MASARYKOVA UNIVERZITA
FAKULTA INFORMATIKY

Nástroje pro správu systému OpenSolaris

DIPLOMOVÁ PRÁCE

Rostislav Beneš

Brno, 2010

Prohlášení

Prohlašuji, že tato diplomová práce je mým původním autorským dílem, které jsem vypracoval samostatně. Všechny zdroje, prameny a literaturu, které jsem při vypracování používal nebo z nich čerpal, v práci řádně cituji s uvedením úplného odkazu na příslušný zdroj.

Vedoucí práce: Ing. Petr Adámek

Shrnutí

Ve své práci se věnuji administraci systému OpenSolaris. Obsahuje spoustu specifických vlastností a technologií, například RBAC, ZFS a zóny, ale ty je potřeba nejprve se naučit ovládat. Jejich správa se provádí prostřednictvím systémových příkazů, které však mohou začátečníky odradit. Tuto situaci se pokouší napravit projekt Visual Panels, jenž přináší nástroje pro správu s grafickým rozhraním určené hlavně nováčkům. Pokrývají však jen omezenou část schopností systému OpenSolaris. Zde leží cíl mé práce, vytvořit nové nástroje, zvýšit tak jejich celkový počet a zpřístupnit systém novým uživatelům.

Klíčová slova

Solaris, OpenSolaris, Visual Panels, Java, JMX, CACAO, RBAC, ZFS, Zóny, správa

Obsah

Kapitola 1

Úvod

Historie operačního systému OpenSolaris začíná v roce 1983, kdy firma Sun vydává svůj SunOS ve verzi 1.0 [14]. Jako základ byl použit BSD Unix. K přejmenování na Solaris dochází v roce 1992 v souvislosti s přechodem na UNIX System V Release 4 (SVR4). Místo SunOS verze 5.0 tak vyšel Solaris 2.0. V roce 2005 dospěl do stávající verze 10, která již do dnes obdržela osm aktualizací.

Ve stejné době se objevují plány na otevření zdrojových kódů pod názvem OpenSolaris [7]. Tak se tomu i stalo a to pod licencí CDDL (Common Development and Distribution License). Na jejich základě by měla vznikat nová vydání Solaris. Pod stejným vzniká i binární distribuce založená na zdrojových kódech OpenSolaris s kódovým jménem projekt Indiana [6]. Prvního vydání se dočkala v květnu roku 2008. Na OpenSolaris zdrojových kódech stavěl i starší Solaris Express (projekt Nevada), který zahrnoval navíc i programy dostupné jen v binární formě. V blízké době by však jeho vývoj měl být ukončen na úkor právě projektu Indiana. Proto se ve své práci věnuji výhradně této distribuci.

OpenSolaris získal ze svého předchůdce spoustu zajímavých vlastností, kvůli kterým by ho mohli chtít vyzkoušet noví uživatelé. Před jejich použitím je však potřeba přečíst si příslušnou dokumentaci k příkazům či konfiguračním souborům. Daleko lépe se schopnosti systému nováčkům prezentují nástroji s grafickým rozhraním (GUI), ve kterých snadno vidí, co vše systém poskytuje. V tom spatřuji jejich největší přínos a s tímhle cílem jsem svou práci započal.

Kapitola 2

Vlastnosti systému OpenSolaris

OpenSolaris v mnoha ohledech výrazně rozšiřuje původní SVR4. Týká se to hlavně správy uživatelů a služeb. Přichází také s novým souborovým systémem, i když ten už není výsadou jen systémů postavených na jádru Solaris. Ojedinělou vlastnost přinášejí samostatné běhové prostředí nazývané zóny.

2.1 Správa uživatelů

Celý koncept uživatelů a systému OpenSolaris vychází ze standardu POSIX. Každému uživateli se nastaví jméno, pod kterým se přihlašuje do systému, a unikátní číselný identifikátor (UID). Ještě obdrží svůj domovský adresář a zařadí se do jedné či více skupin. Skupiny vyjadřují podobnost vlastností či pravomocí mezi uživateli.

Z pohledu, jak kdo může se systémem zacházet, se uživatelé rozdělují do dvou kategorií. První jsou běžní uživatelé, kteří počítač používají jako svůj pracovní prostředek, ale z administrativních činností smí provádět pouze nezbytné minimum. Druhou méně početnější kategorii tvoří pouze jediný superuživatel vždy s identifikátorem 0 a pojmenovaný většinou *root*. Jemu je dovoleno se systémem provádět prakticky cokoliv bez omezení. A zde vzniká často problém. Spousta úkonů ve skutečnosti potřebuje pouze pár specifickým pravomocí. Přidělením všech, i zbytečných, ústí v důsledku v menší bezpečnost celého systému.

Tomuto problému čelí OpenSolaris pomocí Role-based Access Control (RBAC) [2]. Rozděluje pravomoci superuživatele mezi ostatní podle jejich potřeb. Přesněji zavádí pojem role a k ní soubor činností, které smí se systémem vykonávat. Na jejich základě se roli přidělí jen přesně ty pravomoci, které potřebuje. Vybraní uživatelé pak se systémem operují nikoliv pod svým účtem, ale pod účtem role. Tento přístupu dovoluje zavést přesnější a detailnější bezpečnostní politiky. Na druhou stranu je konfigurace RBAC náročnější a přináší větší prostor k chybám.

2.1.1 Implementace RBAC

Role je implementována téměř shodně jako uživatelský účet. Jediným rozdíl spočívá ve vlastnosti *type* s hodnotami *normal* a *role*. Pod účtem role však není povoleno se do systému přímo přihlásit, lze jen změnit identitu uživatele příkazy jako například `su`. Všechna přihlášení do role jsou samozřejmě zaznamenávána a tak se snadno nahlédne,

kdo pod určitou rolí v daný čas pracoval. Zde je vidět další posun od modelu se superuživatelem v případě, kdy jeho účet sdílí více fyzických osob a nešlo dohledat kdo přesně konkrétní akci provedl. Pro práci s nimi platí jedno důležité pravidlo. Uživatel vystupuje vždy maximálně pod jednou rolí. Nikdy se mu tak nepodaří spojit dohromady pravomoci z různých rolí.

Konkrétní pravomoci se mezi role rozdělují prostřednictvím autorizací, privilegií a profilů. Autorizace opravňuje uživatele vykonávat nějakou činnost. Systém OpenSolaris přímo definuje již desítky autorizací související s jeho správou. Například *solaris.admin.usermgr.pswd* povoluje měnit hesla uživatelů. K nim lze doplnit i své vlastní. Oproti nim privilegia jsou vlastnosti procesu a existuje jich přesně daný počet. Právě privilegia předávají některé pravomoci superuživatele dalším uživatelům. Profily pak spojují dohromady související autorizace a privilegia. Podporují hierarchii, tedy jeden profil může obsahovat a rozšiřovat jiné.

Autorizace se identifikuje svým jménem zapsaným jako doménové jméno v obráceném pořadí. Speciální funkci mají všechny autorizace končící na *grant*, jenž dovolují předat autorizace jinému uživateli. Při jejich výčtu lze požít hvězdičku (*) jako zástupný znak s významem cokoliv mimo *grant*. Odpovědnost za kontrolu autorizací je přenechána na aplikaci. Tento mechanismus implementují pouze systémy založené na Solaris, takže pokud aplikace nebyla pro ně přímo vyvíjena nebo aspoň upravená, nebude jej podporovat.

Autorize	**Popis**
solaris.admin.usermgr.read	Povolí číst údaje o uživatelích.
*solaris.admin.usermgr.**	Povolí kompletní manipulaci s účty uživatelů.
solaris.admin.usermgr.grant	Opravňuje přidělovat autorize příslušící do *solaris.admin.usermgr* jiným uživatelům.

Privilegia se v jistém smyslu podobají autorizacím, také dovolují a zakazují různé činnosti, ne však uživatelům, ale procesům. Také jejich význam leží o úroveň níže. Kontrolu tentokrát neprovádí aplikace, ale přímo jádro systému. Mezi privilegii se nachází jednak původní pravomoci superuživatele, jednak i pravomoci normální uživatelů, například *net_privaddr* pro přístup k privilegovaným síťovým portů nebo *proc_fork*, jenž povoluje vytvářet nové procesy. Lze tak uživatele, respektive jimi spouštěné programy ještě více omezit. Každému procesu náleží čtyři množiny privilegií a naplní se podle toho jak a kým je aplikace spouštěna.

- Děděné (Inheritable) - Obdrží jako výchozí všichni potomci procesu.
- Povolené (Permitted) - Maximální množina privilegií, jenž smí proces získat.
- Efektivní (Effective) - Množina právě nastavených privilegií.
- Omezující (Limit) - Maximální množina, jenž potomci mohou získat.

Autorizace a privilegia jsou však příliš podrobné na to, aby se spravovaly jednotlivě. Proto se seskupují do profilů a až ty se přidělují rolím. Navíc profily ještě dále rozšiřují možnosti nastavení. Ke každému profilu přísluší jeho jméno a pak podle potřeby autorizace, výchozí privilegia pro procesy a jména další profilů, z kterých přebírá a kombinuje vlastnosti. Další součást profilu tvoří seznam příkazů a definice pod jakým účtem se budou spouštět, případně která další privilegia navíc obdrží. Podle tohoto předpisu se řídí příkaz `pfexec`, jenž dostane jako argument již skutečnou aplikaci. Postupně prochází seznam všech profilů patřících uživateli a hledá v nich první definici pro požadovaný program. Tento seznam se obvykle ukončuje profilem *All*, jenž platí pro všechny aplikace, ale bez zvláštních nastavení tak, aby uživatel mohl pracovat s běžnými programy. Naopak profil *System Administrator* zahrnuje spoustu jiných profilů a opravňuje ke správě systému. Jedním z nich je například *Printer Management* věnující se administraci tiskáren.

2.2 ZFS

V ZFS spočívá jedna z klíčových a nejznámějších vlastností OpenSolaris a důvod, proč spousta lidí vůbec tento systém vyzkouší. Představuje nový pohled na problematiku souborových systémů. Po prvním vyzkoušení však může být část experimentátorů trochu zklamána, neboť ZFS není výkonnější (ale ani horší) něž jiné souborové systémy. Jeho přednosti se týkají odolnosti a jednoduché správy, navíc se spoustou možností.

ZFS opouští klasický model s rozdělením na disky, správce svazků, kdy se svazek chová stejně jako disk, a jednotlivé souborové systémy. Prakticky došlo ke sloučení posledních dvou částí. Ve skutečnosti obsahuje vlastní vnitřní dělení, v němž existuje paralela k původnímu modelu, jen s jiným rozhraním.

Nejprve je však nutné vysvětlit princip klasickému modelu a jeho slabiny. Disk představuje základní jednotku, kterou je sice možné ještě dělit na oddíly, ale to není podstatné. Správce svazků mapuje disky do svazků. Mapování může být i jedna k jedné, ale nejčastěji se spojuje kapacita disků, vytváří se zrcadlení nebo jiné RAID zapojení. Souborový systém nakonec dostane svazek, případně přímo disk (správce svazků se pak úplně vynechá) a je jen na něm, jak s ním naloží. Naneštěstí ve skutečnosti nemusí vůbec tušit, kam svoje data ukládá, ani odkud je následně čte, což limituje jeho schopnosti.

V případě, že správce svazků pouze spojuje více disků dohromady, hrozí ztráta části nebo všech dat při selhání jen jednoho disku podle toho, zda se disky zaplňují postupně nebo současně. Větší riziko sebou přináší i větší rychlost. Zrcadlení se naopak chlubí větší odolností při selhání jednotlivých zařízení a rychlím čtením, naproti tomu však s výrazně menší kapacitou. Funguje dobře do doby, než se poškodí část dat na jednom disku. Pak záleží čistě na náhodě, jestli svazek jako celek vrátí správná nebo špatná data. A když už souborový systém zjistí, že obdržel špatná data, nedokáže nijak získat jejich správnou verzi z kopií z nepoškozených zařízení.

