ul acestora și poate apela metodele unui serviciu folosind o interfața GUI prietenoasă.

2.2.2 Implementatorii de PnP UDDI

Cei ce doresc să contruiască dispozitive PnP UDDI trebuie să aiba în vedere implementarea firmware-ului dispozitivului, legarea acestuia de componentele respective precum și realizarea suportului pentru sistemele de operare dorite. Astfel trebuie implementat:

- IOCTL în firmware pentru listarea serviciilor web disponibile.
- dispozitiv USB cu cel putin o configuratie și o interfață cu două endpointuri, unul de OUT pentru primirea de query-uri SOAP și unul de IN pentru trimiterea de răspunsuri înapoi la consumatorii de servicii.
- comunicația la nivel de firmware cu serviciile web. Această comunicație este realizată in exemplele date printr-o conexiune TCP la un server Apache Axis.
- device driver pentru fiecare sistem de operare în care se doreste accesarea dispozitivului. Astfel device driver-ul va expune prin intermediul API-ului existent aplicațiilor userspace un device file cu funcționalitățile descrise în capitolul precedent. Practic în implementarea dată ca exemplu s-a folosit un driver USB pentru Windows NT, *usb2virt.sys*, în care este hardcodat IOCTL-ul pentru enumerarea de servicii web și știe să înainteze request-urile de read și write către endpointurile Bulk de INPUT și OUTPUT.

Chiar și framework-ul de implementare al dispozitivului PnP UDDI are părți care se pot refolosi. De exemplul driverul usb2virt.sys folosit pentru conectarea dispozitivului la platforme Windows NT (exemplu Windows Vista) este acelaș pentru cele două implementări de firmware date ca exemplu:

- firmware simulat folosind Driver Simulation Framework din WDK sub Windows Vista
- firmware emulat în Qemu, într-un mediu virtualizat folosind hypervisor-ul Xen.

În acelaș mod, pentru extinderea dispozitivului PnP UDDI pentru platforme Linux, trebuie implementat doar un device driver pentru kernelul de Linux, folosind API-ul UNIX corespunzător pentru dispozitive USB, device driver ce are hardcodat IOCTL-urile, și Endpointurile Bulk de IN și OUT.

2.2.3 Specificația dispozitivului PnP UDDI

Dispozitivul PnP UDDI reprezintă o componentă USB fizică, simulată sau emulată atașată la mașina unde se dorește accesare unui set de servicii web. Dupa cum spuneam acesta va fi atașat unui host controller USB, OHCI, UHCI sau EHCI deja existent in configurația hardware a mașinii sau emulat de un framework de simulare (DSF în Windows Vista) sau de un emulator într-un mediu virtualizat (Qemu sub Xen). Pentru ca un dezvoltator de device driver pentru dispozitiv să poată comunica cu device-ul este necesar ca firmware-ul acestuia să suporte anumite specificații și o anumită configurație.

Pentru aceasta se iau în considerare următoarele:

Caracteristici generice

- **VendorID** 0x02A0
- **ProductID** 0xA123
- **USBVersion** 2.0
- DeviceClass 0xFF
- DeviceSubClass 0xFF
- DeviceProtoocol 0xFF
- MaxPacketSize0 0x40(64)

Primele două caracteristici identifică unic device-ul între toate device-urile prezente pe piață. Valoare 0xFF pentru descrierea clasei de comunicație reprezintă faptul că tipul de access la device nu este unul generic(gen Human Interface Device), ci implementat de vendor. *MaxPacketSize0* reprezintă valoarea maximă a unui pachet pentru endpointul de control 0.

Caracteristici interfață

Un dispozitiv USB poate avea mai multe configurații, una singură fiind activă la un moment dat. O configurație prezintă mai multe interfețe, fiecare interfață corespunzând unui device file văzut de o aplicație userspace pe care se pot efectua operații de IOCTL, read/write. Dispozitivul PnP UDDI trebuie să prezinte o configurație cu o singură interfață cu două endpointuri Bulk, de IN și OUT, precum și un IOCTL pentru enumerarea de servicii web:

- Current Config Value: 0x01
- Device Bus Speed: High
- **IOCTL 0x55006000** – întoarce (***server***\0***modalitate_access***\0)*

- Endpointuri Deschise: 2

  EndpointAddress: **0x81 IN**

  Transfer Type: Bulk

  MaxPacketSize: 0x0400 (1024)

  EndpointAddress: **0x02 OUT**

  Transfer Type: Bulk

  MaxPacketSize: 0x0400 (1024)

Modalitatea de lucru cu device file-ul expus de sistemul de operare pentru comunicația cu dispozitivul începe prin accesarea IOCTL-ului pentru enumerarea de servicii. De exemplu, dacă device-ul PnP UDDI a fost conectat la un server Axis acesta va întoarce:

Axis 1.4 on Tomcat 6.0\0GET /axis/servlet/AxisServlet HTTP1/1\r\nHost: localhost\r\n\0

ca mai apoi, o aplicație userspace să poată efectua o operație de write folosind sintagma **GET /axis/servlet/AxisServlet HTTP1/1\r\nHost: localhost\r\n** pe device file-ul corespunzător dispozitivului, care va fi trimis de device driver-ul din kernel pe endpoint-ul de OUT către firmware-ul dispozitivului. Acesta se va conecta la server prin modalități cunoscute de firmware și va întoarce pe endpointul IN rezultatul dat de query-ul HTTP. Acesta va putea fi accesat de aplicație prin efectuarea unui apel de read pe device file-ul respectiv.

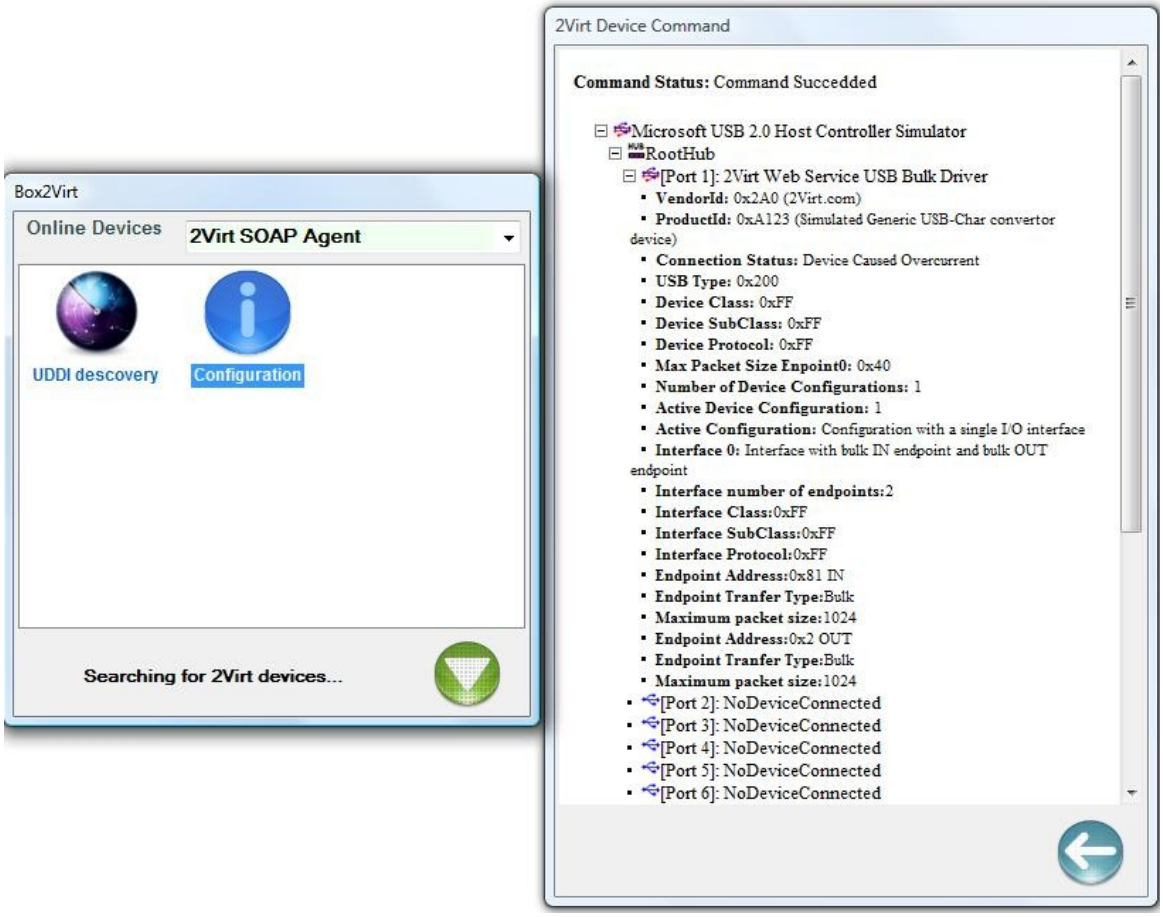
Pentru comunicația efectivă a mesajelor HTTP sau SOAP peste HTTP s-a ales folosirea unui endpoint de tipul Bulk pentru că caracteristica unui astfel de endpoint impune trimiterea de chunk-uri mari către destinație, cum ar fi cazul unui pachet ce conține mesajul HTTP. Ceea ce trebuie notat este că în acest caz nu se mai folosește TCP pentru comunicație astfel că mesajul HTTP nu va mai fi encapsulat în protocoalele stivei TCP/IP ca în cazul unui apel pe o interfață Ethernet, ci va fi direct introdus într-un pachet USB de tipul transmisiei de date Bulk. Acelaș lucru este adevărat și pentru răspuns, firmware-ul dispozitivului trebuind să intoarcă răspunsuri doar la nivel de protocol aplicație, gen HTTP sau SOAP.

3 Aplicația Box2Virt

Pentru exemplificarea funcționalității dispozitivului PnP UDDI s-a creat o aplicație sample .Net 3.0, scrisă folosind limbajul C#. Practic aplicația caută apariția de dispozitive PnP UDDI căutând interfețe de device-uri cu GUID-ul **A9503447-6B3B-4581-99CF-663557F7E73A** corespunzător acestui tip de funcționalități.

