## Diskutujte využití anycastu přímo v prostředí sítí IPv6. Proč je využíván? Využití krátce. Srovnejte s tím, jak je analogický problém (existuje-li) rešen v IPv4.

Metoda routování a adresace, packet je doručen nejbližšímu uzlu skupiny pro kterou je určen. Původně určen především pro nespojové UDP, ale s deterministickým výběrem cíle je možní i spojové TCP (ikdyž cílový uzel se může s časem měnit). Používá se pro load balancing a distribuované služby s mirrorovanými daty jako je například DNS, nebo AD controller. DNS je výborný příklad anycastingu, protože jeho root servery jsou rozdistribuovány po celém světě, anycast zabezpečuje spolehlivý load balancing. Anycast lze implementovat metodou 6to4. Anycast se používá také při content delivery (streamování, atd..) odolné proti DDoS útokům. Umí zajistit automatický failover, k tomu využívá metodiku heartbeat aby klienti neposílali requesty na nefunkční uzel.

## Popište základní komponenty CORBA modelu. Co je a jakou úlohu hraje ORB (Object Request Broker)?

Common Object Request Broker Architecture - jedná se o standard, který byl vyvinut konsorciem více než 700 společností - OMG). Poskytuje **framework nezávislý na platformě a jazyku** (programovací jazyk, hardware a software platformy, síť) pro psaní distribuovaných objektových aplikací. Jedná se **pouze o specifikaci**, nikoliv nástroj. Nicméně existují konkrétní implementace (SOM, DSOM). Komponentami jsou:

**Object Request Broker** (ORB): Jádro, součást předchůdce CORBA. Distribuovaná služba, implementující dotazy na vzdálené objekty. Má za úkol najít vzdálený objekt, vyřídí mu ho a poté odešle odpověď klientovi. Location transparency - nezáleží na umístění, požadavek je vyřízen stejně. Klient může být napsán v jiném jazyce než dotazovaný objekt. Klient i daný objekt jsou od ORBu odizolovány. IIOP - standart pro komunikaci mezi ORBy.

**Interface Definition Language** - Objekty jsou specifikovány rozhraním (kontrakty). Odizolovává objekty a ORB. Používá syntax C++ (s rozšířením). Definuje mapování mezi ostatními programovacími jazyky.

**Dynamic Invocation Interface** - Podpora CORBA pro dynamické volání. U statického volání je vše dáno při kompilaci. U dynamického volání není znám při kompilaci typ objektu.

**Dynamic Skeleton Interface** - Analogie klientských DII, ale pro servery.

**Interface Repository** - Komponenta, která za běhu dynamicky zjišťuje informace o typech IDL. Zajišťuje dynamickou rezoluci dotazu.

**Object Adapters** - Rozhraní mezi ORBem a serverovým procesem. Mapuje dotazy s danou instancí objektu. Poskytuje API pro implementaci objetků. Existují dva typy: Basic - BOA - základní, nepodporuje všechny features. Portable - POA - podporuje přenositelnost mezi ORBy, perzistenci…

## Diskutujte, jaké problémy působí distribuovaným systémům, pokud se omezí pouze na synchronní metody komunikace. Srovnejte RPC (Remote Procedure Call) a CORBA model. Jakým způsobem se tyto problémy řeší? A je toto řešení již bezproblémové?

Problémem na nejnižší úrovni je určitě synchronizace procesorů a času. Synchronizace reálného času, se dá řešit nejlépe UTC, propagovaným přes signál. Využít můžeme i build-in hodiny, ale zde můžeme narazit na problém s fyzikálními problémy s konstrukcí hodin (rozdílná struktura krystalů, rozdíly napětí, vlhkost vzduchu,…). Stejný problém řešíme při synchronizaci událostí a operací. Máme několik technik pro synchronizaci hodin jako Berkeleyho algoritmus, Network time protocol, časové známky, apod… V Berkeleyho algoritmu je v síti časový démon, který se dotazuje strojů na čas a provádí korekci. NTP propaguje čas stromovou architekturou, kde v kořeni je „přesný“ čas. Přesnost je určena úrovni ve stromu. Úroveň 1 má přesnější čas než úroveň 2.

## Jakým způsobem se v dynamických distribuovaných systémech řeší problém různého způsobu interní reprezentace objektu (v závislosti na hardware i operacích systémech)?