Další dnes často používané zapojení je RAID-5, kde se vyhradí kapacita jednoho disku pro paritní informace. Ty se použijí až při opravě celého svazku, v případě když v něm nějaký disk selže. Při běžném čtení dat se však ke kontrole jejich správnosti nevyužívá, protože by se tak výrazně snížila výkonnost celého svazku. S tímto zapojením vzniká nový problém označovaný jako *write hole*. Nastane při náhlém vypnutí nebo restartu stroje, jenž způsobí přerušení zápisu na disky. Hrozí, že data na všech zařízeních svazku se ocitnou v nekonzistentním stavu. Obzvláště nepříjemná situace nastane, když se neuloží paritní informace. Špatnou paritu nelze detekovat za normálního běhu systému[1], ale projeví se až při případné opravě svazku.

Cíl návrhu ZFS právě spočíval v odstranění těchto a dalších nedostatků. Dosahuje něj pomocí dvou základních metod [9].

1. Data se nikdy nepřepisují na místě, ale všechny změny se zapisují do nových bloků a mění se pouze odkazy. ZFS tak dociluje kompletně transakční sémantiky a nevyžaduje žurnál k udržení konzistence. Projevuje se při tom často jeden zajímavý vedlejší efekt, kdy se všechny zápisy provedou sekvenčně, neboť se přidělily po sobě následující bloky.

2. Počítá a ukládá kontrolní součty všech dat a metadat. Brání se tak proti různým chybám, které mohou při čtení a zápisu dat nastat, například přečtení poškozených dat, špatný kabel k disku, chyba v diskovém řadiči nebo jeho ovladači.

Disky se v ZFS taky organizují do svazků, nazývaných však ZPool[2]. Podporuje běžně používaná či obdobná zapojení jako předchozí správce svazků, tedy spojování kapacity, zrcadlení a variaci na RAID-5 a RAID-6 pod jménem RAID-Z. Oproti původnímu správci ale nelze kombinovat zrcadlení a jiný RAID. RAID-Z existuje ve třech variantách, podle toho jakou paritu požadují a to od jednoduché a až po trojitou. Ke svazku lze přiřadit rezervní (spare) disk, jenž nalezne uplatnění tehdy, když některý z původních disků selže, do doby, než je špatný vyměněn. Rezervní disky mohou být sdíleny mezi více svazky. Ke zvýšení výkonu slouží Intent Log[3] a doplňková přídavná paměť. Intent Log uspokojuje požadavek standardu POSIX na synchronní transakce, používané například databázemi, kdy systémové volání vrací, až když jsou data na disku. Obvykle se pro Intent Log vyhradí část přímo ve svazku. Avšak vyhrazením speciálního zařízení, například NVRAM, lze dosáhnout vyššího výkonu. Přídavná vyrovnávací paměť pomáhá v případě, kdy převažuje čtení velkého množství dat, jenž se již najednou nevejdou do operační paměti.

Na každém disku jsou uloženy čtyři kopie záznamu, obsahující základní informace o svazku, ke kterému disk patří [4]. Dva se nacházejí na začátku a dva na konci disku. Kvůli pevnému umístnění se jedná o jedinou výjimku, kdy se data přímo přepisují a porušuje se tak sémantika kopírování při zápisu. Mechanizmus aktualizací těchto záznamů však stále splňuje podmínky transakčního zpracování.

---

1. Parita se nikdy nepočítá ze všech částí, ale jen se aktualizuje podle změn.
2. Běžně se používá tvar ZPool, ale snažil jsem se vyhnout jeho skloňování v textu.
3. Doslovný překlad zní úmyslný či vynucený záznam. Nenalezl jsem však vhodnou českou alternativu.

Místo ve svazku přiděluje vrstva Storage Pool Allocator (SPA) ve formě bloků. Data Managment Unit (DMU) následně organizuje bloky do objektů. Právě objekty představují základní prvek splňující transakční sémantiku za pomoci kopírování při zápisu. Tedy jakákoliv provedená operace s daty se buď celá zaznamená, nebo se data vůbec nezmění. Zde se dobře demonstruje rozdíl mezi klasickým přístupem a ZFS. Zatímco v klasickém modelu si souborový systém alokuje bloky, v případě ZFS jejich funkci zastávají objekty.

Kombinace SPA a DMU dovoluje další, v klasickém přístupu nemožné akce. Například automatické opravování poškozených data z kopií v případě zrcadlení. Nejprve špatná data rozpozná za pomoci kontrolního součtu. Jelikož ví, že někde existuje jejich kopie, zkusí ji přečíst. Pokud je kopie v pořádku, vrátí ji a zároveň se pokusí opravit chybná data. U ZFS není redundance dat vázaná pouze na zrcadlení či RAID-Z. I při jednoduchém spojení více disků dohromady, lze vynutit vytváření kopií dat na různých zařízeních, celková kapacita se tím ale přirozeně sníží. I v tomto případě však funguje automatická oprava dat.

Zbývá už jen poslední krok, tedy z objektů vytvořit souborový systém. Tenhle úkol plní ZFS POSIX Layer (ZPL). Ta organizuje soubory do adresářů, ukládá jejich obsah, přístupová práva a informace o vlastnících. V jednom svazku lze vytvořit prakticky neomezený počet souborových systémů a založení dalšího nového proběhne téměř okamžitě. V ZFS totiž souborový systém připomíná spíše chytřejší adresář s několika novými speciálními vlastnostmi. Všechny souborové systémy sdílí celou kapacitu svazku. To lehce mate při zobrazení volného místa jednotlivých souborových systémů, neboť každý ukazuje volnou kapacitu právě celého svazku. Prakticky to nijak nevadí ale pokud přece, lze správným nastavení kvót vyhradit pro každý souborový systém přesnou velikost.

Kopírování při zápisu nabízí ještě jednu zajímavou vlastnost. Původní data totiž zůstávají zachována, jen k nim nevede žádný dosažitelný odkaz. Přímo se tak nabízí možnost si tento odkaz někam schovat a kdykoliv se k němu vrátit a tím i k původním datům. ZFS dovoluje pořizovat snímky souborového systému, ve kterých se uchovávají aktuální informace a obsah všech souborů a adresářů. Snímky tak slouží jako zálohy ve smyslu, že v nich lze najít starší verze nebo smazané soubory, případně obnovit celý souborový systém do původního stavu. Taky ale usnadňují celkové zálohování na jiná zařízení, neboť umí kamkoliv uložit a opět nahrát buď celý snímek nebo jen rozdíl mezi snímky. Řeší i zálohování běžících počítačů, protože snímek zachycuje celý souborový systém v jednom okamžiku a po jeho pořízení ho nelze už jinak změnit. Není nutné pak po dobu zálohování omezit zápisy na disk nebo přenechat starost o zálohy na konkrétních aplikacích.

Existují situace, kdyby se však hodila možnost nějak snímky měnit. Tuto funkci plní klony. Klon vznikne jako nový souborový systém, ovšem s vazbou na snímek, ze kterého pochází. Klon může být dokonce povýšen a zbavit se závislosti na původním souborovém systému, ten ale naopak klesne a stane se klonem. Tohoto se v systému OpenSolaris využívá při správě zaváděcích prostředí (Boot Environments). Při kompletní aktualizaci se vytvoří nový klon souborového systému, do nějž se zapíšou změny, a

nastaví se jako výchozí. Po restartu pak systém naběhne do aktualizovaného prostředí. Pokud se však start nezdaří, neaktualizovaná, ale funkční verze systému je stále k dispozici.

ZPL nepředstavuje jedinou volbu, jak s objekty z DMU pracovat. Vedle ní stojí ještě vrstva ZFS Volume Emulator (ZVE). Ta simuluje bloková zařízení a používá například pro odkládací prostor (swap) nebo pro aplikace, které vyžadují přímí přístup k disku, typicky databáze. Pro databáze však poskytuje ZFS ještě jedno řešení a to postavit své úložiště přímo nad DMU pomocí objektů.

2.3 Správa služeb

OpenSolaris pečuje o své služby daleko více než jiné UNIX systémy. Výrazně upravuje koncept inicializačních skriptů, úrovní běhu a jednotlivých konfiguračních souborů. Základní jednotku nepředstavuje služba, ale její instance. Od jedné služby může zároveň běžet několik instancí, buď se stejným nebo odlišným nastavením. Služby nejsou jen aplikace typu webový server, ale i například síťové rozhraní, informace o konfiguraci jádra nebo milníky. O vše se stará technologie Service Management Facility (SMF), jenž oproti jiným UNIX systémům přidává tyto vlastnosti:

- Namísto spouštění služeb v daném pořadí se při startu systému využívá definice závislostí. Část služeb se tak spouští současně a zvlášť na strojích podporujících paralelní zpracování dochází ke zrychlení celé zaváděcí procedury.
- Hlídá a automatický restartuje služby, které selžou. Současně s nimi restartuje i služby, které na nich závisí.
- Poskytuje prostor a funkce pro čtení a ukládání konfigurace služeb a jejich instancí. V ideální případě se nastavení sdílí i mezi službami podobného typu, což zjednoduší výměnu konkrétní implementace, například webového nebo poštovního serveru. Bohužel, každá služba se musí patřičně upravit, než je schopná takto spolupracovat. Taky tato forma ukládání nastavení není vhodná pro databázově orientované typy dat jako seznamy uživatelů.
- Usnadňuje zálohování, obnovování či rušení změn konfigurace. Automaticky vytváří snímky nastavení.
- Všechny standardní a chybové výstupy, jenž služby produkují, shromažďuje do společného adresáře `/var/svc/log`. Když služba nepracuje správně, snadněji se hledá příčina.
- Funkci úrovní běhu přebírají speciální služby, milníky. Každá služba by měla záviset minimálně na některém z milníků, když už ne na jiných službách.

Inicializační skripty zůstávají podporovány, ale nemohou využít žádné vlastnosti SMF. Tedy i služby nekompatibilní s SMF pracují pod OpenSolaris v pořádku.

Zvláštní službu představuje *inetd* démon, který spouští na požádání vybrané síťové služby. Jeho nastavení bylo integrováno přímo jako součást SMF a ignoruje staré konfigurační soubory.

2.4 Zóny

Zóny představují způsob, jakým od sebe oddělit běžící aplikace, aby o sobě navzájem nevěděly [11]. Poskytují prostředí s virtualizovanými systémovými službami. Nepředstavuje tak bezpečnostní riziko, přidělit v nich procesům větší pravomoci. Nikdy však neuvidí procesy mimo svoji zónu ani nedostanou přístup přímo k hardwaru počítače s výjimkou síťových karet. Využívají se především jednak jako alternativa k příkazu `chroot`, a jednak ke sdílení počítače, kdy každý uživatel či skupina obdrží vlastní instalaci systému OpenSolaris. Nahrazuje tak plnohodnotnou virtualizaci, ale o proti ní vyžaduje méně prostředků, omezuje se však pouze na OpenSolaris.

Zóny se rozdělí na globální a ostatní. Globální má jako jediná přístup k hardwaru a smí jej konfigurovat. Vidí též všechny procesy i z ostatních zón. Jedině z globální zóny se vytváří, upravují a spouští ostatní zóny. Systém vždy startuje v globální zóně. Naopak ostatní, neglobální právě reprezentují samostatná prostředí.

K zónám se přistupuje přes konzoli nebo přes síťové rozhraní. Konzole slouží převážně při prvotním nastavením nebo když jiné způsoby nefungují. Jinak se doporučuje k přihlášení do zóny používat protokol SSH.