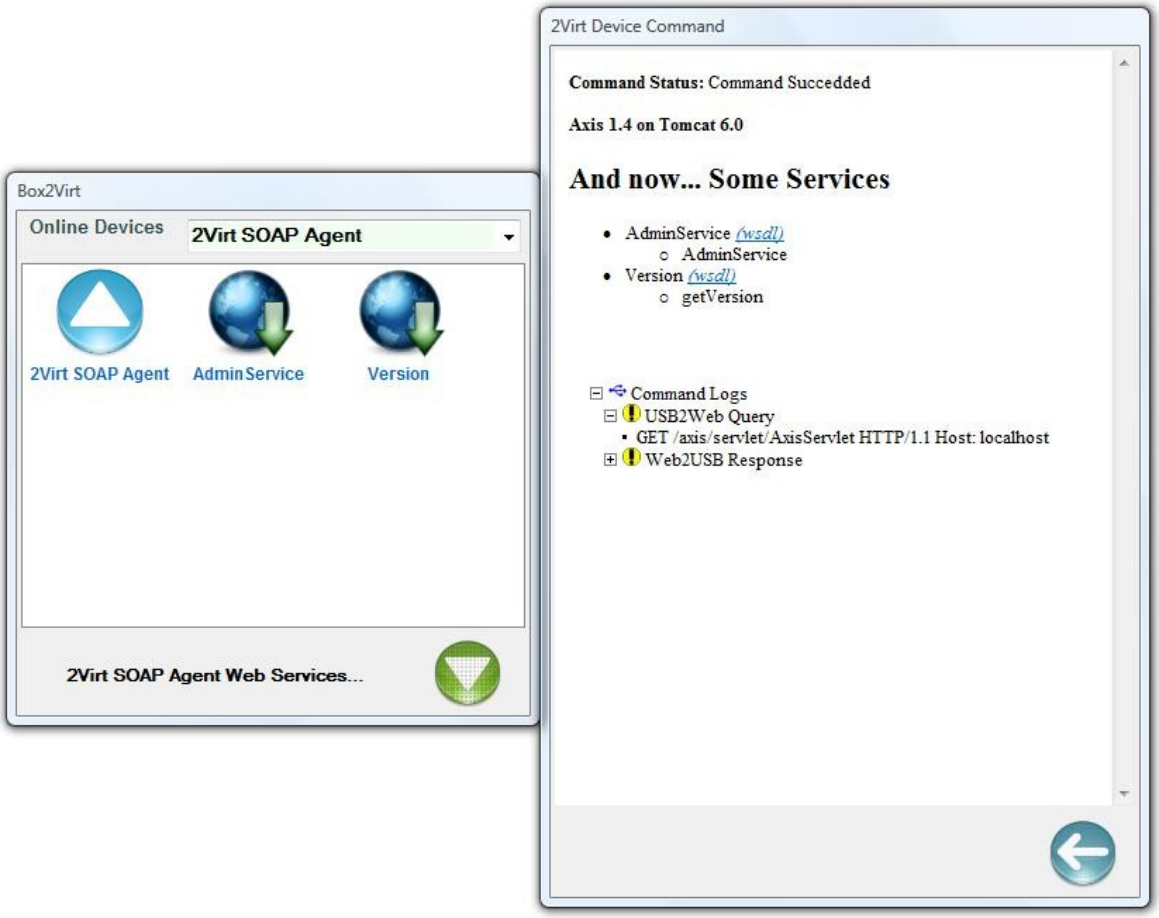
3.1 Prezentarea funcționalităților GUI

3.1.1 Identificarea dispozitivelor



După cum se poate vedea din imaginea de mai sus, odată ce un dispozitiv a fost identificat ca fiind conectat la mașina curentă, utilizatorul poate vedea configurația acestuia, pornind de la controller-ul host în care aceasta a fost conectat. În cazul de mai sus este vorba de Windows Vista cu Device Simulation Framework instalat.

3.1.2 Descoperirea și încărcarea serviciilor web

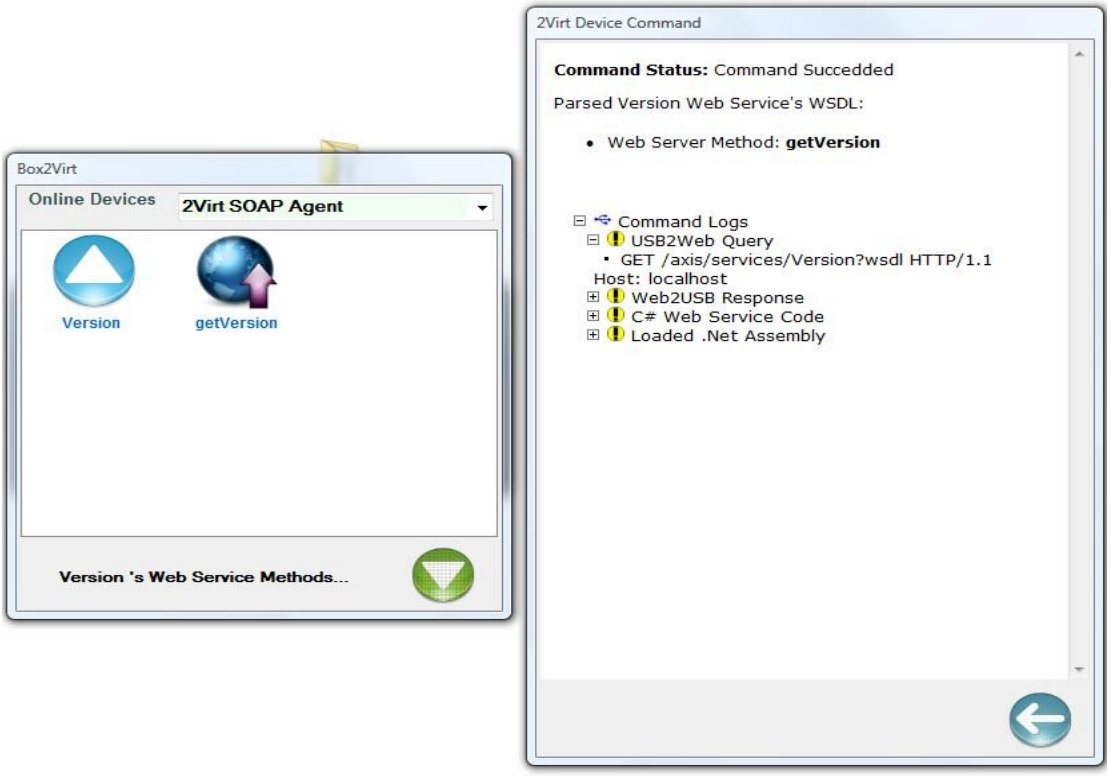


Pasul următor este utilizarea facilității de UDDI Discovery. Aceasta va efectua asupra device file-ului dispozitivului PnP UDDI un IOCTL pentru enumerarea serverelor web la care dispozitivul este conectat, urmând să se efectueze un apel corespunzător metodei de interogare a serverului web pentru listarea serviciilor. Dupa cum se vede in logul de mai sus *USB2WebQuery* conține un request HTTP către servletul din serverul Axis răspunzător pentru listarea serviciilor. Aplicația va parsa acest reply HTML și va încărca serviciile aflate pe serverul Axis prin intermediul dispozitivului PnP UDDI. Cele două servicii default sunt prezentate în figura de sus, *AdminService* si *Version*, care vin cu instalarea default a serverului Apache Axis.

3.1.3 Încărcarea unui serviciu web

Un utilizator poate apoi să încarce un serviciu web în aplicație pentru a avea posibilitatea utilizării metodelor acesteia. Să presupunem că utilizatorul a ales folosirea serviciului Version. Pașii corespunzători folosirii serviciului de către aplicație se pot vedea urmărind log-ul Comenzii:

- Utilizatorul selectează serviciul web *Version.*



Conform cu informațiile primite recent prin fisierul HTML dat de servletul AxisServlet se efectuează un query HTTP pentru opținerea WSDL-ului serviciului:

GET /axis/services/Version?wsdl HTTP1/1\r\nHost: localhost

- Aplicația va citi apoi de pe endpointul IN, prin intermediul apelului de read pe device file următorul WSDL:

```
Web2USB Response
 HTTP/1.1 200 OK Server: Apache-Coyote/1.1 Content-Type: text/xml;charset=utf-8 Transfer-Encoding: chunked Date: Sat, 30 Jan 2010 19:00:16 GMT 7fc
-<wsdl:definitions targetNamespace="http://localhost/axis/services/Version">
 +<!-- -->
  <wsdl:message name="getVersionRequest"/>
 -<wsdl:message name="getVersionResponse">
    <wsdl:part name="getVersionReturn"type="xsd:string"/>
  </wsdl:message>
 -<wsdl:portType name="Version">
  -<wsdl:operation name="getVersion">
     <wsdl:input message="impl:getVersionRequest"name="getVersionRequest"/>
     <wsdl:output message="impl:getVersionResponse"name="getVersionResponse"/>
    </wsdl:operation>
  </wsdl:portType>
 -<wsdl:binding name="VersionSoapBinding"type="impl:Version">
    <wsdlsoap:binding style="rpc"transport="http://schemas.xmlsoap.org/soap/http"/>
  -<wsdl:operation name="getVersion">
     <wsdlsoap:operation soapAction=""/>
    -<wsdl:input name="getVersionRequest">
       <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"namespace="http://axis.apache.org"use="encoded"/>
     </wsdl:input>
    -<wsdl:output name="getVersionResponse">
       <wsdlsoap:body
         encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"namespace="http://localhost/axis/services/Version"use="encoded"/>
     </wsdl:output>
    </wsdl:operation>
  </wsdl:binding>
 -<wsdl:service name="VersionService">
  -<wsdl:port binding="impl:VersionSoapBinding"name="Version">
     <wsdlsoap:address location="http://localhost/axis/services/Version"/>
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```

- WSDL-ul complet primit va fi convetit în cod C#:

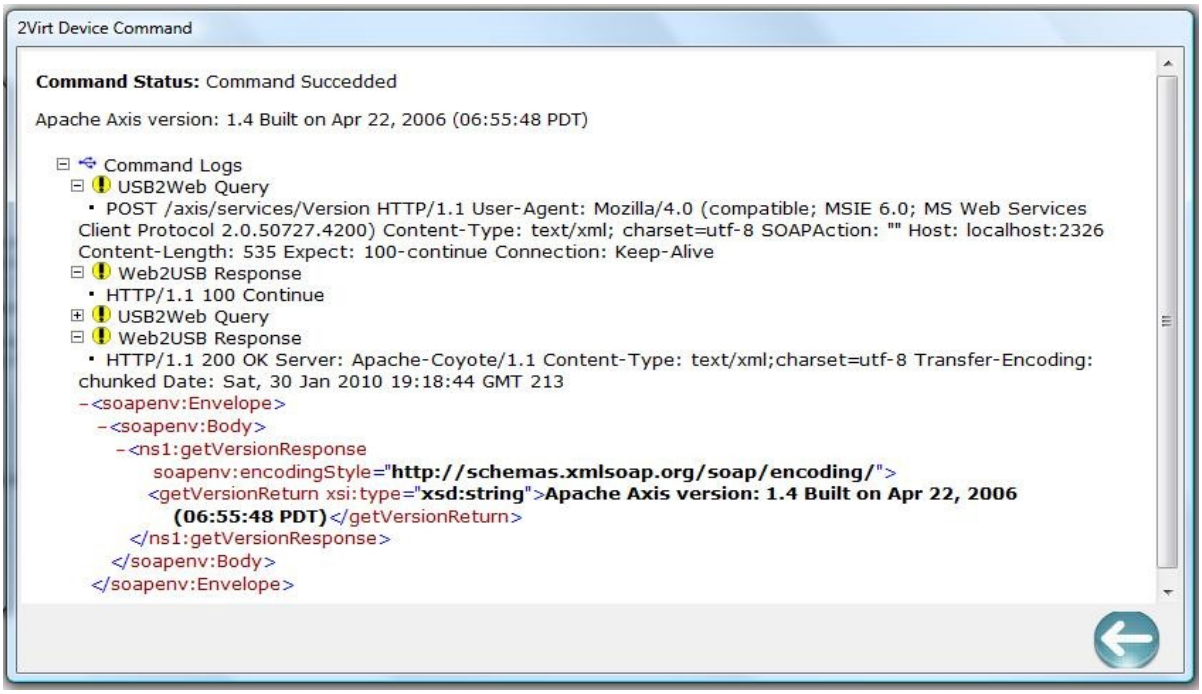
```
//------------------------------------------------------------------------------
// <auto-generated>
//     This code was generated by a tool.
//     Runtime Version:2.0.50727.4200
//
//     Changes to this file may cause incorrect behavior and will be lost
//     if the code is regenerated.
// </auto-generated>
//------------------------------------------------------------------------------

namespace Usb2WebService0 {
    using System.Diagnostics;
    using System.Web.Services;
    using System.ComponentModel;
    using System.Web.Services.Protocols;
    using System;
    using System.Xml.Serialization;