RPC tento problém řeší pomocí eXternal Data Representaiton (XDR), které obsahuje primitivní funkce (xdr_int(), xdr_enum(),…) a agregační funkce (xdr_array(), xdr_string(),…). Jeho použití je ale velmi komplikované. Nicméně kód klienta/serveru lze generovat pomocí dobře definovaných rozhraní. Server rozhraní implementuje a klient volá metody rozhraní. Existují kompilátory rpcgen a jrpcgen. RPC řeší jen volání procedur.

RMI je v podstatě RPC, ale operuje s objekty místo s procedurami. Server vytvoří objekt a zpřístupní ho - zaregistruje. Klientská aplikace může přistoupit k objektu a umožní mu volání jeho metod. Metoda na vzdáleném objektu má stejnou syntax jako na lokálním objektu. A stejně jako u RPC rozhraní poskytují stuby (u klientů) a skeletony (u serveru). Remote object reference je unikátní identifikátor pro distribuovaný systém. Je nutné jeho unikátnost zajistit. K objektům jde přistoupit konkurentně, musí se ale zajistit synchronizace (semafory, atd...). Transakce je stejná jako u RPC. Vzdálené objekty jsou předány referencí.

CORBA má objekty plně zapouzdřeny a jsou přístupné jen pomocí rozhraní (viz. DII, DSI, DR). Reference objektů CORBA je podobná ukazatelům. Definuje Interoperable Object Reference který je unikátní. IOR obsahuje fixní klíč obsahující: ID repozitáře, ID instance, porty a metadata ORBu serveru. O mapování objektů se starají OA.

## Jaký má význam protokol průzkumu okolí v protokolu IPv6? Které úkoly plní a jakým způsobem? Používá se někde v jeho kontextu anycast?

Neighbor Discovery Protokol - Má na starosti konfiguraci adres uzlů, odhalování nových uzlů, detekci duplicit, apod... Protokol definuje pět speciálních druhů packetů pro funkci v IPv6. Přivádí zlepšení oprati IPv4, například robustnost doručení paketů, v přítomnosti chybové linky/uzlu. Anycastové adresy jsou použity pro rezoluci pomocí NDP protokolu.

## Model RPC (Remote Procedure Call) má charakter klient-server. Jakým způsobem se klient se v tomto případě klient dozví, který server a jak má použít? Jak vůbec naváže klient spojení se serverem?

Na klientské straně je vzdálená procedura reprezentována **stub**em, který parametry zakóduje do zprávy, kterou odešle. Pak čeká na odpověď. Na straně serveru je to **skeleton**, který přeparsuje zprávu a zpracuje proceduru lokálně, výsledek opět zabalí do zprávy a odešle klientovi. Klienti potřebují znát číslo portu, na kterém služba běží, k tomu je použit **Portmapper**. Server zaregistruje ID programu, verze a portu do své lokální instance portmapperu. Klient zjistí ID portu odesláním dotazu na portmapper (port 111).

## Popište, jakou roli má GSH (Grid Service Handle) v kontextu Gridových služeb. Proč Gridové služby uvažují dva různé druhy „továren" (Factory versus factory)? Co je to soft state a k řešení jakého problému se používá?

Jedná se o URI gridové služby. Ale přez GSH nelze se sloužbou komunikovat. Místo toho se používá k rezoluci Grid Service Reference (GSR), která už obsahuje konkrétní endpoint, který lze pro komunikaci využít. Ve stejný čas může existovat více GSR. Pro dynamické vytváření instance gridové služby se používá factory pattern. Vyrezolvované GSR lze použít k přístupu k factory, která poskytne instanci služby, popřípadě vytvoří novou instanci. Specifikaci sémantiky obstarávají OGSI přes portType Factory, použití factory ale není striktně vyžadováno.

Oproti RPC, které pracuje se hard state (musíme se spolehnout na to že fungují - doufat), ale RMI pracuje se soft state (systémy nemusí fungovat). V soft state si musíme ověřovat, jestli služba funguje. Prostě aktualita stavu připojení je vázaná na časovou známku a degraduje, není-li obnovována. Pokud nedostane odpověď na obnovovací signál (heartbeat) nebo na počadavky, spojení se zruší. Opačný problém je blokování zdrojů na serveru, ty také timeoutují (narozdíl od RPC).

