AFS i Coda
Rozproszone systemu plików

Marcin Poturalski (mp181259@zodiac.mimuw.edu.pl)

7 stycznia 2003

1 AFS

1.1 Wprowadzenie

AFS – Andrew File System – to rozproszony system plików, opracowany w Carnegie-Mellon University jako część rozproszonego środowiska obliczeniowego *Andrew*. Obecnie *AFS* jest rozwijany przez IBM.

1.2 Założenia projektowe

AFS jest rozproszonym system plików, wobec tego w jego założeniach projektowych znalazły się typowe cechy systemu rozproszonego[1]:

przezroczystość dostępu – użytkownik nie powinien być świadomy rozproszenia plików; dostęp do plików lokalnych i rozproszonych powinien się odbywać przez ten sam interfejs;

przezroczystość położenia – użytkownik powinien widzieć jednolitą przestrzeń nazw plików, niezależnie od miejsca zalogowania; zmiana (fizycznego) położenia pliku nie powinna wpływać na jego położenie logiczne (ścieżkę dostępu);

przezroczystość współbieżności – wielu użytkowników może pracować nad tym samym plikiem; mechanizm blokad (*lock*)

przezroczystość awarii – poprawne działanie serwera po awarii klienta oraz poprawne działanie klienta pomimo zagubienia komunikatów lub chwilowych przerw w działaniu serwera;

[1] implementowane przez większość współczesnych rozproszonych systemów plików

przezroczystość wydajności – programy klienta powinny wykonywać się z zadawalającą prędkością pomimo zmian obciążeń serwerów;

przezroczystość sprzętu i systemu operacyjnego – niezależność systemu od sprzętu i systemu operacyjnego; serwery/maszyny klienckie współdzielące rozproszony system plików mogą działać pod różnymi systemami operacyjnymi i na różnym sprzęcie;

skalowalność – powinna być możliwość stopniowego zwiększania liczby serwerów i maszyn klienckich w ramach jednego rozproszonego systemu plików, przy zachowaniu zadawalającej wydajności;

bezpieczeństwo – użytkownik ma pewność, że komunikuje się z prawdziwym serwerem, a serwer, że nikt nie podszywa się pod użytkownika; system zapewnia ochronę plików użytkownika przed innymi użytkownikami, w zakresie, który użytkownik sobie zażyczy.

Szczególny nacisk projektanci *AFS* położyli na skalowalność.

Dodatkowo wśród założeń projektowych *AFS* pojawiły się:

przezroczystość zwielokrotnienia – jeden plik może posiadać kilka kopii znajdujących się na różnych serwerach; umożliwia to dzielenie obciążenia pomiędzy serwerami oraz polepsza tolerowanie uszkodzeń;

przezroczystość wędrówki – przemieszczanie pliku nie powoduje konieczności zmian w programach klienta, ani modyfikacji systemowych tablicach administracyjnych po stronie klienta;

architektura klient–serwer – pliki dzielone są przechowywane tylko na serwerach; dostęp do tych plików daje użytkownikowi klient.

Innym założeniem projektowym *AFS* jest semantyka sesji. Oznacza to, że komunikacja między serwerem a klientem ma miejsce tylko podczas otwierania i zamykania pliku.

Jeszcze innym ważnym, ale i trudnym do zrealizowania założeniem jest semantyka jednej kopii – w systemie powinna się znajdować tylko jedna, aktualna wersja pliku.

Cechą wyróżniającą *AFS* wśród rozproszonych systemów plików są usługi całoplikowe. Założenie to wzięło się z następujących obserwacji, dokonanych w środowiskach akademickich:

- większość plików jest mała
- operacje czytania są znacznie częstsze niż pisania
- większość plików jest czytana i pisana przez jednego użytkownika; jeżeli z pliku korzysta więcej użytkowników, to najczęściej tylko jeden dokonuje w nim zmian
- dostęp do plików jest najczęściej sekwencyjny

- odwołania do plików są skumulowane – jeśli odwołanie do pliku było niedawno, to istnieje spora szansa, że nastąpi ponownie.

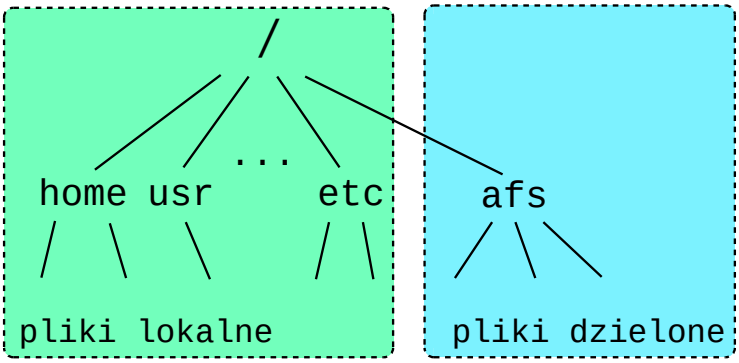
Obserwacje te są prawdziwe dla większości zastosowań rozproszonych systemów plików – z wyjątkiem baz danych. Ze względu na zupełnie inne wymagania względem „zwykłych" rozproszonych systemów plików i baz danych, projektanci założyli, że *AFS* nie będzie tworzony pod kątem używania jako baza danych.

