

JAVA

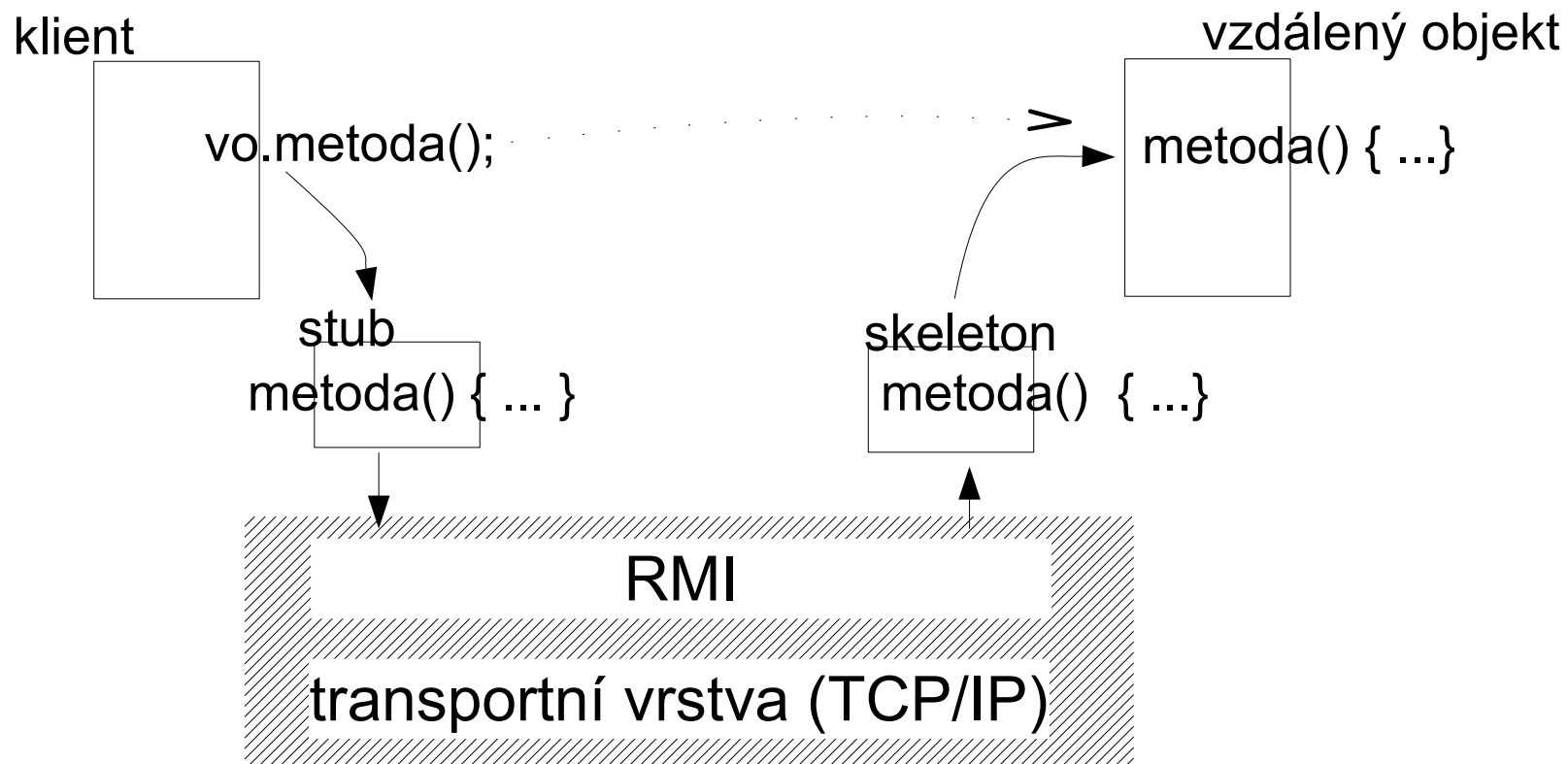
RMI

Přehled

- Remote Method Invocation
- používání vzdálených objektů
 - objekty v jiné VM (na stejném počítači i přes síť)
- jako by to byly lokální objekty (téměř)
 - pouze volání trvají déle

- `java.rmi` modul

Vzdálené volání obecně



Příklad: interface

1. interface pro vzdálený objekt
 - musí dědit od `java.rmi.Remote`
 - `java.rmi.RemoteException` u každé metody

```
import java.rmi.Remote;  
import java.rmi.RemoteException;  
  
public interface Hello extends Remote {  
    String sayHello() throws RemoteException;  
}
```

Příklad: implementace

2. implementace interfacu

```
public class HelloImpl extends UnicastRemoteObject
    implements Hello {

    public HelloImpl() throws RemoteException {}

    public String sayHello() throws RemoteException{
        return "Hello, world!";
    }
}
```

Příklad: vytvoření objektu

3. vytvořit objekt

4. zaregistrovat objekt

```
public class HelloImpl implements Hello
                                extends UnicastRemoteObject {
    ...

    public static void main(String args[]) {
        try {
            HelloImpl obj = new HelloImpl();
            Naming.rebind("Hello", obj);
        } catch (Exception e) {
            ...
        }
    }
}
```

Příklad: klient

```
public class HelloClient {  
  
    public static void main(String[] args) {  
        try {  
            Hello robj = (Hello) Naming.lookup("Hello");  
            String mesg = robj.sayHello();  
            System.out.println(mesg);  
        } catch (Exception e) {  
            .....  
        }  
    }  
}
```

5. získání reference na vzdálený objekt
6. používání objektu

Příklad: kompilace a spuštění

7. kompilace

- zcela normálně

8. spuštění

a) `rmiregistry`

b) `java -Djava.rmi.server.codebase=file:/.../ HelloImpl`

- `codebase` ~ cesta ke class souborům

c) `java HelloClient`

Příklad: implementace objektu

- jiný způsob implementace objektu
 - když nelze dědit od UnicastRemoteObject

```
public class HelloImpl implements Hello {
    ...
    public static void main(String args[]) {
        try {
            HelloImpl obj = new HelloImpl();
            Hello robj = (Hello)
                UnicastRemoteObject.exportObject(obj, 0);
            Naming.rebind("Hello", robj);
        } catch (Exception e) {
            ...
        }
    }
}
```

