

new/usr/src/lib/libzfs/common/libzfs_sendrecv.c

```
*****
85483 Mon Jun 17 09:37:34 2013
new/usr/src/lib/libzfs/common/libzfs_sendrecv.c
3819 zfs receive can fail due to processing order
*****  
_____ unchanged_portion_omitted _____  
687 /*  
688 * recursively generate nvlists describing datasets. See comment  
689 * for the data structure send_data_t above for description of contents  
690 * of the nvlist.  
691 */  
692 static int  
693 send_iterate_fs(zfs_handle_t *zhp, void *arg)  
694 {  
695     send_data_t *sd = arg;  
696     nvlist_t *nvfs, *nv;  
697     int rv = 0;  
698     uint64_t parent_fromsnap_guid_save = sd->parent_fromsnap_guid;  
699     uint64_t guid = zhp->zfs_dmustats.dds_guid;  
700     char guidstring[64];  
  
702     VERIFY(0 == nvlist_alloc(&nvfs, NV_UNIQUE_NAME, 0));  
703     VERIFY(0 == nvlist_add_string(nvfs, "name", zhp->zfs_name));  
704     VERIFY(0 == nvlist_add_uint64(nvfs, "parentfromsnap",  
705             sd->parent_fromsnap_guid));  
  
707     if (zhp->zfs_dmustats.dds_origin[0]) {  
708         zfs_handle_t *origin = zfs_open(zhp->zfs_hdl,  
709             zhp->zfs_dmustats.dds_origin, ZFS_TYPE_SNAPSHOT);  
710         if (origin == NULL)  
711             return (-1);  
712         VERIFY(0 == nvlist_add_uint64(nvfs, "origin",  
713             origin->zfs_dmustats.dds_guid));  
714     }  
  
716     /* iterate over props */  
717     VERIFY(0 == nvlist_alloc(&nv, NV_UNIQUE_NAME, 0));  
718     send_iterate_prop(zhp, nv);  
719     VERIFY(0 == nvlist_add_nvlist(nvfs, "props", nv));  
720     nvlist_free(nv);  
  
722     /* iterate over snaps, and set sd->parent_fromsnap_guid */  
723     sd->parent_fromsnap_guid = 0;  
724     VERIFY(0 == nvlist_alloc(&sd->parent_snaps, NV_UNIQUE_NAME, 0));  
725     VERIFY(0 == nvlist_alloc(&sd->snapprops, NV_UNIQUE_NAME, 0));  
726     (void) zfs_iter_snapshots_sorted(zhp, send_iterate_snap, sd);  
727     (void) zfs_iter_snapshots(zhp, send_iterate_snap, sd);  
728     VERIFY(0 == nvlist_add_nvlist(nvfs, "snaps", sd->parent_snaps));  
729     VERIFY(0 == nvlist_add_nvlist(nvfs, "snapprops", sd->snapprops));  
730     nvlist_free(sd->parent_snaps);  
    nvlist_free(sd->snapprops);  
  
732     /* add this fs to nvlist */  
733     (void) snprintf(guidstring, sizeof (guidstring),  
734         "0x%llx", (longlong_t)guid);  
735     VERIFY(0 == nvlist_add_nvlist(sd->fss, guidstring, nvfs));  
736     nvlist_free(nvfs);  
  
738     /* iterate over children */  
739     if (sd->recursive)  
        rv = zfs_iter_filesystems(zhp, send_iterate_fs, sd);  
  
742     sd->parent_fromsnap_guid = parent_fromsnap_guid_save;  
  
744     zfs_close(zhp);
```

1

```
new/usr/src/lib/libzfs/common/libzfs_sendrecv.c  
745         return (rv);  
746     }  
_____ unchanged_portion_omitted _____  
1934 static int  
1935 recv_incremental_replication(libzfs_handle_t *hdl, const char *tofs,  
1936     recvflags_t *flags, nvlist_t *stream_nv, avl_tree_t *stream_avl,  
1937     nvlist_t *renamed)  
1938 {  
1939     nvlist_t *local_nv, *deleted = NULL;  
1940     nvlist_t *local_nv;  
1941     avl_tree_t *local_avl;  
1942     nvpair_t *fselem, *nextfselem;  
1943     char *fromsnap;  
1943     char newname[ZFS_MAXNAMELEN], guidname[32];  
1943     char newname[ZFS_MAXNAMELEN];  
1944     int error;  
1945     boolean_t needagain, progress, recursive;  
1946     char *s1, *s2;  
  
1948     VERIFY(0 == nvlist_lookup_string(stream_nv, "fromsnap", &fromsnap));  
  
1950     recursive = (nvlist_lookup_boolean(stream_nv, "not_recursive") ==  
1951             ENOENT);  
  
1953     if (flags->dryrun)  
1954         return (0);  
  
1956 again:  
1957     needagain = progress = B_FALSE;  
1959     if ((error = gather_nvlist(hdl, tofs, fromsnap, NULL,  
1960             recursive, &local_nv, &local_avl)) != 0)  
1961         return (error);  
  
1963     deleted = fnvlist_alloc();  
1965 #endif /* ! codereview */  
1966     /*  
1967      * Process deletes and renames  
1968      */  
1969     for (fselem = nvlist_next_nvpair(local_nv, NULL);  
1970         fselem; fselem = nextfselem) {  
1971         nvlist_t *nvfs, *snaps;  
1972         nvlist_t *stream_nvfs = NULL;  
1973         nvpair_t *snapelem, *nextsnapelem;  
1974         uint64_t fromguid = 0;  
1975         uint64_t originguid = 0;  
1976         uint64_t stream_originguid = 0;  
1977         uint64_t parent_fromsnap_guid, stream_parent_fromsnap_guid;  
1978         char *fsname, *stream_fsname;  
  
1980         nextfselem = nvlist_next_nvpair(local_nv, fselem);  
  
1982         VERIFY(0 == nvpair_value_nvlist(fselem, &nvfs));  
1983         VERIFY(0 == nvlist_lookup_nvlist(nvfs, "snaps", &snaps));  
1984         VERIFY(0 == nvlist_lookup_string(nvfs, "name", &fsname));  
1985         VERIFY(0 == nvlist_lookup_uint64(nvfs, "parentfromsnap",  
1986             &parent_fromsnap_guid));  
1987         (void) nvlist_lookup_uint64(nvfs, "origin", &originguid);  
1989         /*  
1990          * First find the stream's fs, so we can check for  
1991          * a different origin (due to "zfs promote")  
1992          */  
1993         for (snapelem = nvlist_next_nvpair(snaps, NULL);
```

2

```

1994     snapelem; snapelem = nvlist_next_nvpair(snaps, snapelem) {
1995         uint64_t thisguid;
1996
1997         VERIFY(0 == nvpair_value_uint64(snapelem, &thisguid));
1998         stream_nvfs = fsavl_find(stream_avl, thisguid, NULL);
1999
2000         if (stream_nvfs != NULL)
2001             break;
2002     }
2003
2004     /* check for promote */
2005     (void) nvlist_lookup_uint64(stream_nvfs, "origin",
2006         &stream_originqid);
2007     if (stream_nvfs && originguid != stream_originqid) {
2008         switch (created_before(hdl, local_avl,
2009             stream_originqid, originguid)) {
2010             case 1:
2011                 /* promote it! */
2012                 zfs_cmd_t zc = { 0 };
2013                 nvlist_t *origin_nvfs;
2014                 char *origin_fnname;
2015
2016                 if (flags->verbose)
2017                     (void) printf("promoting %s\n", fnname);
2018
2019                 origin_nvfs = fsavl_find(local_avl, originguid,
2020                     NULL);
2021                 VERIFY(0 == nvlist_lookup_string(origin_nvfs,
2022                     "name", &origin_fnname));
2023                 (void) strlcpy(zc.zc_value, origin_fnname,
2024                     sizeof(zc.zc_value));
2025                 (void) strlcpy(zc.zc_name, fnname,
2026                     sizeof(zc.zc_name));
2027                 error = zfs_ioctl(hdl, ZFS_IOC_PROMOTE, &zc);
2028                 if (error == 0)
2029                     progress = B_TRUE;
2030                 break;
2031             default:
2032                 break;
2033             case -1:
2034                 fsavl_destroy(local_avl);
2035                 nvlist_free(local_nv);
2036                 nvlist_free(deleted);
2037             #endif /* ! codereview */
2038             #endif /* ! codereview */
2039             return (-1);
2040         }
2041         /*
2042          * We had/have the wrong origin, therefore our
2043          * list of snapshots is wrong. Need to handle
2044          * them on the next pass.