AFS działa pod różnymi systemami operacyjnymi, ale dla uproszczenia opiszę jak wygląda w systemach unixopodobnych.

1.3 Jednolita przestrzeń nazw

AFS udostępnia użytkownikom jednolitą przestrzeń nazw. Znaczy to tyle, że niezależnie od miejsca zalogowania, ścieżki dostępu do plików[2] są takie same.

Przestrzeń plików rozproszonych *AFS*ma strukturę drzewiastą, zupełnie taką sama jak zwykły unixowy system plików. Posiada oczywiście korzeń, który zwyczajowo jest montowany w lokalny system plików na maszynach klienckich pod ścieżką */afs/*.



1.4 Komórka

Komórka (*cell*) to zbiór maszyn (komputerów) – serwerów i klientów, które dzielą ustawienia konfiguracyjne. Oprócz maszyn, do komórek przypisani są użytkownicy. W przeciwieństwie do maszyn, które przynależą do co najwyżej jednej komórki, użytkownik może mieć konto w wielu komórkach.

Użytkownik pracujący na maszynie w komórce, aby mieć pełny dostęp do swoich plików, musi się zalogować. Nie musi jednak być ograniczony tylko do zawartości tej komórki. Administrator komórki może „otworzyć ją na świat" – umożliwić dostęp do danych przechowywanych w

[2] z przestrzeni plików *AFS*

ramach komórki innym komórkom. W tym celu musi jednak przestrzegać ogólnie przyjętej konwencji: korzeń systemu plików *AFS* komórki[3] jest zamontowany w lokalnym systemie plików na każdej maszynie pod adresem */afs/domena/*, gdzie *domena* jest nazwą domeny instytucji, która jest właścicielem komórki (np. */afs/mimuw.edu.pl/*). Po spełnieniu tego warunku[4], może podmontować w katalogu */afs/* pliki[5] z innych komórek[6] (stanie się widoczne np. */afs/ds2.uw.edu.pl/*). Potencjalnie można by stworzyć jednolitą, ogólnoświatową przestrzeń plików w *AFS*, na przeszkodzie stoi jednak wydajność – skalowalność *AFS*ograniczona jest do sieci *WAN*.

Domyślnie użytkownik w obcej komórce (w sensie maszyny, na której pracuje) nie posiada żadnej autentyfikacji[7]. Jeśli posiada konto w obcej komórce, może się zalogować również tam (użytkownik może być zalogowany jednocześnie do kilku komórek) i zyskać dostęp do tych katalogów i plików obcej komórki, do których miałby dostęp logując się na maszynie z owej komórki.

1.5 Klient–serwer

AFS jest zrealizowany w architekturze klient–serwer. Podział ról między serwerem a klientem jest wyraźny. Serwer przechowuje wszystkie pliki i dostarcza je na życzenie klientom. Klient (czyli oprogramowanie zainstalowane na maszynach klienckich) zajmuje się dostarczaniem użytkownikom żądanych plików zdalnych.

W *AFS* serwer nazywa się *vice*, a klient – *venus* (bądź *Cache Manager*).

Serwer dostarcza klientom prostych usług plikowych. Znaczy to tyle, że pliki identyfikuje nie po nazwach i ścieżkach dostępu, a unikalnych identyfikatorach. W *AFS*takim unikalnym indetyfikatorem jest *fid*. Zadaniem klienta (*Cache Manager*) jest dostarczenie użytkownikowi usług katalogowych, czyli tłumaczenia nazw plików i ścieżek dostępu na zrozumiałe dla serwera *fid'y*.

Serwery *AFS* oprócz plików i ich metadanych przechowują informacje o użytkownikach i grupach, wspomagają proces wzajemnej autentyfikacji, są odpowiedzialne za ochronę plików (*ACL*), przechowują bazę danych o lokalizacji tomów, wspomagają przenoszenie, replikowanie, tworzenie i usuwanie tomów, tworzenie ich kopii zapasowych oraz wiele więcej. Pojęcia powyższe są wyjaśnione w dalszej części dokumentu.

[3] czyli główny tom
[4] a także kilku innych, technicznych i niestotnych z punktu widzenia tej prezentacji
[5] a właściwie tomy
[6] oczywiście tylko takich, które również „otworzyły się na świat"
[7] jest anonimowy, czyli jest w grupie *system:anyuser*

1.6 Cache'owanie plików

Jednym z najważniejszych pomysłów w *AFS* jest *cache'owanie* plików po stronie klienta. Każdy klient musi przeznaczyć pewną ilość dysku na bufor plików (*cache*).

Otwieranie plików lokalnych (tzn. z lokalnego systemu plików) odbywa się normalnie. Natomiast gdy użytkownik próbuje otworzyć plik zdalny, *Cache Manager* sprawdza, czy jego aktualna (uwierzytelniona) kopia znajduje się w *cache'u*. Jeśli nie, *Cache Manager* ściąga CAŁY plik z odpowiedniego serwera i umieszcza go w *cache'u*. Następnie jest on otwierany lokalnie, zwracany jest deskryptor, i wszystkie operacje na pliku są wykonywane lokalnie. Dopiero zamknięcie pliku powołuje odesłanie go odpowiedniego serwera, oczywiście tylko jeśli został zmodyfikowany. Lokalna kopia pliku nie jest usuwana i może zostać ponownie użyta (otwarta), jeśli jest nadal ważna. Kopie plików są usuwane przez *Cache Manager* dopiero wtedy, gdy potrzebne jest miejsce na nowe pliki.

