Mariusz Zaborski

ZFS – rewolucja w systemach plików

Zapisując lub odczytując dane z dysku twardego, oczekujemy, że dana operacja wykona się prawidłowo i do tego w krótkim czasie. Niezbędnym komponentem systemu operacyjnego, który m.in. dba o integralność danych, jak i krótki czas wykonania zadań, jest system plików. Często nie jesteśmy świadomi problemów, jakie są z nim związane, oraz funkcjonalności, które nam oferuje. W artykule przyjrzymy się ZFSowi, który zrewolucjonizował architekturę systemów plików. Przede wszystkim postaramy się odpowiedzieć na następujące pytania: Co w ZFS jest takiego wyjątkowego? Po co nam kolejny system plików? Jak go używać? I najważniejsze – czy możemy jakoś wykorzystać jego funkcjonalności do własnych celów?

WSTĘP

ZFS to system plików zaprezentowany przez firmę SUN Microsystems w 2005 roku. Jego autorzy twierdzą, że porzucili 20 lat historii i założeń dotyczących systemów plików, a następnie zbudowali zupełnie nowy od podstaw. Jednym z pierwotnych założeń ZFSa było umożliwienie przechowywania aż do zetabajta danych (1 miliarda terabajtów) – stąd też wzięła się jego nazwa – Zettabyte Filesystem, w skrócie ZFS. Obecnie pozwala on na obsługę woluminów o rozmiarze do 256 zetabajtów. ZFS zrewolucjonizował architekturę systemów plików, łącząc ze sobą menadżer woluminów oraz system plików oraz wprowadzając wiele ciekawych funkcjonalności takich jak *Copy-on-write* czy *snapshoty*.

Omawiany system plików jest szeroko wykorzystywany w różnego typu rozwiązaniach serwerowych, gdzie liczy się niezawodność w dostępie do danych. Jeden z najpopularniejszych otwartych systemów NAS[1] – FreeNAS – bazuje właśnie na nim.

ZFS kryje w sobie wiele ciekawych i unikalnych funkcjonalności, które wcześniej nie były wykorzystywane w systemach plików. Niniejszy artykuł przedstawia kilka najciekawszych z nich, a także opisuje API ZFSa dostępne dla programistów. Na końcu poruszymy kilka problemów, które związane są z jego architekturą.

ŚRODOWISKO PRACY

Większość przykładów w niniejszym artykule będzie ZFS odnosić się do systemu FreeBSD, który w standardowej instalacji dostarcza narzędzia do zarządzania ZFSem – *zpool(8)* oraz *zfs(8)*. Jednakże nic nie stoi na przeszkodzie, aby korzystać z ZFSa na innych systemach **nixowych*. Istnieje projekt „ZFS on Linux", który dostarcza wsparcie dla systemów z jądrem Linuksa (http://zfsonlinux.org/). Aby korzystać z ZFS na systemach z rodzinny Ubuntu, wystarczy wykonać polecenia:

```
# apt-get install python-software-properties
# apt-add-repository --yes ppa:zfs-native/stable
# apt-get update
# apt-get install ubuntu-zfs
```

Ponadto istnieje także port ZFS dla Mac OS, którego można pobrać ze strony https://code.google.com/p/maczfs/. Dodatkowo z ZFS można korzystać na systemach z rodzinny Solaris, Illumos oraz NetBSD.

1 Systemy NAS (Network-attached storage) służą do udostępniania zasobów w sieciach komputerowych przy wykorzystaniu takich protokołów jak iSCSI, NFS czy SMB.

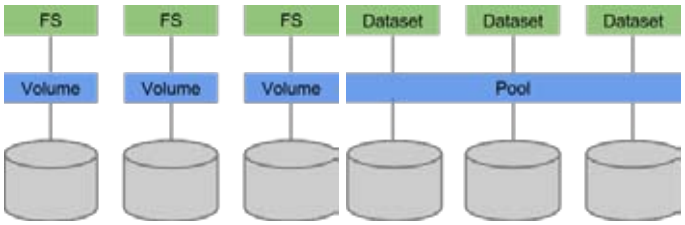
WŁAŚCIWOŚCI ZFS

Połączenie menadżera woluminów oraz systemu plików

Głównym zadaniem systemu plików (ang. *file system*, FS) jest dostarczenie użytkownikowi funkcji służących do tworzenia, usuwania, otwierania, czytania, zapisywania oraz rozszerzania plików na dysku. Natomiast celem menadżera woluminów (ang. *logical volume manager*, LVM) jest grupowanie wielu urządzeń blokowych (dysków) ze sobą, w taki sposób, aby można było utworzyć system plików wykorzystujący więcej niż jeden dysk. ZFS jako pierwszy połączył te dwie funkcjonalności ze sobą.

Ze względu na to innowacyjne podejście w ZFSie najpierw tworzymy *poole*, które grupują ze sobą dyski, a następnie na nich tworzymy *datasety*, które dzielą między sobą przestrzeń dyskową *poola*. W tradycyjnych systemach plików z LVMem określaliśmy konkretnie, które sektory mają należeć do którego FS, natomiast w ZFS przydział sektorów odbywa się dynamicznie w obrębie *poola*. Możemy na przykład utworzyć *pool*, który ma 10GB przestrzeni dyskowej i dowolną ilość *datasetów*, z czego każdy będzie miał 10GB wolnej przestrzeni. Wraz z użyciem pamięci na jednym z *datasetów*, ilość dostępnej pamięci pozostałych również będzie zmniejszona. Dodatkowo istnieje wbudowany mechanizm *quota* w ZFS, dzięki któremu możemy ograniczyć rozrost danych w danym *datasecie*. Na Rysunku 1 zostały przedstawione dwa różne podejścia w zarządzaniu dyskami. W pierwszej metodzie dla każdego dysku tworzymy oddzielne LVM oraz FS. W tradycyjnym podejściu nawet gdybyśmy LVMem połączyli kilka dysków, obszar zajmowany przez FS byłby statyczny. W ZFS (druga metoda) łączymy wszystkie dyski w jeden ogromny *pool*, a następnie dodajemy do niego odpowiednie *datasety*.