2045         */
2046         needagain = B_TRUE;
2047         continue;
2048     }
2049
2050     for (snapelem = nvlist_next_nvpair(snaps, NULL);
2051         snapelem; snapelem = nextsnapelem) {
2052         uint64_t thisguid;
2053         char *stream_snapname;
2054         nvlist_t *found, *props;
2055
2056         nextsnapelem = nvlist_next_nvpair(snaps, snapelem);
2057         VERIFY(0 == nvpair_value_uint64(snapelem, &thisguid));
2058         found = fsavl_find(stream_avl, thisguid,

```

```

2060             &stream_snapname);
2061
2062             /* check for delete */
2063             if (found == NULL) {
2064                 char name[ZFS_MAXNAMELEN];
2065
2066                 if (!flags->force)
2067                     continue;
2068
2069                 (void) snprintf(name, sizeof (name), "%s@%s",
2070                     fnname, nvpair_name(snapelem));
2071
2072                 error = recv_destroy(hdl, name,
2073                     strlen(fnname)+1, newname, flags);
2074                 if (error)
2075                     needagain = B_TRUE;
2076                 else
2077                     progress = B_TRUE;
2078                 (void) sprintf(guidname, "%llu",
2079                     (u_longlong_t)thisguid);
2080                 fnvlist_add_boolean(deleted, guidname);
2081             #endif /* ! codereview */
2082             continue;
2083         }
2084
2085         stream_nvfs = found;
2086
2087         if (0 == nvlist_lookup_nvlist(stream_nvfs, "snapprops",
2088             &props) && 0 == nvlist_lookup_nvlist(props,
2089             stream_snapname, &props)) {
2090             zfs_cmd_t zc = { 0 };
2091
2092             zc.zc_cookie = B_TRUE; /* received */
2093             (void) snprintf(zc.zc_name, sizeof(zc.zc_name),
2094                 "%s@%s", fnname, nvpair_name(snapelem));
2095             if (zcmd_write_src_nvlist(hdl, &zc,
2096                 props) == 0) {
2097                 (void) zfs_ioctl(hdl,
2098                     ZFS_IOC_SET_PROP, &zc);
2099                 zcmd_free_nvlists(&zc);
2100             }
2101         }
2102
2103         /* check for different snapname */
2104         if (strcmp(nvpair_name(snapelem),
2105             stream_snapname) != 0) {
2106             char name[ZFS_MAXNAMELEN];
2107             char tryname[ZFS_MAXNAMELEN];
2108
2109             (void) snprintf(name, sizeof (name), "%s@%s",
2110                 fnname, nvpair_name(snapelem));
2111             (void) snprintf(tryname, sizeof (name), "%s@%s",
2112                 fnname, stream_snapname);
2113
2114             error = recv_rename(hdl, name, tryname,
2115                     strlen(fnname)+1, newname, flags);
2116             if (error)
2117                 needagain = B_TRUE;
2118             else
2119                 progress = B_TRUE;
2120         }
2121
2122         if (strcmp(stream_snapname, fromsnap) == 0)
2123             fromguid = thisguid;
2124     }

```

```

2126     /* check for delete */
2127     if (stream_nvfs == NULL) {
2128         if (!flags->force)
2129             continue;
2130
2131         error = recv_destroy(hdl, fname, strlen(tofs)+1,
2132                             newname, flags);
2133         if (error)
2134             needagain = B_TRUE;
2135         else
2136             progress = B_TRUE;
2137         (void) sprintf(guidname, "%llu",
2138                         (u_longlong_t)parent_fromsnap_guid);
2139         fnvlist_add_boolean(deleted, guidname);
2140 #endif /* ! codereview */
2141         continue;
2142     }
2143
2144     if (fromguid == 0) {
2145         if (flags->verbose) {
2146             (void) printf("local fs %s does not have "
2147                         "fromsnap (%s in stream); must have "
2148                         "been deleted locally; ignoring\n",
2149                         fname, fromsnap);
2150     }
2151     continue;
2152 }
2153
2154 VERIFY(0 == nvlist_lookup_string(stream_nvfs,
2155                                 "name", &stream_fnsame));
2156 VERIFY(0 == nvlist_lookup_uint64(stream_nvfs,
2157                                 "parentfromsnap", &stream_parent_fromsnap_guid));
2158
2159 s1 = strrchr(fname, '/');
2160 s2 = strrchr(stream_fnsame, '/');
2161
2162 /*
2163 * Check if we're going to rename based on parent guid change
2164 * and the current parent guid was also deleted. If it was then
2165 * the rename will fail so avoid this and force an early retry
2166 * to determine the new parent_fromsnap_guid.
2167 */
2168 if (stream_parent_fromsnap_guid != 0 &&
2169     parent_fromsnap_guid != 0 &&
2170     stream_parent_fromsnap_guid != parent_fromsnap_guid) {
2171     (void) sprintf(guidname, "%llu",
2172                     (u_longlong_t)parent_fromsnap_guid);
2173     if (nvlist_exists(deleted, guidname)) {
2174         progress = B_TRUE;
2175         needagain = B_TRUE;
2176         goto doagain;
2177     }
2178 }
2179
2180 /*
2181 * Check for rename. If the exact receive path is specified, it
2182 * does not count as a rename, but we still need to check the
2183 * datasets beneath it.