Mechanizm *cache'owania* radykalnie zmniejsza ilość przesyłanych danych, jeżeli klient odwołuje się wielokrotnie do pliku zanim zostanie usunięty z *cache'u*. Oczywiście gdy klient odwołuje się do pliku często, to jest mała szansa, że zostanie on usunięty, o ile dobrze dobrano rozmiar *cache'u*. Założenia projektowe *AFS* mówią zaś, że odwołania są skumulowane. *Cache'owanie* plików zapewnia więc znaczny wzrost wydajności.

Fakt, że plik podlega aktualizacji na serwerze dopiero w wyniku operacji zamknięcia (semantyka sesji), powoduje pewną utratę przezroczystości. Jeżeli jakaś aplikacja w czasie obróbki pliku ma go cały czas otwarty, to oczywiście operacja zapisania będzie powodować jedynie zmianę lokalej kopii w cache'u. Aby mieć pewność, że wprowadzone zmiany zostaną zapisane na serwerze i nie zostaną utracone w wyniku awarii stacji roboczej, użytkownik musi mieć świadomość, w jaki sposób aplikacja zapisuje plik – jeśli jest on taki jak opisałem powyżej, co jakiś czas powinien ręcznie zamykać plik. Z punktu widzenia twórców oprogramowania użytkowego, AFS *wyróżnia jeden ze sposobów zapisywania pliku (zapis, zamknięcie i ponowne otwarcie), podczas gdy dla plików z lokalnego systemu są one równoważne.*

Spójność *cache'u* Gdy serwer wysyła *Cache Manager* plik, do którego jakiś użytkownik ma prawo pisania, dodatkowo załącza obietnicę zawiadomienia (*callback promise*). Jest to gwarancja, że *Cache Manager* zostanie powiadomiony, gdy dany plik na serwerze zostanie zmieniony (przez innego użytkownika). Obietnica jest przechowywana przez *Cache Manager* razem z plikiem, którego dotyczy. Ma ona dwa stany: ważności i unieważnienia.

Gdy serwer otrzyma od jakiegoś klienta nową wersję pliku, wysyła zawiadomienie (*breaks callback*) każdemu z klientów, którzy otrzymali *callback promise*. Stan obietnicy zmieniany jest na nieważny. Gdy użytkownik spróbuje otworzyć unieważniony plik, *Cache Manager* ponownie ściąga z serwera „opiekującego się" tym plikiem jego aktualną wersję.

W celu zabezpieczenia się przed utratą zawiadomień w wyniku awarii sieci, muszą być one odnawiane co T sekund. T jest stałą systemową; jej typowa wartość to kilka minut.

Mechanizm *callback* został wprowadzony w wersji 2.0 *AFS*. Daje on lepszą skalowalność niż stosowany we wcześniejszych wersjach *AFS* (oraz np. w *NFS*) mechanizm znaczników czasu, ponieważ nie jest konieczne sprawdzanie wersji pliku przy każdym otwarciu. W zamian tracona jest bezstanowość serwerów. Muszą one pamiętać obietnice zawiadomienia. Ponieważ nie mogą one zostać utracone w wyniku awarii serwera, są pamiętane na dysku i aktualizowane transakcjami (operacjami niepodzielnymi).

Opisany tu mechanizm nie zapewnia kontroli współbieżnych aktualizacji. Jeśli kilku użytkowników otworzy ten sam plik i będzie go jednocześnie modyfikować, a następnie zamknie plik, to na serwerze znajdzie się tylko wersja użytkownika, który zamknął go jako ostatni. Nie wystąpią przy tym żadne ostrzeżenia, ani błędy. Sterowanie współbieżnością, jeśli jest potrzebne użytkownikom, musi być zapewnione przez dodatkowe oprogramowanie.

Semantyka aktualizacji Jednym z założeń projektowych *AFS* było przybliżenie znanej z Unixa semantyki jednej kopii. Dokładne jej uzyskanie nie jest oczywiście możliwe w systemach rozproszonych dużej skali bez rażącego spadku wydajności.