Stuby a skeletony

- generují se automaticky
- JDK 1.4
 - automaticky jen skeletony
 - stuby nutno vygenerovat ručně
 - **rmic** překladač
 - pustí se po **javac** na implementace Remote objektů
 - při spuštění servru se musí nastavit **codebase**
 - **-Djava.rmi.server.codebase=.....**
 - codebase odkazuje, kde jsou stuby
 - klient si je sám podle codebase stáhne
 - codebase je typicky file:, ftp://, http://
 - musí končit /
 - musí se nastavit security policy
 - **-Djava.security.policy=....**
 - musí být nastavený security manager
 - **System.setSecurityManager(new SecurityManager());**

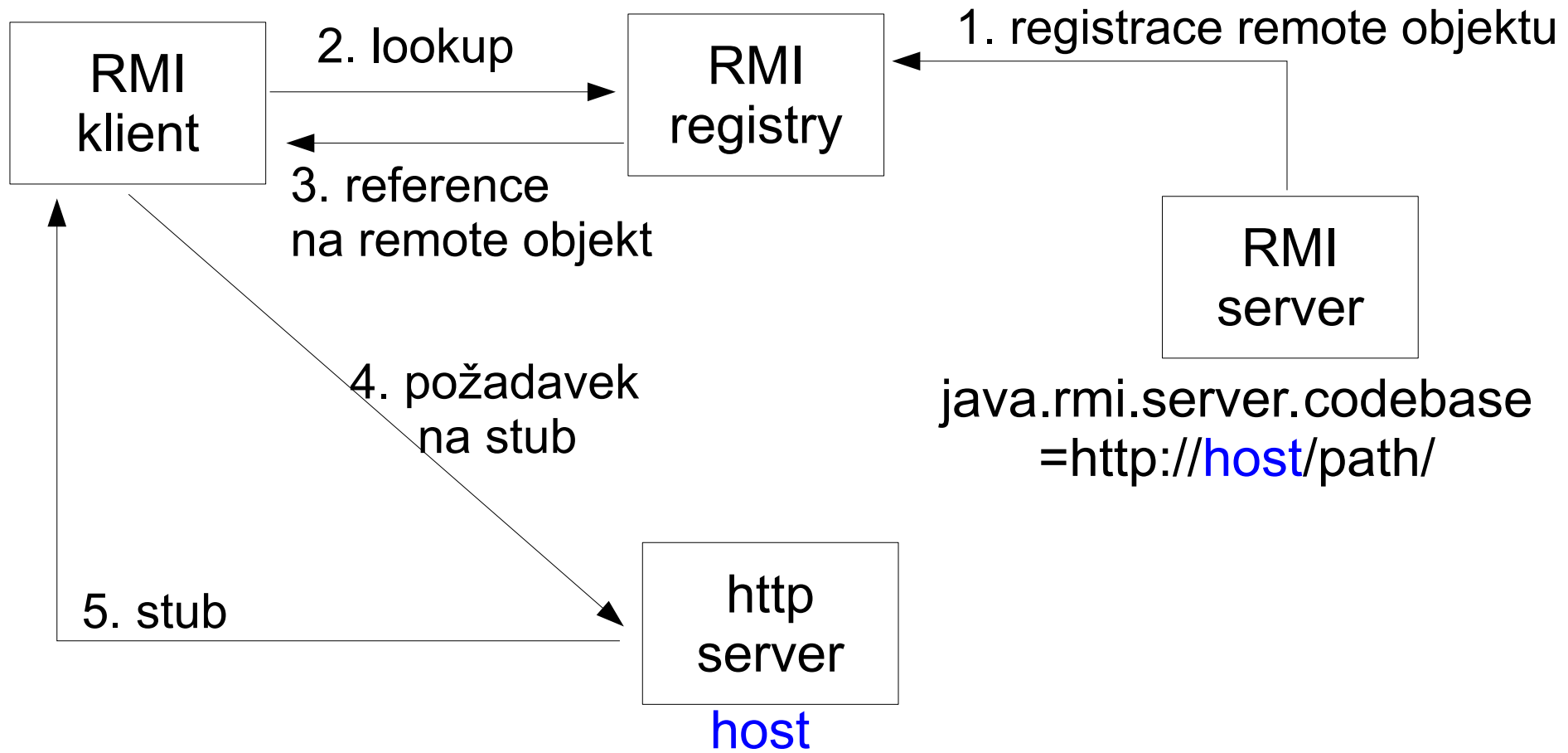
Stuby a skeletony

- JDK 1.4 (pokrač.)
 - **rmiregistry** nesmí mít nastavenou CLASSPATH, ve které jsou třídy, které se mají stahovat
- JDK 1.5
 - pokud jsou stuby k dispozici => negenerují se
 - vždy generovat stuby
 - property `java.rmi.server.ignoreStubClasses` nastavit na `true`
- JDK 1.1
 - negenerují se ani skeletony
 - **rmic** vygeneruje oboje

Stuby a skeletony a codebase

- POZOR
 - od JDK 7 Update 21 změna chování
 - property `java.rmi.server.useCodebaseOnly` implicitně nastavena na **true**
 - dříve byla na **false**
 - při nastavení na **true** je automatické stahování kódu povoleno pouze z **lokálně** nastavené codebase
 - tj. codebase se musí nastavit i pro rmiregistry nebo jí nastavit `useCodebaseOnly` na **false**