2184 */
2185 if ((stream_parent_fromsnap_guid != 0 &&
2186     parent_fromsnap_guid != 0 &&
2187     stream_parent_fromsnap_guid != parent_fromsnap_guid) ||
2188     ((flags->isprefix || strcmp(tofs, fname) != 0) &&
2189     (s1 != NULL) && (s2 != NULL) && strcmp(s1, s2) != 0)) {
2190     nvlist_t *parent;
2191 }
```

```

2192
2193
2194     char tryname[ZFS_MAXNAMELEN];
2195
2196     parent = fsavl_find(local_avl,
2197                          stream_parent_fromsnap_guid, NULL);
2198
2199     /*
2200      * NB: parent might not be found if we used the
2201      * tosnap for stream_parent_fromsnap_guid,
2202      * because the parent is a newly-created fs;
2203      * we'll be able to rename it after we recv the
2204      * new fs.
2205     */
2206     if (parent != NULL) {
2207         char *pname;
2208
2209         VERIFY(0 == nvlist_lookup_string(parent, "name",
2210                                         &pname));
2211         (void) snprintf(tryname, sizeof (tryname),
2212                         "%s%s", pname, strrchr(stream_fnsame, '/'));
2213
2214     } else {
2215         tryname[0] = '\0';
2216         if (flags->verbose) {
2217             (void) printf("local fs %s new parent "
2218                         "not found\n", fname);
2219     }
2220
2221     newname[0] = '\0';
2222
2223     error = recv_rename(hdl, fname, tryname,
2224                         strlen(tofs)+1, newname, flags);
2225
2226     if (renamed != NULL && newname[0] != '\0') {
2227         VERIFY(0 == nvlist_add_boolean(renamed,
2228                                       newname));
2229     }
2230
2231     if (error)
2232         needagain = B_TRUE;
2233     else
2234         progress = B_TRUE;
2235 }
2236
2237 #endif /* ! codereview */
2238     fsavl_destroy(local_avl);
2239     nvlist_free(local_nv);
2240     nvlist_free(deleted);
2241 #endif /* ! codereview */
2242
2243     if (needagain && progress) {
2244         /* do another pass to fix up temporary names */
2245         if (flags->verbose)
2246             (void) printf("another pass:\n");
2247         goto again;
2248     }
2249
2250 }
2251
2252 static int
2253 zfs_receive_package(libzfs_handle_t *hdl, int fd, const char *destname,
2254                      recvflags_t *flags, dmu_replay_record_t *dr, zio_cksum_t *zc,
2255                      char **top_zfs, int cleanup_fd, uint64_t *action_handlep)
2256 {
2257     nvlist_t *stream_nv = NULL;
```

```

2258     avl_tree_t *stream_avl = NULL;
2259     char *fromsnap = NULL;
2260     char *cp;
2261     char tofs[ZFS_MAXNAMELEN];
2262     char sendfs[ZFS_MAXNAMELEN];
2263     char errbuf[1024];
2264     dmu_replay_record_t drre;
2265     int error;
2266     boolean_t anyerr = B_FALSE;
2267     boolean_t softerr = B_FALSE;
2268     boolean_t recursive;
2269
2270     (void) sprintf(errbuf, sizeof (errbuf), dgettext(TEXT_DOMAIN,
2271         "cannot receive"));
2272
2273     assert(drr->drr_type == DRR_BEGIN);
2274     assert(drr->drr_u.drr_begin.drr_magic == DMU_BACKUP_MAGIC);
2275     assert(DMU_GET_STREAM_HDRTYPE(drr->drr_u.drr_begin.drr_versioninfo) ==
2276         DMU_COMPOUNDSTREAM);
2277
2278     /*
2279      * Read in the nvlist from the stream.
2280      */
2281     if (drr->drr_payloadlen != 0) {
2282         error = recv_read_nvlist(hdl, fd, drr->drr_payloadlen,
2283             &stream_nv, flags->byteswap, zc);
2284         if (error) {
2285             error = zfs_error(hdl, EZFS_BADSTREAM, errbuf);
2286             goto out;
2287         }
2288     }
2289
2290     recursive = (nvlist_lookup_boolean(stream_nv, "not_recursive") ==
2291         ENOENT);
2292
2293     if (recursive && strchr(destname, '@')) {
2294         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
2295             "cannot specify snapshot name for multi-snapshot stream"));
2296         error = zfs_error(hdl, EZFS_BADSTREAM, errbuf);
2297         goto out;
2298     }
2299
2300     /*
2301      * Read in the end record and verify checksum.
2302      */
2303     if (0 != (error = recv_read(hdl, fd, &drre, sizeof (drre),
2304         flags->byteswap, NULL)))
2305         goto out;
2306     if (flags->byteswap) {
2307         drre.drr_type = BSWAP_32(drre.drr_type);
2308         drre.drr_u.drr_end.drr_checksum.zc_word[0] =
2309             BSWAP_64(drre.drr_u.drr_end.drr_checksum.zc_word[0]);
2310         drre.drr_u.drr_end.drr_checksum.zc_word[1] =
2311             BSWAP_64(drre.drr_u.drr_end.drr_checksum.zc_word[1]);
2312         drre.drr_u.drr_end.drr_checksum.zc_word[2] =
2313             BSWAP_64(drre.drr_u.drr_end.drr_checksum.zc_word[2]);
2314         drre.drr_u.drr_end.drr_checksum.zc_word[3] =
2315             BSWAP_64(drre.drr_u.drr_end.drr_checksum.zc_word[3]);
2316     }
2317     if (drre.drr_type != DRR_END) {
2318         error = zfs_error(hdl, EZFS_BADSTREAM, errbuf);
2319         goto out;
2320     }
2321     if (!ZIO_CHECKSUM_EQUAL(drre.drr_u.drr_end.drr_checksum, *zc)) {
2322         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
2323             "incorrect header checksum"));
2324

```

```

2324         error = zfs_error(hdl, EZFS_BADSTREAM, errbuf);
2325         goto out;
2326     }
2327
2328     (void) nvlist_lookup_string(stream_nv, "fromsnap", &fromsnap);
2329
2330     if (drr->drr_payloadlen != 0) {
2331         nvlist_t *stream_fss;
2332
2333         VERIFY(0 == nvlist_lookup_nvlist(stream_nv, "fss",
2334             &stream_fss));
2335         if ((stream_avl = fsavl_create(stream_fss)) == NULL) {
2336             zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
2337                 "couldn't allocate avl tree"));
2338             error = zfs_error(hdl, EZFS_NOMEM, errbuf);
2339             goto out;
2340         }
2341
2342         if (fromsnap != NULL) {
2343             nvlist_t *renamed = NULL;
2344             nvpair_t *pair = NULL;
2345
2346             (void) strlcpy(tofs, destname, ZFS_MAXNAMELEN);
2347             if (flags->isprefix) {
2348                 struct drr_begin *drrb = &drr->drr_u.drr_begin;
2349                 int i;
2350
2351                 if (flags->isstail) {
2352                     cp = strrchr(drrb->drr_toname, '/');
2353                     if (cp == NULL) {
2354                         (void) strlcat(tofs, "/", ZFS_MAXNAMELEN);
2355                     } else {
2356                         i = 0;
2357                     }
2358                 } else {
2359                     i = (cp - drrb->drr_toname);
2360                 }
2361             }
2362             /* zfs_receive_one() will create_parents() */
2363             (void) strlcat(tofs, &drrb->drr_toname[i],
2364                 ZFS_MAXNAMELEN);
2365             *strchr(tofs, '@') = '\0';
2366         }
2367
2368         if (recursive && !flags->dryrun && !flags->nomount) {
2369             VERIFY(0 == nvlist_alloc(&renamed,
2370                 NV_UNIQUE_NAME, 0));
2371         }
2372
2373         softerr = recv_incremental_replication(hdl, tofs, flags,
2374             stream_nv, stream_avl, renamed);
2375
2376         /* Unmount renamed filesystems before receiving. */
2377         while ((pair = nvlist_next_nvpair(renamed,
2378             pair)) != NULL) {
2379             zfs_handle_t *zhp;
2380             prop_changelist_t *clp = NULL;
2381
2382             zhp = zfs_open(hdl, nvpair_name(pair),
2383                 ZFS_TYPE_FILESYSTEM);
2384             if (zhp != NULL) {
2385                 clp = changelist_gather(zhp,
2386                     ZFS_PROP_MOUNTPOINT, 0, 0);
2387                 zfs_close(zhp);
2388                 if (clp != NULL) {
2389

```

```

2390                     softerr |=
2391                         changelist_prefix(clp);
2392                         changelist_free(clp);
2393                 }
2394             }
2395         }
2396         nvlist_free(rename);
2397     }
2398 }
2399 */
2400 /*
2401 * Get the fs specified by the first path in the stream (the top level
2402 * specified by 'zfs send') and pass it to each invocation of
2403 * zfs_receive_one().