Dzięki połączeniu menadżera woluminów oraz systemu plików w ZFS otrzymujemy kilka dodatkowych korzyści. Między innymi w bardzo łatwy sposób możemy powiększać naszą przestrzeń dyskową bez potrzeby zmieniania rozmiaru systemu plików – wystarczy dodać nowy dysk do istniejącego *poola*, a wszystkie *datasety* otrzymają dodatkową przestrzeń. Co więcej połączenie menadżera woluminów oraz systemu plików skraca czas odzyskiwania danych, gdy zostanie uszkodzony jeden z dysków w macierzy RAID. Wynika to z tego, że ZFS wie, które konkretnie sektory są aktualnie używane i dzięki temu może zreplikowanać je na nowy dysk. W tradycyjnym podejściu, gdy LVM oraz FS są rozdzielone, wymagałoby to skopiowania wszystkich sektorów, nawet jeżeli nie są one aktualnie używane. W Ramce 1 zostały pokazane kroki, które należy wykonać, aby utworzyć partycję opartą o ZFS dla systemu Linux oraz FreeBSD.



Rysunek 1. Diagram przedstawiający różnicę pomiędzy podejściem zarządzania dyskami tradycyjną metodą a ZFSem. Tradycyjna metoda (LVM + FS) – po lewej i ZFS Pool po prawej

GNU/Linux

0. Utworzenie partycji wykonujemy przy użyciu komendy na przykład „cfdisk". Tworząc partycję „sdb1", wybieramy typ bf (Solaris) i wyłączamy opcję boot. Następnie zapisujemy zmiany

1. Stworzenie poola
```
# zpool create zdata /dev/sdb1
```

2. Utworzenie pierwszego *datasetu*
```
# zfs create zdata/data
```

FreeBSD

0. Usunięcie istniejącego schematu z dysku
```
# gpart destroy -F da0
```

1. Utworzenie schematu gpt na dysku
```
# gpart create -s gpt da0
```

2. Utworzenie partycji ZFSa
```
# gpart add -t freebsd-zfs da0
```

3. Stworzenie poola
```
# zpool create zdata /dev/da0p1
```

4. Utworzenie pierwszego *datasetu*
```
# zfs create zdata/data
```

Ramka 1. Przedstawienie krok po kroku tworzenia partycji z ZFSem dla systemu GNU/Linux dla dysku o nazwie „sdb" oraz FreeBSD dla dysku o nazwie „dao"

Copy-on-write

ZFS wykorzystuje model *Copy-on-write* (COW), co oznacza, że nie nadpisuje on „żywych danych". Gdy chcemy zapisać dane na dysk, są one zapisywane w nowych sektorach. W następnym kroku w nowym miejscu zapisywane są metadane. Na końcu zostaje wykonany zapis aktualizujący wskaźnik na dane, dzięki czemu nowe dane są widoczne w systemie, a stare bloki zostają oznaczone jako wolne. Dzięki zastosowaniu tego modelu, ZFS zapewnia nam także transakcyjność operacji. Jeżeli nastąpi awaria systemu, w momencie zapisu, pod wskaźnikiem wciąż znajdują się stare dane. Nie ma momentu, w którym używa się danych, które zostały tylko w pewnej części zapisane na dysk.

W tradycyjnych systemach jednym ze sposobów zapewnienia spójności danych na dysku jest tak zwany mechanizm dzienników (ang. *journals*). Przykładem systemów plików, które wykorzystują dzienniki, są ext3 bądź UFS z SU+J. Dzienniki przechowują metadane na temat czynności, które zostają przeprowadzone na systemie plików. Wpis do dziennika odbywa się przed wykonaniem faktycznej operacji. W przypadku braku zasilania bądź błędu systemu operacyjnego dzienniki umożliwiają kontynuację przerwanej operacji. Jeżeli działanie systemu zostało przerwane w trakcie tworzenia wpisu do dziennika, wpis taki zostaje usunięty. W ten sposób dzienniki dbają o integralność danych.

Wiele systemów plików dostarcza także programy służące do sprawdzania integralności danych, zwane fsck (ang. *file system consistency check*). Jednakże schemat ich działania wykracza poza zakres tego artykułu.

ZFS, dzięki modelowi COW, dba o integralność danych, przez co dzienniki nie są wykorzystywane w ten sposób, a narzędzia służące do sprawdzania integralności systemu plików nie są potrzebne.

Snapshoty i klony

Snapshoty w ZFSie to kopie tylko do odczytu wybranego *datasetu*. Tworzenie *snapshota*, dzięki modelowi COW, jest bardzo tanie. Gdy robiony jest *snapshot,* ZFS zapamiętuje tylko czas, w którym zostaje on stworzony. W momencie, w którym zapisujemy dane na dysku, są one umieszczane w nowym miejscu, natomiast stare dane pozostają bez zmian (model COW). Następnie ZFS sprawdza, czy czas utworzenia (ang. *birth time*) starych bloków jest mniejszy czy większy od czasu zapisanego przy ostatniem *snapshocie.* Jeżeli czas utworzenia bloku jest większy, blok zostaje zwolniony. Jeżeli czas jest mniejszy, dany blok zostaje dodany do listy bloków powiązanej ze *snapshotem* (ang. *list of dead blocks*). W momencie usunięcia *snapshota*, bloki zostają przepięte do innej listy bądź zwolnione.