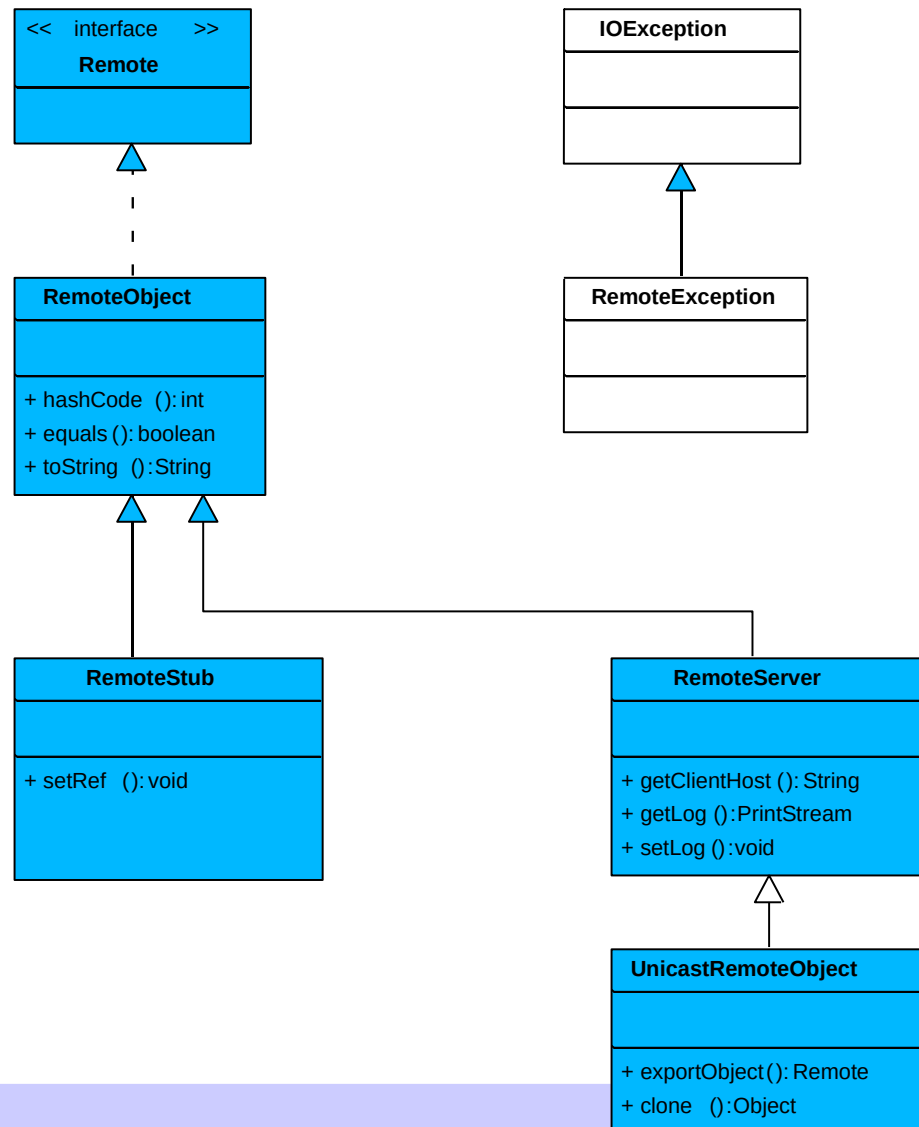
Stahování kódu



Distributed Object Model

- v čem se **neliší** od normálního Java Object Modelu
 - reference na remote objekty lze předávat jako parametry metod
 - remote objekty lze přetypovat na remote interface
 - lze používat **instanceof** na testování remote interfacu
- v čem **se liší**
 - klient vždy pracuje s remote objektem přes remote interface
 - tj. nelze přímo přistupovat k atributům
 - ne remote parametry se předávají hodnotou
 - některé metody z **java.lang.Object** jsou předefinovány
 - hashCode, equals
 - metody vyhazují RemoteException

Hierarchie tříd



Vlákna

- není žádná garance, jak se jednotlivá volání na remote objektu asociují s vlákny
- vzdálená volání na stejném remote objektu se mohou vykonávat paralelně

Naming

- získání iniciální reference na vzdálený objekt
- jednoduchá adresářová služba
 - reference na objekty registrovány s řetězci
- implementace – ***rmiregistry***
 - reference v registry je buď
 - do její explicitní odstranění nebo
 - do ukončení ***rmiregistry***
 - reference na objekt může být registry i když už objekt neběží
- dostupná také přes RMI
- jak získat referenci na ***rmiregistry***?
 - (problém vejce-slepice)
 - reference na ***rmiregistry*** se vytvoří z adresy počítače a čísla portu, kde ***rmiregistry*** běží

Naming

- lze používat najednou několik rmiregistry
 - na různých počítačích
 - na různých portech
- do rmiregistry mohou registrovat pouze procesy, které běží na stejném počítači
 - číst z registry lze odkudkoliv
 - jak obejít?
 - vytvořit remote objekt, který poběží na stejném počítači jako rmiregistry a bude registrovat objekty běžící jinde

rmiregistry

- program **rmiregistry**
 - jeden parametr – port
 - implicitně 1099
 - typické použití
 - unix
 - rmiregistry &
 - Win
 - start rmiregistry

Naming: přístup

- **java.rmi.Naming**
- pouze statické metody
 - bind, rebind, unbind
 - lookup
 - list
- první parametr je typu String – určuje jméno objektu, případně i registry
 - //host:port/jmeno
 - **host** a **port** nejsou povinné
 - implicitně localhost a 1099

Naming: přístup

- balík **java.rmi.registry**
 - třída `LocateRegistry`
 - získání reference na registry
 - vytvoření registry
 - interface `Registry`
 - stejné metody jako na třídě **Naming**
 - první parametr určuje pouze jméno objektu
- JNDI – Java Naming and Directory Interface
 - jednotný přístup k různým adresářovým službám
 - podporuje i trading (žluté stránky)
 - `java.naming` module
 - `javax.naming` package

Vlastní sokety

- lze určit, jaké sokety se budou používat pro RMI
- vytvořit vlastní *socket factory*
 - client socket factory
 - implementuje `RMIClientSocketFactory` a `Serializable`
 - server socket factory
 - implementuje `RMIWebSocketFactory`
- při vytváření remote objektu určit socket factory
- typické použití – šifrování
 - `javax.rmi.SSL`
 - `SSLRMIWebSocketFactory`
 - `SSLRMIClientSocketFactory`

Aktivace

- aktivace objektu, až když jsou potřeba
- **rmid**
 - activation daemon
 - "databáze" aktivačních záznamů
- objekty
 - dědit od třídy **java.rmi.activation.Activatable**
 - nebo pomocí ní exportovat objekt
 - navíc – zaregistrovat aktivační záznam do rmid
- při registraci aktivačního záznamu nutno **explicitně** specifikovat **oprávnění** (permissions)
 - AllPermissions nestačí