2404 */
2405 (void) strlcpy(sendfs, drr->drr_u.drr_begin.drr_toname,
2406 ZFS_MAXNAMELEN);
2407 if ((cp = strchr(sendfs, '@')) != NULL)
2408     *cp = '\0';
2409
2410 /* Finally, receive each contained stream */
2411 do {
2412     /*
2413     * we should figure out if it has a recoverable
2414     * error, in which case do a recv_skip() and drive on.
2415     * Note, if we fail due to already having this guid,
2416     * zfs_receive_one() will take care of it (ie,
2417     * recv_skip() and return 0).
2418     */
2419     error = zfs_receive_impl(hdl, destname, flags, fd,
2420         sendfs, stream_nv, stream_avl, top_zfs, cleanup_fd,
2421         action_handlep);
2422     if (error == ENODATA) {
2423         error = 0;
2424         break;
2425     }
2426     anyerr |= error;
2427 } while (error == 0);
2428
2429 if (drr->drr_payloadlen != 0 && fromsnap != NULL) {
2430     /*
2431     * Now that we have the fs's they sent us, try the
2432     * renames again.
2433     */
2434     softerr = recv_incremental_replication(hdl, tofs, flags,
2435         stream_nv, stream_avl, NULL);
2436 }
2437
2438 out:
2439     fsavl_destroy(stream_avl);
2440     if (stream_nv)
2441         nvlist_free(stream_nv);
2442     if (softerr)
2443         error = -2;
2444     if (anyerr)
2445         error = -1;
2446     return (error);
2447 }
2448
2449 static void
2450 trunc_prop_errs(int truncated)
2451 {
2452     ASSERT(truncated != 0);
2453     if (truncated == 1)

```

```

2456             (void) fprintf(stderr, dgettext(TEXT_DOMAIN,
2457                             "1 more property could not be set\n"));
2458         else
2459             (void) fprintf(stderr, dgettext(TEXT_DOMAIN,
2460                             "%d more properties could not be set\n"), truncated);
2461     }
2462
2463     static int
2464     recv_skip(libzfs_handle_t *hdl, int fd, boolean_t byteswap)
2465     {
2466         dmu_replay_record_t *drr;
2467         void *buf = malloc(1<<20);
2468         char errbuf[1024];
2469
2470         (void) snprintf(errbuf, sizeof (errbuf), dgettext(TEXT_DOMAIN,
2471                             "cannot receive:"));
2472
2473         /* XXX would be great to use lseek if possible... */
2474         drr = buf;
2475
2476         while (recv_read(hdl, fd, drr, sizeof (dmu_replay_record_t),
2477                         byteswap, NULL) == 0) {
2478             if (byteswap)
2479                 drr->drr_type = BSWAP_32(drr->drr_type);
2480
2481             switch (drr->drr_type) {
2482                 case DRR_BEGIN:
2483                     /* NB: not to be used on v2 stream packages */
2484                     if (drr->drr_payloadlen != 0) {
2485                         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
2486                             "invalid sublist header"));
2487                         return (zfs_error(hdl, EZFS_BADSTREAM, errbuf));
2488                     }
2489                     break;
2490
2491                 case DRR_END:
2492                     free(buf);
2493                     return (0);
2494
2495                 case DRR_OBJECT:
2496                     if (byteswap) {
2497                         drr->drr_u.drr_object.drr_bonuslen =
2498                             BSWAP_32(drr->drr_u.drr_object.
2499                                     drr_bonuslen);
2500                     }
2501                     (void) recv_read(hdl, fd, buf,
2502                         P2ROUNDUP(drr->drr_u.drr_object.drr_bonuslen, 8),
2503                         B_FALSE, NULL);
2504                     break;
2505
2506                 case DRR_WRITE:
2507                     if (byteswap) {
2508                         drr->drr_u.drr_write.drr_length =
2509                             BSWAP_64(drr->drr_u.drr_write.drr_length);
2510                     }
2511                     (void) recv_read(hdl, fd, buf,
2512                         drr->drr_u.drr_write.drr_length, B_FALSE, NULL);
2513                     break;
2514
2515                 case DRR_SPILL:
2516                     if (byteswap) {
2517                         drr->drr_u.drr_write.drr_length =
2518                             BSWAP_64(drr->drr_u.drr_spill.drr_length);
2519                     }
2520                     (void) recv_read(hdl, fd, buf,
2521                         drr->drr_u.drr_spill.drr_length, B_FALSE, NULL);
2522                     break;
2523             }
2524         }
2525     }

```

new/usr/src/lib/libzfs/common/libzfs_sendrecv.c

11

```

2522         case DRR_WRITE_BYREF:
2523         case DRR_FREEOBJECTS:
2524         case DRR_FREE:
2525             break;
2526
2527     default:
2528         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
2529                         "invalid record type"));
2530         return (zfs_error(hdl, EZFS_BADSTREAM, errbuf));
2531     }
2532 }
2533
2534 free(buf);
2535 return (-1);
2536 }

2537 /*
2538 * Restores a backup of tosnap from the file descriptor specified by infd.