    /// <remarks/>
    [System.CodeDom.Compiler.GeneratedCodeAttribute("Box2Virt",
"0.0.0.1")]
    [System.Diagnostics.DebuggerStepThroughAttribute()]
    [System.ComponentModel.DesignerCategoryAttribute("code")]
    [System.Web.Services.WebServiceBindingAttribute(Name="VersionSoapBind
ing", Namespace="http://localhost/axis/services/Version")]
    public partial class VersionService :
System.Web.Services.Protocols.SoapHttpClientProtocol {

        /// <remarks/>
        public VersionService() {
            this.Url = "http://localhost/axis/services/Version";
        }

        /// <remarks/>
        [System.Web.Services.Protocols.SoapRpcMethodAttribute("",
RequestNamespace="http://axis.apache.org",
ResponseNamespace="http://localhost/axis/services/Version")]
        [return:
System.Xml.Serialization.SoapElementAttribute("getVersionReturn")]
        public string getVersion() {
            object[] results = this.Invoke("getVersion", new object[0]);
            return ((string)(results[0]));
        }
    }
}
```

De notat este că codul generat folosește clasa *SoapHttpClientProtocol* pentru interogarea serviciului web. Deși această clasă a fost gândită să folosească stream-uri TCP/IP pentru trimiterea de apeluri SOAP, ele vor fi redirecționate către device file-ul asociat cu dispozitivul USB PnP UDDI.

- Mai departe aplicația va compila codul generat într-un assembly .dll specific .Net pe care îl va încărca, putând astfel chema prin reflexie metodele web corespunzătoare serviciului, cum ar metoda C# public string getVersion() aparținând clasei VersionService

3.1.4 Apelarea unei metode SOAP

Din acest moment serviciul a fost încărcat în aplicație și un utilizator poate apela metoda ***getVersion*** a serviciului. Aceasta va însemna trimiterea și primirea de mesaje SOAP prin intermediul operațiilor de write și read pe device file. Acestea vor fi trimise fără să fie encapsulate în TCP/IP pe endpointurile device-ului. Log-urile comenzii pot arăta acest lucru:



După cum se poate observa s-a primit ca răspuns versiunea și data la care a fost compilat serverul Axis.

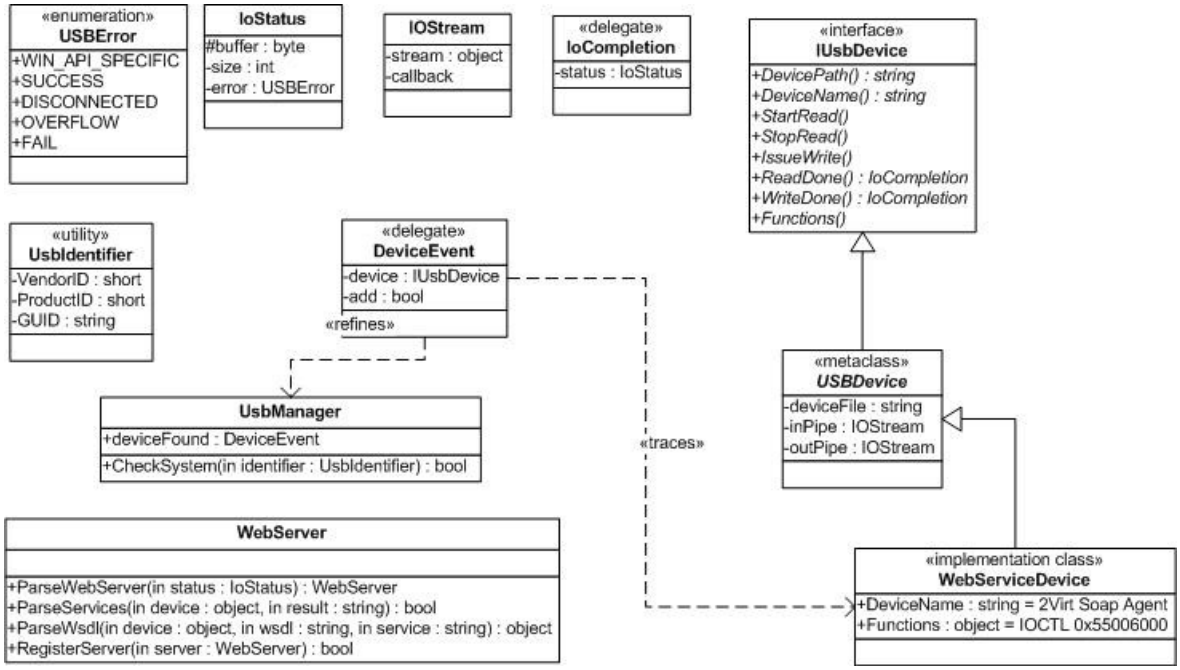
Pe final ceea ce trebuie notat este că aplicația ***Box2Virt*** are capacitatea de a apela orice metodă din orice serviciu web atâta timp cât există o metodă de a procura WSDL-ul ce descrie componența serviciului.

În următorul capitol voi da o scurtă descriere a structurii aplicației din punctul de vedere al arhitecturii OOP.

3.2 Arhitectura OOP a implementării

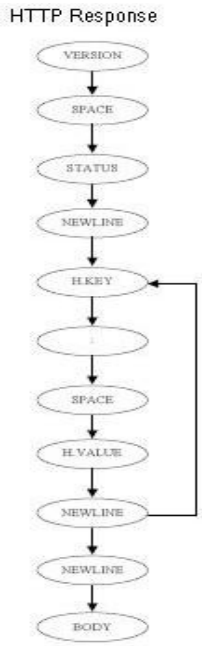
Aplicația Box2Virt este momentan împărțită în două assembly-uri .Net, *2Virt* și *Box2Virt* corespunzătoare comunicării cu dispozitive USB generice, respectiv ocupându-se de implementarea GUI-ului și al componentelor de comunicare cu serviciile web accesibile prin device-ul USB. Practic cea mai importantă caracteristică a implementării aplicației a stat in separarea funcționalităților specifice unui dispozitiv USB de folosirea

acesteia de către GUI-ul aplicației. Această funcționalitate este realizată în assembly-ul *2Virt*, diagrama UML a acesteia fiind prezentată mai jos:



Practic tot ce va trebui să facă un client al assembly-ului *2Virt* este să adauge clasei singleton *UsbManager* un *UsbIdentifier* ce conține informațiile despre dispozitivul pe care îl caută în sistem (product id, vendor id, interface GUID). Daca un eveniment corespunzător unui astfel de dispozitiv a fost identificat, va fi lansat envenimentul ***deviceFound*** la care se pot subscrie consumatori. Odata identificat un dispozitiv, acesta poate fi accesat prin interfața *IUsbDevice*, ce conține metode pentru lucrul cu device-ul ca cu un stream asincron, precum și informații cu privire la IOCTL-urile acestuia. Pentru dispozitivul PnP UDDI, abstractizarea acestuia se regăsește in implementarea de *WebServiceDevice*.

Una dintre problemele care au trebuit să fie rezolvate a fost determinarea faptului că un query HTTP sau SOAP s-a terminat, că s-au primit toate datele de la dispozitivul USB pentru a putea fi procesate mai departe. Cum dispozitivul USB nu dispune decât de o interfața de stream-uri, fiecare tip de comandă asociată dispozitivul trebuie să implementeze o metodă pentru a determina faptul că a primit toate datele. Pentru comenzile HTTP acest lucru nu a fost posibil decât instrumentat, prin parsarea răspunsului conform stărilor descrise la dreapta, fiind conștient și de encodingul răspunsului **Content-Type: length/chunked.**



4 Device Driverul PnP UDDI

Pentru a oferi acea abstractizare la nivel de fișier pe care un dezvoltator de aplicații să-l folosească pentru a comunica cu un dispozitiv PnP UDDI, un sistem de operare va trebui sa încarce un driver în spațiul de kernel. În funcție de cum se privește acest lucru, aceasta poate fi considerată atât un avantaj cât și un dezavantaj. Avantajul este că pentru fiecare familie de sisteme de operare acest driver trebuie să fie dezvoltat o singură dată, și majoritatea sistemelor utilizate în masă sunt forward compatible, ceea ce înseamnă că driverele scrise pentru o anumită versiune de kernel vor continua să funcționeze și pe kernele mai noi. Partea proastă este ca dezvoltarea driverului trebuie realizată pentru fiecare sistem în parte, ceea ce înseamnă că implementatorul efectivi al software-ului trebuie să fie familiarizați cu arhitectura și API-ul specific fiecărui sistem de operare dorit.

4.1 Implementarea generică a driverului

Pentru o mai bună înțelegere a ceea ce se realizează la acest nivel trebuie arătat ce trebuie implementat, toate cele enumerate mai jos realizându-se într-o manieră specifică fiecărui sistem de operare:

- să implementeze modalitatea de adăugare, scoatere a dispozitivului respectiv din punctul de vedere al integrării acestuia cu componentele sistemului de operare
- să configureze dispozitivul, setarea caracteristicilor hardware (viteza de comunicare, selectarea unui configurări)
- să asigure expunerea unei interfețe, conform cu paradigma sistemului de operare pentru endpointurile acestuia, sistemul de IOCTL-uri
- să asigure redirecționarea operațiilor de read/write/IOCTL către endointurile pe care driverul știe că dispozitivul le prezintă.

În cele din urmă, ceea ce este important de precizat este că implementarea efectivă a device driverului este strâns legată de cunoașterea caracteristicilor firmware-ului dispozitivului USB, în acelaș mod cum un implementator de sistem de operare trebuie să cunoască specificațiile platformei pe care va rula acesta. În cazul dispozitivului PnP UDDI aceastea au fost enumerate în paragraful 2.2.3, și anume numărul IOCTL-ului pentru listarea serviciilor web, precum și faptul că există un endpoint Bulk OUT în care se vor trimite request-urile de write și un endpoint Bulk IN pentru apelurile de read.

4.2 Driverul usb2virt.sys pentru Windows NT

Pentru a demonstra o astfel de implementare de device driver pentru dispozitivul PnP UDDI s-a ales utilizarea Windows Driver Kit sub Windows Vista. În continuare nu ne vom axa pe prezentarea platformei WDK pentru dispozitive, alte surse online și nu numai fiind mult mai indicate, ci doar vom prezenta cum elementele specificate în capitolul precedent s-au regasit în această implementare. Ceea ce trebuie subliniat este că asocierea dintre device-ul PnP UDDI și driverul usb2virt.sys, sub Windows o va face chiar utilizatorul. Atunci cand un dispozitiv PnP UDDI este conectat prima dată, sistemul va localiza sau va cere utilizatorul să identifice o locație unde se află driverul. Asocierea dintre *ProductID/VendorID* și *usb2virt.sys* este realizată de dezvoltator prin fisierul *usb2virt.inf* ce trebuie să se regăsească la aceeași locație cu driverul.

Primul lucu pe care o va face implementarea de driver este să seteze funcționalitățile necesare când un device nou PnP UDDI este adăugat:

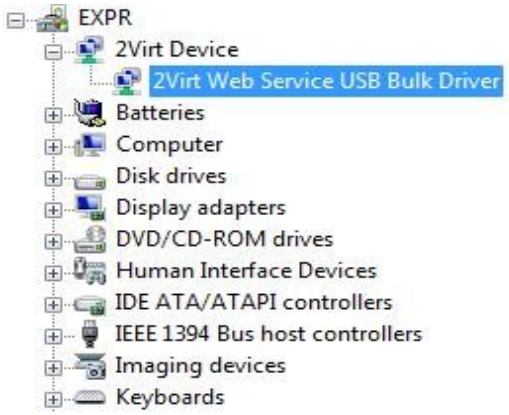
```
// Se creaza un handle pentru device-ul USB pentru a comunica cu
// stiva USB. Handle-ul va fi folosit pentru a configura si seta
// propietatile device-ului
if (pDeviceContext->UsbDevice == NULL) {
    status = WdfUsbTargetDeviceCreate(Device,
                                      WDF_NO_OBJECT_ATTRIBUTES,
                                      &pDeviceContext->UsbDevice);
}
// Se cauta informatii depre versiunea USB, portul la care este
// conectat si capabilitatile device-ului
WDF_USB_DEVICE_INFORMATION_INIT(&deviceInfo);
status = WdfUsbTargetDeviceRetrieveInformation(
                               pDeviceContext->UsbDevice,
                               &deviceInfo);
// Se inregistreaza interfata device-ului care poate fi obtinuta din
// userspace sub forma unui device file folosind GUID-ul interfetei
status = WdfDeviceCreateDeviceInterface(device,
                     (LPGUID) &GUID_DEVINTERFACE_USB2VIRT,
                     NULL);
// Se selecteaza configuratia device-ului dintre cele existente
WDF_USB_DEVICE_SELECT_CONFIG_PARAMS_INIT_SINGLE_INTERFACE( )
status = WdfUsbTargetDeviceSelectConfig(
                               pDeviceContext->UsbDevice,
                               WDF_NO_OBJECT_ATTRIBUTES, &configParams);
// Se obtin pipe-urile de scriere/citire
if(WdfUsbPipeTypeBulk == pipeInfo.PipeType &&
            WdfUsbTargetPipeIsInEndpoint(pipe))
        pDeviceContext->BulkReadPipe = pipe;
if(WdfUsbPipeTypeBulk == pipeInfo.PipeType &&
            WdfUsbTargetPipeIsOutEndpoint(pipe))
        pDeviceContext->BulkWritePipe = pipe;
```

Pentru a exemplifica și modalitatea de comunicare pe care driverul trebuie să o implementeze ca funcționalitate vom demonstra un apel de write. Mai întâi driverul va înregistra la interfața creată mai sus o coadă pentru a prelucra apelurile de write făcute de o aplicație pe un device file ce abstractizează această interfață:

```
    // se configureaza o coada pentru a procesa apelurile de write
    // intr-un mod secvential
    WDF_IO_QUEUE_CONFIG_INIT(&ioQueueConfig,
                                WdfIoQueueDispatchSequential);
    ioQueueConfig.EvtIoWrite = BulkIoWrite;
    ioQueueConfig.EvtIoStop  = EvtIoStop;
    status = WdfIoQueueCreate(device, &ioQueueConfig,
                 WDF_NO_OBJECT_ATTRIBUTES, &queue); // queue handle
     status = WdfDeviceConfigureRequestDispatching(
                      device, queue, WdfRequestTypeWrite);
    // BulkIoWrite va fi apelata pentru WDFREQUEST-urile din coada
VOID BulkIoWrite(
    __in WDFQUEUE Queue, __in WDFREQUEST Request, __in size_t Length)
    // se obtine endpointul OUT pentru write
    pDeviceContext = GetDeviceContext(WdfIoQueueGetDevice(Queue));
    pipe = pDeviceContext->BulkWritePipe;
    // se formateaza memoria requestului de write (bufferul)
    status = WdfRequestRetrieveInputMemory(Request, &reqMemory);
    status = WdfUsbTargetPipeFormatRequestForWrite(pipe,
                            Request, reqMemory, NULL);
    // se seteaza rutina care va fi apelata cand firmware-ul a confirmat
    WdfRequestSetCompletionRoutine( Request,
                            EvtRequestWriteCompletionRoutine, pipe);
    // trimite asincron requestul catre firmware-ul dispozitivului prin
    // endpointul de OUT
    WdfRequestSend( Request,
              WdfUsbTargetPipeGetIoTarget(pipe), WDF_NO_SEND_OPTIONS)
```

Mesajul primit de la aplicație și pus de kernel în coada de write va fi procesat ca mai sus, driverul PnP UDDI trimitând mesajul mai departe în stiva de procesare USB către subsistemul care se ocupă de comunicația cu controller-ul host USB. Când firmware-ul dispozitivului va anunța controller-ul că a terminat de primit pachetul, un nou eveniment va fi procesat de driverul *usb2virt.sys*, și anume routina de terminare a mesajului, setată mai sus:

```
VOID EvtRequestWriteCompletionRoutine(__in WDFREQUEST Request,
    __in WDFIOTARGET Target,
    __in PWDF_REQUEST_COMPLETION_PARAMS CompletionParams,
    __in WDFCONTEXT Context)
    // se va trimite catre subsistemul care se ocupa de apelul de write
    // din kernel cati bytes au fost scrisi. In functie de cum a fost
    // facut apelul pe device file de aplicatia userspace (sincron sau
    // asincron aceasta va determina ca threadul userspace sa revina
    // in userspace
    bytesWritten =  usbCompletionParams->Parameters.PipeWrite.Length;
    WdfRequestCompleteWithInformation(Request, status, bytesWritten);
```



Pentru a ilustra cum va vedea un utilizator dispozitivul după ce a fost adăugat la mașina curentă putem vedea cum este prezentat de utilitarul Device Manager pe un Windows Vista.

5 Dispozitiv PnP UDDI Simulat

Rezumat. În acest capitol se va prezenta simularea firmware-ului unui device PnP UDDI folosind platforma Driver Simulation Framework de la Microsoft. Acest mediu are avantajului de a fi unul propice pentru un implementator de dispozitive PnP UDDI pentru ca are posibilitatea de a dezvolta întregul sistem (dispozitiv + driver + aplicație) pe aceeași mașină, sistem de operare.

În următoarele paragrafe vom descrie cum s-a implementat un device USB în acest framwork, prezentând în linii mari caracteristicile simulării precum și câteva aspecte tehnice.

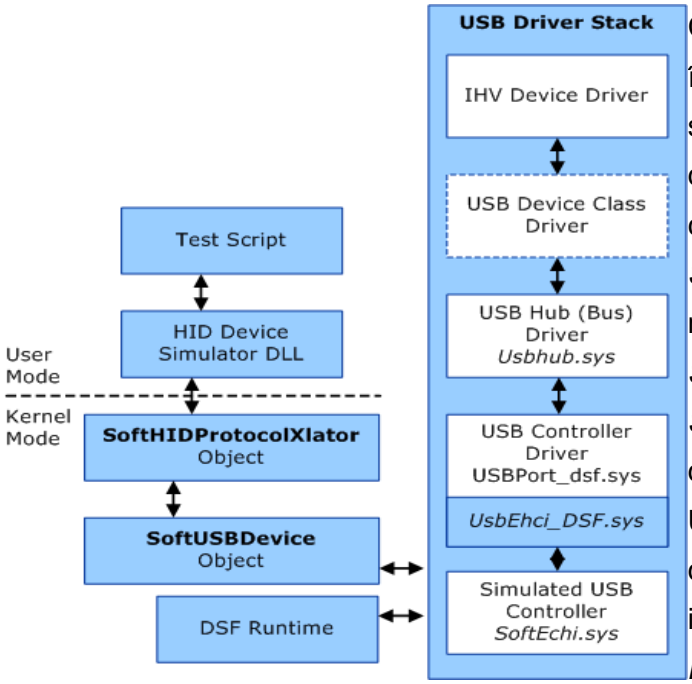
5.1 Device Simulation Framework

Platforma DSF pentru dispozitive USB reprezintă un framework arhitectural de programare pentru implementarea și simularea dispozitivelor pe o mașină gazdă folosind doar componente software. Acest framework este alcătuit dintr-un controller host EHCI simulat, precum și dintr-o colecție de obiecte COM (Component Object Model) pentru configurarea și implementarea funcționalității device-ului. Componentele COM implementează protocolul USB din perspectiva dispozitivului simulat, iar controllerul EHCI ce se comportă ca un simulator trebuie să mute pachetele USB de și la obiectele COM care reprezintă endpointurile device-ului. De exemplu o aplicație test (în cazul nostru scriptul visual basic *RunUSBProxy.wsf*) crează controller-ul EHCI simulat, crează și conectează device-ul la controller și apoi foloseste device-ul (atât prin funcționalitatea interfețelor expuse de obiectele COM, pe partea de implementare a dispozitivului, cât și prin apeluri de read/write/IOCTL pe device file-ul din userspace expus de sistem pentru a fi folosit de aplicații) pentru a simula interacțiunea dintre hardware-ul dispozitivului și sistemul de operare (Windows NT) la care acesta e conectat.

Avantajul folosirii obiectelor COM este acela ca dispozitivul poate fi implementat sau poate fi apelat de orice limbaj de programare pentru care exista facilitatea de OLE-automation. De exemplu s-a implementat funcționalitățile de firmware in C++, extinzând obiectele COM disponibile prin API-ul DSF, iar utilizarea efectivă a acestora s-a realizat în Visual Basic Script. Practic device-ul este compilat intr-o bibliotecă partajată, *SoftUSBProxy.dll*, care prin intermediul facilităților COM va fi încărcată într-o aplicație, cum ar fi scriptul visual basic.

Versiunea specifică implementată de DSF corespunzătoare unui controller EHCI trebuie se regăseste într-un driver, Usbehci_dsf.sys, încărcat în kernelul NT. Acesta se comportă ca un dispozitiv hardware, înterceptând accesele la registrii si memorie (de tip DMA – Direct Memory Access) și generând întreruperi hardware simulate.

În figura de mai jos simulatorul de device-uri USB se regăsește sub forma unui server COM inter-proces, adăugat prin încărcarea unui biblioteci partajate. Un script test crează o instanță a obiectului de simulat în constructorul căruia se creaza și se configurează obiectul *SoftUSBDevice*. Acesta din urmă expune mai multe interfețe (cum ar fi *SoftUsbConfiguration*, *SoftUsbInterface*, *SoftUsbEndpoint*) ce encapsulează diferite funcționalități ale unui dispozitiv USB. După ce obiectul corespunzător dispozitivului USB a fost creat, se intanțiază un obiect *DSF*, a cărui metodă *IDSF::HotPlug* va fi folosită pentru a conecta obiectul dispozitiv la hubul părinte al host controller-ului EHCI emulat de framework. Simulatorul în care rulează controller-ul EHCI va modifica anumiți registrii corespunzători cu specificația de implementare hardware a acestuia și va simula o întrerupere hardware. Aceasta va fi preluată de implementarea de miniport EHCI din kernelul NT care va ridica evenimentul mai sus în stiva USB implementată de codul kernelului. Ideea de bază din spatele acestei arhitecturi este că implementarea generică de miniport EHCI, precum și kernelul nu va face diferența între un dispozitiv real și unul simulat.



Când miniportul EHCI va submite o transacție pentru a fi executată de controller-ul host simulat, acesta va scrie datele într-o zonă de memorie pe care simulatorul le va citi asincron prin verificarea periodică a acelei zone de memorie, ce mapează anumiți registrii interni ai unui controller real EHCI. Simulatorul aflând în acest mod de existența a noi pachete USB va trimite evenimentul la obiectele COM ce abstractizează implementarea dispozitivului USB, ajungând în cele din urmă la implementarea de *ISoftUsbEndpoint*, care va putea să dea comenzi simulatorului să citească acea zonă de memorie din spațiul de adrese al kernelului de Windows.

5.2 Arhitectura DSF a dispozitivului PnP UDDI

Pentru a implementa acest dispozitiv a trebuit ca elementele enumerate în paragraful 2.2.3 să se regăsească și în configurația și implementarea obiectelor COM, corespunzătoare framework-ului DSF, precum și ca să se creeze un client TCP care să fie concetat la un server web (Apache Axis) pentru a trimite mai departe mesajele SOAP care vin prin funcționalitatea expusă de interfața *ISoftUsbEndpoint*.

Dispozitivul PnP UDDI este realizat sub forma unui obiect scris in C++, *CUSBProxyDevice*, care moșteneste interfețele expuse de platforma DSF:

```
class ATL_NO_VTABLE CUSBProxyDevice :
    public CComObjectRootEx<CComSingleThreadModel>,
    public CComCoClass<CUSBProxyDevice, &CLSID_USBProxyDevice>,
    public IConnectionPointContainerImpl<CUSBProxyDevice>,
    public IConnectionPointImpl<CUSBProxyDevice,
&__uuidof(IUSBProxyDeviceEvents)>,
    public ISoftUSBEndpointEvents, // DSF USB device interface
    public IDispatchImpl<IUSBProxyDevice, &IID_IUSBProxyDevice,
&LIBID_SoftUSBProxyLib, /*wMajor =*/ 1, /*wMinor =*/ 0>
```

În codul corespunzător instațierii obiectului se vor seta propietățile firmware-ului:

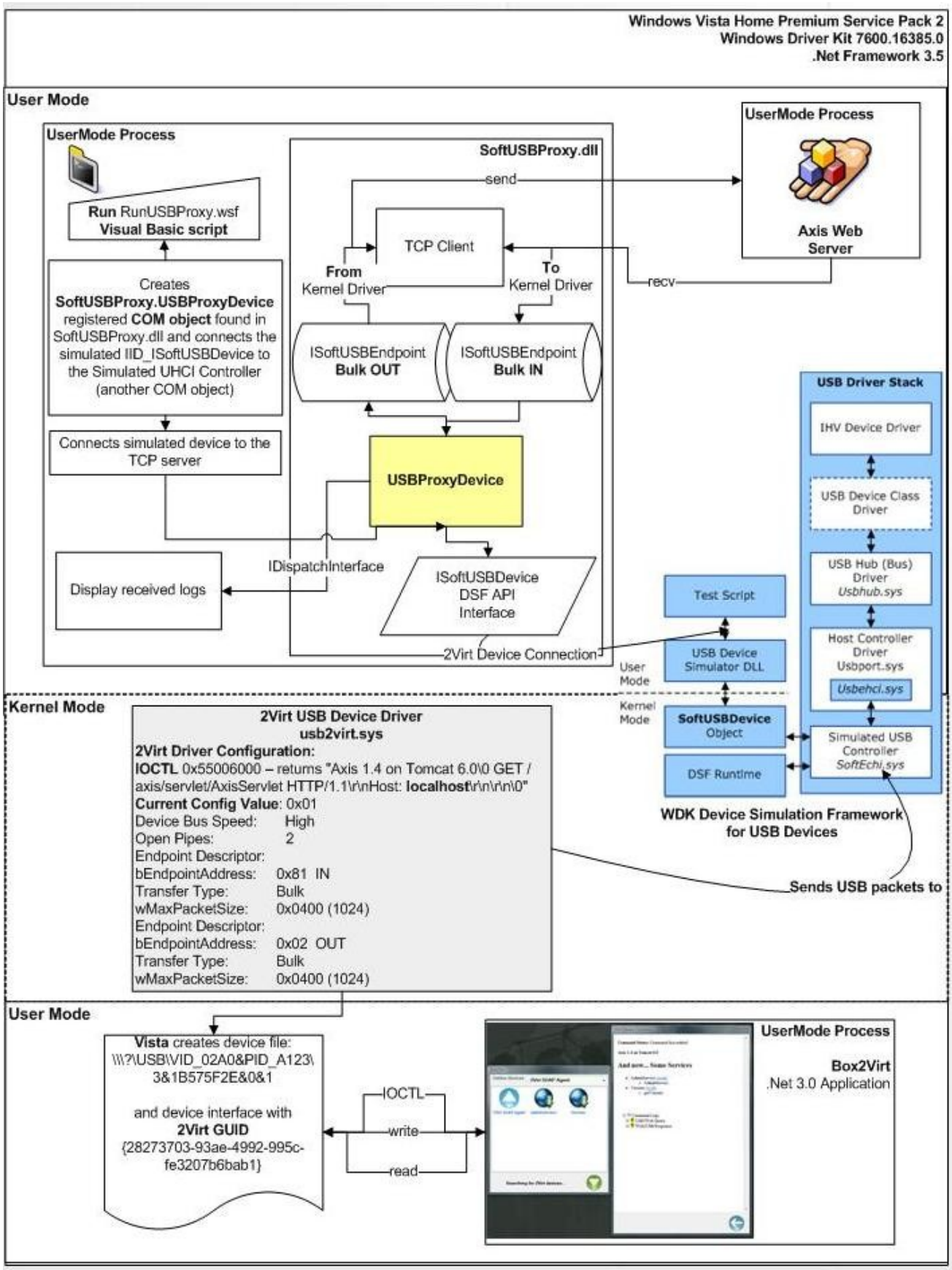
```
CHECK_FAIL( pDevice->put_MaxPacketSize0(64) );
CHECK_FAIL( pDevice->put_Vendor(pUsbId->vendorId) );   // 0x02A0
CHECK_FAIL( pDevice->put_Product(pUsbId->productId) ); // 0xA123
// Se creaza si configureaza o interfata
CHECK_FAIL( CoCreateInstance(CLSID_SoftUSBInterface, NULL,
            CLSCTX_INPROC_SERVER, __uuidof(ISoftUSBInterface),
            reinterpret_cast<void**>(&piInterface)) );
// Se adauga si cele 2 endpointuri Bulk IN si OUT
CHECK_FAIL( CoCreateInstance(CLSID_SoftUSBEndpoint,
            NULL, CLSCTX_INPROC_SERVER, __uuidof(ISoftUSBEndpoint),
            reinterpret_cast<void**>(&piEndpoint)) );
// Se seteaza tipul Bulk IN/OUT + maxPacketSize=1024
CHECK_FAIL( piEndpoint->put_EndpointAddress(pEndpoint->address) );
CHECK_FAIL( piEndpoint->put_Attributes(pEndpoint->endpointAttr.Byte) );
CHECK_FAIL( piEndpoint->put_MaxPacketSize(pEndpoint->maxPacketSize) );
```

O descriere destul de intuitivă a componentelor impementării dispozitivului PnP UDDI în acest mediu de dezvoltare sub DSF se regăseste în figura imediat următoare. După cum se poate observa toate componentele sunt doar obiecte software aflate pe aceeași mașină ce rulează un Windows NT. Pentru o mai bună înțelegere a îmbinării componentelor arhitecturii PnP UDDI să vedem care sunt pașii pe care îi va parcurge sistemul pentru conectarea unui astfel de dispozitiv simulat:

- Utilizatorul rulează scriptul visual basic: *cscript RunUSBProxy.wsf*

```
Dim USBProxyDev    : Set USBProxyDev    =
WScript.CreateObject("SoftUSBProxy.USBProxyDevice", "USBProxyEvents_")
Dim USBProxyDSFDev : Set USBProxyDSFDev = USBProxyDev.DSFDevice
Dim USBProxyUSBDev : Set USBProxyUSBDev =
USBProxyDSFDev.Object(IID_ISoftUSBDevice)
WriteLine "Connecting simulated USB Proxy device to simulated EHCI controller"
Dim Bus : Set Bus = DSF.HotPlug(USBProxyDSFDev, "USB2.0")
```

5 Dispozitiv PnP UDDI Simulat



- Servicul de OLE automation a sistemului Windows va încărca, în spațiul de adrese al procesului corespunzător scriptului visual basic, DLL-ul conținând serverul COM intra-process, ce va creea obiectul *CUSBProxyDevice* simulând dispozitivul PnP UDDI. Acesta se va conecta la serverul web aflat pe mașina locală printr-un socket tcp deschis pe portul de listen al serverului. Această conexiune, reprezentată în figură de elementul *TCP Client* va fi folosită pentru a trimite la server mesajele primite prin endpoint.
- Simulatorul astfel creat pentru obiectul *SoftUsbDevice* din .DLL va modifica zona de memorie mapată ca controller EHCI de framework-ului DSF, cu datele necesare conectării unui nou device, ca și în cazul când această zonă de adrese PCI ar fi fost actualizată de un controller EHCI real.
- Implementarea de miniport EHCI, *Usbehci.sys* din kernel va detecta că un nou dispozitiv a fost adăugat controller-ului, va interoga informații despre noul device, în special perechea *VendorId/ProductId* apoi va încărca în kernel driverul *usb2virt.sys* descris în capitolul 4.2.
- Driverul va expune în userspace un device file pe care aplicația .Net *Box2Virt*, descrisă în capitolul 3, îl poate folosi pentru a comunica cu device-ul PnP UDDI, fără să știe faptul că dispozitivul, de fapt, reprezintă o componentă software simulată.

Să presupunem că aplicația Box2Virt va face un apel de write pe device file pentru a trimite un mesaj SOAP:

- apelul va ajunge în driver care îl va înainta pe endpointul de OUT. Acest apel va fi prelucrat de stiva USB din kernel și va ajunge în cele din urmă să se regăsească în date scrise în zona de memorie corespunzătoare controller-ului EHCI emulat.
- Simulatorul DSF corespunzător implementării obiectului *CUSBProxyDevice* va descoperi acest apel pentru dispozitiv și îl va trimite prin cod la metoda *OnWriteTransfer* a interfeței *ISoftUsbEndpoint* implementată de obiectul nostru, care le va trimite mai departe către web server:

```
STDMETHODIMP CUSBProxyDevice::OnWriteTransfer(
     BYTE DataToggle, BYTE *pbDataBuffer, ULONG cbDataBuffer, BYTE *pbStatus)
{

    // Trimite datele pentru a fi afisate pe consola aplicatiei script
    if (m_CharDevice->GetLogger() != NULL)
       CHECK_FAIL( m_CharDevice->GetLogger()->Log(pbDataBuffer, cbDataBuffer) )
    // Trimite requestul mai departe catre socketul tcp deschis la Axis
    CHECK_FAIL( m_CharDevice->Write(pbDataBuffer, cbDataBuffer) );
    // Seteaza faptul ca firmware-ul a procesat tot pachetul USB
    *pbStatus = USB_ACK;
    return hr;
}
```

Pentru a procesa datele primite de la web server obiectul *CUSBProxyDevice* ascultă pe un thread diferit mesajele de pe socket. Când s-a primit un mesaj pe socket acesta va înainta pe endpointul de IN, expus de Simulator prin interfața de *ISoftUsbEndpoint*:

```
HRESULT CUSBProxyDevice::CharDeviceData(void *pUser, BYTE* data, ULONG size)
{
    // Se vor fragmeta pachetele in dimensiunea maxima acceptata de endpointul IN
    CHECK_FAIL( pThis->m_piINEndpoint->get_MaxPacketSize (&sMaxPacketSize) );
    do
    {
       toSend = ((SHORT)size < sMaxPacketSize) ? (SHORT)size : sMaxPacketSize;
       // Se va apela metoda de a trimite pachetul a interfetei ISoftUsbEndpoint
       CHECK_FAIL( pThis->m_piINEndpoint->QueueINData(pCopy, toSend, bINStatus,
SOFTUSB_FOREVER) );
       size -= toSend;
    } while (size > 0);
    return hr;
}
```

- datele vor fi scrise de Simulator în zona de memorie corespunzătoarea controllerului host simulat. Implementarea de miniport USB din kernel va detecta pachetele noi USB și le va înainta catre coada de primire a pachetelor unde va fi rulată logica implementată de device driverul dispozitivului pentru care pachetul era destinat
- În cazul PnP UDDI driverul *usb2virt.sys* le va trimite, implementând API-ul de kernel către device file-ul dispozitivului
- În cazul în care aplicația făcuse un apel de read pe device file acesta se va întoare prezentând informațiile primite de la firmware.

Ceea ce este important de luat în considerare din acest capitol este modalitatea de a lega serverul web printr-un char device (legatura socket de tip tcp) cu firmware-ul simulat și traseul mesajelor de la device file din userspace la server. Abstractizare legăturii este sub forma unui stream asincron între cele două entități, respectiv aplicatia Box2Virt și serverul web Apache Axis. Dacă ne propunem să examinăm contextul acestui traseu vom avea următoarele elemente:

Box2Virt [*userspace*] read/write pe device file ↔ syscall-uri Driver **usb2virt.sys** [*kernel*] ↔ driver EHCI [*kernel*] ↔ **Simulatorul DSF** [*kernel*] USB ↔ CProxyUSBDevice **SoftUSBProxy.dll** [*userspace*] read/write socket ↔ **Apache Axis** [*userspace*]

6 Dispozitiv PnP UDDI Emulat

Rezumat. În acest capitol se va prezenta emularea firmware-ului unui device PnP UDDI folosind Qemu pentru emularea dispozitivelor I/O ale unei mașini virtuale ce rulează Windows Vista, folosind ca software de virtualizare hypervizorul Xen. Se va prezenta o scurtă introducere despre conceptul de virtualizare și arhitectura specifică a implementării Xen pentru mașini virtualizate total folosind suport de IVT (Intel Virtualization Technology) , precum și atașarea unui dispozitiv PnP UDDI la această platformă, explicând componentele și interacțiunea lor în această configurație.

6.1 Concepte de virtualizare

Ce reprezintă virtualizarea unui sistem? Virtualizarea este o combinație de suport hardware și software specializat ce asigură execuția unui software existent în aceeași modalitate ca și masina pentru care software-ul a fost dezvoltat. Mediul mașinei virtuale în care software-ul se execută poate avea resurse diferite de mașina reala, atât în cantitate cat si in tipul acestora.

Monitorii de mașini virtuale ofera o modalitate buna pentru a partaja hardware-ul fizic între mai multe entități ale unor sisteme de operare. Sistemele de operare au realizat acest lucru de ani buni permitând utilizatorilor să ruleze diferite aplicații în acelaș timp. Totuși, fară virtualizare, un anumit hardware va putea suporta, la un moment dat, un singur sistem de operare, ceea ce poate fi insuficient (de exemplu dintr-un anumit motiv trebuie să ai instalat un Linux pe post de router, dar iți dorești să rulezi și aplicații într-un mediu nativ Windows). Monitorii de mașini virtuale, denumiți hypervizori, devin din ce în ce mai importanți în configurarea unui sistem modern, deoarece aceștia abstractizeaza resursele sistemului fizic, în entitați ce pot fi alocate unor anumite mașini guest. De asemenea monitorii asigură și un nivel de izolare între entitățile guest ce rulează în acelaș timp.

Virtualizarea are un rol important în mediile enterprise. Ea asigură consolidarea mai multor servere virtuale pe aceeași mașina fizică, fără să piardă din securitatea de a avea medii complet izolate. Cloud computing are la bază arta virtualizării, ce permite, printre altele și migrarea sau clonarea diferitelor mașini virtuale, asigurandu-se astfel necesitatea de a avea servere la cerere. Un alt avantaj important este acela al consumului de energie, un aspect destul de cercetat în ultimul timp, datorita costurilor mari impuse de a mentine o fermă de servere. Astfel un server idle poate fi oprit fără a opri mașina fizică

respectivă sau mai multe servere pot ocupa aceeași masină până când rata lor de consum al resurselor creste.

În acest mediu mixt un rol important îl reprezintă partea de alocare a resurselor virtuale unui sistem guest, care poate fi realizată fie prin partajarea unei resurse fizice fie prin emularea software totală a acesteia. Astfel un dispozitiv de tip disc poate fi emulat unei mașini virtuale fie la nivel de partiție fie la nivel de fișier într-un sistem de fisiere apațînând unui alt sistem de operare privilegiat. În cazul emulării unui controller host USB se poate folosi doar software care simulează comportamentul hardware al acestui dispozitiv pe o zonă de adrese din sistemul guest, pe care acesta le vede ca un spațiu de adrese PCI.

Diferitele detalii tehnice ale virtualizării în general sunt similare, totuși mai multe abordări există pentru a rezolva problemele existente în diferitele implementări. Patru tipuri de arhitecturi de virtualizare există în lumea de azi ce reusesc sa dea iluzia unui sistem de sine stătător: emularea, virtualizarea totală, para-virtualizarea și virtualizarea la nivelul de sistem de operare. Sistemele de operare moderne de obicei au o formă usoară de izolare a proceselor individuale, și au servicii pentru a partaja date inter-proces. Majoritatea sistemelor de operare au fost gândite pentru a fi utilizate de un singur utilizator, de acceea facilitățile pentru partajarea resurselor au fost mai importante decât cele pentru izolare. Totuși rolul unui hypervizor este tocmai acela de a izola accesul la resurse între mașinile virtuale. Fiecare din modalitățile de virtualizare enumerate mai sus, în implementarea lor, conferă un nivel de izolare mai redus pentru a permite partajarea de resurse intre mașinile guest. Tipic, un grad de izolare mare inseamnă performanță mai redusă, datorită overhead-ului introdus de logica mecanismelor de izolare în softul de hypervisor.

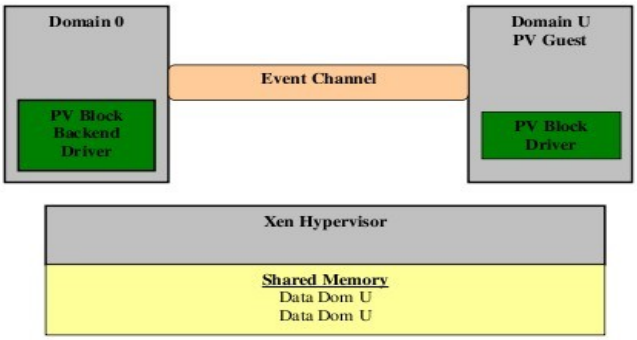
În capitolul următor vom prezenta arhitectura Xen pentru mașini virtualizate total, ce folosește atât tehnice de virtualizare pentru partajarea timpului de procesor, a memoriei și anumite componente ale mașinii fizice, ca Advanced Programmable Interrupt Controller, cât și mecanisme de emulare pentru simularea funcționării și a funcționalităților diferitelor dispozitive I/O, cum ar fi placa video, disc IDE, controller host USB.

6.2 Arhitectura XEN pentru Hardware Virtual Machine