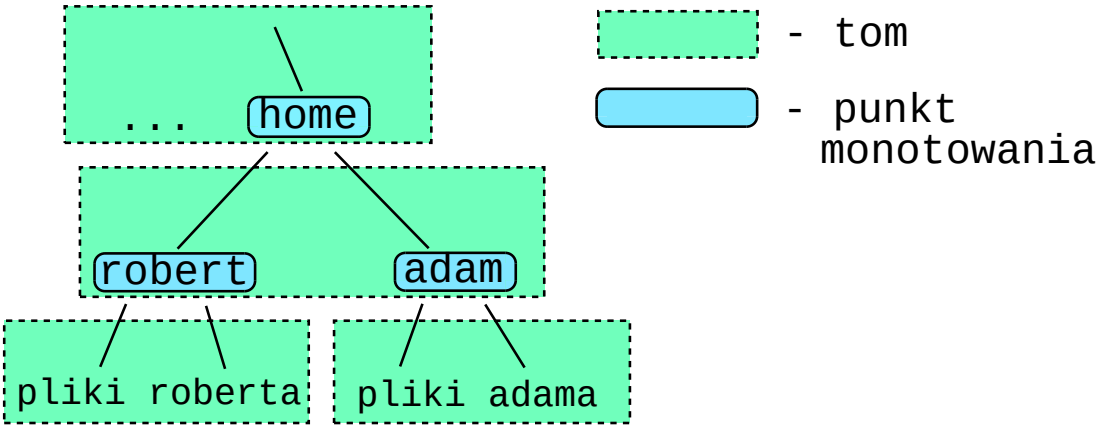
W systemie *AFS* z mechanizmem obietnic powiadomienia (czyli od wersji 2.0) uzyskano następującą gwarancję aktualności (dla operacji otwarcia pliku):

- plik jest aktualny
- komunikat z zawiadomieniem (*callback break*) został zagubiony w ciągu ostatnich *T* sekund, plik znajduje się w *cache'u* i plik jest przestarzały o nie więcej niż *T* sekund.

T to stała systemowa, która mówi po jakim czasie obietnice powiadomień muszą być odnowione.

1.7 Tomy

Jedną z ważniejszych idei w *AFS* jest pogrupowanie plików w tomy (*volumes*). Tom jest w pewnym sensie podobny do katalogu. Zawiera poddrzewo zbudowane z plików, katalogów oraz punktów montowania (o których za chwilę). Zawartość tomu powinna być ze sobą logicznie powiązana – bardzo dobrym przykładem jest katalog domowy jednego użytkownika.



Wspomniane punkty montowania to specjalny rodzaj plików w *AFS*. Dla użytkownika wyglądają jak katalogi, ale w rzeczywistości są dowiązaniami do tomów (tzn. zawierają identyfikator tomu). Użytkownik wędrując po drzewie katalogów może wielokrotnie przejść punkt montowania, za każdą zmianą czerpiąc dane z innego tomu, położonego być może na innym serwerze, w ogóle tego nie zauważając (przezroczystość położenia). Tomy są więc niewidoczne dla normalnie pracującego użytkownika, choć ich istnienie nie jest przed nim ukrywane – ma dostęp do poleceń określających w jakim tomie znajduje się dany plik, a nawet na jakim serwerze dany tom leży.

Z punktu widzenia użytkownika ma znaczenie fakt, że rozmiar tomu jest ograniczony (*quota*). Rozmiar ustala administrator. Umożliwia to mu proste ograniczenie rozmiaru katalogu domowego każdego użytkownika. Rozmiar tomu nie powinien być zbyt duży, ponieważ nie może się on musi się on w całości zmieścić w partycji.

Każdym tomem „opiekuje się” jeden serwer. Ponieważ w punktach montowanie przechowuje się nazwy, a nie położenie tomu, nie ma znaczenia gdzie się on fizycznie znajduje. Tomy mogą być zatem przenoszone między partycjami serwera oraz między serwerami. W *AFS* jest to bardzo proste i wygodne, dokonuje się w czasie normalnej pracy systemu, a przenoszony tom jest niedostępny tylko przez chwilę.

Co więcej, tomy mogą być replikowane. Znaczy to tyle, że kopia tomu zawierającego popularne pliki może się znajdować na wielu serwerach. Pozwala to wydatnie zmniejszyć obciążenie serwera przechowującego oryginał. Oryginał jest jedyną wersją modyfikowalną, kopie są tylko do odczytu, co może być pewną niedogodnością, ale takie rozwiązanie wynika z założeń projektowych (obserwacje o typowym dostępie do plików). Zmiana oryginału powoduje konieczność zmiany kopii znajdujących się na innych serwerach – *AFS*potrafi zrobić to automatycznie.

W *AFS* istnieją narzędzia służące do robienia kopii zapasowych i one również działają na poziomie tomów.

1.8 Awarie

Awaria klienta (*Cache Manager*) uniemożliwia dostęp do *AFS*użytkownikowi siedzącemu przy danej maszynie klienckiej. Dodatkowo, po ponownym uruchomieniu *Cache Manager* dla bezpieczeństwa zakłada, że wszystkie pliki znajdujące się w *cache'u* zostały unieważnione, ponieważ mogło dojść do utraty komunikatów *callback*.

Na czas awarii serwera niedostępne stają się tomy, które się na nim znajdują. W szczególności:

- Pliki zawarte w tomach, które są kopiami, są nadal dostępne, spada jedynie wydajność.
- Plików zawartych w tomach zapisywalnych, ale posiadających repliki, nie można pobrać do zapisu. *Cache Manager* nie jest w stanie zarządać zapisania nowej wersji pliku stworzonej przez jakiegoś klienta.
- Pliki zawarte w tomach niezreplikowanych stają się niedostępne.

Dodatkowo niemożliwe staje się namierzenie po ścieżce dostępu plików zawartych w tomach dostępnych, jeśli ścieżka dostępu przechodzi przez punkty montowania tomów niedostępnych w wyniku awarii.