2539 */
2540 static int
2541 zfs_receive_one(libzfs_handle_t *hdl, int infd, const char *tosnap,
2542 recvflags_t *flags, dmu_replay_record_t *drr,
2543 dmu_replay_record_t *drr_noswap, const char *sendfs,
2544 nvlist_t *stream_nv, avl_tree_t *stream_avl, char **top_zfs, int cleanup_fd,
2545 uint64_t *action_handlep)
2546 {
2547     zfs_cmd_t zc = { 0 };
2548     time_t begin_time;
2549     int ioctl_err, ioctl_errno, err;
2550     char *cp;
2551     struct drr_begin *drrb = &drr->drr_u.drr_begin;
2552     char errbuf[1024];
2553     char prop_errbuf[1024];
2554     const char *choppprefix;
2555     boolean_t newfs = B_FALSE;
2556     boolean_t stream_wantsnewfs;
2557     uint64_t parent_snapguid = 0;
2558     prop_changelist_t *clp = NULL;
2559     nvlist_t *snapprops_nvlist = NULL;
2560     zprop_errflags_t prop_errflags;
2561     boolean_t recursive;
2562
2563     begin_time = time(NULL);
2564
2565     (void) snprintf(errbuf, sizeof (errbuf), dgettext(TEXT_DOMAIN,
2566                 "cannot receive"));
2567
2568     recursive = (nvlist_lookup_boolean(stream_nv, "not_recursive") ==
2569                 ENOENT);
2570
2571     if (stream_avl != NULL) {
2572         char *snapname;
2573         nvlist_t *fs = fsavl_find(stream_avl, drrb->drr_toguid,
2574                         &snapname);
2575         nvlist_t *props;
2576         int ret;
2577
2578         (void) nvlist_lookup_uint64(fs, "parentfromsnap",
2579                         &parent_snapguid);
2580         err = nvlist_lookup_nvlist(fs, "props", &props);
2581         if (err)
2582             VERIFY(0 == nvlist_alloc(&props, NV_UNIQUE_NAME, 0));
2583
2584         if (flags->canmountoff) {
2585             VERIFY(0 == nvlist_add_uint64(props,
2586                         zfs_prop_to_name(ZFS_PROP_CANMOUNT), 0));
2587         }
2588     }
2589 }
```

new/usr/src/lib/libzfs/common/libzfs_sendrecv.c

```

2588
2589
2590
2591
2592     }
2593     ret = zcmdd_write_src_nvlist(hdl, &zc, props);
2594     if (err)
2595         nvlist_free(props);
2596
2597     if (0 == nvlist_lookup_nvlist(fs, "snapprops", &props)) {
2598         VERIFY(0 == nvlist_lookup_nvlist(props,
2599                                         snapname, &snapprops_nvlist));
2600     }
2601
2602     if (ret != 0)
2603         return (-1);
2604
2605     cp = NULL;
2606
2607     /*
2608      * Determine how much of the snapshot name stored in the stream
2609      * we are going to tack on to the name they specified on the
2610      * command line, and how much we are going to chop off.
2611      *
2612      * If they specified a snapshot, chop the entire name stored in
2613      * the stream.
2614      */
2615     if (flags->istail) {
2616         /*
2617          * A filesystem was specified with -e. We want to tack on only
2618          * the tail of the sent snapshot path.
2619          */
2620         if (strchr(tosnap, '@')) {
2621             zfs_error_aux(hdl, dgettext(TEXT_DOMAIN, "invalid "
2622                                     "argument - snapshot not allowed with -e"));
2623             return (zfs_error(hdl, EZFS_INVALIDNAME, errbuf));
2624         }
2625
2626         chopprefix = strrchr(sendfs, '/');
2627
2628         if (chopprefix == NULL) {
2629             /*
2630              * The tail is the poolname, so we need to
2631              * prepend a path separator.
2632              */
2633             int len = strlen(drrb->drr_toname);
2634             cp = malloc(len + 2);
2635             cp[0] = '/';
2636             (void) strcpy(&cp[1], drrb->drr_toname);
2637             chopprefix = cp;
2638         } else {
2639             chopprefix = drrb->drr_toname + (chopprefix - sendfs);
2640         }
2641     } else if (flags->isprefix) {
2642         /*
2643          * A filesystem was specified with -d. We want to tack on
2644          * everything but the first element of the sent snapshot path
2645          * (all but the pool name).
2646          */
2647         if (strchr(tosnap, '@')) {
2648             zfs_error_aux(hdl, dgettext(TEXT_DOMAIN, "invalid "
2649                                     "argument - snapshot not allowed with -d"));
2650             return (zfs_error(hdl, EZFS_INVALIDNAME, errbuf));
2651         }
2652
2653         chopprefix = strchr(drrb->drr_toname, '/');
2654         if (chopprefix == NULL)
2655             chopprefix = strchr(drrb->drr_toname, '@');
2656     } else if (strchr(tosnap, '@') == NULL) {
2657
2658
2659
2660
2661
2662
2663
2664
2665
2666
2667
2668
2669
2670
2671
2672
2673
2674
2675
2676
2677
2678
2679
2680
2681
2682
2683
2684
2685
2686
2687
2688
2689
2690
2691
2692
2693
2694
2695
2696
2697
2698
2699
2700
2701
2702
2703
2704
2705
2706
2707
2708
2709
2710
2711
2712
2713
2714
2715
2716
2717
2718
2719
2720
2721
2722
2723
2724
2725
2726
2727
2728
2729
2730
2731
2732
2733
2734
2735
2736
2737
2738
2739
2740
2741
2742
2743
2744
2745
2746
2747
2748
2749
2750
2751
2752
2753
2754
2755
2756
2757
2758
2759
2760
2761
2762
2763
2764
2765
2766
2767
2768
2769
2770
2771
2772
2773
2774
2775
2776
2777
2778
2779
2780
2781
2782
2783
2784
2785
2786
2787
2788
2789
2790
2791
2792
2793
2794
2795
2796
2797
2798
2799
2800
2801
2802
2803
2804
2805
2806
2807
2808
2809
2810
2811
2812
2813
2814
2815
2816
2817
2818
2819
2820
2821
2822
2823
2824
2825
2826
2827
2828
2829
2830
2831
2832
2833
2834
2835
2836
2837
2838
2839