Aktivace

```
public interface MyRemoteInterface extends Remote {  
    ... }  

```

```
public class MyRemoteImpl extends Activatable  
    implements MyRemoteInterface {  
    public MyRemoteImpl(ActivationID id, MarshaledObject m)  
        throws RemoteException {  
        super(id, 0);  
    }  
    ....  
}
```

nebo

```
public class MyRemoteImpl implements MyRemoteInterface {  
    public MyRemoteImpl(ActivationID id, MarshaledObject m)  
        throws RemoteException {  
        Activatable.exportObject(this, id, 0);  
    }  
    ....  
}
```


Aktivace

- **registrace**
 - **vytvořit registrační záznam**
 - `public ActivationDesc(ActivationGroupID groupID, String className, String location, MarshalledObject data)`
 - **registrovat**
 - `static Remote Activatable.register(ActivationDesc desc)`
 - vrátí stub
 - **registrovat stub v rmiregistry**
 - běžným způsobem

Distribuovaný garbage collector

- garbage collector v distribuovaném prostředí
- počítání referencí
- "leases"
- objekt může být odstraněn, pokud na něj není žádná reference nebo vypršel "lease" a nebyl prodloužen
- VMID – identifikátor VM
 - jednoznačný (unique)
 - obsahuje ho "lease"

RMI-IIOP

- transportní protokol – JRMP
 - Java Remote Message Protocol
- lze použít IIOP
 - interoperabilita s CORBou
 - CORBA client – RMI server
- použití
 - balík javax.rmi (modul java.corba)
 - implementaci objektu dědit od PortableRemoteObject
 - ne od UnicastRemoteObject
 - používat **rmic** s parametrem **-iiop**
 - používat CORBA naming
 - javax.naming.... (JNDI)
 - místo **rmiregistry** používat **orbd**

*java.corba
odstraněno v Java 11*

JAVA

Jiná “RMI”

gRPC

- <https://grpc.io/>
- multiplatformní
 - Java, Python, C#, C++,...
- interfacy ~ protocol buffers

```
service Greeter {  
  rpc SayHello (HelloRequest) returns (HelloReply) {}  
}  
message HelloRequest {  
  string name = 1;  
}  
message HelloReply {  
  string message = 1;  
}
```

- protokol – HTTP + WebSockets

Další

- ...

JAVA

Security (Access control)

Přehled

- původně v Javě – „sandbox“ model
- postupně přidávány další služby
 - dále – omezení/povolení přístupu ke „zdrojům“

Security Manager

- `java.lang.SecurityManager`
 - před přístupem ke „zdroji“ se zkontroluje, zda je dostatečné oprávnění
 - implicitně není nainstalován
 - pro aplikace spouštěné „normálně“
 - pro spouštěné přes JNLP je implicitně nainstalován
- oprávnění – `java.security.Permission`
 - během načítání třídy, classloader přiřadí třídám oprávnění
- `java.security.Policy`
 - sada oprávnění
 - ve VM jen jedna
 - typicky se nastavuje přes textový soubor

Security Manager

- nastavení SM
 - buď v kódu
 - `System.setSecurityManager(sm)`
 - nebo z příkazové řádky
 - `-Djava.security.manager`
 - nastaví implicitní sm
 - `-Djava.security.manager=org.foo.SM`
- implicitní SM
 - implementován přes `java.security.AccessControler`
 - při přístupu ke „zdroji“ testuje všechny „elementy“ v posloupnosti volání
- lze si naimplementovat vlastní

Policy

- nastavení Policy
 - Djava.security.policy=file.policy
- formát

```
grant [SignedBy "signer_names"] [, CodeBase "URL"]
    [, Principal [principal_class_name] "principal_name"]
    [, Principal [principal_class_name] "principal_name"] ... {
    permission permission_class_name [ "target_name" ]
        [, "action"] [, SignedBy "signer_names"];
    permission ...
};
```

- příklad

```
grant codeBase "file:/home/sysadmin/" {
    permission java.io.FilePermission "/tmp/abc", "read";
};
```