1.9 Bezpieczeństwo

Bezpieczeństwo do jeden z ważniejszych wymogów stawianych przez jakimkolwiek systemem rozproszonym. Wynika to z natury rozproszenia – jedną z jego warstw jest sieć, czyli nośnik bardzo wrażliwy na podgląd przekazywanych informacji. Jeśli nie poczyni się odpowiednich zabezpieczeń, bardzo łatwo można się podszyć pod użytkownika, czy program.

Użytkownicy systemu *AFS* mogą zabezpieczać swoje pliki przed niepożądanym dostępem innych użytkowników. Oczywiście, aby sens miało pojęcie „swoje" pliki, użytkownik musi mieć możliwość zalogowania do systemu. Jak wspomniano wcześniej użytkownik posiada konto w ramach komórki. Dostęp do niego jest chroniony hasłem. Po podaniu właściwego hasła użytkownik staje się autentyfikowany, a jego *Cache Manager* otrzymuje *token* – jest on dowodem na to, że użytkownik jest tym, za kogo się podaje.

Wzajemna autentyfikacja Jednym z wymogów bezpieczeństwa jest, aby wymiana informacji (tu: operacji na plikach i samych plików) odbywała się tylko między prawdziwym serwerem i prawdziwym klientem. W tym celu, każdą wymianę danych poprzedza wzajemna autentyfikacja (*mutual authentication*) serwera i klienta (lub ewentualnie dwóch serwerów). Idea tego procesu polega na tym, że serwer i klient posiadają sobie tylko znany „sekret". W czasie komunikacji muszą obie strony wykazać, że go znają.

Wzajemna autentyfikacja jest wykorzystywana podczas logowania do komórki. „Sekretem” jest wówczas hasło użytkownika.

Skutkiem ubocznym implementacji wzajemnej autentyfikacji w *AFS*, ale jednocześnie celem samym w sobie, jest zakodowanie przesyłanej wiadomości, tak, że osoba śledząca transmisję danych nie może tych danych odczytać.

ACL *Access Control Lists*, czyli *ACL* służą do definiowania dostępu do plików różnym grupom użytkowników. Są zorganizowane inaczej niż pełniące w lokalnym systemie plików Unix tę samą funkcję pola bitowe.

Pierwszą różnicą jest to, że *ACL* jest związany z katalogiem, a nie plikiem, i dotyczy wszystkich plików w nim się znajdujących. Przy tworzeniu podkatalogu w jakimś katalogu *ACL* nowo utworzonego pliku jest kopiowany do rodzica. Można go oczywiście później zmienić.

Tak jak w Unixie są trzy grupy, dla których można określać prawa dostępu (*owner*, *group*, *others*), tak *ACL* udostępnia do definiowanie praw dostępu dla 20 różnych bytów: użytkowników, grup lub, co ciekawe, maszyn. Same grupy funkcjonują trochę inaczej niż w Unixie, ponieważ może je zakładać każdy użytkownik, a nie tylko administrator. Standardowo zdefiniowane są trzy grupy użytkowników:

system:anyuser	wszyscy użytkownicy, w tym niezalogowani (odpowiednik *anonymous*)
system:authuser	wszyscy użytkownicy zalogowani
system:administrators	administratorzy; mają prawo ’a’ do każdego katalogu

Inne są także same prawa, które można nadawać. Jest ich aż 7:

	dotyczące katalogu	
l	*lookup*	pozwala obejrzeć zawartość katalogu
i	*insert*	wstawianie pliku do katalogu
d	*delete*	usuwanie pliku z katalogu
a	*administer*	zmiana *ACL*
	dotyczące zawartych w nim plików	
r	*read*	czytanie
w	*write*	pisanie
k	*lock*	zakładanie blokady

Prawa te można nadawać normalnie (jak w Unixie) i negatywnie – odbierając dane prawo użytkownikowi/grupie/maszynie.

1.10 RPC

AFS wykorzystuje *RPC* (*Remote Procedue Call*). Dzięki temu jest niezależny od konkretnego protokołu sieciowego, takiego jak np. *TCP/IP*. *RPC* jest szerzej omówione w ramach wykładu z SO oraz w prezentacji o NFS.

2 Coda

2.1 Wprowadzenie i założenie projetkowe

Rozproszony system plików *Coda* powstał na bazie *AFS*. Projektanci *Coda*[8] stworzyli rozproszony system plików wykorzystując większość dobrych rozwiązań *AFS*, a zmiany wprowadzając w miejscach, które po kilku latach używania *AFS* na CMU okazały się jego słabymi stronami. Są to głównie: słabe tolerowanie uszkodzeń, a nawet planowych wyłączeń serwerów oraz „wąskie gardło" w postaci tylko jednej zapisywalnej wersji tomu.

Pojawiło się też nowe założenie projektowe, a mianowicie praca w odłączeniu od sieci. Powstało z myślą o komputerach przenośnych i mówi tyle, że po odłączeniu komputera od sieci użytkownik ma mieć możliwość normalnej pracy, czyli mieć dostęp do plików zdalnych, których będzie chciał używać.