## Co je to difuzní přístup v kontextu problému rovnoměrného rozložení zátěže? Na jakém principu pracuje a pro které typy úloh je vhodný?

Jedná se o přístup distribuovaných front, což je self-scheduling pro distribuovanou paměť, kde každý uzel si drží frontu. V každém kroku difuzního přístupu se spočítá cena zbývajících tasků na každém procesoru. Procesory si pak vymění informace a balancují se. Na lokalitě **nesmí** záležet.

## K popisu čeho je a čeho není vhodné použít *Specification Description Language* (SDL)? Uveďte příklady a uveďte fázi vývoje nebo implementaci komunikačního protokolu.

SDL **není vhodné** použít na hi-level popis systému. Demonstraci dobrého (špatného) chování, specifikaci testů (všechno to na co se hodí MSC).

Nejprve použijeme MSC na popsání typických komunikačních scénářích. Pak použijeme opět MSC pro chybné sekvence. Plně toto specifikujeme pomocí HMSC, ale toto není nutné. Pak napíšeme distribuovanou specifikaci pomocí SDL. S takovým modelem již máme nástroje pro implementaci.

## Zrovnajte stručne princípy *symetrickej a asymetrickej* kryptografie. Ktorý systém používa dlhšie kľúče a prečo (uveďte bežnú dnes používanú veľkosť kľúčov u asymetrickej kryptografii)? Ktorý z týchto systémov je možné použiť pre digitálny podpis a na akom princípe digitálny podpis funguje?

Symetrická kryptografie používá jeden klíč pro šifrování i dešifrování zprávy. Má sice nízké výpočetní nároky a tím se hodí pro dlouhé zprávy, ale je nutné tento klíč předat. Asymetrická má dva klíče, vždy se dá s jedním zašifrovat a ten druhý dešifruje. Jeden z nich se typicky zvolí jako veřejný a zveřejní se, lze jím zprávu zašifrovat a pouze držitel druhého klíče ho umí rozšifrovat (jde to i naopak). Výhoda je, že nevzniká riziko odposlechnutí klíče při předávání. Spoléhá na složitost matematických operací pro dešifrování hrubou silou (ví se jak na to, ale trvá to - nevyplatí se). Problém je s man-in-the-middle útoky, protože se sdílí klíč veřejně. Navíc je výpočetně náročné.

Delší klíče mají typicky symetrické přístupy, protože čím delší klíč - tím bezpečnější, asymetrické spoléhají na matematiku.

Pro digitální podpis se používá asymetrická kryptografie. Zpráva je zašifrována privátním klíčem a může být dešifrována pouze a jenom veřejným klíčem dané osoby.

## Čo je to *Remote Procedure Call* (RPC) a ako sa používa? Ktoré nedostatky RPC rieši *Remote Method Invocation* (RMI) v jazyku Java? Ako sa Java RMI vysporiada s predaním ukazateľov vzdialeným objektom? Je vôbec možné ukazatele predávať?

Idea jak budou klienti komunikovat se vzdálenou službou. Procedury jsou zakryté rozhraními (stub u klienta a skeleton u serveru). Client zavolá proceduru na stubu, klasickým způsobem, stub zabalí dotaz do zprávy a pošle ho skeletonu, ten rozbalí zprávu a provede proceduru. Odpověď zabalí a odešle čekajícímu stubu. RPC funguje jen na procedurách, ale protože OOP přístup je žádoucí, přichází na řadu RMI, které již je objektové. Přistupuje se tedy ke vzdálenému objektu.

RMI pro „globání“ ukazatel, platný pro celý distribuovaný systém používá remote object refrence. To je identifikátor objektu, který obsahuje IP adresu, port, čas, číslo objektu, apod... Při konstrukci se musí zajistit jeho unikátnost.

## Čo rozumiete pod pojmom *Webové služby*? S akými základnými protokolmi sa v ich súvislosti môžeme stretnúť?

Jedná se o službu, která je poskytována nějakým zařízením jinému zařízení přes WWW. Předávání zpráv se typicky realizuje přes XML/JSON. Web service je softwarový systém, který je navržen jako interoperabilní interakce stroj-stroj přes síť. Má rozhraní (kontrakt) popsán v strojově

zpracovatelném formátu (WSDL). Interakce se službou probíhá přes zprávy použitím SOAP protokolu, ve spolupráci s ostatními webovými standarty.