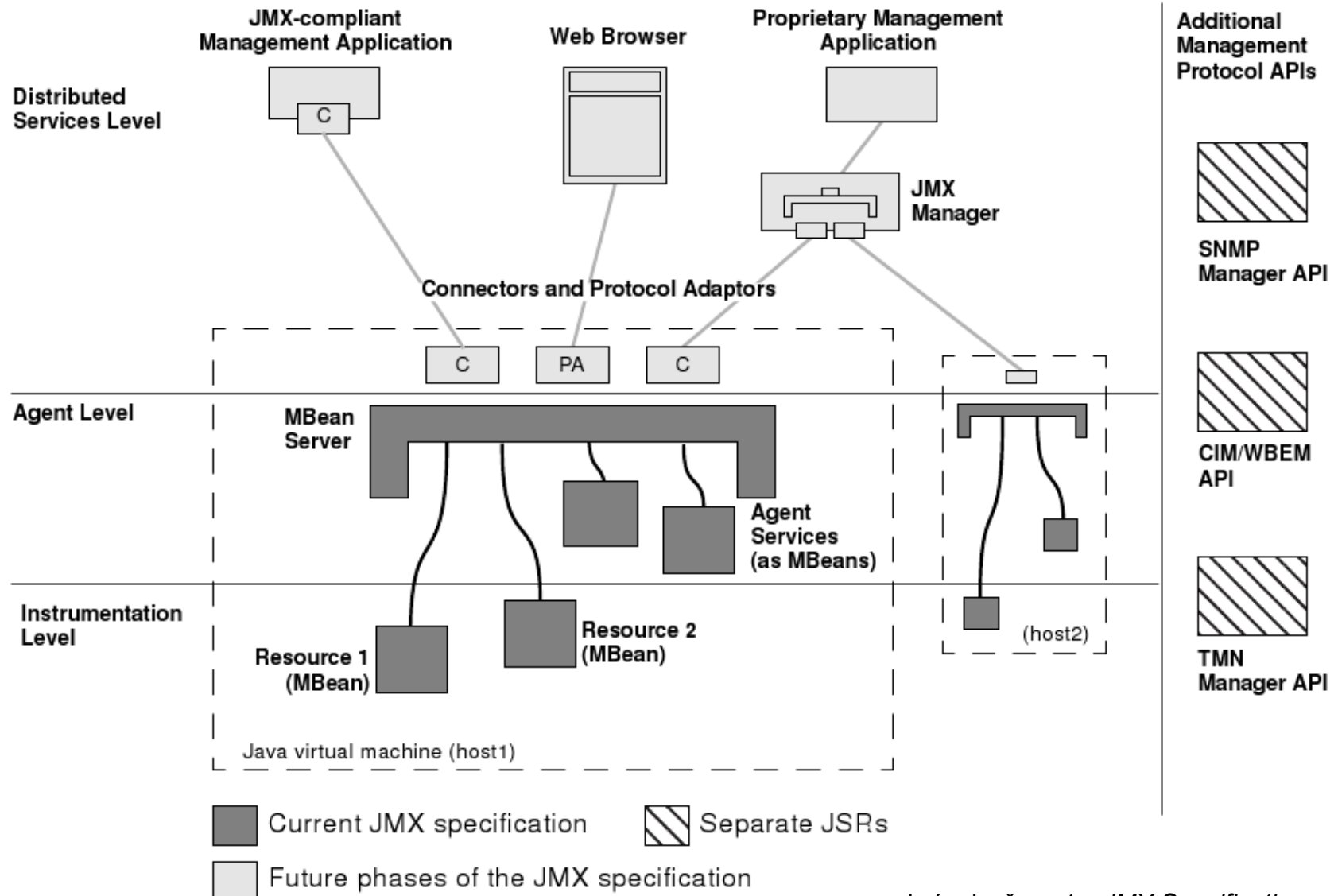
JAVA

Java Management Extensions JMX

Přehled

- součást JDK od 5
 - dříve externí sada jar archivů
- MBean = Managed Java Bean
 - beans určené pro správu něčeho (zařízení, aplikace, cokoliv)
 - poskytují rozhraní jako std. beans
 - vlastnosti (get a set metody)
 - normální metody
 - notifikace pomocí událostí
 - několik druhů
 - standard
 - dynamic
 - open
 - model
- (nejen) univerzální klient – JConsole

Architektura



obrázek převzat z *JMX Specification, version 1.4*

Druhy MBeans

- Standard
 - nejjednodušší druh
 - interface jsou všechny metody
- Dynamic
 - musejí implementovat určitý interface
 - mnohem flexibilnější
 - za runtime lze měnit
- Open
 - dynamic
 - ale používají pouze základní typy
 - nemusejí mít žádný speciální deskriptor
- Model
 - dynamic
 - plně konfigurovatelné za běhu

Standard MBean

- definované explicitně interfacem a implementací (třídou)
 - interface se musí jmenovat stejně jako třída plus přípona MBean
 - vše co je v MBean interfacu je poskytováno
 - metody, které třída implementuje, ale nejsou v interfacu, nejsou přes JMX viditelné
 - pravidla pro pojmenovávání „vlastností“ a metod jsou stejná jako u normálních bean
 - interface je za běhu programu získán introspekcí

Příklad std. MBean

```
package example.mbeans;

public interface MyClassMBean {
    public int getState();
    public void setState(int s);
    public void reset();
}
```

```
package example.mbeans;

public class MyClass
    implements MyClassMBean {
    private int state = 0;
    private String hidden = null;
    public int getState() {
        return(state);
    }
    public void setState(int s) {
        state = s;
    }
    public String getHidden() {
        return(hidden);
    }
    public void setHidden(String h) {
        hidden = h;
    }
    public void reset() {
        state = 0;
        hidden = null;
    }
}
```

Použití MBean

```
package example.mbeans;

import java.lang.management.*;
import javax.management.*;

public class Main {

    public static void main(String[] args) throws Exception {

        MBeanServer mbs = ManagementFactory.getPlatformMBeanServer();

        ObjectName name = new
            ObjectName("example.mbeans:type=MyClass");

        MyClass mbean = new MyClass();
        mbs.registerMBean(mbean, name);

        System.out.println("Waiting forever...");
        Thread.sleep(Long.MAX_VALUE);
    }
}
```

Dynamic MBean

- určené pro měnící se interface
- implementují **DynamicMBean** interface
 - interface je za běhu získán přes volání metod tohoto interfacu

```
interface DynamicMBean {
    MBeanInfo getMBeanInfo();
    Object getAttribute(String attribute);
    AttributeList getAttributes(String[] attributes);
    void setAttribute(Attribute attribute);
    AttributeList setAttributes(AttributeList
                                attributes);
    Object invoke(String actionName, Object[] params,
                  String[] signature);
}
```

Dynamic MBean

- MBeanInfo
 - popisuje MBean interface
 - při každé zavolání getMBeanInfo se výsledek může lišit
 - pak ale (většinou) nelze použít univerzální JMX klienty

Identifikace

- třída `ObjectName`
 - reprezentuje jméno mbean nebo vzor při vyhledávání
 - skládá se z domény a vlastností
 - doména
 - string
 - nesmí obsahovat dvojtečku a //
 - vlastnosti
 - jméno-hodnota páry
 - type – typ mbean
 - name – jméno
 - ...

JMX notifikace

- MBean může generovat události
 - např. při změně stavu
 - podobně jako obyčejné bean
- třída Notification
 - reprezentuje událost
 - potomek od `java.util.EventObject`
 - lze používat přímo
 - většinou ale přes potomky (opět podobně jako u obyčejných bean)
- interface NotificationListener
 - lze zaregistrovat pro poslouchání na události
- interface NotificationBroadcaster
 - MBean generující události musí implementovat tento interface
 - lépe implementovat NotificationEmitter
 - potomek od NotificationBroadcaster

JMX notifikace

- interface NotificationFilter
 - filtr na notifikace
 - listener si ho zaregistruje
- typy událostí
 - neplést s třídami
 - vlastnost události (String)
 - hierarchické
 - JMX.<neco> rezervované pro JMX
- vlastnosti události (třídy Notification)
 - typ
 - pořadové číslo
 - časová značka (kdy byla událost vygenerována)
 - zpráva
 - uživatelská data

JMX notifikace

- NotificationEmitter
 - void addNotificationListener(NotificationListener listener, NotificationFilter filter, Object handback)
 - handback
 - pomocný objekt
 - emitter ho nepoužívá ani nemění
 - je předáván při doručování událostí
 - void removeNotificationListener(NotificationListener listener)
 - void removeNotificationListener(NotificationListener listener, NotificationFilter filter, Object handback)
 - MBeanNotificationInfo[] getNotificationInfo()