Ponieważ *Coda* jest bardzo podobna do *AFS* , omówię tylko rzeczy nowe, bądź inaczej zrealizowane.

2.2 Operacja odłączenia

Możliwa jest sytuacja, w której wszystkie serwery są niedostępne – gdy wszystkie ulegną awarii lub padnie sieć. Z drugiej strony całkowite odłączenie od sieci może być celowo spowodowane przez użytkownika, który np. zabiera komputer przenośny na weekend do domu. *Coda* nawet w takiej sytuacji ma umożliwiać użytkownikowi pracę na plikach rozproszonych. Istnieje specjalna operacja odłączenia (którą można wywołać odpowiednią komendą), przełączająca *Venus* w tryb odłączony.

Żeby praca była wówczas możliwa trzeba zagwarantować, by wszystkie pliki, których użytkownik będzie używał podczas odłączenia, znajdowały się w *cache'u*. W ogólności nie jest to oczywiście możliwe, ale *Coda* stara się minimalizować liczbę odwołań do plików nieobecnych w *cache'u* (niedostępnych w czasie odłączenia). Konieczna jest do tego współpraca z użytkownikiem, który wskaże, które pliki są mu potrzebne. Odbywa się to w procesie nazywanym *hoarding*. Użytkownik podaje *Venus* listę plików wraz z priorytetami. Im wyższy priorytet pliku, tym mniejsza szansa, że zostanie usunięty z *cache'u*, a zatem tym większa, że będzie on dostępny podczas odłączenia.

Kiedy po odłączeniu następuje ponowne połączenie z siecią, *Venus* automatycznie integruje pliki z *cache'u* z serwerami – zapisuje pliki zmodyfikowane, etc. Jeżeli operacja ta się nie powiedzie, *Coda* informuje użytkownika, o błędach, które uniemożliwiły integrację – potrzebna jest wówczas ręczna ingerencja użytkownika.

Operacja odłączenia jest też użyteczna, gdy chwilowo sieć wolno działa.

[8] te same osoby, które projektowały *AFS*

2.3 Zwielokrotnienie tomów zapisywalnych

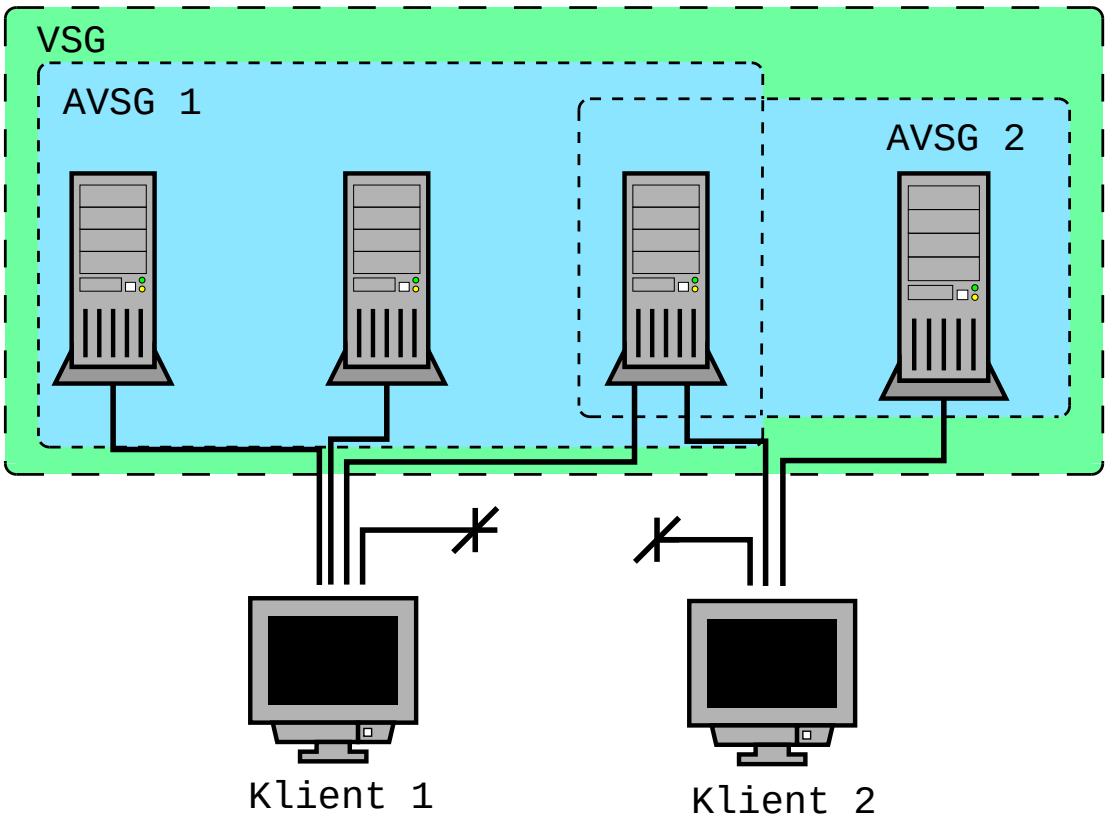
W *AFS* tom zapisywalny mógł mieć kopie, ale tylko do odczytu. *Coda* wprowadza prawdziwe zwielokrotnienie tomów zapisywalnych, co zwiększa dostępność plików, zarówno poprzez zwiększenie wydajności, jak i większą odporność na awarie. Płaci się za to możliwością powstania rozbieżności w wersjach tego samego pliku przechowywanego na kilku serwerach, których rozwiązanie wymaga ingerencji użytkownika lub administratora oraz bardziej skomplikowanym mechanizmem ściągania pliku przez *Venus* – jest on opisany poniżej.