Ke každé zóně lze přiřadit libovolný počet síťových rozhraní. Celé zóna pak pracuje v jednou ze dvou módů: sdíleném či exklusivním, jenž ovlivňuje, co vše procesy v zóně smí provádět se sítí. Ve sdíleném módu se u všech přidělených rozhraní nastavuje pevná IP adresa a volitelně i adresa výchozího směrovače. IP vrstva se sdílí mezi globální a ostatními zónami a smí posílat pouze TCP, UDP a ICMP zprávy, a to pouze s vlastní zdrojovou adresou. Nemá vůbec přístup k L2 síťové vrstvě.

V exklusivním módu obdrží zóna síťové rozhraní výhradně pro sebe a nedělí se o něj s žádnou jinou zónou, ani s globální. Spravuje si svoji vlastní IP vrstvu, nezávislou na IP vrstvě globální zóny. Oproti sdílení nabízí tyto vlastnosti:

- DHCP a IPv6 autokonfigurace
- IP Filter i s překladem adres.
- IP Network Multipathing (IPMP)
- IP routování
- IPsec

Exklusivní mód představuje ale větší bezpečnostní riziko. Případný útočník získá přístup k síťovému rozhraní na stejné úrovní jako v globální zóně.

OpenSolaris obsahuje jiného správce balíčků než Solaris, což se odráží v konfiguraci zón. Solaris totiž rozlišuje mezi kompletně samostatných prostředím a zónou sdílející

některé adresáře s globální. U sdílené se ukládají pouze rozdíly, jako konfigurace zóny nebo navíc instalované balíčky. V OpenSolaris je k dispozici pouze varianta se samostatným prostředím. Uvažuje se však, že by zóny sdílely aspoň dočasné úložiště pro balíčky, aby je nestahovaly opakovaně. Zrychlila by se tak hlavně instalace zón.

Kapitola 3

Dostupné nástroje

Zmíněné vlastnosti představují pouze jednu část. Aby jich uživatelé reálně vyžívali, musí se snadno a přehledně spravovat. O tento úkol se starají různé systémové příkazy, ruční editace konfiguračních souborů se téměř vůbec nevyskytuje. Samozřejmě je nutné si nejprve přečíst jejich dokumentaci, ale pro správce zvyklého listovat manuálovými stránkami by to neměl být vážnější problém.

3.1 Správa uživatelů

OpenSolaris podporuje tři způsoby, kde uchovávat informace o uživatelských účtech.

- Lokální databáze v souborech v adresáři `/etc` a `/etc/security`.
- NIS (Network Information Service) - Sychronizuje konfigurační soubory mezi počítači.
- LDAP - Adresář k uchovávání informací o různých objektech. Typickým použitím je právě shromažďování uživatelských účtů.

Věnovat se budu pouze lokální databázi, neboť se na ní dobře demonstruje v jakých záznamech se konkrétní vlastnosti nacházejí. Veškeré údaje se ukládají do těchto textových souborů:

- `/etc/passwd`
- `/etc/shadow`
- `/etc/group`
- `/etc/user_attr`
- `/etc/security/auth_attr`
- `/etc/security/prof_attr`
- `/etc/security/exec_attr`

První tři jmenované definuje přímo norma POSIX. Všechny používají stejnou syntaxi, kdy každý řádek obsahuje jeden záznam. Jednotlivé údaje se oddělují od sebe znakem `:` (dvojtečka) a jejich počet se nikdy nemění. Pokud vlastnost představuje seznam hodnot, oddělují se od sebe znakem `,` (čárka). U některých záznamů se definuje vlastnost obsahující seznam atributů. Atribut tvoří dvojice jméno a hodnota ve tvaru jméno=hodnota. Uchovávají se tak nepovinné údaje.

3.1.1 `/etc/passwd`

Obsahuje základní informace o uživatelském účtu.

- Jméno uživatele.
- Již nepoužívaný haš hesla, nahrazený znakem x.
- Uživatelovo ID (UID).
- ID primární skupiny.
- Další informace o uživateli, někdy je označováno jako celé, skutečné jméno uživatele.
- Domovský adresář.
- Výchozí interpretr příkazů (shell), spuštěný po přihlášení.

3.1.2 `/etc/shadow`

Doplňuje k uživatelskému účtu vlastnosti související se správou hesel. Záznamy by měly být uvedeny ve stejném pořadí jako v `/etc/passwd`.

- Jméno uživatele.
- Haš hesla, kromě haše může obsahovat ještě speciální řetězce:

  NP - Značí nepřiřazené žádné heslo. K účtu se nelze přihlásit.

  LK - Představuje zamknutý účet. Po něm následuje původní heslo

  Žádný - Prázdné heslo
- Poslední změna hesla ve dnech od 1. ledna 1970.
- Minimální počet dnů mezi změnami hesla.
- Maximální počet dnů, kdy zůstává heslo platné.
- Po kolika dnech je uživatel varován před vypršením platnosti hesla.

- Maximální počet dnů, po které se hlídá nečinnost (nepřihlášení) uživatele.
- Za kolik dnů skončí platnost hesla.
- Počet neúspěšných pokusů o přihlášení.

3.1.3 `/etc/group`

Definuje dostupné skupiny. Skupiny obecně neoplývají příliš mnoha údaji.

- Jméno skupiny.
- Dobrovolný haš hesla.
- ID skupiny (GID).
- Seznam uživatelů obsahující jen uživatele, kteří nemají danou skupinu jako svou primární.

3.1.4 `/etc/user_attr`

Přiděluje vlastnosti související s RBAC k účtům. Počet ani pořadí záznamů nemusí odpovídat `/etc/passwd`, stačí uvést jen vybrané.

- Jméno uživatele.
- Seznam atributů z výběru:

  auths - Seznam autorizací přiřazených k účtu.

  profiles - Seznam profilů. Na rozdíl od jiných seznamů, v tomto má význam i pořadí v jakém jsou jeho položky uvedeny.

  roles - Seznam rolí, pod kterými může uživatel vystupovat.

  type - Rozhoduje, zda účet představuje uživatele nebo roli.

  project - Výchozí projekt po přihlášení uživatele.

  defaultpriv - Výchozí množina privilegií nastavená při přihlášení.

  limitpriv - Maximální množina privilegií, kterou může libovolný proces získat.

  lock_after_retries - Po kolika špatných přihlášení dojde k zablokování uživatelského účtu.

3.1.5 `/etc/security/auth_attr`

Obsahuje seznam všech známých autorizací.

- Jméno autorizace, rozděluje se na prefix a sufix. Pokud sufix chybí, jedná se jen o informativní záznam[1] o skupině autorizací se stejným prefixem. V takové formě nelze přiřadit žádnému uživateli.
- Zkrácený popis, určený hlavně pro GUI aplikace. Měl by se vejít na jeden řádek.
- Úplný popis, určený pro GUI aplikace. Obsahuje rozšířený popis, jenž se většinou zobrazuje až po vyžádání.
- Seznam atributů, aktuálně jediný podporovaný atribut je jen *help* odkazující na dokument s nápovědou.

3.1.6 `/etc/security/prof_attr`

Definuje dostupné profily a jejich základní vlastnosti.

- Jméno profilu.
- Popis profilu, měl by vysvětlit k čemu profil slouží.
- Sezname atributů z výběru:

  help - Jméno dokumentu s nápovědou.

  auths - Seznam autorizací přiřazených profilu.

  profiles - Jména dalších profilů, jenž zahrnuje.

  privs - Seznam privilegií, jenž se mohou požít při spuštění příkazů přes `pfexec`.

3.1.7 `/etc/security/exec_attr`

Druhá část nastavení profilu určuje s jakými privilegii se program spustí prostřednictvím `pfexec`.

- Jméno profilu, stejné jako v `prof_attr`. Typicky pro jeden profil je více záznamů v `exec_attr` se stejným jménem.
- Volba politika (policy) mezi hodnotami *suser* a *solaris*. Politika *suser* je původní chování superuživatele, politika *solaris* již rozeznává privilegia.
- Volba typu mezi *cmd* a *act*, přičemž *act* je dostupná pouze v rozšíření Trusted Solaris.

1. Využíjí jej například GUI aplikace, jenž spravují autorizace.

- Úplná cesta k příkazu, lze použít znak hvězdička v obvyklém smyslu.
- Seznam atributů z výběru:

  euid a *uid* - Mohou být jak jméno, tak přímo identifikátor uživatele. Atribut *euid* slouží jako alternativa k *setuid bit* na spustitelných souborech, *uid* je užitečný v případě skriptů.

  egid a *gid* - Maji obdobný význam jako *euid* a *uid*, jenom pro skupiny
- Seznam privilegií, o která se rozšíří výchozí privilegia před spuštěním příkazu.
- Seznam limitních privilegií, jež se nastaví před spuštěním příkazu.

Poslední dva údaje jsou platné pouze pro politiku *solaris*.

Pro každý soubor existuje sada systémových funkcí pro čtení jejich obsahu s jednotným rozhraním. Jak ilustrační příklad poslouží funkce pro `/etc/passwd`.

`getpwnam` - Vyhledá záznam o uživateli podle jména.

`getpwuid` - Vyhledá záznam o uživateli podle UID.

`setpwent` - Připraví procházení všech uživatelů.

`getpwent` - Postupně prochází záznamy všech uživatelů.

`fgetpwent` - Prochází všechny záznamy o uživatelích ze zadaného souboru

`endpwent` - Ukončí procházení uživatelů.

Obecně se nedoporučuje používat funkci `getpwent`, neboť může způsobit nadměrnou zátěž, případně nemusí být vůbec podporována. Doporučuje listovat jen pomocí `fgetpwent` s parametrem přímo `/etc/passwd`. Funkce pro soubory v adresáři `/etc` mimo `user_attr` jsou součásti normy POSIX. Rozhraní zbylých funkcí se lehce odlišuje, například pro soubor `auth_attr`.

`getauthnam` - Vyhledá záznam o autorizaci podle jména.

`setauthent` - Připraví procházení autorizací.

`getauthent` - Prochází postupně všechny autorizace.

`endauthent` - Ukončí procházení autorizací.

`free_authattr` - Uvolní záznam o autorizaci. Na rozdíl od prvního příkladu, vracené záznamy se alokují dynamicky a ne staticky

`chauthattr` - Zkontroluje, zda uživatel má přidělenou požadovanou autorizaci.

Díky dynamické alokaci jsou tyto funkce již bezpečné z pohledu více vláknových aplikací.

Pro změnu údajů o uživatelích už ale systém žádné funkce neposkytuje a zbývají jen speciální příkazy nebo přímá editace souborů. S lokálními soubory pracují tyto programy: `useradd`, `usermod`, `userdel`, `roleadd`, `rolemod`, `roledel`, `groupadd`, `groupmod` a `groupdel`. Příkazy pro uživatele a role se téměř shodují svými schopnostmi, například nic nebrání změnit uživatelský účet na roli a zpět pomocí `usermod`, respektive `rolemod`. Bohužel, žádný z nich nedovoluje vyzkoušet si změny na nečisto a zkontrolovat si tak více po sobě následující volání dopředu, zda se v nich nevyskytuje neplatné se špatnými parametry. Taky neexistují žádné programy pro editaci profilů a autorizací. K nastavení hesla a souvisejících údajů slouží příkaz `passwd`. Na rozdíl od předchozích však podporuje i vzdálená úložiště.

3.2 ZFS administrace

Správa ZFS se omezuje pouze na dva příkazy: `zpool` a `zfs`. První se zaobírá svazky (ZPool), druhý pak souborovými systémy.