Snapshoty umożliwiają nam cofnięcie stanu aktualnego *datasetu.* Można je wykorzystać na przykład w momencie aktualizacji systemu operacyjnego do nowszej wersji. Jeżeli jakaś funkcjonalność w systemie przestanie działać, można po prostu cofnąć stan dysków do wersji sprzed aktualizacji. Klony, w odróżnieniu od *snapshotów*, są kopiami *datasetu* z możliwością zapisu nowych danych.

Do tworzenia *snapshotów* wykorzystuje się polecenie `zfs snapshot`. Jako argument komendy podaje się nazwę *poola*, *datasetu* oraz nazwę *snapshota* w postaci *nazwapool/nazwadataset@nazwasnapshota*. W celu utworzenia *snapshota* o nazwie *new* na *datasecie users* i *poolu zdata* należy wykonać polecenie:

```
# zfs snapshot zdata/users@new
```

Jeżeli posiadalibyśmy bardziej złożoną hierarchiczną strukturę (na przykład taką jak na Listingu 1, w której nasz *dataset* ma kilku potomków), możemy utworzyć *snapshota* rekurencyjnie, tzn. utworzyć *snapshota* dla każdego potomka. Służy do tego opcja '-r' w poleceniu `zfs snapshot`

```
# zfs snapshot -r zdata/users@new
```

W celu wyświetlenia istniejących *snapshot*ów należy użyć komendy:

```
# zfs list -t snapshot
```

Komenda `zfs rollback` umożliwia nam przywrócenie *datasetu* do stanu, w którym wybrany *snapshot* został stworzony. Domyślnie komenda ta pozwala przywrócić stan tylko z ostatnio stworzonego *snapshota*. Jeżeli chcielibyśmy cofnąć się dalej, należy usunąć wszystkie późniejsze *snapshoty*. Alternatywnym wyjściem jest stworzenie nowego *datasetu* na podstawie *snapshota* (przy użyciu komendy `zfs clone`).

Listing 1. Wynik działania komendy zfs list. Dataset users zawiera trzy potomne datasety

```
$ zfs list
NAME                    USED  AVAIL  REFER  MOUNTPOINT
zdata/users             576K  100G   144K   /home/
zdata/users/a           144K  100G   144K   /home/a
zdata/users/b           144K  100G   144K   /home/b
zdata/users/c           144K  100G   144K   /home/c
```

Kompresja danych

ZFS umożliwia kompresję danych. Każdy *dataset* może używać innego algorytmu bądź w ogóle może nie kompresować danych. Warto zauważyć, że kompresja odbywa się na poziomie systemu plików, przez co jest ona całko-

wicie niezauważalna przez użytkownika końcowego (np. aplikację). Do wyboru mamy trzy algorytmy:

- lz4[2]
- lzjb
- gzip[3]

Korzystanie z kompresji umożlwia nam dużą oszczędność miejsca. Jeżeli wiemy, że *dataset* będzie zawierał dane, które dobrze się kompresują (np. logi systemu operacyjnego), możemy uruchomić na takim *datasecie* jeden z algorytmów.

Włączenie kompresji danych przy tworzeniu *datasetu* wykonujemy poprzez dodanie opcji `compression`:

```
# zfs create -o compression=lzjb zdata/data
```

Jeżeli utworzyliśmy już *dataset* i chcemy włączyć na nim kompresję dla nowych danych, możemy to zrobić za pomocą komendy:

```
# zfs set compression=lz4 zdata/data
```

Deduplikacja

Kolejną techniką oszczędzania miejsca w ZFS jest deduplikacja. Technika ta polega na eliminowaniu powtarzających się danych poprzez zastępowanie ich wskaźnikami do jednej kopii. Istnieją trzy typy deduplikacji – na poziomie plików, bloków, bądź bajtów. W ZFSie wykorzystuje się deduplikacje na poziomie bloków. Jest ona zaimplementowana za pomocą tablicy, która mapuje sumę kontrolną fragmentów danych na wskaźnik oraz licznik referencji. Kiedy zapisywane są dane, zamiast alokować nową przestrzeń na dysku, zostaje przekazana referencja do danych znajdujących się w tablicy. Przy mapowaniu używa się algorytmów o niskim prawdopodobieństwie kolizji oraz silnych kryptograficznie (np. SHA256, którego prawdopodobieństwo kolizji wynosi 10^{-77}).

Algorytm silny kryptograficznie to taki, w którym nie jesteśmy w stanie w łatwy (ani trudny) sposób wygenerować sumy kontrolnej o zadanej wartości. Założenie to wynika z bezpieczeństwa danych przechowywanych na dysku. Możemy wyobrazić sobię sytuację, w której atakujący przesyła nam spreparowane zdjęcie, a my zapisujemy je na dysku. Mogłoby się zdarzyć, że zamieścił on w nim blok kodu maszynowego odpowiadający sumie kontrolnej nowszej wersji jakiegoś popularnego programu, np. *sudo(8)*. W momencie aktualizacji systemu operacyjnego, przy włączonej deduplikacji i sprawdzaniu tylko sumy kontrolnej, nasz system plików zamiast zaktualizować program zacząłby korzystać ze wskaźników, które dostarczył nam atakujący. W momencie uruchomienia programu *sudo(8)* wykonałby się kod dostarczony przez atakującego.