WSDL - jazyk ve formátu XML popisující rozhraní webové služby. Současná verze je 2.0. Službu popisuje jako kolekci endpointů, které mají přiřazené bindingy a ty využívají jednotlivá rozhraní. Rozhraní mají své operace se vstupy a výstupy, ale také faulty - chyby.

SOAP - protokol pro výměnu správ založených na XML, sloužící pro komunikaci mezi webovými službami.

REST API - architektura s velmi dobrou škálovatelnosti. Určena pro manipulaci se vzdálenými zdroji. Je stateless.

## Popíšte princípy a rozdiely politík transferu úloh v kontexte vyrovnávania záťaže (*Senderinitiated a Receiver-initiated Policies*)

*Poznámka: Hranicí je zde myšlena konstanta, představující únosnou/přípustnou zátěž.*

Sender-initiated - **rozhoduje odesílatel**

1. Tansfer - Pokud by překročil hranici na dalším uzlu, pošle jinam.
2. Selection - nově příchozí procesy.
3. Location
    a. Náhodně - může generovat hodně transferů.
    b. Threshold - sekvenčně proskenuje n uzlů a task předá té, která nepřekračuje hranici.
    c. Shortest - pooluje nody paralelně, vybere tu nejméně zatíženou, která má místo a předá.

Reciever-initiated - **rozhoduje přijímatel**

1. Transfer - pokud překračuje hranici, pošle dál.
2. Selection - nově příchozí nebo částečně provedené procesy.
3. Location
    a. Threshold - sekvenčně proskenuje n uzlů a task si vezme od té, která překračuje hranici.
    b. Shortest - pooluje nody paralelně, a bere si od té která je nejvytíženější.

Symmetric - kombinace obou, hranicí mezi výběrem strategie je průměrná zátěž.

1. Nad hranicí -> sender
2. Pod hranicí -> receiver

## Diskutujte stručne problematiku migrácie – aké typy prístupov poznáte? Popíšte nejakú metódu migrácie bežiaceho kódu v distribuovaných systémoch (teda migrácie už spusteného programu z počítača A na počítač B).

Klíčové jsou preformace a flexibilita. Flexibilita znamená, že klienti nepotřebují speciální předinstalovaný software (download on demand) a konfigurace je dynamická.

1. Migrace procesu - poskytuje vysokou mobilitu. Je v něm obsažen kód, data, a zásobník
2. Migrace kódu - poskytuje nízkou mobilitu. Program vždy začne od začátku. Ale přenáší se toho méně.

V heterogenních systémech je běžně dostupná jen slabá mobilita a je nutno překompilovat kód. U virtuálních strojů (JVM) je toto jednoduší.

## Čo je cieľom systémov správy sietí (*Network Management*)? Čo je v tejto súvislosti zpravidla sledované?

Nejzákladnější cíl je rozhodně řešit incidenty. Tedy co udělat když se „něco“ pokazilo - co se pokazilo, proč se to pokazilo, jak to řešit. Se správným managementem můžeme zjistit, jestli se neměnila konfigurace, která mohla způsobit problém. Nebo jestli nedošlo k bezpečnostnímu incidentu. Nebo jestli nějaká komponenta nevykazovala delší dobu podezřelé chování. Tohoto nejsme schopni dosáhnout bez automatizovaných systémů a nástrojů, ale také musíme mít správně podchycenou metodiku správy.

Chceme sledovat: Síťové prvky, traffic, informace o routování, podezřelé chování (divný traffic, útoky, latence, problémy s disky,…).

## Princip AAA

- Autentizace - Zjišťujeme, kdo se přihlašuje, o koho se jedná. Řešíme, jestli je entita opravdu ta za kterou se vydává - hesla, pin, biometrika, certifikát, bezpečnostní otázka, apod… Ale také můžeme řešit kdo je tato entita. Navíc pod autentizaci řadíme i ověření odesílatele zprávy (digitální podpis).
- Autorizace - Zjišťujeme na co má autorizovaná entita oprávnění. Muže vstoupit? Může používat zdroj?
- Accounting - Vedeme si podrobné logy, kdo/co se přihlašoval, k čemu přistupoval, atd… Důležité při bezpečnostních incidentech. Je nutné si dát pozor na legislativu ohledně osobních informací. Také se zde řeší účtování za služby.