JMX notifikace

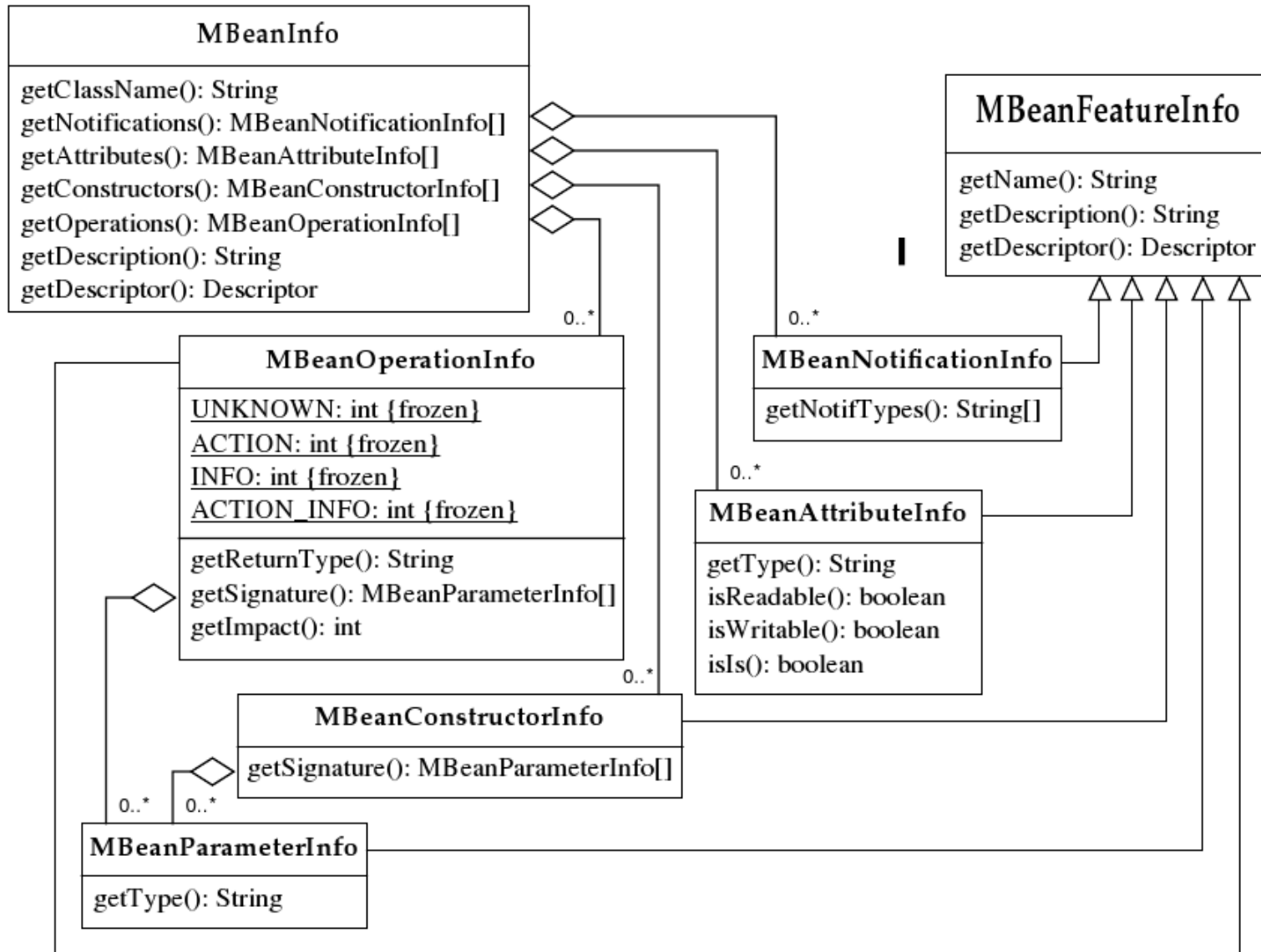
- NotificationListener
 - void handleNotification(Notification notification, Object handback)
- NotificationFilter
 - boolean isEnabled(Notification notification)
- podpora pro oznamování změn atributů
 - AttributeChangeNotification
 - AttributeChangeNotificationFilter
- třída NotificationBroadcasterSupport
 - předpřipravená implementace pro NotificationBroadcaster

Příklad notifikace

```
public class Hello extends
    NotificationBroadcasterSupport implements HelloMBean {
    ....
    public synchronized void setCacheSize(int size) {
        int oldSize = this.cacheSize;
        this.cacheSize = size;
        Notification n = new AttributeChangeNotification(this,
            sequenceNumber++, System.currentTimeMillis(), "CacheSize
            changed", "CacheSize", "int", oldSize, this.cacheSize);
        sendNotification(n);
    }

    public MBeanNotificationInfo[] getNotificationInfo() {
        String[] types = new String[] {
            AttributeChangeNotification.ATTRIBUTE_CHANGE
        };
        String name = AttributeChangeNotification.class.getName();
        String description = "An attribute of this MBean has changed";
        MBeanNotificationInfo info = new MBeanNotificationInfo(types,
                                                                    name, description);
        return new MBeanNotificationInfo[] {info};
    }
}
} Java, letní semestr 2020
```

MBeanInfo



obrázek převzat z JMX Specification, version 1.4

Open MBean

- dynamické MBean
- používají ale pouze omezenou množinu datových typů
 - basic data types
 - primitivní typy (wrapper typy)
 - String
 - BigDecimal, BigInteger
 - Date
 - javax.management.openbean.CompositeData
 - javax.management.openbean.CompositeTabular
 - pole těchto typů
- lze mít univerzální klienty
 - není potřeba překládat klienty při změně interfacu

Open MBean

- `javax.management.openbean.CompositeData`
 - interface
 - reprezentace složených typů
 - „struktury“
 - podobné jako hashovací tabulka
- `javax.management.openbean.CompositeTabular`
 - interface
 - reprezentace polí
- `OpenMBeanInfo`
 - rozšíření `MBeanInfo`
 - i další „Open“ deskriptory
 - `OpenMBeanOperationInfo`,...