2.4 VSG i AVSG

W systemie *Coda*wprowadzono następujące pojęcia:

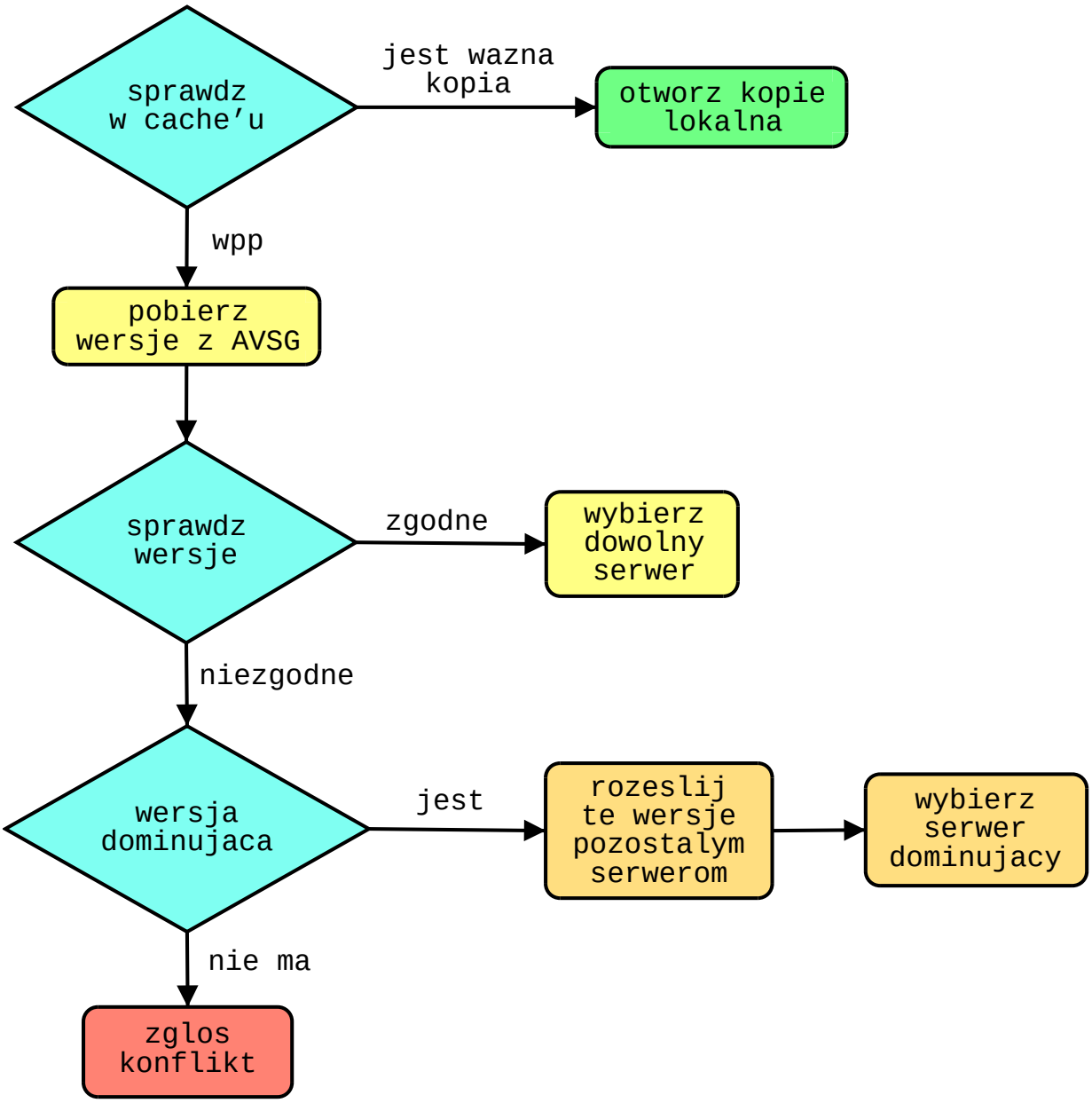
VSG – *Volume Storage Group* – zbiór serwerów przechowujących dany tom. Jest to pojęcie globalne.

AVSG – *Available Storage Group* – zbiór dostępnych serwerów przechowujących dany tom. Jest to pojęcie lokalne dla klienta.