Până la introducerea suportullui de virtualizare în procesoarele de mainstream, singura modalitate de a putea rula concomitent mai multe sisteme de operare pe aceeași mașină era numai prin metodele cunoscute ca para-virtualizare sau binary rewriting (rescrierea codului compilat). Ambele dintre aceste metode necesitau o întelegere bună a arhitecturii

sistemului guest virtualizat. În cazul sistemelor guest para-virtualizate era necesar modificarea codului sursă a acestora, ceea ce însemna că kernelul sistemului de operare trebuia recompilat. Framework-ul de virtualizare Xen suportă atât guesti para-virtualizați cât și sisteme nemodificate, ca sistemul Windows, pentru care nu există decât parțial surse de cod publice, cu ajutorul extensiilor de virtualizare existente pe procesoarele mai noi, cum ar fi Intel Virtualization Extension (Intel-VT) sau AMD-V.

Principala diferență dintre un guest para-virtualizat și unul total, denumit HVM (Hardware Virtual Machine), o reprezintă modalitatea de a accesa resursele hardware ale mașinii fizice, în special cele de I/O cum ar fi, de exemplu, controller-ul SCSI. Un guest para-virtualizat va efectua hypercall-uri (asemănătoare apelurilor de sistem făcute de o aplicație către kernel), care vor fi procesate de software-ul de virtualizare, în cazul nostru hypervizorul Xen în ring0, care va înainta apelurile către driverele reale aflate în domeniul privilegiat Linux, denumit Dom0. Aceste drivere reale, aflate în acest domeniu, au access total la dispozitivele I/O și vor efectua apelurile necesare către dispozitivele hardware.

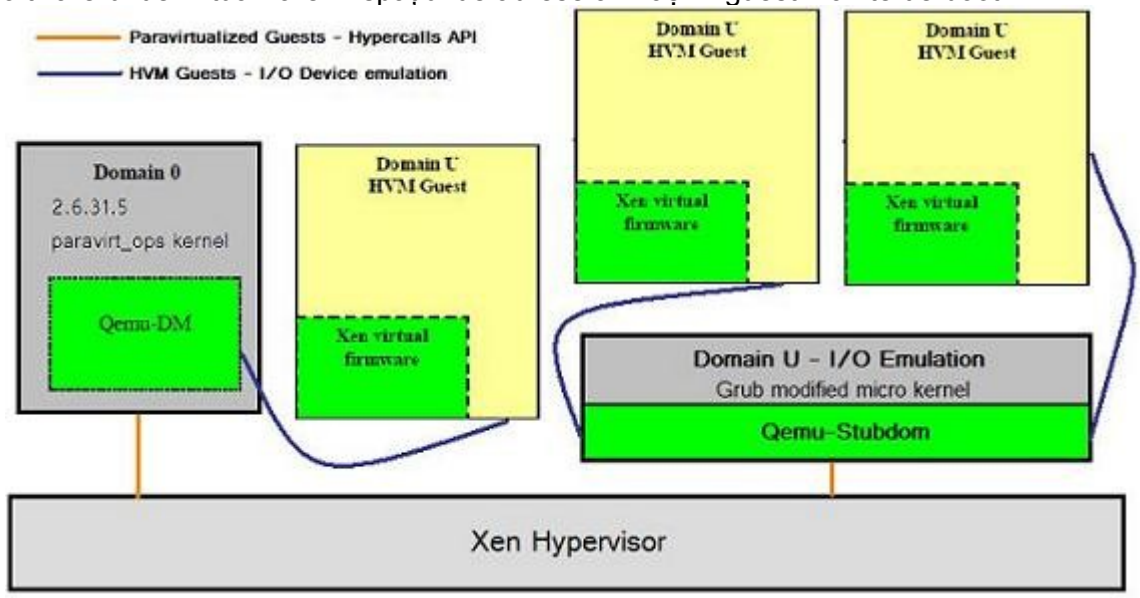


Pentru că dispozitivul PnP UDDI a fost emulat pentru un HVM ne vom axa pe prezentarea arhitecturii din spatele virtualizării acestui tip de guest în lumea Xen. Ceea ce trebuie să avem în vedere pe arhitecturile x96 și ia64 este că toate excepțiile și întreruperile externe sunt configurate să fie luate de hypervizorul Xen, software ce ruleaza în domeniul hardware cel mai privilegiat și anume ring0. Unul dintre lucrurile pe care orice software de virtualizare trebuie să-l facă este acela de a separa zonele de memorie ale mașinilor virtuale ce rulează, așa cum un sistem de operare o face pentru procesele userspace, precum și de a emula intrucțiunile la care guestul virtualizat nu are acces la nivelul hardware de privilegiu la care rulează.

De exemplu să presupunem că un kernel Windows NT, ce nu știe că rulează sub un software de virtualizare pentru a face un hypercall, dorește să mapeze o zonă de memorie într-un TLB. Într-un mod real atunci când se accesează o zonă de memorie nemapată hardware-ul va ridica o excepție de TLB miss care va fi procesată de rutina pentru această excepție a kernelului NT, care va parcurge tabela PTE a procesului

respectiv și va utiliza o intrucțiune specifică de ring0 pentru a adăuga acea mapare în TLB. Maparea pe care o va introduce kernelul este de la adresa virtuală a procesului la adresa fizică pe care kernelul o știe pentru acea zonă de memorie. Dar cum acum kernelul NT rulează într-un ring de VM entry (conform descrierii Intel) și procesorul virtual pe care rulează guest-ul nu are access la acea instrucțiune de mapare în TLB, se va ridica o excepție de VM exit care va fi prelucrată de software-ul de virtualizare. Aceasta rutină va efectua adevărata mapare din spațiul de adrese virtual al aplicației userspace a guest-ului NT la adevărata adresa fizică la care rulează mașina virtuală. Ceea ce mai trebuie înțeles este că nici întreruperea de TLB miss nu se duce direct la kernelul NT, ci mai întâi este luată de software-ul de virtualizare care o va injecta mașinii guest, știind care este rutina de tratare a acestei întreruperi a sistemului NT în spațiul de adrese al acestuia.

Aceeași emulare trebuie să fie realizată și în cazul resurselor de I/O. Deoarece hardware-ul fizic este partajat de toate mașinile virtuale ce rulează pe platformă, trebuie să existe o formă de multiplexare a accesului la dispozitivele disponibile, aceasta realizându-se, în cazul Xen, în domeniul 0 Linux privilegiat. Pentru aceasta, accesul la spațiul de I/O, cum ar fi adresele porturilor PCI, trebuie să fie prins de software-ul de virtualizare și emulat. Pentru ca hypervizorul și mașina HVM să aibă aceeași imagine despre layout-ul device-urilor pe magistrale, sau pentru cunoașterea adreselor routinelor de întrerupere setate de kernelul HVM, un firmware (BIOS) virtual este adăugat de software-ul de virtualizare în spațiul de adrese al mașinii guest înainte de boot.



Acest firmware conține secvențele și serviciile de bootare ca orice alt BIOS, diferența fiind că device-urile pe care le pune la dispoziție mașinii guest sunt unele virtuale, cum sunt ele prezentate in device-model-ul mașinii care se rulează. De exemplu acesta

conține un bus PCI, ACPI, APIC virtual (folosit pentru a ști la ce adrese kernelul își instalează routinele de întrerupere) și altele. Pe adresele PCI pot exista dispozitive ca o placă video Cirrus Logic, sau Vesa, un disk IDE, o interfață Ethernet, un controller USB UHCI. Acesta din urmă va fi folosit pentru atașarea dispozitivului PnP UDDI.

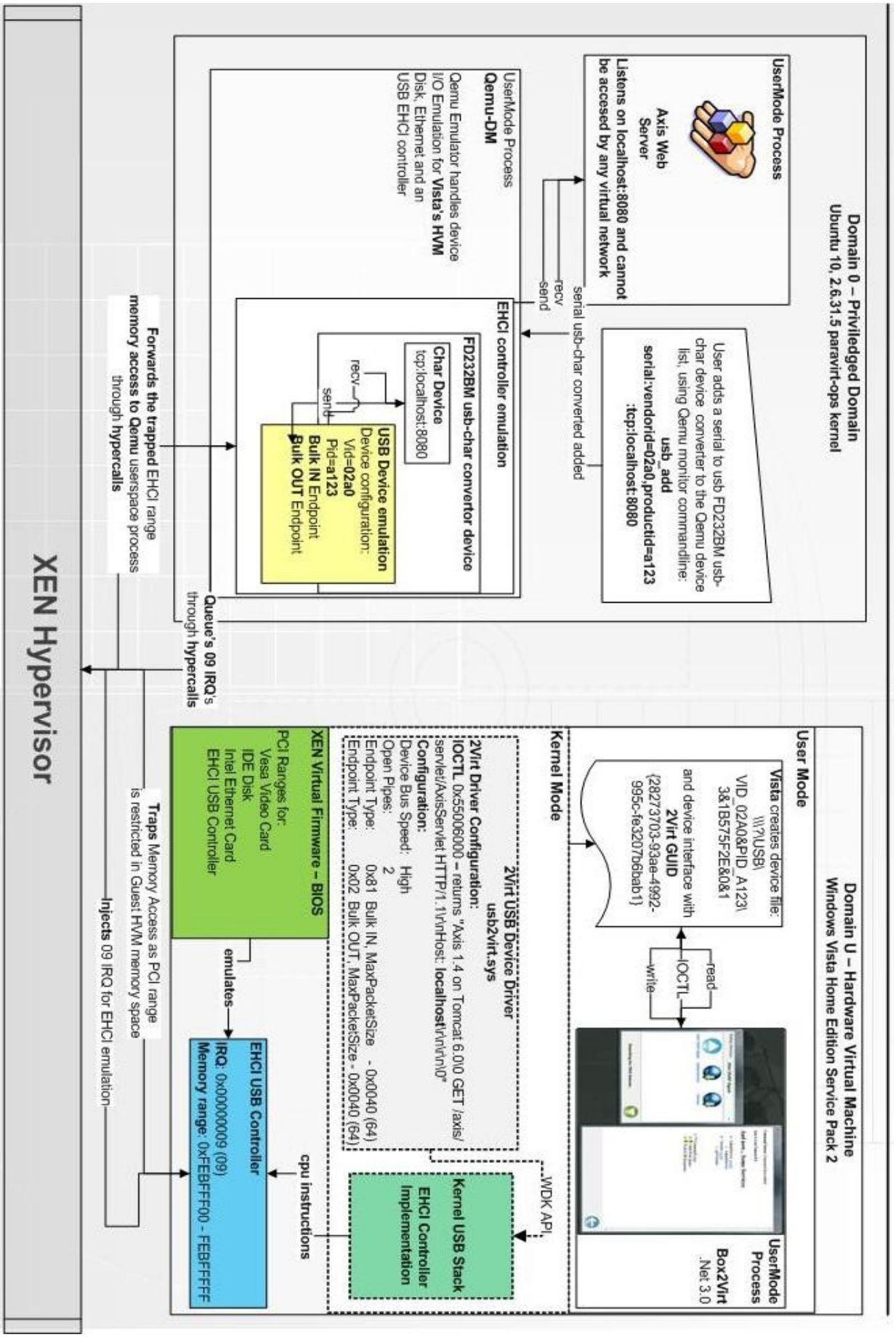
Pentru emularea accesului la dispozitivele I/O platforma Xen folosește emulatorul Qemu, rulând ca un process userspace în domeniul privilegiat Dom0 Linux. De exemplu să presupunem că codul kernelului HVM NT va dori să realizeze o operație de DMA corespunzătoare controllerului IDE. Cum kernelul știe la ce adrese PCI acesta se găsește, datorită BIOS-ului pus la dispoziție de Xen, va realiza operații de load/store pe aceste zone de adrese PCI. Cum hypervizorul a restricționat accesul la aceste adrese, tocmai pentru a emula dispozitivele I/O, acest access va fi identificat de Xen prin ridicarea hardware a unei excepții la aceste instrucțiuni de procesor. Hypervizorul va determina faptul că acest HVM are emularea dispozitivului IDE realizată de Qemu și va injecta acestuia acest access. Mai departe, Qemu, rulând ca process userspace in Dom0, va decodifica instrucțiunea curentă, și o va emula prin apelarea facilităților oferite de sistemul Linux pentru accesul la disk, practiv va efectua în Dom0 un apel de read pe un device file corespunzător partiției pe care o foloseste mașina virtuală. Acest apel de read va fi prelucrat de sistemul Linux Dom0 (device drivere reale din kernel + apel hardware) și se va întoarce în Qemu. Qemu apoi poate informa mașina guest că apelul s-a terminat, scriind, prin intermediul facilităților oferite de hypervisor datele întoarse de read în zona de memorie DMA din spațiul de adrese al mașinii HVM guest, pe care a vrut-o kernelul NT, și va injecta întreruperea de DMA guest-ului. Acesta din urmă va rula routina de tratare a apelului de DMA, care va putea citi datele așa cum ar fi făcut-o cu un dispozitiv IDE real.

6.3 Emularea dispozitivului USB PnP UDDI

Acest paragraf va arăta interacțiunea dintre componentele necesare emulării acestui dispozitiv într-o mașina HVM guest rulând Windows Vista, sub hypervisorul Xen. Ce este important de reținut, este că emularea dispozitivului este realizată doar prin software și nu necesită implementarea vreunui hardware, nici măcar existența vreunui controller USB pe mașina fizică.

Nu vom intra în detaliile setării environmentului și mașinii guest, acestea pot fi regăsite pe *www.2virt.com/blog*. Ce este important de reținut este că mașinii guest i s-a adăugat în fișierul de configurare, faptul că emularea dispozitivelor este realizată de emulatorul Qemu, și că are adăugat un controller UHCI la care se pot conecta ulterior dispozitve

6 Dispozitiv PnP UDDI Emulat



USB. Această specificație de controller UHCI este adăugată firmware-ului cu care este bootată mașina guest Windows Vista. După cum se poate observa emularea efectivă a controller-ului UHCI se realizează în procesul Qemu, aflat in spațiul de user al Dom0 Linux. Să enumerăm pașii necesari pentru adăugarea unui dispozitiv PnP UDDI:

1. Utilizatorul porneste în **domeniul privilegiat Dom0 Linux**, rulând sub Xen. Acesta are un server web **Apache Axis** ce ascultă doar pe interfața ***localhost*** și nu poate fi accesat de oricare altă rețea.
2. Se pornește mașina virtuală HVM Windows Vista, folosind fișierul de configurare.
3. Guestul Vista bootează, kernelul NT își configurează stiva USB pentru controller-ul UHCI găsit prin intermediul firmware-ului (știe zona mapată de controller și intreruperea asociată cu acesta)
4. Utilizatorul pornește în spațiul userspace al mașinii HVM aplicația *Box2Virt*, pune la dispoziția kernelului Vista driverul *usb2virt.sys* pentru dispozitivul PnP UDDI
5. În Dom0 Linux, utilizatorul în consola monitorului Qemu pentru mașina Vista va adăuga dispozitivul PnP UDDI, rulând comanda:
   usb_add serial:vendorid=02a0,productid=a123:tcp:localhost:8080
   Aceasta va informa emulatorul Qemu că se doreste adăugarea unui convertor FTDI FT232BM USB-serial la mașina virtuală. Acesta va avea caracteristicile unui dispozitiv PnP UDDI (vendorid/productid) și va converti mesajele USB primite de la device-ul emulat la un char device (seriala) dat de conexiunea tcp la serverul Axis. Practic acest convertor va expune un firmware de dispozitiv USB ce conține o configurație cu o interfața cu doua endpointuri Bulk de IN, OUT ce vor fi serializate către char device-ul respectiv.
6. Guestul HVM Vista va vedea un dispozitiv USB cu caracteristicile PnP UDDI și va încărca driverul *usb2virt.sys* în kernel. Acesta va expune aplicațiilor userspace un device file, ca și în exemplul PnP UDDI simulat, pe care acestea îl vor putea folosi.

Pentru a prezenta un flux de date între entitățile componete să presupunem că aplicația *Box2Virt* vrea să apeleze o metodă a unui servici web. Pachetul SOAP emis de aplicație va trece prin următoarele etape:

- Aplicația .Net Box2Virt userspace în HVM Vista face un apel de write cu pachetul SOAP pe device file-ul dispozitivului.
- Apelul va ajunge în kernelul Vista și va fi redirecționat către coada de write a driverului *usb2virt.sys.*

- Codul din driver procesează requestul de write și îl înaintează pe endpointul Bulk de OUT conform API-ului WDK de Windows NT.
- Stiva USB din kernel va prelucra pachetul USB, corespunzător endpointului Bulk OUT al dispozitivului, driverul pentru controllerul UHCI din kernel dorind să scrie acest pachet pe magistrala USB. Pentru aceasta el va face apeluri de store la anumite adrese PCI, corespunzătoare informațiilor citite din firmware-ul virtual despre controller-ului UHCI.
- Aceste apeluri vor fi prinse de hypervisor, în ring0, routina de acces invalid le va înainta la emulatorul Qemu, din spațiul userspace al Dom0 Linux.
- Acesta le va decodifica și va emula accesul atât din privința unui controller UHCI, cât și al unui firmware PnP UDDI. Astfel va scrie în zona de memorie a controllerului UHCI din mașina guest că a primit pachetul USB, și va înainta pachetul USB SOAP de pe endpointul Bulk la convertorul USB-serial aflat tot în emulator.
- Convertorul USB-serial, ce reprezintă disozitivul PnP UDDI emulat, va decodifica pachetul USB pentru un enpoint Bulk, și va înainta datele primite pe dispozitivul char, reprezentat de conexiunea TCP la serverul web Axis de pe localhost.
- Serverul web va răspunde tot pe socket, convertorul USB-serial din procesul Qemu va encapsula datele într-un pachet USB pentru un endpoint Bulk IN și va înainta pachetul pentru emulare controllerului UHCI din Qemu.
- Emulatorul de UHCI din Qemu va scrie datele pachetului USB în spațiul de adresă al mașinii guest Vista, cu ajutorul hypervisorului, și va injecta o întrerupere în acea mașină.
- Handlerul de întrerupere din Vista va apela driverul de UHCI, care va înainta pachetul USB driverului PnP UDDI, usb2virt.sys, care îl va pune în coada de read a device file-ului, pe care aplicația Box2Virt poate citi pentru a primi răspunsul apelului SOAP, tot în acelaș format.

Practic calea parcursă de un pachet SOAP, pentru o arhitectură x86_64 este:

Box2Virt [*userspace HVM Vista ring1*] read/write pe device file ↔ syscall-uri Driver **usb2virt.sys** [*kernel HVM Vista ring1*] ↔ driver UHCI [*kernel HVM Vista ring1*] ↔ **Hypervizor Xen** [*ring0*] trampOline ↔ **Qemu UHCI emulator** [*userspace Dom0 Linux ring1*] USB ↔ **Qemu PnP UDDI serial-usb convertor** [*userspace Dom0 Linux ring1*] ↔ **Apache Axis** [[*userspace Dom0 Linux ring1*]]

7 Concluzii

Dispozitivul PnP UDDI adresează una dintre problemele tehnologiei Web Service, aceea a necesității existenței unui registry comun pentru toate serviciile web, precum și securitatea în asamblu. Dacă, până acum, acest lucru era realizat prin tot felul de extensii de securitate la protocoalele utilizate, sau prin modalități standard de interogare, ca cele oferite de registrii UDDI, PnP UDDI realizează toate aceste lucruri prin migrarea acestor nivele de implementare într-un dispozitiv USB care le asigură într-un mod transparent din perspectiva unui consumator de servicii web. Platforma PnP UDDI a fost împartită în elementele necesare a fi implementate de un consumator de servicii web, anume o aplicație specifică fiecărui sistem care stie să comunice cu dispozitivul PnP UDDI la nivel de primitivelor userspace ale sistemului de operare, și cele necesare realizării de către un dezvoltator de framework: driver, firmware.

Practic principalele avantaje pe care tehnologia PnP UDDI le aduce sunt:

- reduce nivelul de cunoastere a arhitecturii serviciilor web a unui consumator prin necesitatea doar de a comunica cu un device.
- tehnologia USB conferă posibilitatea de Plug And Play a device-urilor, având posibilitatea de a adăuga sau înlătura în mod dinamic serviciile web de pe mașina curentă.
- Autentificarea și securizarea accesului la servicii web este realizată chiar de dispozitiv. Pentru transmiterea mesajelor SOAP corespunzătoare tehnologiei Web Service nu mai este necesar encapsularea acestora peste SSL, TCP/IP, putând fi trimise direct device-ului.
- Prin implementarea de enpointuri Bulk IN/OUT se asigură realizarea unei abstractizări de streaming între un consumator și un serviciu web, ca și în cazul conectării pe un char device, cum ar fi un socket tcp.
- Cele doua implementări de dispozitive PnP UDDI, simulat și emulat, prezintă aspectul practic al tehnologiei, prin realizarea firmware-ului dispozitivului doar prin elemente software, fără să fie nevoie de suport hardware. În ambele implementări doar firmware-ul a trebuit să fie regândit, aplicația .Net Box2Virt precum și device driverul usb2virt.sys rămânând neschimbate pentru același sistem de operare, Windows NT. Acest fapt pune în evidență ușurința cu care se poate migra de la o implementare de dispozitiv la alta, fără a fi nevoie să fie modificate celelalte componente ale arhitecturii PnP UDDI.

Bibliografia

[**1**] **Matei, Mihai**. *Tutoriale Xen*. 2009. http://www.2virt.com/blog/?cat=3

[**2**] Revista online **EE Herald** - *USB interface tutorial covering basic fundamentals.* http://www.eeherald.com/section/design-guide/esmod14.html

[**3**] **Windows Hardware Developer Central**. *Windows Driver Kit*. http://www.microsoft.com/whdc/devtools/wdk/default.mspx

[**4**] **Windows Hardware Developer Central**. *Device Simulation Framework*. http://www.microsoft.com/whdc/devtools/DSF.mspx

[**5**] **Xen Wiki**. *Documentație*. http://wiki.xensource.com/xenwiki/XenDocs

[**6**] **Qemu**. *Documentation*. http://wiki.qemu.org/Manual

[**7**] Dong, Yaozu; Li, Shaofan; Mallick, Asit; Nakajima, Jun; Tian, Kun; Xu, Xuefei; Yang, Fred; Yu, Wilfred**.** *Extending Xen* with Intel® VT*. http://www.intel.com/technology/itj/2006/v10i3/3-xen/4-extending-with-intel-vt.htm

[**8**] **Chisnall, David**. 2007. *The Definitive Guide to the Xen Hypervisor*.

[**9**] Matthews , Jeanna Neefe; Dow, Eli M.; Deshane, Todd; Hu, Wenjin; Bongio,Jeremy; Wilbur, Patrick F. 2008. *A Hands-on Guide to the Art of Virtualization*