3.2.1 `zpool`

Program `zpool` vytváří, spravuje a ruší svazky. Před tvorbou nového svazku dovoluje ověřit si výslednou konfiguraci. Zpřístupňuje informace o svazku a chybách, které se vyskytly, a následně je řešit, typicky výměnou vadného disku či spuštěním kontroly nebo obnovy. Též zobrazí kompletní historii všech příkazů (`zpool` a `zfs`), které s ním jakkoliv manipulovaly.

Svazek se skládá z jednoho nebo více virtuálních zařízení z následujícího seznamu:

disk - Konkrétní diskové zařízení.

file - Použije soubor místo disku. Slouží hlavně ke zkoušení a testování, nikoliv pro reálné nasazení.

mirror - Dva a více disků stejné velikosti uspořádá do zrcadlení.

raidz(1-3) - Sestaví disky do RAID-Z zapojení s jednoduchou, dvojitou či trojitou paritou. K tomu potřebuje aspoň o jeden disk více, než je zvoléná parita.

spare - Rezervní disk pro případ výpadku nějakého disku z hlavních.

log - Speciální zařízení pro Intent Log.

cache - Zařízení zvětšující kapacitu vyrovnávací paměti.

Celý svazek i jednotlivé zařízení se nachází vždy v jednom z těchto stavů, reprezentující jeho zdraví:

ONLINE - Zařízení pracuje v pořádku.

OFFLINE - Zařízení bylo odstaveno

REMOVED - Fyzické zařízení bylo odpojeno. Detekce odebrání závisí od konkrétního hardwaru.

UNAVAIL - Zařízení se nepodařilo nalézt při otvírání svazku.

DEGRADED - Při práci se zařízením se vyskytly chyby. Stále však existuje dostatek správných kopií, ze kterých lze svazek opravit.

FAULTED - Horší stádium předchozího, když už došlo ke ztrátě dat.

3.2.2 `zfs`

Příkaz `zfs` se podobá předešlému, jen se zaměřením na souborové systémy. Do kategorie správa však tentokrát patří více činností. Stará se o snímky, což představuje jejich pořízení, uložení, nahrání a vrácení souborového systému zpět. Dále pak provádí operace *mount* a *umount* a nahrazuje tak stejnojmenné programy. Ještě povoluje a zakazuje sdílení přes NFS či CIFS.

Souborové systémy a snímky sdílí jednak způsob pojmenování a jednak některé vlastnosti. Jméno začíná názvem svazku a pokračuje hierarchií souborových systémů oddělovaných znakem / (stejně jako adresáře). Název celého snímku vznikne doplněním původního jména souborového systému o znak `@` a konkrétního jména snímku.

Souborový systém v ZFS se výrazně podobá obyčejnému adresáři. Předpokládá se, že s ním bude i podobně manipulováno a k tomu jsou zapotřebí dostatečné pravomoci. K tomuto úkolu se příliš nehodí pouhé autorizace z RBAC, neboť jsou příliš obecné a neomezují přístup ke konkrétnímu souborovému systému. Používá se obdoba řízení přístupu jako u souborů, tedy forma ACL, ale zjednodušená. Uživatelům se přidělují oprávnění k různým činnostem nebo úpravám vlastností. Pokud však uživatel dostane nějaké oprávnění k souborovému systému, obdrží jej zároveň taky pro všechny jeho potomky bez možnosti jak mu jej odepřít. Celkový počet oprávnění dosahuje hranice třiceti, naštěstí je lze organizovat do množin a přiřazovat přímo je. Některá z oprávnění prakticky ani nedávají smysl samostatně přidělit.

3.3 Zóny

Základní nástroje pro zóny zachovávají podobné rozhraní jako v případě ZFS a jsou to `zoneadm` a `zonecfg`. První instaluje, spouští, ukončuje a odinstalovává zóny, druhý pak nastavuje jejich vlastnosti. Každá neglobální zóna se nachází v jednom z šesti stavů:

Configured - Zóna byla vytvořena a její nastavení je v pořádku a uloženo. Následuje instalace a po jejím dokončení dodatečná nastavení při prvním startu.

Incomplete - Během instalace či odinstalace zóny se stav nastaví na *Incomplete*. Taktéž se do tohoto stavu dostane, pokud se zóna nějakým způsobem poškodí.

Installed - Reprezentuje úspěšně na instalovanou zónu, která je připravena ke spuštění. Po skončení instalace již nejsou některá nastavení dostupná.

Ready - Systém připraví zónu ke spuštění.

Running - Zóna byla úspěšně spuštěna a běží.

Shutting down nebo *Down* - Zóna se vypíná. Jedná se o dočasný stav, po ukončení přejde zóna do stavu Installed. Pokud se ovšem zónu nepodařilo zastavit, v tomto stavu setrvá.

3.3.1 `zoneadm`

Program zoneamd je poměrně jednoduchý a kromě již zmíněných schopností přenáší zóny mezi různými počítači. Dále pak nabízí alternativu k čisté instalaci a to zkopírování již existujícího prostředí. Při tom z výhodou využívá klonování souborových systému v ZFS.

3.3.2 `zonecfg`

Příkaz `zonecfg` se soustředí pouze na konfiguraci zón, ale za to velmi podrobně a hlavně prakticky pro interakci s jinými aplikacemi. Všechny změny v nastavení se ukládají současně a před zapsáním proběhne jejich kontrola. Pracuje ve dvou režimech. Buď dostane popis úprav jako své argumenty při spuštění. Pak vše provede a skončí. Nebo se spustí jako interaktivní aplikace.

Oba doplňuje ještě program `zlogin`, prostřednictvím kterého se připojuje ke konzoli zóny. Právě přes konzoli se dokončuje nastavení zóny po instalaci.

3.4 Solaris Managmet Console

Pro Solaris 10 existují i jiné nástroje pro správu, jenž se nedostali do OpenSolaris z licenčních důvodů. Jedním z nich je Solaris Managment Console (SMC) [10]. Jedná se především o GUI aplikaci, ale umožňuje i zadání konkrétních povelů při spuštění. Přímo podporuje taky vzdálenou správu, neboť se skládá ze serverové a klientské části. Věnuje se hlavně následujícím oblastem:

- Sledování různých informacích o systému:
  - Čas a datum.
  - Záznamy z logů.
  - Bežící procesy.

  - Výkon a přidělení prostředků.

- Správa uživatelů a projektů.
- Vytváření a kontrolování úkolů pro plánovač *cron*.

Vedle SMC byla v jedné z aktualizací Solaris 10 přidána webová aplikace pro správu ZFS. V hezké grafice provede všemi nastaveními svazků a souborových systémů. Před vykonání určitého kroku navíc zobrazí jeho zápis v podobě příkazu `zpool` nebo `zfs` spolu s argumenty.

3.5 Visual Panels

OpenSolaris tedy nezdědil žádné komplexnější nástroje pro správu. Proto vznikl projekt Visual Panels psaný v jazyce Java, který se snaží situaci napravit [12]. A i když se ještě výrazně vyvíjí a mění, některé jeho komponenty se již dodávají v aktuálním vydání OpenSolaris.

Vedle Visual Panels existuje podobný, ale menší projekt SimplePanels [8], tentokrát však v jazyku Python. Bohužel se zdá už více jak rok neaktivní.

Nabízely se mi dvě cesty, kudy svou práci vést. Buď začít kompletně novou vlastní aplikaci nebo nějakým způsobem rozšířit funkcionalitu Visual Panels. Moje původní představa spočívala v komplexním nástroji, jenž by začínajícího správce provedl základní konfigurací a poskytl mu jednoduše přehled o systému. V principu by se podobal SMC, ale s větším důrazem pro nastavení zón.

Naproti tomu Visual Panels se více soustředí na správu jednotlivých vlastností systému, aktuálně hlavně na služby, například webový server Apache a databázi MySQL. Dále se vyvíjí ještě nástroj pro firewall, sdílení souborů a nastavení času. Aktuálně však chybí jakákoliv hlubší provázanost samostatných aplikací, přesněji není vůbec cílem.

Zvolil jsem nakonec variantu s Visual Panels, i když byla vzdálenější od mého původního cíle. Poskytovala totiž větší naději, že z mé práce se aspoň část uchytí. Nový projekt zase představoval větší výzvu a více oblastí pro rozhodování o konkrétní implementaci. S tím však i větší riziko neúspěchu, které jsem prostě nechtěl podstoupit.

Kapitola 4

Visual Panels

Ve Visual Panels se klade důraz na vzdálenou správu, tedy nejedná se jen o jednoduché nástavby nad již existujícími programy. Každý nástroj se tak skládá ze serverové a klientské části, komunikace mezi nimi probíhá podle standardu JMX [13]. Prozatím se klientské rozhraní omezuje jen na GUI v rámci Swing, ale do budoucnosti se počítá i s jinými alternativami. Poskytuje však detailnější integraci s prostředím GNOME, například při hledání systémových ikon.

4.1 JMX

Název Java Management Extensions (JMX) zastřešuje technologii pro kontrolu a správu aplikací, systémových objektů, zařízení a služeb. Každému takovému prostředku přísluší objekty typu *MBean* (Managment Bean), respektive *MXBean*. V případě *MXBean* se klade větší omezení na argumenty metod na úkor jednoduché implementace vzdáleného volání. Povoleny jsou pouze základní typy (řetězce, čísla, kolekce a podobné), odkazy na jiné *MXBean* a s omezením i obecné Java Bean. Ty musí ke každé vlastnosti přístupné přes `get` metodu nabízet příslušnou párovou `set` metodu, respektive označit konstruktor správně *@ConstructorProperties* anotací.

Objekty *MBean* mohou generovat události. Některé typické druhy definuje již standard, jako například *AttributeChangeNotification* pro změnu hodnoty vlastnosti. Samozřejmě definici vlastních událostí nic nebrání. Bohužel zřejmě neexistuje spolehlivý způsob, jak prostřednictvím JMX přímo zjistit, zda někdo události odebírá.

Za zmínku ještě stojí podpora autentizace pro vzdálený přístup a pomocná třída *MBeanInfo*, která reprezentuje podrobnější popis objektu *MBean* za běhu aplikace místo prostého výpisu jmen metod a vlastností.

4.2 CACAO

Common Agent Container (CACAO) představuje kompletní server určený ke správě *MXBean* [5]. Normálně se každá aplikace stará o vlastní *MXBean* objekty sama a nepotřebuje další, aby jí s tím jakkoliv pomáhala. Ale ve Visual Panels se pracuje s nezávislými systémovými objekty a žádnému takovému programu nepřísluší. Proto je nutné ho pro běh *MXBean* nejprve vytvořit a následně všechny objekty do něj umístit. Psát

vlastní server jen pro potřeby Visual Panels však postrádá smysl a tak padla volba na CACAO.

Největší přínos CACAO spočívá v rozšíření možností autentizace o RBAC a PAM. Prakticky tak dovoluje opravdové přihlášení do systému. Využívá k tomu své nativní programy, neboť provést obdobný úkol čistě z JVM[1] je nemyslitelné. Při startu aplikace jí nastaví správného uživatele podle předešlé autentizace. Řeší tak problém pří vzniku podprocesů, které se spouští pod stejným uživatelem, pod kterým běží server, tedy superuživatelem.

Zajímavá situace panuje u dokumentace. Pro JMX jsem nalezl spoustu zdrojů, jen občas opomíjí zmínit existenci události. CACAO spoléhá hlavně na dokumentaci ke třídám a pár ilustračních příkladů. A však mimo jeho domovskou stránku jsem si o něm nikde moc nepřečetl. Projekt Visual Panels však již dokumentaci výrazně postrádá. Občas některé třídy obsahují úplnou dokumentaci, ale u většina není uvedený ani krátký komentář s jejich účelem. Dva krátké návody určené pro začátečníky situaci lehce zachraňují, bohužel už trochu zastaraly. Zbývá tak už jen čtení přímo zdrojových kódů.