Model MBean

- dynamické
- generické a plně konfigurovatelné za běhu programu
 - nevytváří se statický interface, ale dynamicky se přidávají elementy

Model MBean příklad

```
MBeanServer mbs = ...
```

```
HashMap map = new HashMap();
```

```
Method getMethod = HashMap.class.getMethod("get", new Class[]  
    {Object.class});
```

```
ModelMBeanOperationInfo getInfo =  
    new ModelMBeanOperationInfo("Get value for key", getMethod);
```

```
ModelMBeanInfo mmbi =  
    new ModelMBeanInfoSupport(HashMap.class.getName(),  
        "Map of keys and values",  
        null, // no attributes  
        null, // no constructors  
        new ModelMBeanOperationInfo[]{getInfo},  
        null); // no notifications
```

```
ModelMBean mmb = new RequiredModelMBean(mmbi);  
mmb.setManagedResource(map, "ObjectReference");
```

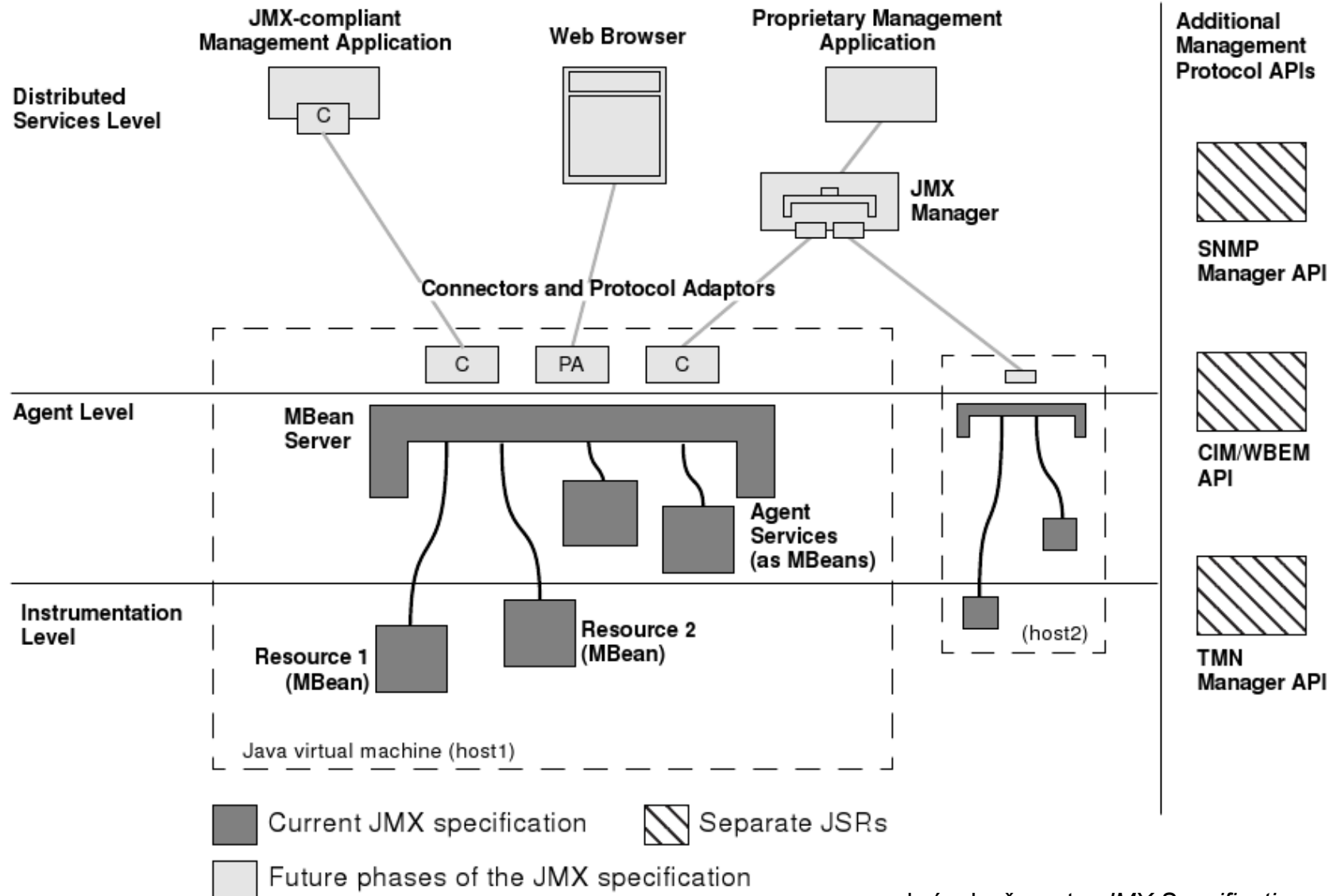
```
ObjectName mapName = new ObjectName(":type=Map,name=whatever");  
mbs.registerMBean(mmb, mapName);
```

```
mbs.invoke(mapName, "get", new Object[] {"key"}, new String[]  
    {Object.class.getName()});
```

MXBean

- nový druh MBean
 - od JDK 6 (částečně už v 5)
- standardní MBean
- navíc dodržují pravidla pro Open MBean
 - tj. používají jen omezenou množinu typů
- MXBean je třída implementující `<něco>MXBean` interface
 - třída se ale může jmenovat jakkoliv
- koncovku `MXBean` lze nahradit anotací `@MXBean`
 - naopak lze také pomocí `@MXBean(false)` nastavit, že daný interface není JMX interface i když má koncovku `MXBean`

Architektura (opak.)