Włączenie deduplikacji odbywa się przez komendę:

```
# zfs set dedup=on zdata/var/log
```

Jeżeli pomimo użycia algorytmu SHA256 obawiamy się o nasze dane, możemy uruchomić deduplikację z trybem `verify`, który spowoduje, oprócz porównania sum kontrolnych, werfikację także danych spod wskaźnika:

```
# zfs set dedup=verify zdata/var/log
```

Istnieje bardzo przyjazne narzędzie *zdb(8)*, które umożliwia analizę *poola/datasetu* w celu stwierdzenia, czy deduplikacja jest na nim opłacalana. Opis tego narzędzia wykracza jednak poza zakres niniejszego artykułu.

RAID1 oraz RAIDZ{1,2,3}

ZFS posiada wsparcie dla RAID1 (tzw. mirror), a także ma on własny typ redundancji danych zwany RAIDZ – który jest podzielony na trzy poziomy. RAIDZ1 jest podobny do RAID3, jednak został w nim wyeliminowany problem zwany „write hole". Rysunek 2 przedstawia schemat RAID3.

Najczęściej na redundantnym dysku zapisywany jest wynik operacji xor bloków danych z pozostałych dysków. W razie awarii jednego z dysków przez użycie tego operatora ponownie jesteśmy w stanie odtworzyć dane znajdujące się na brakującym dysku. Schamet zapisu na dysku w tej konfiguracji wygląda następująco:

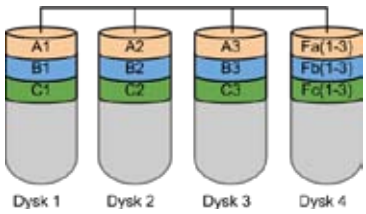
1. Odczytywana jest wartość spod bloku, który ma zostać zmieniony.
2. Odczytywany jest kod parzystości z dysku redundantnego.
3. Na odczytanych wartościach wykonywana jest operacja xor. Wynikiem tej operacji jest kod parzystości z danych znajdujących się na pozostałych dyskach,
4. Kod parzystości jest aktualizowany przez nowe dane, które mają zostać zapisane.
5. Zapisywane są nowe dane na dysk.
6. Zapisywany jest nowy kod parzystości.

Problem „write hole" występuje wtedy, gdy na dysk zostały zapisane nowe dane, ale kod parzystości z jakiegoś powodu nie został zapisany (na przykład z powodu odcięcia zasilania serwera). W tym wypadku jeżeli wystąpi awaria jednego z dysków, odzyskane dane nie będą zgodne z tym, co znajdowało się na uszkodzonym dysku. Ze względu na to, że ZFS, a co za tym idzie RAIDZ, nigdy nie nadpisuje aktywnych bloków danych, problem ten nie występuje (model COW).

Oprócz RAIDZ1 istnieją także RAIDZ2 oraz RAIDZ3, które zwiększają niezawodność. Tabela 1 porównuje właściwości poszczególnych poziomów RAIDZ.

W celu utworzenia *poola* w jednej z wymienionych konfiguracji należy wykonać komendę:

```
# zpool create data mirror/raidz{1,2,3} /dev/da0s1 ...
```



Rysunek 2. Schemat podziału danych w konfiguracji RAID3. Na dysku 4 zapisany jest kod parzystości (wynik funkcji xor)

Rodzaj macierzy	Minimalna liczba dysków	Sugerowana minimalna liczba dysków	Dyski parzystości / liczba dysków, które mogą ulec awarii	Dyski z danymi
RAIDZ1	2	3	1	n – 1
RAIDZ2	3	4	2	n – 2
RAIDZ3	4	6	3	n – 3

Tabela 1. Porównanie właściwości RAIDZ

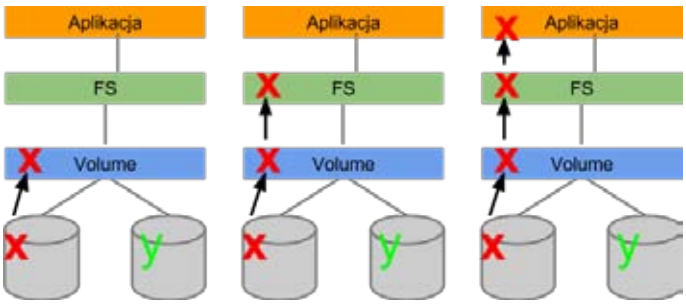
Sumy kontrolne oraz self-healing

Wszystkie dane zapisywane na dysku posiadają własną sumę kontrolną – zarówno dane aplikacji, jak i jej metadane. W trakcie odczytu danych z nośnika, ZFS przelicza sumy kontrolne i porównuje je z tymi zapisanymi na dysku. Umożliwia to wykrycie cichych uszkodzeń danych (czyli takich, które są spowodowane przez błędy urządzeń fizycznych, kable, firmware dysków itp.). Ponadto w konfiguracji RAID dzięki sumom kontrolnym oraz połączeniu menadżera woluminów i systemu plików w ZFS możliwe jest wykorzystanie funkcjonalności zwanej *self-healing*. Umożliwia ona weryfikację poprawności danych, i w przypadku wykrycia błędu – naprawienia ich na wadliwym dysku.