Visual Panels se aktuálně zaměřují na služby, tedy SMF. Teoreticky by se tak pokryly všechny oblasti nastavení systému. Avšak ne vše je či bude konfigurovatelné přes SMF. Chybí právě správa uživatelů, ZFS a zón a na to jsem se zaměřil ve své práci.

1. Java Virtual Machine

Kapitola 5

Tvorba mých panelů

Na začátku jsem si napsal svůj vlastní malý nástroj, ve Visual Panels nazývaný jednoduše panel. Potřeboval jsem si zjistit základní schopnosti a omezení celého rámce. Panel se pouští příkazem `vp` s jeho jménem jako argumentem. Tedy nelze využít ladících nástrojů vývojového prostředí. Obdobné platí i pro serverovou část, kde jsem se musel spokojit s logem do souborů v `/var/cacao/instaces/default/logs`.

5.1 Logins panel

Jako první jsem si vybral správu uživatelů. Hlavně z důvodu, že jsem o tomto tématu již něco věděl a zdálo se mi nejjednodušší. Ze svých pokusů s Visual Panels jsem ale nabyl dojmu, že bude výhodnější naprogramovat prvně samostatnou aplikaci a z ní pak vytvořit panel. Samozřejmě už při jejím vývoji vymezit hranice mezi částí klienta a serveru.

Následně jsem se rozhodoval, co vše bude aplikace schopna spravovat z výběru lokální databáze uživatelů, NIS či LDAP. Zvolil jsem tu nejjednodušší variantu a to pouze lokální databázi. A však s dostatečnou úrovní abstrakce, tak aby nástroj byl lehce rozšiřitelný o správu zbylých úložišť.

Existují dva způsoby, jak přistupovat k lokální souborům s uživatelskými účty. Za prvé přímý přístup, kdy si manipulaci se soubory obstarává program sám. To v důsledku znamená jednak zamykání souborů, aby se vyloučily souběžné úpravy jinými aplikacemi. A jednak řízení přístupu podle RBAC, což není ani v prostředí JVM přímo implementovatelné a musel bych si připravit vlastní podpůrné nativní programy.

Druhý způsob spojuje systémové funkce pro čtení záznamů o účtech a již existující příkazy k jejich editaci. Funkce jsou dostupné pro všechny typy záznamů včetně profilů, ale příkazy jsou dostupné jen pro úpravu účtů uživatelů a skupin. Nejedná se však o zásadní omezení. Daleko větší problém představuje jejich výkon, neboť při jednom volání se edituje vždy jen jeden účet a zřejmě se tak zbytečně provádí některé operace opakovaně při hromadných úpravách. Objevil jsem též jednu nepříjemnou vlastnost, kdy při přejmenování uživatele se neaktualizuje jeho jméno v souboru `/etc/group` a tím ztratí členství ve skupinách. Obcházím to tak, že nejdříve ho zbavím členství ve všech skupinách, přejmenuji jej a následně původní členství obnovím. Navzdory uvedeným nedostatků představuje druhý způsob lepší volbu než přímá editace souborů a založil jsem na něm svou aplikaci.

Čtení údajů tak zajišťují systémové funkce. Z prostředí JVM se volají pomocí technologie Java Native Interface (JNI) [3]. Ta dovoluje implementovat metody třídy v jiných jazycích než Java, především v C a C++. Tyto metody se pak uloží ve formátu standardní dynamické knihovny a načítají se za běhu programu. Já jsem provedl pouze jednoduché mapování typů argumentů systémových funkcí z jazyka C do jazyka Java. Jelikož jejich rozhraní není nijak složité, jediný obtížnější úkol spočíval v převodu struktur. Jako jejich paralelu jsem vybral neměnné (immutable) Java Beans. Java Beans proto, že žádná jiná rozumná varianta prostě neexistuje. Jejich neměnná varianta vyžaduje uvést konstruktor s anotací *@ConstructorProperties*. Rozdíl mezi nimi demonstruje příklad 5.1. Dalé jsem spojil trojici metod (`set...`, `get...` a `end...`) pro listování záznamy do jediné, která vrací výsledný seznam. Tak vznikly třídy přistupující k obsahu souborů s údaji o uživatelích v adresářích `/etc` a `/etc/security`, pro každý soubor jedna. U akcí, jenž nebyly bezpečné s ohledem na souběžný přístup jsem doplnil synchronizaci.

Následovaly příkazy pro editaci údajů. Externí program se z JVM spustí prostřednictví třídy *Runtime* a reprezentuje jej objekt typu *Process*. CACAO dodává vlastní třídu *UserProcess* se stejnými metodami jako *Process*, ale příkaz se spustí s právy uživatele přihlášeného k serveru. Parametry se předávají jako pole řetězců, takže není nutné se starat o problematické znaky. Program s jeho argumenty lze sloučit i do jednoho řetězce, ale neměl jsem s tím dobrou zkušenost. Parametry se sestavují na základě provedených změn, proto si aplikace musí pamatovat původní i novou hodnotu. Neupravené údaje se nepřepisují. Specifický případ představuje příkaz `passwd`, jenž nastavuje hesla. Jedná se o interaktivní program využívající rozšířené schopnosti terminálu přesahující možnosti třídy *Process*. Nelze s ním tedy nijak komunikovat. Našel jsem však aplikaci `expect` [1], která simuluje interakci s uživatelem a podle jednoduchého skriptu předá příkazu `passwd` nové heslo.

Aktuální a původní údaje spravuji ve třídě *AbstractLogins*. Informace o konkrétním uživateli či skupině se uchovávají v pomocných třídách *User* a *Group*, ale editují se právě pomocí *AbstractLogins*. Je to poměrně netypické rozdělení schopností, ale překvapivě ne úplně špatné. *AbstractLogins* obsahuje po dvou seznamech uživatelů a skupin. V jednom si pamatuje aktuální údaje a v druhém původní hodnoty změněných účtů. Nový uživatelé se nachází v obou dvou, ale odlišují se od pouze upravených hodnotou UID nastavenou na *-1* v druhém jmenovaném seznamu. Naopak odstraněný účet zmizí ze seznamu aktuálních hodnotu a zůstane jen v seznamu původních. Dále hlídá, aby nedošlo při přejmenování k výměně jmen mezi uživateli, neboť by při ukládání musely dočasně existovat dva účty se stejným jménem, což není přípustné.

Z třídy *AbstractLogins* dědí třída *LocalLogins*, která ji rozšiřuje o manipulaci s lokální databází uživatelů. Při čtení provádí kontrolu, zda jsou všechny údaje konzistentní. Například, jestli všichni uživatelé náležící ke skupině existují, či zda přidělené role jsou skutečně role a ne běžní uživatelé. Pokud nějakou chybu zjistí, zeptá se, jak ji vyřešit. Při ukládání změn se ale vyskytuje několik slabin. Nelze nijak dopředu zkontrolovat, že všechny úpravy budou úspěšné. Jediný způsob jak vrátit zpět všechny modifikace spo-

```
/* Běžná Java Bean, hodnoty x a y se mohou kdykoliv
 * upravovat set metodami. */
public class Point {
    private int x;
    private int y;

    public int getX() {
        return x;
    }
    public int getY() {
        return y;
    }

    public void setX(int nx) {
        x = nx;
    }
    public void setY(int ny) {
        y = ny;
    }
}

/* Immutable Java Bean, hodnoty x a y se nastaví pouze
 * při vytvoření a již se nikdy nezmění. */
public final class Point {
    private final int x;
    private final int y;

    @ConstructorProperties ("x", "y")
    public Point(int nx, int ny) {
        x = nx;
        y = ny;
    }

    public int getX() {
        return x;
    }
    public int getY() {
        return y;
    }
}
```

Příklad 5.1: Upravitelná a neměnná varianta třídy *Point*

čívá v pořízení kopií všech souborů s účty jako zálohy. Z ní pak obnovit původní stav a doufat, že současně nikdo další neprováděl změny, neboť o ně taky přijde. Souběžnost obecně představuje neřešitelný problém, protože i když v aplikaci lze zabezpečit výlučnost, z pohledu systému se bude jednat o samostatné a nezávislé volání příkazů. Nikdy tak nelze stávajícími prostředky výlučnosti přístupu dosáhnout.

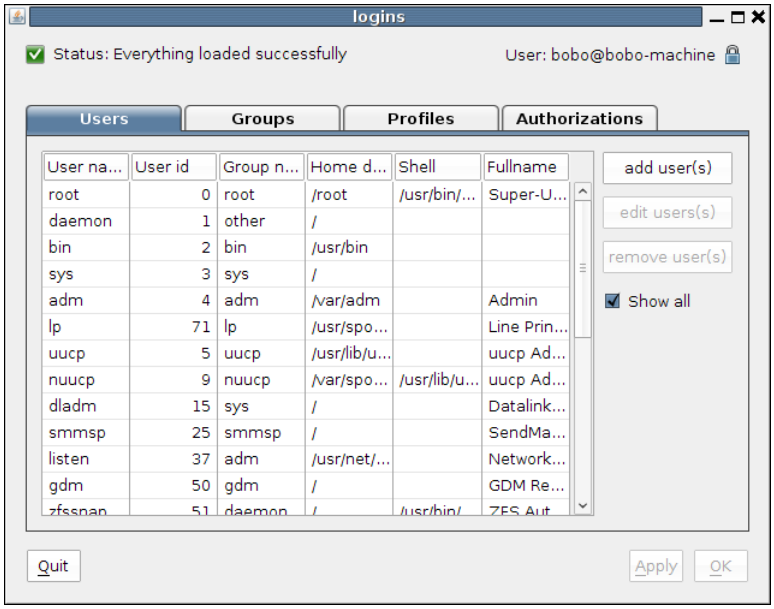
Poslední část tvoří poměrně jednoduché GUI. Jedno hlavní okno (obrázek 5.2) s tabulkami uživatelů, skupin, profilů a autorizací a několika dialogovými okny (obrázek 5.3) pro editaci jejich údajů.

Jakmile jsem aplikaci dokončil to stavu, se kterým jsem byl spokojený, pustil jsem se do další části. Udělat z ní panel na správu uživatelů. Nejprve jsem hledal kudy vést hranici mezi serverou a klientskou částí. První myšlenka spočinula na třídě *LocalLogins*. Chtěl jsem ji přepsat na *MXBean*. Narazil jsem však na dvě překážky. Kontrola konzistence očekává odezvu od uživatele. Dočasně by si tak museli klient a server prohodit role a klient poskytnou *MXBean* pro interakci s uživatelem. Navíc jsem pří návrhu třídy *LocalLogins* nepředpokládal, že by existovala pouze její jediná instance sdílená mezi všemi připojenými panely. Takže jsem postoupil o úroveň níže a vytvořil jsem *MXBean Logins*, která jen prostě agreguje všechny potřebné systémové funkce a příkazy.