2840
2841
2842
2843
2844
2845
2846
2847
2848
2849
2850
2851
2852
2853
2854
2855
2856
2857
2858
2859
2860
2861
2862
2863
2864
2865
2866
2867
2868
2869
2870
2871
2872
2873
2874
2875
2876
2877
2878
2879
2880
2881
2882
2883
2884
2885
2886
2887
2888
2889
2890
2891
2892
2893
2894
2895
2896
2897
2898
2899
2900
2901
2902
2903
2904
2905
2906
2907
2908
2909
2910
2911
2912
2913
2914
2915
2916
2917
2918
2919
2920
2921
2922
2923
2924
2925
2926
2927
2928
2929
2930
2931
2932
2933
2934
2935
2936
2937
2938
2939
2940
2941
2942
2943
2944
2945
2946
2947
2948
2949
2950
2951
2952
2953
2954
2955
2956
2957
2958
2959
2960
2961
2962
2963
2964
2965
2966
2967
2968
2969
2970
2971
2972
2973
2974
2975
2976
2977
2978
2979
2980
2981
2982
2983
2984
2985
2986
2987
2988
2989
2990
2991
2992
2993
2994
2995
2996
2997
2998
2999
2999
3000
3001
3002
3003
3004
3005
3006
3007
3008
3009
3010
3011
3012
3013
3014
3015
3016
3017
3018
3019
3020
3021
3022
3023
3024
3025
3026
3027
3028
3029
3030
3031
3032
3033
3034
3035
3036
3037
3038
3039
3040
3041
3042
3043
3044
3045
3046
3047
3048
3049
3050
3051
3052
3053
3054
3055
3056
3057
3058
3059
3060
3061
3062
3063
3064
3065
3066
3067
3068
3069
3070
3071
3072
3073
3074
3075
3076
3077
3078
3079
3080
3081
3082
3083
3084
3085
3086
3087
3088
3089
3090
3091
3092
3093
3094
3095
3096
3097
3098
3099
3099
3100
3101
3102
3103
3104
3105
3106
3107
3108
3109
3110
3111
3112
3113
3114
3115
3116
3117
3118
3119
3120
3121
3122
3123
3124
3125
3126
3127
3128
3129
3130
3131
3132
3133
3134
3135
3136
3137
3138
3139
3140
3141
3142
3143
3144
3145
3146
3147
3148
3149
3149
3150
3151
3152
3153
3154
3155
3156
3157
3158
3159
3160
3161
3162
3163
3164
3165
3166
3167
3168
3169
3170
3171
3172
3173
3174
3175
3176
3177
3178
3179
3180
3181
3182
3183
3184
3185
3186
3187
3188
3189
3189
3190
3191
3192
3193
3194
3195
3196
3197
3198
3199
3199
3200
3201
3202
3203
3204
3205
3206
3207
3208
3209
3209
3210
3211
3212
3213
3214
3215
3216
3217
3218
3219
3219
3220
3221
3222
3223
3224
3225
3226
3227
3228
3229
3229
3230
3231
3232
3233
3234
3235
3236
3237
3238
3239
3239
3240
3241
3242
3243
3244
3245
3246
3247
3248
3249
3249
3250
3251
3252
3253
3254
3255
3256
3257
3258
3259
3259
3260
3261
3262
3263
3264
3265
3266
3267
3268
3269
3269
3270
3271
3272
3273
3274
3275
3276
3277
3278
3279
3279
3280
3281
3282
3283
3284
3285
3286
3287
3288
3289
3289
3290
3291
3292
3293
3294
3295
3296
3297
3298
3299
3299
3300
3301
3302
3303
3304
3305
3306
3307
3308
3309
3309
3310
3311
3312
3313
3314
3315
3316
3317
3318
3319
3319
3320
3321
3322
3323
3324
3325
3326
3327
3328
3329
3329
3330
3331
3332
3333
3334
3335
3336
3337
3338
3339
3339
3340
3341
3342
3343
3344
3345
3346
3347
3348
3349
3349
3350
3351
3352
3353
3354
3355
3356
3357
3358
3359
3359
3360
3361
3362
3363
3364
3365
3366
3367
3368
3369
3369
3370
3371
3372
3373
3374
3375
3376
3377
3378
3379
3379
3380
3381
3382
3383
3384
3385
3386
3387
3388
3389
3389
3390
3391
3392
3393
3394
3395
3396
3397
3398
3398
3399
3399
3400
3401
3402
3403
3404
3405
3406
3407
3408
3409
3409
3410
3411
3412
3413
3414
3415
3416
3417
3418
3419
3419
3420
3421
3422
3423
3424
3425
3426
3427
3428
3429
3429
3430
3431
3432
3433
3434
3435
3436
3437
3438
3439
3439
3440
3441
3442
3443
3444
3445
3446
3447
3448
3449
3449
3450
3451
3452
3453
3454
3455
3456
3457
3458
3459
3459
3460
3461
3462
3463
3464
3465
3466
3467
3468
3469
3469
3470
3471
3472
3473
3474
3475
3476
3477
3478
3479
3479
3480
3481
3482
3483
3484
3485
3486
3487
3488
3489
3489
3490
3491
3492
3493
3494
3495
3496
3497
3498
3498
3499
3499
3500
3501
3502
3503
3504
3505
3506
3507
3508
3509
3509
3510
3511
3512
3513
3514
3515
3516
3517
3518
3519
3519
3520
3521
3522
3523
3524
3525
3526
3527
3528
3529
3529
3530
3531
3532
3533
3534
3535
3536
3537
3538
3539
3539
3540
3541
3542
3543
3544
3545
3546
3547
3548
3549
3549
3550
3551
3552
3553
3554
3555
3556
3557
3558
3559
3559
3560
3561
3562
3563
3564
3565
3566
3567
3568
3569
3569
3570
3571
3572
3573
3574
3575
3576
3577
3578
3579
3579
3580
3581
3582
3583
3584
3585
3586
3587
3588
3589
3589
3590
3591
3592
3593
3594
3595
3596
3597
3598
3598
3599
3599
3600
3601
3602
3603
3604
3605
3606
3607
3608
3609
3609
3610
3611
3612
3613
3614
3615
3616
3617
3618
3619
3619
3620
3621
3622
3623
3624
3625
3626
3627
3628
3629
3629
3630
3631
3632
3633
3634
3635
3636
3637
3638
3639
3639
3640
3641
3642
3643
3644
3645
3646
3647
3648
3649
3649
3650
3651
3652
3653
3654
3655
3656
3657
3658
3659
3659
3660
3661
3662
3663
3664
3665
3666
3667
3668
3669
3669
3670
3671
3672
3673
3674
3675
3676
3677
3678
3679
3679
3680
3681
3682
3683
3684
3685
3686
3687
3688
3689
3689
3690
3691
3692
3693
3694
3695
3696
3697
3698
3698
3699
3699
3700
3701
3702
3703
3704
3705
3706
3707
3708
3709
3709
3710
3711
3712
3713
3714
3715
3716
3717
3718
3719
3719
3720
3721
3722
3723
3724
3725
3726
3727
3728
3729
3729
3730
3731
3732
3733
3734
3735
3736
3737
3738
3739
3739
3740
3741
3742
3743
3744
3745
3746
3747
3748
3749
3749
3750
3751
3752
3753
3754
3755
3756
3757
3758
3759
3759
3760
3761
3762
3763
3764
3765
3766
3767
3768
3769
3769
3770
3771
3772
3773
3774
3775
3776
3777
3778
3779
3779
3780
3781
3782
3783
3784
3785
3786
3787
3788
3789
3789
3790
3791
3792
3793
3794
3795
3796
3797
3798
3798
3799
3799
3800
3801
3802
3803
3804
3805
3806
3807
3808
3809
3809
3810
3811
3812
3813
3814
3815
3816
3817
3818
3819
3819
3820
3821
3822
3823
3824
3825
3826
3827
3828
3829
3829
3830
3831
3832
3833
3834
3835
3836
3837
3838
3839
3839
3840