Korzystając z pojęcia AVSG *można powiedzieć, że gdy operacja odłączenia ma miejsce,* AVSG *dla wszystkich tomów stają się puste.*

Otwierając plik nieobecny w *cache'u Venus* wysyła do wszystkich serwerów z *VSG* zapytanie o wersję pliku, którą posiadają. Serwery, które odpowiedzą to *AVSG*. Jeśli jest zgodność co do wersji pliku *Cache Manager* wybiera dowolny serwer. Jeśli są jakieś różnice *Cache Manager* wybiera serwer spośród posiadających najnowszą wersję, a do nieposiadających rozsyła informację, że mają przestarzała wersję pliku. Plik jest ściągany z wybranego serwera i tylko ten serwer daje obietnicę zawiadomienia.



Przy porównywaniu wersji może okazać się, że występuje konflikt. Oznacza to tyle, że nie da się automatycznie znaleźć najnowszej wersji. Operacja otwierania kończy się wówczas niepowodzeniem, a odkryty konflikt jest zgłaszany użytkownikowi.

W momencie zamknięcia pliku, jeśli użytkownik wprowadził w nim zmiany, nowa wersja jest rozsyłana do wszystkich serwerów z *AVSG*. Jeżeli *AVSG* nie jest równe *VSG* nie wszystkie serwery dostają nową wersję pliku.

2.5 Wersje pliku

Z każdym plikiem związany jest znacznik czasu oraz *CVV – Coda Version Vector*. Jest to wektor o długości równej liczności zbioru *VSG* złożony z liczb całkowitych. Każdemu elementowi wektora odpowiada jeden serwer, a liczba w tym elemencie szacuje liczbę zmian wersji pliku na tym serwerze.

CVV jest używany do automatycznego rozwiązywania niespójności wersji na serwerach. Jeśli dla jakiegoś pliku jego *CVV* na jednym serwerze „dominuje" nad jego *CVV* na drugim serwerze, to wersja z pierwszego serwera jest nowsza. „Dominacja" oznacza, że wszystkie elementy pierwszego wektora są nie mniejsze niż odpowiadające im elementy drugiego wektora (np. $[2,2,3] > [1,2,2]$). Jeśli żaden wektor nie dominuje na drugim (np. $[1,2,3]$ i $[2,2,2]$), konieczna jest ingerencja użytkownika bądź administratora – różne wersje pliku są umieszczane w katalogu podobnym do unixowego *lost+found*, w tomie uzupełniającym (*covolume*) związanym z każdym tomem.

Rozbieżność wersji może wystąpić w wyniku rozspójnienia sieci. Jeżeli *AVSG* jednego użytkownika zmieniającego plik jest różne od *AVSG* innego, zmieniającego ten sam plik, to powstanie rozbieżność wersji.

2.6 Semantyka aktualizacji

W systemie *Coda* uzyskano następujące gwarancje aktualności:

- Udane otwarcie:
  AVSG jest niepuste i plik jest aktualny
  lub
  AVSG jest niepuste, plik jest przestarzały o co najwyżej *T* sekund i jest w *cache'u* oraz przez ostatnie *T* sekund stracono zawiadomienia
  lub
  AVSG jest puste i plik jest w *cache'u*.

- Nieudane otwarcie:
  AVSG jest niepuste i są konflikty

lub
AVSG jest puste i pliku nie ma w *cache'u*.

- Udane zamknięcie:
  AVSG jest niepuste i plik pomyślnie zaktualizowano
  lub
  AVSG jest puste
- Nieudane zamknięcie:
  AVSG jest niepuste i wykryto konflikty

Aby powyższe gwarancje dało się zrealizować, *Venus* musi po czasie co najwyżej *T* wykrywać:

- powiększenie *AVSG*
- pomniejszenie *AVSG*
- utratę zawiadomień

Uzyskuje to, dzięki rozsyłaniu co *T* sekund komunikatu próbnego do serwerów w *VSG*, dla każdego pliku przechowywanego w *cache'u*. Odpowiadają na niego tylko serwery z aktualnego *AVSG*. Aby wykryć utratę zawiadomień w odpowiedź zawiera *CVV*. Wykrycie niezgodności w *CVV* pochodzących do różnych serwerów oznacza, że któryś z nich posiadają wersje pliku nieaktualne. *Venus* w tym momencie unieważnia obietnice zawiadomienia dotyczące takich plików.

2.7 Przechowywanie plików

Same pliki są zarówno na serwerze, jak i u klienta, przechowywane w lokalnym systemie plików Unix.

Metadane natomiast są trzymane w specjalnych plikach lub partycjach (*RVM*), gdzie aktualizacja odbywa się na zasadzie transakcji – nie ma możliwości częściowego (czyli niepoprawnego) zaktualizowania danych. Metadane to obietnice zawiadomień, *ACL*, *CVV*, etc.

2.8 Zmiany w jądrze

W systemie Unix częścią jądra odpowiedzialną za obsługę żądań dotyczącą plików jest *VFS – Virtual File System*. *Coda*, czy raczej jej podsystem wkompliowany w jądro, współpracuje z *VFS* w sposób analogiczny do *NFS*. Jest to dokładnie opisane w ramach wykładu z SO oraz prezentacji o *NFS*, zatem nie będę się wgłębiał w tę tematykę.

Opisany tu sposób przechowywania plików oraz zmiany wprowadzone w jądrze najprawdopodobniej wyglądają podobnie w AFS*, ale nie udało mi się znaleźć konkretów na ten temat.*