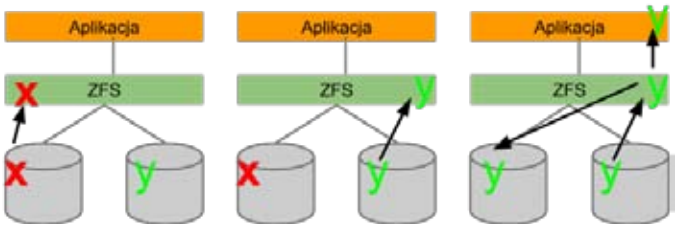
2 http://wiki.illumos.org/display/illumos/LZ4+Compression
3 http://www.gzip.org/algorithm.txt

Na Rysunku 3 został zaprezentowany schemat odczytu danych z macierzy RAID1 w tradycyjnym systemie plików. Gdy aplikacja zażąda danych z dysków, dane są odczytywane z jednego z nich. Menadżer woluminów nic nie wie o formacie danych, więc nie może zwerfikować ich poprawności. Dane zostają przekazane do systemu plików. FS mógłby na przykład sprawdzić sumy kontrolne żądanych danych, ale nie wie, że pracuje w macierzy, więc nie ma możliwośći pobrania danych z drugiego dysku. W ostateczności błędne dane zostają przekazane do aplikacji.

Na Rysunku 4 mamy przedstawiony odczyt danych z ZFSa, przy tej samej konfiguracji RAID. Ze względu na to, że ZFS wie, w jakiej macierzy pracuje, oraz ma świadomość istnienia sum kontrolnych, podczas odczytu danych werfikuje je. Jeżeli suma kontrolna jest błędna, dane są pobierane z drugiego dysku i również weryfikowane. Gdy ZFS stwierdzi, że dane są poprawne, przekazuje je do aplikacji, a błędne dane zastają zastąpione poprawnymi.



Rysunek 3. Odczyt danych z macierzy RAID 1 przy oddzielnym systemie plików i menadżerze woluminów



Rysunek 4. Zaprezentowana metoda self-healing. W momencie odczytu dane są weryfikowane i przy wykryciu nieprawidłowości naprawiane

ZVOL

W ramach *poola* istnieje możliwość stworzenia urządzenia blokowego zwanego ZVOL (ang. *ZFS Emulated Volume*). Na takim urządzeniu możemy na przykład utworzyć nowy system plików i wykorzystywać go w programach do wirtualizacji (np. w QEMU). Dodatkowo zachowujemy wszystkie pozostałe funkcjonalności, jakie oferuje nam ZFS (np. *snapshoty* czy klony). Jeżeli chcielibyśmy utworzyć ZVOLa o nazwie „windows" i wielkości 100GB, wystarczy wywołać komendę:

```
# zfs create -V 100G zdata/windows
```

Teraz nowo powstałe urządzenie możemy już sformatować i utworzyć jakiś nowy system plików. Jeżeli chcielibyśmy zamontować takie urządzenie blokowe w QEMU wystarczy uruchomić je z opcją `-hda`:

```
# qemu-system-x86_64 -hda /dev/zvol/zdata/windows
```

APLIKACJA REPLIKUJĄCA DANE

Zapoznawszy się z wieloma interesującymi właściwościami ZFS, spróbujemy teraz wykorzystać go w praktyce. Naszym zadaniem będzie implementacja aplikacji klient-serwer, która umożliwi replikację danych pomiędzy dwoma maszynami. Taki program może posłużyć na przykład do stworzenie *time machine*, którego celem jest utworzenie kopii zapasowej danych z naszego komputera.

Wiele języków, takich jak Java, Python czy Go, posiada biblioteki wspierające ZFSa. My przyjrzymy się API, jakie jest dostępne w języku C. Niestety w odróżnieniu od pozostałych języków, *libzfs* dla języka C nie ma żadnej oficjalnej dokumentacji. Powodem jest to, że *libzfs* jest uznawany za bibliotekę niestabilną, tzn. taką, której interfejsy mogą się zmieniać. Dla języka C istnieje także biblioteka *libzfs_core*, która ma wypełnić tę lukę, jednakże jest to wciąż młody projekt, nie udostępniający pełnej funkcjonalności.

Replikacja danych w ZFSie odbywa się na podstawie *snapshotów*, dlatego też zacznijmy od napisania tworzącej je funkcji. W tym celu wykorzystamy trzy funkcje API:

```
libzfs_handle_t *libzfs_init(void);
void libzfs_fini(libzfs_handle_t *hdl);
int  zfs_snapshot(libzfs_handle_t *hdl, const char *path,
boolean_t recursive, nvlist_t *props);
```

Pierwsze dwie funkcje służą odpowiednio: do inicjacji oraz do zwolnienia struktur biblioteki *libzfs*. Trzecia funkcja umożliwia nam stworzenie *snapshota* o wybranej nazwie. `nvlist_t` jest to specjalny typ z prostym API, umożliwiającym tworzenie list. Typ ten jest wykorzystywany bardzo często do przekazywania opcjonalnych argumentów. Niestety opis całego API dla *nvlist* wykracza poza zakres tego artykułu, z tego powodu też typ ten nie będzie w nim wykorzystywany. Na Listingu 2 jest pokazana pierwsza funkcja tworząca *snapshota*, którego nazwa zawiera aktualną datę. Funkcja `zfs_snapshot` umożliwia nam stworzenie także *snapshotów* rekurencyjnych (argument `recursive`).