3841
3842
3843
3844
3845
3846
3847
3848
3849
3849
3850
3851
3852
3853
3854
3855
3856
3857
3858
3859
3859
3860
3861
3862
3863
3864
3865
3866
3867
3868
3869
3869
3870
3871
3872
3873
3874
3875
3876
3877
3878
3879
3879
3880
3881
3882
3883
3884
3885
3886
3887
3888
3889
3889
3890
3891
3892
3893
3894
3895
3896
3897
3898
3898
3899
3899
3900
3901
3902
3903
3904
3905
3906
3907
3908
3909
3909
3910
3911
3912
3913
3914
3915
3916
3917
3918
3919
3919
3920
3921
3922
3923
3924
3925
3926
3927
3928
3929
3929
3930
3931
3932
3933
3934
3935
3936
3937
3938
3939
3939
3940
3941
3942
3943
3944
3945
3946
3947
3948
3949
3949
3950
3951
3952
3953
3954
3955
3956
3957
3958
3959
3959
3960
3961
3962
3963
3964
3965
3966
3967
3968
3969
3969
3970
3971
3972
3973
3974
3975
3976
3977
3978
3979
3979
3980
3981
3982
3983
3984
3985
3986
3987
3988
3989
3989
3990
3991
3992
3993
3994
3995
3996
3997
3998
3998
3999
3999
4000
4001
4002
4003
4004
4005
4006
4007
4008
4009
4009
4010
4011
4012
4013
4014
4015
4016
4017
4018
4019
4019
4020
4021
4022
4023
4024
4025
4026
4027
4028
4029
4029
4030
4031
4032
4033
4034
4035
4036
4037
4038
4039
4039
4040
4041
4042
4043
4044
4045
4046
4047
4048
4049
4049
4050
4051
4052
4053
4054
4055
4056
4057
4058
4059
4059
4060
4061
4062
4063
4064
4065
4066
4067
4068
4069
4069
4070
4071
4072
4073
4074
4075
4076
4077
4078
4079
4079
4080
4081
4082
4083
4084
4085
4086
4087
4088
4089
4089
4090
4091
4092
4093
4094
4095
4096
4097
4098
4098
4099
4099
4100
4101
4102
4103
4104
4105
4106
4107
4108
4109
4109
4110
4111
4112
4113
4114
4115
4116
4117
4118
4119
4119
4120
4121
4122
4123
4124
4125
4126
4127
4128
4129
4129
4130
4131
4132
4133
4134
4135
4136
4137
4138
4139
4139
4140
4141
4142
4143
4144
4145
4146
4147
4148
4149
4149
4150
4151
4152
4153
4154
4155
4156
4157
4158
4159
4159
4160
4161
4162
4163
4164
4165
4166
4167
4168
4169
4169
4170
4171
4172
4173
4174
4175
4176
4177
4178
4179
4179
4180
4181
4182
4183
4184
4185
4186
4187
4188
4189
4189
4190
4191
4192
4193
4194
4195
4196
4197
4198
4198
4199
4199
4200
4201
4202
4203
4204
4205
4206
4207
4208
4209
4209
4210
4211
4212
4213
4214
4215
4216
4217
4218
4219
4219
4220
4221
4222
4223
4224
4225
4226
4227
4228
4229
4229
4230
4231
4232
4233
4234
4235
4236
4237
4238
4239
4239
4240
4241
4242
4243
4244
4245
4246
4247
4248
4249
4249
4250
4251
4252
4253
4254
4255
4256
4257
4258
4259
4259
4260
4261
4262
4263
4264
4265
4266
4267
4268
4269
4269
4270
4271
4272
4273
4274
4275
4276
4277
4278
4279
4279
4280
4281
4282
4283
4284
4285
4286
4287
4288
4289
4289
4290
4291
4292
4293
4294
4295
4296
4297
4298
4298
4299
4299
4300
4301
4302
4303
4304
4305
4306
4307
4308
4309
4309
4310
4311
4312
4313
4314
4315
4316
4317
4318
4319
4319
4320
4321
4322
4323
4324
4325
4326
4327
4328
4329
4329
4330
4331
4332
4333
4334
4335
4336
4337
4338
4339
4339
4340
4341
4342
4343
4344
4345
4346
4347
4348
4349
4349
4350
4351
4352
4353
4354
4355
4356
4357
4358
4359
4359
4360
4361
4362
4363
4364
4365
4366
4367
4368
4369
4369
4370
4371
4372
4373
4374
4375
4376
4377
4378
4379
4379
4380
4381
4382
4383
4384
4385
4386
4387
4388
4389
4389
4390
4391
4392
4393
4394
4395
4396
4397
4398
4398
4399
4399
4400
4401
4402
4403
4404
4405
4406
4407
4408
4409
4409
4410
4411
4412
4413
4414
4415
4416
4417
4418
4419
4419
4420
4421
4422
4423
4424
4425
4426
4427
4428
4429
4429
4430
4431
4432
4433
4434
4435
4436
4437
4438
4439
4439
4440
4441
4442
4443
4444
4445
4446
4447
4448
4449
4449
4450
4451
4452
4453
4454
4455
4456
4457
4458
4459
4459
4460
4461
4462
4463
4464
4465
4466
4467
4468
4469
4469
4470
4471
4472
4473
4474
4475
4476
4477
4478
4479
4479
4480
4481
4482
4483
4484
4485
4486
4487
4488
4489
4489
4490
4491
4492
4493
4494
4495
4496
4497
4498
4498
4499
4499
4500
4501
4502
4503
4504
4505
4506
4507
4508
4509
4509
4510
4511
4512
4513
4514
4515
4516
4517
4518
4519
4519
4520
4521
4522
4523
4524
4525
4526
4527
4528
4529
4529
4530
4531
4532
4533
4534
4535
4536
4537
4538
4539
4539
4540
4541
4542
4543
4544
4545
4546
4547
4548
4549
4549
4550
4551
4552
4553
4554
4555
4556
4557
4558
4559
4559
4560
4561
4562
4563
4564
4565
4566
4567
4568
4569
4569
4570
4571
4572
4573
4574
4575
4576
4577
4578
4579
4579
4580
4581
4582
4583
4584
4585
4586
4587
4588
4589
4589
4590
4591
4592
4593
4594
4595
4596
4597
4598
4598
4599
4599
4600
4601
4602
4603
4604
4605
4606
4607
4608
4609
4609
4610
4611
4612
4613
4614
4615
4616
4617
4618
4619
4619
4620
4621
4622
4623
4624
4625
4626
4627
4628
4629
4629
4630
4631
4632
4633
4634
4635
4636
4637
4638
4639
4639
4640
4641
4642
4643
4644
4645
4646
4647
4648
4649
4649
4650
4651
4652
4653
4654
4655
4656
4657
4658
4659
4659
4660
4661
4662
4663
4664
4665
4666
4667
4668
4669
4669
4670
4671
4672
4673
4674
4675
4676
4677
4678
4679
4679
4680
4681
4682
4683
4684
4685
4686
4687
4688
4689
4689
4690
4691
4692
4693
4694
4695
4696
4697
4698
4698
4699
4699
4700
4701
4702
4703
4704
4705
4706
4707
4708
4709
4709
4710
4711
4712
4713
4714
4715
4716
4717
4718
4719
4719
4720
4721
4722
4723
4724
4725
4726
4727
4728
4729
4729
4730
4731
4732
4733
4734
4735
4736
4737
4738
4739
4739
4740
4741
4742
4743
4744
4745
4746
4747
4748
4749
4749
4750
4751
4752
4753
4754
4755
4756
4757
4758
4759
4759
4760
4761
4762
4763
4764
4765
4766
4767
4768
4769
4769
4770
4771
4772
4773
4774
4775
4776
4777
477
```

```

2654         /*
2655          * If a filesystem was specified without -d or -e, we want to
2656          * tack on everything after the fs specified by 'zfs send'.