## SNMP jak funguje z ceho se sklada

Protokol (nástroj), který dovoluje vzdálenou komunikaci a lokální management jednotek sítě (servery, stanice, routery, switche). Jedná se i o framework architektury. Jak strukturovat, ukládat a vyměňovat si informace v protokolu. Využívá agentů.

Komponenty:

- Jádro SNMP - definuje formát zpráv. Zprávy také vytváří a zapouzdřuje. Dovoluje manažerovi pracovat se zprávami z agentů a agentovy spustit alarm při abnormální situaci.
- Structure of Management Information (SMI) - pravidla pro formátování spravovaných objektů (pojmenovávání, typy, kódování). Objekty pojmenovává (globálně podle hierarchické struktury) a kóduje.
- Management Information Bases (MIB) - Kolekce objektů (virtuální databáze), a jejich vztahy mezi sebou. Stará se také o hledání objektů.

## Autentizace chalange-response

Entita dokáže znalost tajemství, bez toho aniž by ho odhalila. Challenge má časově závislou známku. Často se jedná o aplikaci nějaká funkce.

## Typy Firewallů

Firewall je brána mezi privátní a nezabezpečenou sítí (typicky Internet). Jedná se o hardware/software/obojí. Umí implementovat bezpečnostní politiky, monitorovat. Neposkytuje zabezpečení proti vnitřním hrozbám (jedná se o „celnici“ nikoliv „vnitro“).

- **Packet filters** - Jsou stateless, rozhodují se za každý packet zvlášť, aplikují pravidla podle IP adres zdroje/cíle, portu zdroje/cíle, hlaviček. Jsou jednoduché a nízkonákladové. Nechrání proti útokům cíleným na aplikace.
- **Dynamic packet filters** - Podobně jako obyčejné packet filtry, ale jsou stavové a drží si kontext o komunikaci. Rozhodují se tedy na základě kontextu. Pořád mají nízkou výkonnostní zátěž.
- **Circuit gateways** - Pracují na session layer. Monitorují handshaking a rozhodují se legitimitě session. TCP spojení není end-to-end ale end-to-FW-to-end. Umožnuje skrýt informace o interní síti, vše je uzavřené, dokud to FW nepovolí. Podobnou funkci má NAT v IPv4, ale zde se nejedná o FW.
- **Application gateways** - Podobné circut gateways, ale jsou specifické pro aplikaci. Spojení probíhá přes proxy. Například SMTP proxy - virové scany, anti-spam, apod…

## Co je Distribuovaný System, znáte nějaký?

Základní charakteristiky:

- Atomicita - každý uzel je autonomní s vlastní pamětí.
- Heterogenita - entity (uzly) se mezi sebou liší (HW, SW, programovací jazyky, API,...)
- Concurrency - podpora souběžného přístupu.
- No global clock - neexistují globální hodiny, uzly se synchronizují přes zprávy.
- Independent failures - každá komponenta selže nezávisle na ostatních, zbytek systému pořád běží a vůbec nemusí o výpadku vědět.

Distribuovaný systém chceme dobře škálovat, zabezpečit sdílení zdrojů, zajistit rozšiřitelnost (HW, SW, použití jiného programovacího jazyka), podporovat souběžný přístup, bezpečnost. Také chceme

zabezpečit možné chyby například pomocí redundance HW/SW. Vypadne-li jeden uzel, další ho ihned nahradí. Dále chceme transparentnosti, přihlašování stejným způsobem, mít přístup ke zdroji bez ohledu na lokalitu, migrovat systém bez potíží, změnu konfigurace, škálování, odolnost k chybám.

Služby Google, Amazonu. Grid CERNu, ....

## Problem zhody, co to je a rozdiel medzi silnou s slabou

U Failure Detectors rozlišujeme několik strategií shody. Toto je z důvodu že každý proces si udržuje vlastní FD. Jejich hlášení mohou být nepřesné, neshodující se a mohou se měnit.

**Perfektní (silný) detektor**: Selhávající proces je už vždy podezřelý ostatním a zároveň žádný korektní proces nebude nikdy podezřelý. Těžké na implementaci.

**Slabý detektor**: Proces je podezřelý jen určitou jednotku času. Časem každý selhávající proces bude podezřelý všem ostatním. Jednodušší na implementaci.