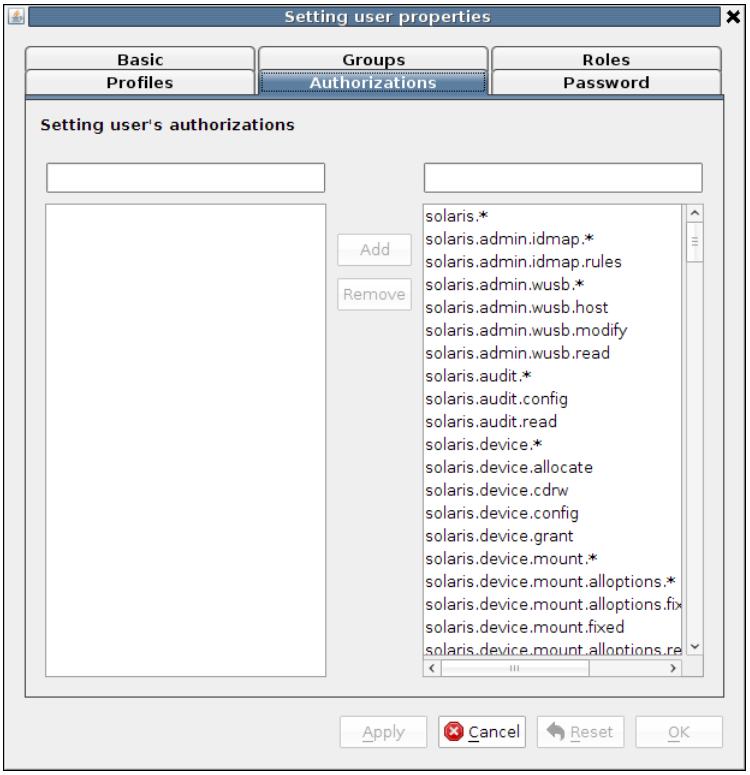
Při úpravách GUI jsem objevil zajímavou vlastnost, na kterou jsem již stihl od prvních pokusů zapomenout. Ke každému vstupnímu prvku, například textovému políčku, náleží ještě schovaná ikonka, jenž se zobrazí pokud uživatel změnil jeho hodnotu. Když ji nastaví zpátky na původní, opět zmizí. Stejný mechanizmus řídí tlačítka jako *OK*, *Použij* i *Obnov* ve spodní části. Jsou dostupná jen v situacích, kdy dává smysl je použít. Vše stojí na rozhraní *Changeable* (příklad 5.4) a jeho potomku *MutableProperty* (příklad 5.5). *Changeable* objekty o sobě ví, zda se jejich hodnota změnila a upozorní na to ostatní. Ještě dovedou aktuální hodnotu uložit nebo obnovit původní. *MutableProperty* definuje přesně typ uchovávané hodnoty a metody k její úpravě. Používají se právě ve vazbě na ovládací prvky, kdy se napojí na model prvku. Například *StringProperty* reprezentuje hodnotu řetězce a třídou *StringPropertySynchornizer* se navzájem synchronizují. Jiný postup implementuje vlastní modely ovládacích prvků s rozhraním *Changeable*. Například u tabulek, kde se model vytváří téměř pokaždé je snazší dopsat pár metod, než zápasit se synchronizací mezi modelem a nejčastěji seznamem záznamů.

Zdárně jsem dokončil svůj první panel, i když úplně spokojený s ním jsem nebyl. Rozhodně bych dnes změnil způsob editace účtů. Vzal bych zdrojové kódy stávajících příkazů a na jejich základě napsal vlastní programy. Doplnil bych je hlavně o hromadné úpravy a s vysokou pravděpodobností by se zvýšil i výkon.

U uživatelského rozhraní mi zase chybí hlubší smysl. Dovede sice nastavit všechny vlastnosti, ale nezohledňuje například to, že profily, autentizace a privilegia se přidělují spíše rolím a až kompletní role uživatelům.



Obrázek 5.2: Hlavní okno Logins panelu



Obrázek 5.3: Nastavení autorizací uživateli

```
public interface Changeable {

    /* Přidá další objekt, jež chce být informován o změnách
     * tohoto objektu. */
    void addChangeListener(ChangeListener listener);

    /* Vrátí, zda byla hodnota objektu změněna. */
    boolean isChanged();

    /* Odebere objekt ze seznamu informovaných. */
    boolean removeChangeListener(ChangeListener listener);

    /* Obnoví původní hodnotu objektu. Aktuální hodnota
     * se ztratí. */
    void reset();

    /* Uloží aktuální hodnotu objektu. Původní hodnota se
     * přepíše aktuální. */
    void save();
}
```

Příklad 5.4: Rozhraní *Changeable*

```
public interface MutableProperty<T> extends Changeable {

    /* Přidá další objekt, jenž bude informován o změnách
     * aktuální nebo původní hodnoty. */
    void addPropertyChangeListener
        (PropertyChangeListener listener);

    /* Odebere objekt ze seznamu informovaných. */
    boolean removePropertyChangeListener
        (PropertyChangeListener listener);

    /* Vrátí konvertor, jenž převádí hodnotu na řetěz a zpět.
     * Používá se k předávání hodnot v jednotném formě. */
    DualConverter<String, T> getConverter();

    /* Vrátí jméno vlastnoti. */
    String getPropertyName();

    /* Vrátí původní hodnotu. */
    T getSavedValue();

    /* Vrátí aktuální hodnotu. */
    T getValue();

    /* Nastaví původní hodnotu. */
    void setSavedValue(T saved);

    /* Nastaví aktuální hodnotu. */
    void setValue(T t);

    /* Nastaví aktuální hodnotu. Pokud se původní a aktuální
     * před změnou shodovali nebo je vynuceno parametrem
     * force, nastaví se na tutéž hodnotu i původní. */
    void update(T value, boolean force);
}
```

Příklad 5.5: Rozhraní *MutableProperty*.

```
public interface ManagedObject<C extends ManagedObject> {

    /* Vrátí unikátní identifikátor. Měl by být konzistentní
     * pro všechny lokalizace i opakovaná spuštění JVM. */
    String getId();

    /* Vrátí popis objektu, pokud možno lokalizovaný. */
    String getDescription();

    /* Vrátí seznam potomků */
    List<C> getChildren();

    /* Vrátí zámek pro synchonizovaný přístup k potomkům. */
    Object getChildrenLock();

    /* Vrátí stav (zdraví) objektu. Může být bud' v pořádku,
     * s varováními nebo s chybami. Může ho ovlivnit i zdraví
     * potomků. Stav nejvýše postaveného se zobrazuje v horní
     * části okna panelu. */
    ManagedObjectStatus getStatus();

    /* Vrátí stav objektu, tentokrát však jeho podrobnější
     * popis. */
    String getStatusText();
}
```

Příklad 5.6: Třída *ManagedObject* (zjednodušená verze)

5.2 ZFS panel

Ovšem největší chybu představovala třída *AbstractLogins*, neboť se kryje s funkcionalitou s Visual Panels. Pro stejný účel existuje rozhraní *ManagedObject* (příklad 5.6). *ManagedObject* obsahuje jednak seznam podřízených objektů (zase typu *ManagedObject*) a jednak seznam vlastností reprezentovaných objekty již zmíněného typu *MutableProperty*.

Opakování stejného omylu jsem se u ZFS panelu snažil vyhnout. Tentokrát jsem vyvíjel už přímo panel a ne samostatnou aplikaci. Rozhraní mezi klientem a serverem jsem postavil na stejném principu jak při uživatelích. Takže jednotlivé metody serverových tříd odpovídají volání příkazu `zfs` s různými argumenty. Správu svazků jsem vynechal. Jednak z důvodu, že výstupy programu `zpool` se hůře zpracovávají oproti příkazu `zfs`, jenž produkuje na požádaní strojově zpracovatelný výstup. A jednak proto, že mi správu svazků nepřipadalo vhodné zpracovat jako GUI aplikaci, maximálně by byla užitečná při zakládání nového svazku.

Z informací, jež klient obdrží od serveru, si poskládá seznam všech souborových systémů. Jeho jednotlivé položky tvoří objekty třídy *DatasetObject*, respektive její potomci *FileSystemObject* a *SnapshotObject*. Ty již právě implementuje rozhraní *ManagedObject*. *DatasetObject* se zaměřuje na funkcionalitu okolo čtení a úprav vlastností datasetů[1]. Základní rysy vlastností jsou dány hierarchickým uspořádání souborových systémů (vztah rodič a potomek). Pro každý typ datasetu existuje sada vlastností, jenž budou vždy dostupné. Část z nich je určena jen ke čtení a reprezentují jeho stav. Zbytek svým nastavením ovlivňuje jeho chování. Hodnota vlastnosti se nastavuje buď přímo u datasetu nebo se dědí z nejbližšího předka. Pokud se ani u jednoho z předků žádná nenalezne, použije se výchozí. Přidělené hodnoty se dataset zbaví tak, že ji zapomene a opět začne dědit. Ke konkrétnímu datasetu lze přidat ještě uživatelské vlastnosti. Pracuje se s nimi úplně stejně, což působí lehce zvláštně. Neexistuje žádný příkaz „přidej vlastnost“, prostě se jen nastaví její hodnota. Podobně při jejím rušení, kdy se jednoduše nechá zdědit z předka.

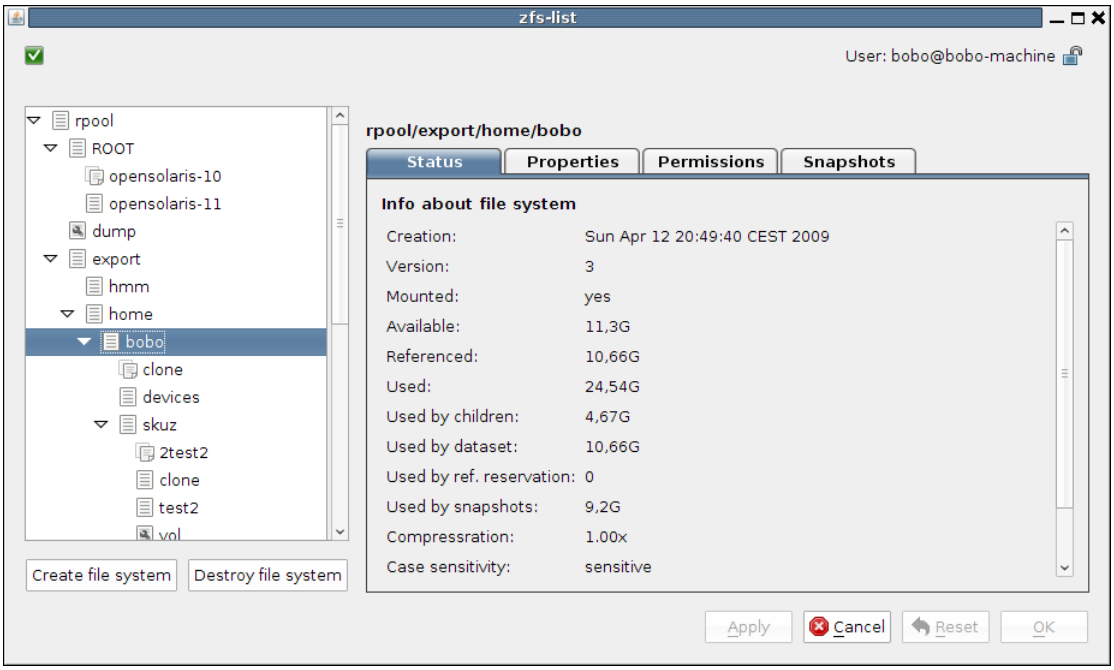
Pro uchovávání vlastností se používám objekty třídy *ZFSProperty*, která vychází z rozhraní *MutableProperty* a spojuje tři *MutableProperty* dohromady. První reprezentuje vlastní hodnotu, druhá, zda se vlastnost dědí z předka a třetí zruší vlastní hodnoty u potomků a vynutí u nich dědění. Načítání hodnot ze serveru probíhá až když je skutečně potřeba. Nikoliv tedy při startu panelu, neboť příliš zdržuje.

FileSystemObject přidává k *DatasetObject* ještě správu oprávnění a snímků. Oprávnění jsou daleko komplikovanější než vlastnosti. Oprávnění vždy někomu patří ať už uživateli nebo skupině. Jedinou výjimku představují speciální oprávnění *Everyone*, jenž platí pro všechny a *Creator*, které obdrží uživatel, jenž vytvoří potomka souborového systému. Dále u každého oprávnění lze zvolit, zda platí jen pro daný souborový systém, jeho potomky nebo nejčastěji pro něj i potomky zároveň. K dispozici jsou pouze definovaná oprávnění, korespondující se schopnostmi příkazu `zfs` a vlastnostmi datasetů. Při úpravách oprávnění se příkaz `zfs` chová trochu nestandardně, kdy chybové hlášky vypisuje na normální výstup. To mě zaskočilo a chvíli mi trvalo, než jsem přišel na to, proč se mi zobrazuje prázdné chybové okno, když v jiných případech fungovalo v pořádku.