Listing 2. Program tworzący snapshota na datasecie zdata/data o nazwie zawierającej aktualną datę

```
#define ZFS_SIZE 64

static int
make_snapshot(libzfs_handle_t *lzfsp, const char *dataset,
  char **sname)
{
  time_t now;
  struct tm *ptime;
  char fullname[1024];

  time(&now);
  ptime = localtime(&now);
  if (ptime == NULL)
    return (-1);

  *sname = malloc(ZFS_SIZE * sizeof(char));
  if (*sname == NULL)
    return (-1);

  if (strftime(*sname, ZFS_SIZE, "%Y%m%d%H%M%S",
    ptime) == 0) {
    goto out;
  }

  if (snprintf(fullname, sizeof(fullname), "%s@%s",
    dataset, *sname) < 0) {
    goto out;
  }

  if (zfs_snapshot(lzfsp, fullname, false,
    NULL) != 0) {
    fprintf(stderr,
      "ZFS snapshot failed: %s.\n",
      libzfs_error_description(lzfsp));
    goto out;
  }

  return (0);
out:
  free(*sname);
  *sname = NULL;
  return (-1);
}

int
main(void)
{
  libzfs_handle_t *lzfs;
  char *sname;
```

```
  lzfs = libzfs_init();
  if (lzfs == NULL)
    return (1);

  if (make_snapshot(lzfs, "zdata/data",
      &sname) == 0) {
    printf("Created snapshot: %s\n", sname);
    free(sname);
    return (0);
  }

  return (1);
}
```

Następnym krokiem jest implementacja funkcji służącej do wysyłania danych. W ZFS możemy wysyłać dane przyrostowo. Oznacza to, że nie musimy za każdym razem przesyłać całego *datasetu*, a tylko zmianę pomiędzy dwoma *snapshotami*. Do generowania strumienia danych dla ZFS służy funkcja:

```
int zfs_send(zfs_handle_t *zhp, const char *fromsnap,
  const char *tosnap, sendflags_t *flags,
  int outfd, snapfilter_cb_t filter_func, void *cb_arg,
  nvlist_t **debugnvp);
```

Dla nas interesujące z tej funkcji są trzy parametry:

- `fromsnap` – nazwa *snapshotu*, od którego ma zostać utworzony strumień danych. Argument może przyjąć wartość `NULL`, co będzie oznaczać, że nie wysyłamy danych przyrostowo.
- `tosnap` – nazwa *snapshota*, do którego strumień ma zostać utworzony.
- `outfd` – deskryptor, do którego strumień danych ma zostać zapisany. Utworzoną różnicę możemy zapisać do pliku, wysłać po różnego rodzaju gniazdach, zapisać do pamięci (*fmemopen(3)*) bądź wypisać na standardowe wyjście (`STDOUT_FILENO`).

Na Listingu 3 został zamieszczony kod klienta naszej aplikacji. Funkcja przyjmuje jako parametr deskryptor, do którego ma wysyłać dane oraz nazwę ostatniego zreplikowanego *snapshota*. Jeżeli nie zostanie podana nazwa ostatniego *snapshota*, cały *dataset* zostanie zreplikowany od zera. Replikacja odbywać się będzie co 60 sekund (*sleep(60)*). W kodzie występują jeszcze dwie nieomówione wcześniej funkcje z biblioteki:

```
zfs_handle_t *zfs_open(libzfs_handle_t *hdl, const char *path,
int types);
void zfs_close(zfs_handle_t *zhp);
```

Funkcja `zfs_open` umożliwia otworzenie wybranego *snapshota* bądź *datasetu*. W kodzie zamieszczonym na Listingu 3 funkcja ta jest wykorzystana do utworzenia uchwytu dla *datasetu*, który replikujemy (*zdata/data*). Funkcja `zfs_close` zwalnia zaalokowaną pamięć.

Listing 3. Implementacja funkcji klienta

```
static int
client(int fd, char *old)
{
  libzfs_handle_t *lzfs;
  zfs_handle_t *zhp;
  char *new;
  int ret;
  sendflags_t flags = { 0 };

  ret = -1;
  new = NULL;
  zhp = NULL;

  lzfs = libzfs_init();
  if (lzfs == NULL) {
    fprintf(stderr,
      "Unable to initialize ZFS library.\n");
    return (-1);
  }

  zhp = zfs_open(lzfs, "zdata/data", ZFS_TYPE_DATASET);
  if (zhp == NULL) {
    fprintf(stderr, "Unable to open dataset.");
    goto out;
  }

  while (true) {
    if (make_snapshot(lzfs, "zdata/data",
        &new) != 0) {
      goto out;
    }

    if (zfs_send(zhp, old, new, &flags, fd,
        NULL, 0, NULL) != 0) {
      fprintf(stderr,
        "ZFS send failed: %s.\n",
        libzfs_error_description(lzfs));
      goto out;
    }

    free(old);
    old = new;
    new = NULL;
    sleep(60);
  }

  ret = 0;
out:
  if (zhp != NULL)
    zfs_close(zhp);
  libzfs_fini(lzfs);
  free(old);
  free(new);
  return (ret);
}
```

Kolejnym etapem jest implementacja funkcji serwera, którego celem będzie przechowywanie kopii bezpieczeństwa. Do rozpakowywania strumienia danych ZFSa służy funkcja:

```
int zfs_receive(libzfs_handle_t *hdl, const char *tosnap,
recvflags_t *flags, int infd, avl_tree_t *stream_avl);
```

W parametrze `tosnap` przekazujemy nazwę *datasetu*, w którym mają znaleźć się dane. Jeżeli *dataset* o żądanej nazwie nie istnieje, zostanie on utworzony. Na Listingu 4 została zaprezentowana przykładowa implementacja funkcji serwera. Należy zwrócić także uwagę na ustawioną flagę `force`, która oznacza, że przed przyjęciem nowych danych *dataset* zostanie cofnięty do ostatniego *snapshota*. Rozwiązuje nam to problem, w którym replikowane dane byłyby zmodyfikowane na maszynie z kopią bezpieczeństwa. Gdyby ta flaga nie była ustawiona, a dane były zmodyfikowane, ZFS zgłosiłby błąd.