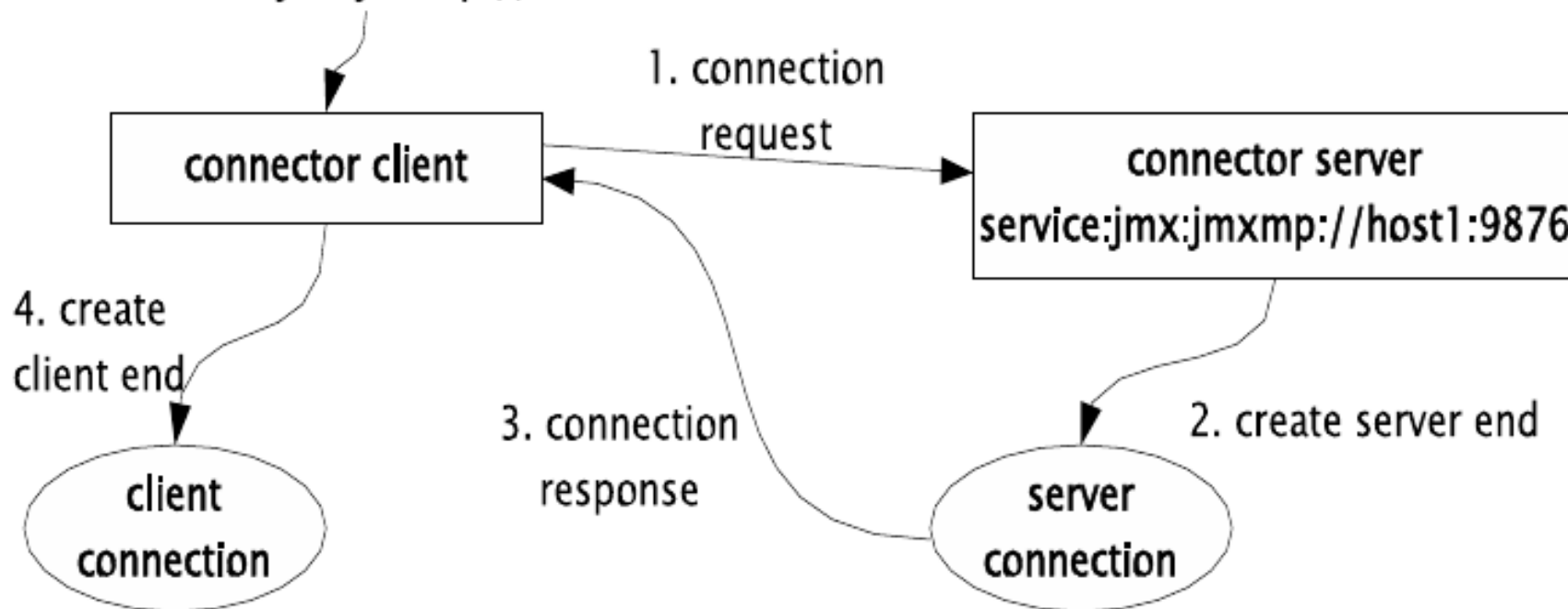
obrázek převzat z *JMX Specification, version 1.4*

JMX Remote

- vzdálený přístup k JMX
- pomocí *konektorů*
 - skládá se
 - connector client
 - connector server
- lze vytvořit konektory nad (v podstatě) čímkoliv
- specifikace definuje 2 konkrétní konektory
 - RMI
 - generic
 - JMX Messaging Protocol (JMXMP)
 - přímo nad TCP
 - implementace je volitelná

Vytváření spojení

connect "service:jmx:jmxmp://host1:9876"



obrázek převzat z *JMX Specification, version 1.4*

JMX Remote

- vytvoření MBean, registrace,... jsou stejné
- navíc vytvoření connector serveru

```
MBeanServer mbs = MBeanServerFactory.createMBeanServer();
```

```
...
```

```
JMXServiceURL url = new  
    JMXServiceURL("service:jmx:rmi:///jndi/rmi://localhost:  
    9999/server");
```

```
JMXConnectorServer cs =  
    JMXConnectorServerFactory.newJMXConnectorServer(url,  
    null, mbs);
```

```
cs.start();
```

```
...
```

```
cs.stop();
```

JMX Remote

- JMXServiceURL
 - url connector serveru
 - je závislé na typu konektoru
 - typická struktura
service:jmx:<protocol>:...
 - při vytváření vlastních konektorů není potřeba dodržovat strukturu
 - ale je to vhodné
- JMX specifikace určuje
 - bufrování zpráv
 - pravidla pro paralelní používání
 - jak se chovat při přenosových chybách
 - dynamické nahrávání tříd
 - bezpečnost
 - ...

JMX Remote – RMI konektor

- povinný konektor
 - každá implementace JMX ho musí obsahovat
- používá normální RMI
- lze určit, zda se má použít JRMP nebo IIOP
- použití RMI konektoru
 - `service:jmx:rmi://host:port`
 - konektor server vytvoří RMI server a vrátí na něj url ve formě `service:jmx:rmi://host:port/stub/XXXX`
 - XXXX je serializovaný RMI server
 - `service:jmx:iiop://host:port`
 - konektor server vytvoří CORBA objekt a vrátí url ve formě `service:jmx:iiop://host:port/ior/IOR:XXXX`
 - XXXX je std ior
 - `service:jmx:rmi://ignoredhost/jndi/rmi://myhost/myname`
 - vytvoří server a zaregistruje ho v naming service
 - místo rmi lze napsat iiop

JMX Remote – Generic konektor

- volitelný konektor
 - implementace ho nemusí obsahovat
- konfigurovatelný
 - cíl je snadné specifikování přenosových protokolů a wrapper objektů pro komunikaci
- definuje komunikaci přes zasílání zpráv
 - navazování spojení
 - zprávy
 - ...
- JMXMP konektor
 - konfigurace generic konektoru pro JMXMP

JMX Remote – klient

- navázání spojení se serverem

```
JMXServiceURL url = new
    JMXServiceURL("service:jmx:rmi:///jndi/rmi://localhost:9999/server");
JMXConnector jmxc = JMXConnectorFactory.connect(url,
    null);
```

```
MBeanServerConnection mbsc =
    jmxc.getMBeanServerConnection();
```

- používání

```
mbsc.queryMBeans(ObjectName name, QueryExp query)
mbsc.getAttribute(ObjectName name, String attrName)
mbsc.setAttribute(ObjectName, Attribute attr)
```


JMX Remote – klient

- vytváření proxy objektu pro přímý přístup
 - nutno znát interface
 - funkční pro standardní mbean

```
T JMX.newMBeanProxy (MBeanServerConnection connection,  
ObjectName objectName, Class<T> interfaceClass)
```

```
T JMX.newMBeanProxy (MBeanServerConnection connection,  
ObjectName objectName, Class<T> interfaceClass,  
boolean notificationBroadcaster)
```



Verze prezentace AJ09.cz.2020.01

Tato prezentace podléhá licenci [Creative Commons Uveďte autora-Neužívejte komerčně 4.0 Mezinárodní License](https://creativecommons.org/licenses/by-nc/4.0/).