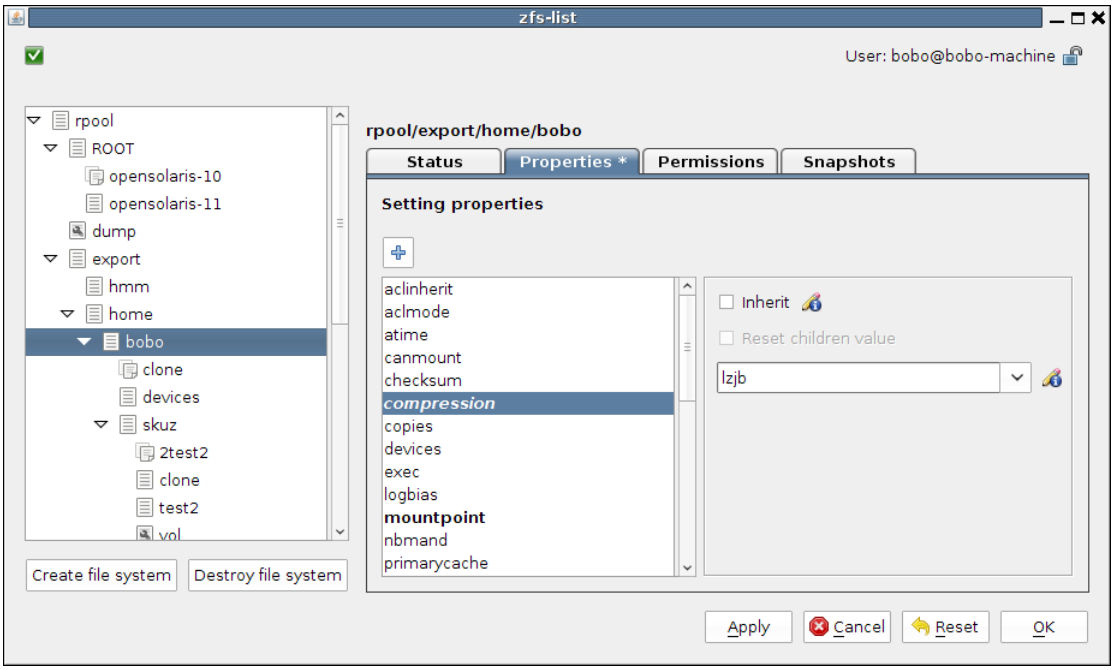
Uživatelské rozhraní po zkušenostech s panelem uživatelů nepředstavovalo už žádnou obtíž. Všechny souborové systémy jsem nasypal do jednoho stromu (*JTree*), takže se při zobrazení uchovává jejich hierarchie (obrázek 5.7). Naproti tomu vlastnosti hierarchii postrádají, takže si vystačí pouze se seznamem. Podle výběru vlastnosti se pak vedle zobrazí vhodný editor (obrázek 5.8). Například spousta z nich má předem omezený výběr povolených hodnot, tedy se ideálně hodí pro použití rozbalovací nabídky (*JComboBox*). Snímkům pro změnu vyhovuje tabulka se základními informacemi jako datum pořízení a zabrané místo (obrázek 5.9).

U zobrazení oprávnění jsem se rozhodoval nejdéle. Původně jsem chtěl mít nahoře jednu rozbalovací nabídku s výběrem mezi oprávněními pro uživatele, skupiny, množinami a speciálními *Everyone* a *Creator*. Pod ní se měl nacházet seznam s položkami

1. Souhrnný název pro souborové systémy a snímky.



Obrázek 5.7: Okno ZFS panel s detailním pohledem



Obrázek 5.8: Nastavení vlastnosti *compression*

podle volby z nabídky a vedle něj tabulka plná zaškrtávacích políček se všemi oprávněními v řádcích a ve sloupcích oblast působnosti (lokální a potomci). Toto uspořádání se však nedočkalo ani kompletní implementace a zavrhl jsem ho. Připadalo mi příliš těžkopádné. Místo toho jsem seznam spojil s nabídkou a vznikl z nich strom s pěti větvemi z kořene kopírující volby z nabídky. Jednotlivé položky ze seznamů tvoří listy pod příslušnými větvemi. Pokud k větvi žádné listy nenáleží, ani se nezobrazí. I tabulku jsem nahradil stromem. Přišla mi příliš veliká a nepřehledná. Spousta oprávnění se věnuje stejné oblasti, například zálohování nebo snímkům a obvykle jsou potřebné všechny, aby nebyl uživatel nijak omezován. Takový vztah se ve stromu dobře zachytí. Na nejvyšší úrovni se dělí na oprávnění pro příkazy a pro vlastnosti. Zjemňování pokračuje směrem dolů až se zastaví na jednotlivých oprávnění. Takže přiřadit oprávnění pro všechny příkazy lze pouze jediným kliknutím. Aktuální stav přiřazení ukazuje vždy zaškrtávací políčko vedle názvu. V pokročilejším režimu zobrazuje hned dvě políčka. Rozlišují tak mezi lokálním přiřazením a potomky. Původně se zobrazovaly vždy obě dvě, ale strom působil na první pohled složitě. A hlavně nijak nenaznačoval jejich účel. Nyní se objeví až po akci *Odděl lokální a potomky*. Výsledek převádí obrázek 5.10.

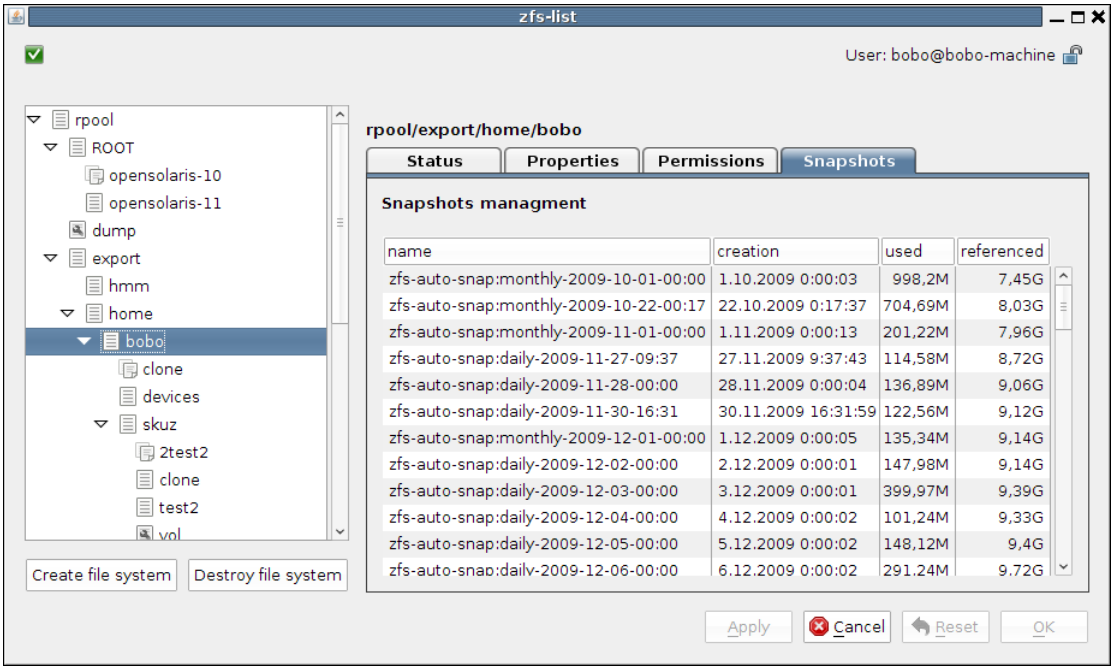
5.3 Panel zón

Po dokončení ZFS panelu jsem si uvědomil slabinu rozhraní mezi serverem a klientem. Klient se nikdy nedozví o změnách, jenž neprovedl on sám. Platí to především o přidání, rušení a přejmenování souborových systémů. Aby byl o nich pravidelně informován, musel by si sám kontrolovat aktuální stav v pravidelných intervalech. Jenže toto řešení způsobuje zbytečnou zátěž serveru, pokud se k němu připojí více klientů naráz. Úplné zahlcení ale nehrozí, neboť zřejmě nenastane situace, kdyby se připojily více jak desítky klientů naráz.

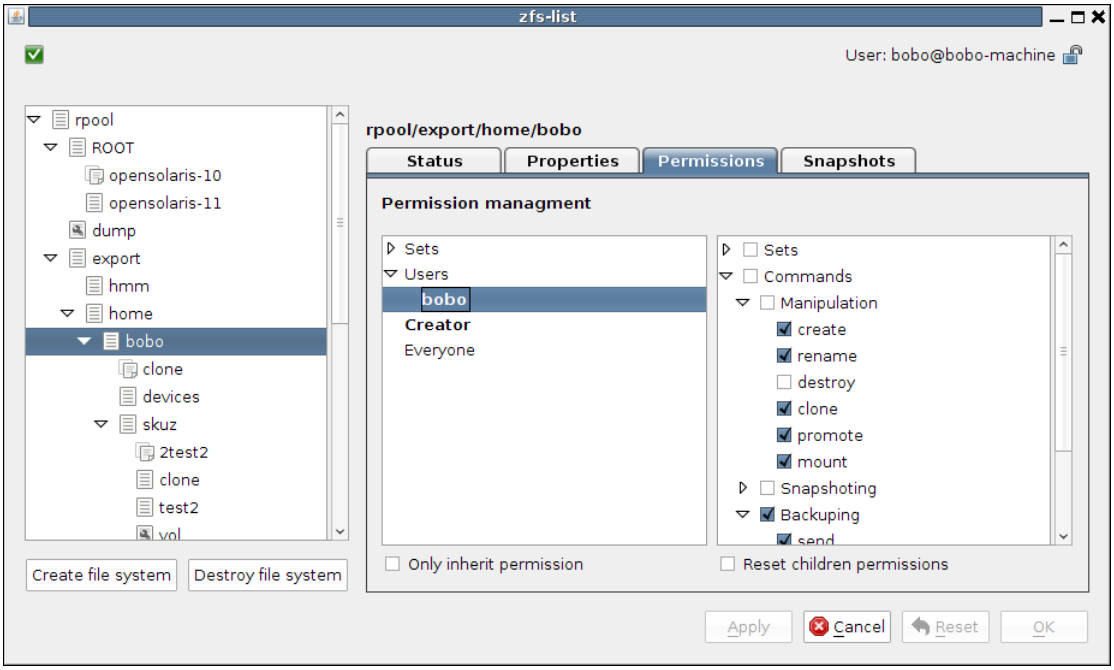
Hledal jsem nějaký nápad, jak z toho ven. Přišel jsem s variantou, že na straně serveru přidám ještě jednu vrstvu a přiblížím ho více klientovi. Každé zóně tak odpovídá jedna *MXBean* se všemi jejími vlastnostmi a funkcemi. Pamatuje si však jen aktuální stav a udržuje jej shodný se skutečným stavem zóny. Bohužel správa zón negeneruje žádné události a tak musí periodicky kontrolovat, či nedošlo ke změnám. Pokud ano, informuje klienta prostřednictvím standardní notifikace v JMX. Postrádá však smysl udržovat aktuální údaje, pokud se k serveru nikdo, kdo by o ně projevoval zájem, nepřipojí. To JMX samo neumí a proto se klient serveru periodicky připomíná. Když server nedostane v požadovaném intervalu žádnou připomínku, přeruší aktualizace dokud se nepřipojí další klient.