Listing 4. Funkcja serwera

```
static int
server(int fd)
{
  libzfs_handle_t *lzfs;
  recvflags_t flags = { 0 };
  int ret;

  ret = -1;
  lzfs = libzfs_init();
  if (lzfs == NULL) {
    fprintf(stderr,
      "Unable to initialize ZFS library.\n");
    return (-1);
  }

  flags.force = true;
  while (true) {
    if (zfs_receive(lzfs, "zdata/backup",
        &flags, fd, NULL) != 0) {
      fprintf(stderr,
        "ZFS receive failed: %s.\n",
        libzfs_error_description(lzfs));
      goto out;
    }
  }

  ret = 0;
out:
  libzfs_fini(lzfs);
  return (ret);
}
```

Następnym krokiem jest stworzenie funkcji `main` programu. Dla uproszczenia i dla testów posłużymy się *unixdomain socketem* oraz instrukcją *fork(2)* służącą do sklonowania aktualnego procesu. Testowy kod funkcji `main` możemy zna-

leźć na Listingu 5. Jeżeli chcielibyśmy, aby nasza aplikacja faktycznie działała jako *time-machine*, powinniśmy użyć protokołu TCP. Użycie tego protokołu pozwoliłoby nam na przesyłanie dnaych w sieci LAN. Dodatkowo protokół ten zapewnia, że pakiety są odbierane w kolejności wysłania. Dodatkowo funkcja z Listingu 5 nie zapamiętuje ostatniego wysłanego *snapshota*, dlatego przy ponowonym uruchomieniu aplikacji będzie ona chciała przesłać cały replikowany *dataset* z maszyny. W tym przypadku ZFS zwróci błąd, ponieważ w kopii zapasowej wciąż będą znajdowały się inne zależności (*snapshoty*), które nie są automatycznie usuwane.

Listing 5. Testowa funkcja główna

```
int
main(void)
{
  int ret, socks[2], status;
  pid_t pid;

  ret = socketpair(PF_UNIX, SOCK_STREAM, 0, socks);
  if (ret < 0) {
    fprintf(stderr,
      "Unable to create sockets.\n");
    return (1);
  }

  pid = fork();
  switch (pid) {
  case -1:
    fprintf(stderr, "fork failed.\n");
    return (1);
  case 0:   /* Child. */
    close(socks[1]);
    ret = server(socks[0]);
    break;
  default: /* Parent. */
    close(socks[0]);
    ret = client(socks[1], NULL);
    kill(pid, SIGKILL);
    wait(&status);
    break;
  }

  return (ret != 0 ? ret : status);
}
```

Dodatkową modyfikacją, jaką wykonamy dla naszej aplikacji *time-machine*, jest usuwanie z klienta starych *snapshotów*. Nasz program będzie tworzył znaczną liczbę *snapshotów*, jednakże na maszynie, z której jest tworzona kopia bezpieczeństwie, są one zbędne. W tym celu użyjemy funkcji `zfs_iter_snapshots_sorted`, która pozwoli nam przeiterować po każdym *snapshotcie* z wybranego *datasetu*.

```
int zfs_iter_snapshots_sorted(zfs_handle_t *zhp, zfs_iter_f
callback, void *data);
```

Ze względu na budowę funkcji będzimy musieli zdefiniować strukturę, która będzie zawierać nazwę aktualnego *snapshota* (tego, którego nie chcemy kasować), oraz strukturę do zarządzania otwartym *datasetem*. Do usuwania *snapshotów* posłuży nam funkcja:

```
int zfs_destroy_snaps(zfs_handle_t *zhp, char *snapname,
boolean_t defer);
```

Kod metody usuwającej *snapshoty*, za wyjątkiem podanego, oraz zdefiniowana struktura znajdują się na Listingu 6. Na Listingu 7 została zaprezentowana zaś zmiana, którą należy dodać do funkcji `client` w celu usunięcia nadmiarowych *snapshotów*.

Listing 6. Funkcja i struktura służąca do usuwania snapshotów

```
typedef struct {
  const char *snapname;
  zfs_handle_t *dataset;
} destroyarg_t;

static int
dodestroy(zfs_handle_t *snapshot, void *arg)
{
  destroyarg_t *destroyarg = (destroyarg_t *)arg;
  char *snapdate;

  if (snapshot == NULL)
    return (1);

  snapdate = strstr(zfs_get_name(snapshot), "@");
```

```
  if (snapdate == NULL)
    return (-1);

  snapdate += 1;

  if (strcmp(destroyarg->snapname, snapdate) != 0) {
    if (zfs_destroy_snaps(destroyarg->dataset,
        snapdate, false) != 0) {
      return (-1);
    }
    return (0);
  }

  return (1);
}
```

Listing 7. Zmiana w funkcji klienta usuwająca wszystkie snapshoty, poza ostatnio replikowanym

```
while (true) {
+    if (old != NULL) {
+        darg.snapname = old;
+        darg.dataset = zhp;
+        if (zfs_iter_snapshots_sorted(zhp,
+            dodestroy, &darg) == -1) {
+            goto out;
+        }
+    }

  if (make_snapshot(lzfs, "zdata/data",
      &new) != 0) {
    goto out;
  }
```

Ostatnim już krokiem jest skompilowanie aplikacji. W zależności od systemu operacyjnego proces ten będzie wyglądał trochę inaczej. ZFS jest mocno zintegrowany z FreeBSD, dlatego też wymagane jest ściągnięcie kodu źródłowego systemu operacyjnego. Na Listingu 7 został przedstawiony plik *Makefile*, który zakłada, że źródła znajdują się w katalogu */usr/src/*.