Ve výsledku na straně serveru komunikaci zabezpečují objekty třídy *Zone* a klient si z nich staví své objekty typu *ZoneObject*. *ZoneObject* se podobá *Zone*, jen u vlastností přidává dělení na původní a novou hodnotu a hlídá jejich změny.

Pokud nějaká akce skončí chybou, zobrazí se uživateli okno obsahující chybový výstup příkazu. Naneštěstí server běží vždy pod výchozí lokalizací a klient tak obdrží zprávu v ní. Proto jsem přidal ke všem akcí třídy *Zone* ještě argument řetězec *lang*, aby



Obrázek 5.9: Tabulka se snímky



Obrázek 5.10: Editace oprávnění

se příkazy spouštěli se správně nastavenou lokalizací. Její hodnota se převezme ze systémové proměnné *LANG*.

Správu zón doprovází jedna obtíž a to, že minimálně při instalaci potřebuje uživatel příkazový řádek, aby provedl iniciální nastavení zóny. To probíhá v jednoduchém programu a však s TUI[2] a nelze tak s ním přímo pracovat ze stejného důvodu jako s příkazem `passwd`. Nabízí se podobné řešení, ale neshledal jsem jej úplně šťastným. Konfigurace neprobíhá totiž vždy stejně a výsledný skript by byl zřejmě o dost složitější. Zbývala mi jediná možnost, nějak jinak přenést vstup a výstup příkazu ke klientovi. První úvaha padla na protokol SSH. Pro mluvila dostupnost a jednoduchost implementace. Na druhou stranu by se uživatel musel pokaždé znovu přihlašovat a to i když spravuje svůj lokální počítač. Zvolil jsem těžší variantu a pokusil jsem implementovat vlastní vzdálené přihlášení k terminálu, ale pomocí JMX. Obával jsem se, že to nebude fungovat, ale podařilo se.

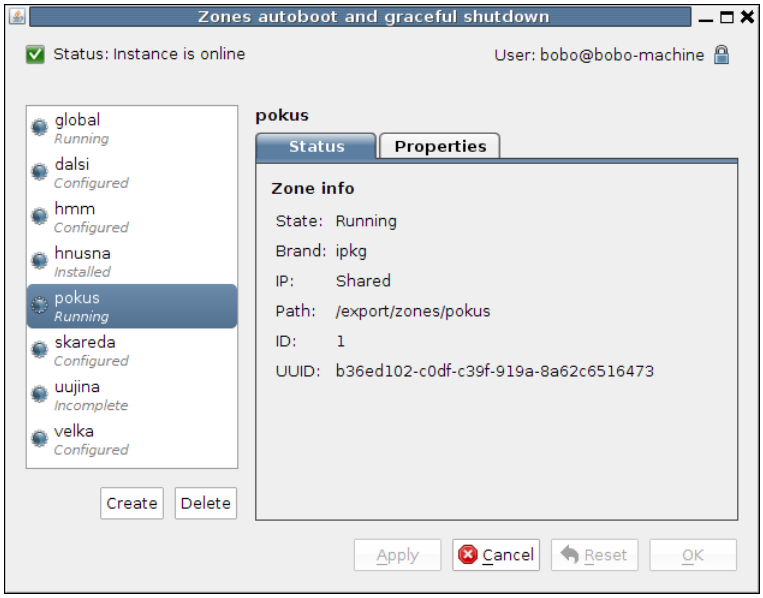
Nejprve jsem potřeboval program, jenž otevře virtuální terminál a napojí na něj zadaný příkaz. Jeho výstupy pak předá serveru a z něj nazpět přečte vstup od klienta. Jedině při splnění těchto podmínek funguje správně příkaz `zlogin`, neboť vyžaduje vždy konkrétní terminálové zařízení. To přímo CACAO ani Java neumí. Na straně klienta jsem chtěl nejdříve integrovat okno s terminálem přímo do panelu, ale nenašel jsem vyhovující Java implementaci emulátoru terminálu. A tak prostě spouští program *xterm* či *gnome-terminal*. V něm pak další aplikaci, jenž pracuje obráceně než její serverový protějšek. Přepne terminál do *raw* módu a zachytává všechny pokyny uživatele, ty obdrží panel a pošle dál serveru. Opačným směrem pak přicházejí výstupy, jenž aplikace vydává za své. Veškerou komunikaci řídí klient, server si jen hlídá, zda ještě někdo projevuje o běžící příkazy zájem.

Uživatelské rozhraní se inspiruje u ZFS panelu a velmi se mu podobá (obrázek 5.11). Je jen chudší, neboť u zón se nastavují jen jejich vlastnosti. Za to jsem jej dotáhl úplně dokonce. Pod editor jsem doplnil plánovaný popis jednotlivých vlastností, poskládaný z manuálových stránek (obrázek 5.12).

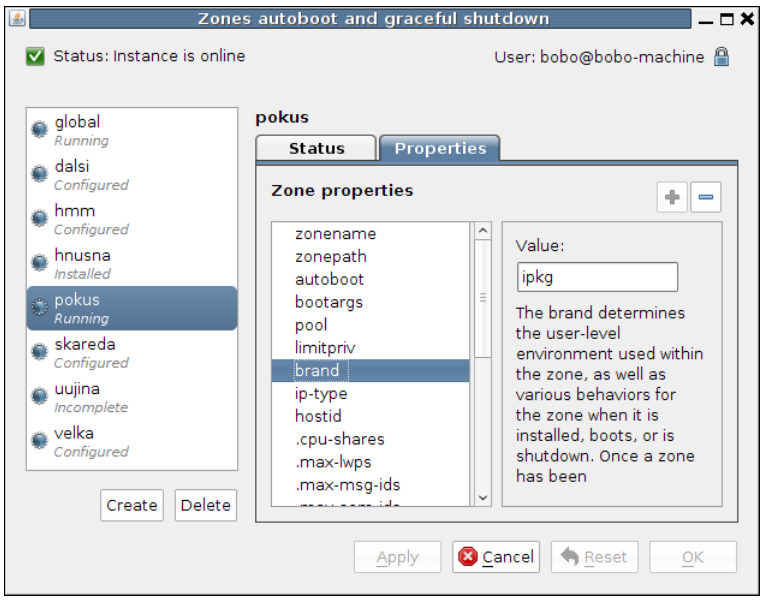
Během práce jsem se setkal s několika problémy a pár nejnepříjemnějších zde uvedu. Aktualizace Visual Panels probíhají poměrně často a stále dochází k zásadním změnám, takže panely napsané pod starší verzí přestanou fungovat. Udržet krok s hlavní větví zabere nezanedbatelnou dobu. Na druhou stranu systémem OpenSolaris distribuuje vždy starší verze, pod kterou zase nefungují novější panely. Vyřešil jsem to nakonec tak, že jednou za čas jsem synchronizoval své zdrojové kódy s hlavní větví a některé soubory standardní instalace přepisoval vlastními.

Objevil jsem nečekanou vlastnost implementace stromů (*JTree*). Když model informoval strom, že se změnila hodnota uzlu a daný uzel byl shodou okolností vybrán, generoval strom události o změně vybraného uzlu. S tím jsem nepočítal a chvíli jsem hledal, proč se mi spouštěly metody, která jsem přímo nevolal. Ocenil bych, kdyby se taková skrytá zřetězená volání nějakou formou dokumentovala.

2. Textová obdoba GUI.



Obrázek 5.11: Okno panelu zón s detailním pohledem



Obrázek 5.12: Nastavení vlastnosti *brand*

Kapitola 6

Závěr

S výsledkem své práce jsem spokojený jen napůl. Panel zón se mi podařil a zbývá na něm provést pár kosmetických úprav. Věřím, že pak by mohl být začleněn do Visual Panels. Nepokusil jsem jej začlenit do Visual Panels jen proto, že jsem se už musel věnovat psání textu práce a ne dolaďovat výsledek podle požadavků vývojářů.

ZFS panelu chybí poslední krok a to upravit rozhraní serveru do podobného tvaru jako u panelu zón. Nejedná se o nijak náročnou činnost, prakticky se jen otrocky zkopírují a lehce upraví kusy kódu z implementace zón.

Panel uživatelů ve stávající podobě nevyhovuje vůbec a prakticky vyžaduje kompletní přepis. Serverovou část očekává podobná transformace jako u ZFS panelu. Klienta bych však v klidu zahodil a napsal znovu, i když GUI by se využít dalo. Jak už jsem však uvedl, ignoruje doporučené postupy, jak správně konfigurovat uživatele podle RBAC. Na druhou stranu nápad, jak jej navrhnout lépe jsem ještě nedostal.

Jaká budoucnost očekává Visual Panels si však nejsem jistý. Ano, OpenSolaris potřebuje nástroje pro snadnou správu začátečníky. Ovšem naučit se základy jeho ovládání lze stihnou i za jeden týden, podle předchozích zkušeností s jinými systémy. Já sám používám občas jen panel pro správu služeb.

Trochu zpětně lituji, že jsem se nepokusil napsat vlastní aplikaci podle svých původních představ. Zřejmě bych ji ale nedotáhl úspěšně do konce, neboť o spoustě problémů, kterým bych musel čelit jsem neměl ani ponětí, ale byla by to zřejmě zajímavější zkušenost, i když zřejmě bez reálného přínosu.

Literatura

[1] Libes, D.: The Expect Home Page [online]. Dostupné z <http://expect.nist.gov>, 2009.

[2] Sun Microsystems, Inc.: *RBAC in the Solaris Operating Environment*. Sun Microsystems, Inc., 4 2001.

[3] Sun Microsystems, Inc.: ava Native Interface Specification [online]. Dostupné z <http://java.sun.com/j2se/1.5.0/docs/guide/jni/spec/jniTOC.html>, 2003.

[4] Sun Microsystems, Inc.: ZFS On-Disk Specification [online]. Dostupné z <http://hub.opensolaris.org/bin/download/Community+Group+zfs/docs/ondiskformat0822.pdf>, 2006.

[5] Sun Microsystems, Inc.: Common Agent Container API [online]. Dostupné z <https://common-agent-container.dev.java.net/nonav/public/doc/api/index.html>, 2007.

[6] Sun Microsystems, Inc.: OpenSolaris OS [online]. Dostupné z <http://www.opensolaris.com/>, 2009.

[7] Sun Microsystems, Inc.: opensolaris.org (Main.WebHome) [online]. Dostupné z <http://hub.opensolaris.org/bin/view/Main>, 2009.

[8] Sun Microsystems, Inc.: Simple Panels (Project simplepanels.WebHome) [online]. Dostupné z <http://hub.opensolaris.org/bin/view/Project+simplepanels>, 2009.

[9] Sun Microsystems, Inc.: *Solaris ZFS Administration Guide*. Sun Microsystems, Inc., 10 2009.

[10] Sun Microsystems, Inc.: *System Administration Guide: Basic Administration*. Sun Microsystems, Inc., 10 2009.

[11] Sun Microsystems, Inc.: *System Administration Guide: Solaris Containers-Resource Management and Solaris Zones*. Sun Microsystems, Inc., 10 2009.

[12] Sun Microsystems, Inc.: Visual Panels (Project vpanels.WebHome) [online]. Dostupné z <http://hub.opensolaris.org/bin/view/Project+vpanels>, 2009.

[13] Sun Microsystems, Inc.: Getting Started with Java Management Extensions (JMX): Developing Management and Monitoring Solutions [online]. Dostupné z <http://java.sun.com/developer/technicalArticles/J2SE/jmx.html>, 2010.

[14] The Trustees of Indiana University: What are SunOS and Solaris? [online]. Dostupné z <http://kb.iu.edu/data/agjq.html>, 2008.