Listing 7. Plik Makefile, dla FreeBSD umożliwiający kompilację testowego programu.

```
PROG=     testzfs
NO_MAN=   1
SRCS=     testzfs.c

CFLAGS+=  -I/usr/src/cddl/contrib/opensolaris/lib/libzpool/common
CFLAGS+=  -I/usr/src/cddl/compat/opensolaris/include
CFLAGS+=  -I/usr/src/cddl/compat/opensolaris/lib/libumem
CFLAGS+=  -I/usr/src/sys/cddl/compat/opensolaris
CFLAGS+=  -I/usr/src/cddl/contrib/opensolaris/head
CFLAGS+=  -I/usr/src/cddl/contrib/opensolaris/lib/libuutil/common
CFLAGS+=  -I/usr/src/cddl/contrib/opensolaris/lib/libzfs/common
CFLAGS+=  -I/usr/src/cddl/contrib/opensolaris/lib/libumem/common
CFLAGS+=  -I/usr/src/cddl/contrib/opensolaris/lib/libnvpair
CFLAGS+=  -I/usr/src/cddl/contrib/opensolaris/lib/libzfs_core/common/
CFLAGS+=  -I/usr/src/sys/cddl/contrib/opensolaris/uts/common
CFLAGS+=  -I/usr/src/sys/cddl/contrib/opensolaris/uts/common/fs/zfs
CFLAGS+=  -I/usr/src/sys/cddl/contrib/opensolaris/uts/common/sys
CFLAGS+=  -DNEED_SOLARIS_BOOLEAN

DPADD=    ${LIBZFS} ${LIBGEOM} ${LIBBSDXML} ${LIBSBUF} \
                    ${LIBM} ${LIBNVPAIR} ${LIBUUTIL} ${LIBUTIL}
LDADD=    -lzfs -lgeom -lbsdxml -lsbuf \
                       -lm -lnvpair -luutil -lutil

.include <bsd.prog.mk>
```

Cały kod programu można znaleść na stronie: http://oshogbo.vexillium.org/codesnap/zfstest.tar. Kolejnymi modyfikacjami, które czytelnik mógłby wykonać sam, byłyby:

- znajdowanie ostatniego wysłanego *snapshota* przy użyciu funkcji `zfs_iter_snapshots_sorted`,
- obsługa błędu w przypadku wysłania niepoprawnego *snapshota* przyrostowego,
- implementacja replikacji ze *snapshotami* rekurencyjnymi,
- przepisanie aplikacji na skrypt *sheelwy* bądź inny język,
- wykorzystanie protokułu TCP do przesyłania danych,
- napisanie programu służącego do odtworzenie stanu klienta z *time-machine*.

WADY ZFS

Poza zaletami i ciekawymi funkcjonalnościami ZFS ma kilka wad wydajnościowych. Przy maszynach mainstreamowych nie odczujemy żadnej różnicy pomiędzy ZFSem a innymi systemami plików. Niestety ZFS wymaga sporej ilości pamięci RAM, dlatego też odradza się stosowanie tego systemu plików w systemach wbudowanych. Drugą wadą jest znaczące zwolnienie ZFS przy dużej zajętości dysków (80% i więcej). Niestety model COW niesie za sobą pewne konsekwencje w postaci luk pomiędzy danymi na dysku twardym, przez co odnajdywanie wolnej przestrzeni do alokacji nowych danych się wydłuża. Drugi problem można rozwiązać poprzez dodanie nowych dysków do istniejącego *poola* w celu zwiększenia ilości wolnego miejsca.

PODSUMOWANIE

Czy da się poprawić coś, co było projektowane i ulepszane przez ponad 20 lat? Okazuje się, że tak. ZFS zmienił sposób postrzegania systemów plików i to, jak wiele można z nimi zrobić. Pokazał też, że przy zastosowaniu prostych modeli można w łatwy sposób zapewnić integralność danych. Wydaje się pewne, że z roku na rok ZFS będzie zyskiwał coraz więcej na popularności wśród osób zajmujących się administracją systemów, ale także wśród programistów; szczególnie dzięki społeczności, która tworzy się wokół projektu (http://open-zfs.org/).

W artykule poruszono kilka najciekawszych funkcjonalności ZFSa, takich jak model COW, połączenie menadżera wolumenów i systemu plików czy *snapshoty*. Poznaliśmy podstawowe komendy służące do zarządzania ZFSem, takie jak *zpool(8)* i *zfs(8)*. Udało nam się także zaimplementować prosty program do replikacji danych w języku C.

Mamy nadzieję, że przedstawiony tutaj zarys ZFSa zachęci czytelnika do głębszego zapoznania się z jego możliwościami.

W sieci:

- Dokumentacja ZFS: http://docs.oracle.com/cd/E19253-01/819-5461/
- Strona społeczności ZFSa: http://open-zfs.org/
- FreeBSD handbook dotyczący ściągania źródeł: https://www.freebsd.org/doc/handbook/svn.html
- ZFS dla Linuksa: http://zfsonlinux.org/
- ZFS dla Mac OSx: https://code.google.com/p/maczfs/

Mariusz Zaborski oshogbo@vexillium.org

Programista w firmie Wheel Systems, w której zajmuje się rozwijaniem produktów związanych z bezpieczeństwem. Hobbystycznie student Politechniki Warszawskiej na wydziale Elektrycznym. W wolnym czasie zaangażowany w rozwój projektu FreeBSD.