2657          */
2658         chopprefix = drrb->drr_toname + strlen(sendfs);
2659     } else {
2660         /* A snapshot was specified as an exact path (no -d or -e). */
2661         if (recursive) {
2662             zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
2663                                         "cannot specify snapshot name for multi-snapshot "
2664                                         "stream"));
2665             return (zfs_error(hdl, EZFS_BADSTREAM, errbuf));
2666         }
2667         chopprefix = drrb->drr_toname + strlen(drrb->drr_toname);
2668     }
2669
2670     ASSERT(strstr(drrb->drr_toname, sendfs) == drrb->drr_toname);
2671     ASSERT(chopprefix > drrb->drr_toname);
2672     ASSERT(chopprefix <= drrb->drr_toname + strlen(drrb->drr_toname));
2673     ASSERT(chopprefix[0] == '/' || chopprefix[0] == '@' ||
2674           chopprefix[0] == '\0');
2675
2676     /*
2677      * Determine name of destination snapshot, store in zc_value.
2678      */
2679     (void) strcpy(zc.zc_value, tosnap);
2680     (void) strncat(zc.zc_value, chopprefix, sizeof (zc.zc_value));
2681     free(cp);
2682     if (!zfs_name_valid(zc.zc_value, ZFS_TYPE_SNAPSHOT)) {
2683         zcmd_free_nvlists(&zc);
2684         return (zfs_error(hdl, EZFS_INVALIDNAME, errbuf));
2685     }
2686
2687     /*
2688      * Determine the name of the origin snapshot, store in zc_string.
2689      */
2690     if (drrb->drr_flags & DRR_FLAG_CLONE) {
2691         if (guid_to_name(hdl, zc.zc_value,
2692                         drrb->drr_fromguid, zc.zc_string) != 0) {
2693             zcmd_free_nvlists(&zc);
2694             zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
2695                             "local origin for clone %s does not exist"),
2696                           zc.zc_value);
2697             return (zfs_error(hdl, EZFS_NOENT, errbuf));
2698         }
2699         if (flags->verbose)
2700             (void) printf("found clone origin %s\n", zc.zc_string);
2701     }
2702
2703     stream_wantsnewfs = (drrb->drr_fromguid == NULL ||
2704                          (drrb->drr_flags & DRR_FLAG_CLONE));
2705
2706     if (stream_wantsnewfs) {
2707         /*
2708          * if the parent fs does not exist, look for it based on
2709          * the parent snap GUID
2710          */
2711         (void) sprintf(errbuf, sizeof (errbuf), dgettext(TEXT_DOMAIN,
2712                                         "cannot receive new filesystem stream"));
2713
2714         (void) strcpy(zc.zc_name, zc.zc_value);
2715         cp = strrchr(zc.zc_name, '/');
2716         if (cp)
2717             *cp = '\0';
2718         if (cp &&
2719             !zfs_dataset_exists(hdl, zc.zc_name, ZFS_TYPE_DATASET)) {

```

```

2720         char suffix[ZFS_MAXNAMELEN];
2721         (void) strcpy(suffix, strrchr(zc.zc_value, '/'));
2722         if (guid_to_name(hdl, zc.zc_name, parent_snapguid,
2723                         zc.zc_value) == 0) {
2724             *strchr(zc.zc_value, '@') = '\0';
2725             (void) strcat(zc.zc_value, suffix);
2726         }
2727     } else {
2728         /*
2729          * if the fs does not exist, look for it based on the
2730          * fromsnap GUID
2731          */
2732         (void) sprintf(errbuf, sizeof (errbuf), dgettext(TEXT_DOMAIN,
2733                                         "cannot receive incremental stream"));
2734
2735         (void) strcpy(zc.zc_name, zc.zc_value);
2736         *strchr(zc.zc_name, '@') = '\0';
2737
2738         /*
2739          * If the exact receive path was specified and this is the
2740          * topmost path in the stream, then if the fs does not exist we
2741          * should look no further.
2742          */
2743         if ((flags->isprefix || (*chopprefix = drrb->drr_toname +
2744             strlen(sendfs)) != '\0' && *chopprefix != '@') &&
2745             !zfs_dataset_exists(hdl, zc.zc_name, ZFS_TYPE_DATASET)) {
2746             char snap[ZFS_MAXNAMELEN];
2747             (void) strcpy(snap, strrchr(zc.zc_value, '@'));
2748             if (guid_to_name(hdl, zc.zc_name, drrb->drr_fromguid,
2749                             zc.zc_value) == 0) {
2750                 *strchr(zc.zc_value, '@') = '\0';
2751                 (void) strcat(zc.zc_value, snap);
2752             }
2753         }
2754     }
2755
2756     (void) strcpy(zc.zc_name, zc.zc_value);
2757     *strchr(zc.zc_name, '@') = '\0';
2758
2759     if (zfs_dataset_exists(hdl, zc.zc_name, ZFS_TYPE_DATASET)) {
2760         zfs_handle_t *zhp;
2761
2762         /*
2763          * Destination fs exists. Therefore this should either
2764          * be an incremental, or the stream specifies a new fs
2765          * (full stream or clone) and they want us to blow it
2766          * away (and have therefore specified -F and removed any
2767          * snapshots).
2768          */
2769         if (stream_wantsnewfs) {
2770             if (!flags->force) {
2771                 zcmd_free_nvlists(&zc);
2772                 zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
2773                                         "destination '%s' exists\n"
2774                                         "must specify -F to overwrite it"),
2775                               zc.zc_name);
2776                 return (zfs_error(hdl, EZFS_EXISTS, errbuf));
2777             }
2778             if (ioctl(hdl->libzfs_fd, ZFS_IOC_SNAPSHOT_LIST_NEXT,
2779                         &zc) == 0) {
2780                 zcmd_free_nvlists(&zc);
2781                 zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
2782                                         "destination has snapshots (eg. %s)\n"
2783                                         "must destroy them to overwrite it"),
2784                               zc.zc_name);
2785             }
2786         }
2787     }

```

```

2786         return (zfs_error(hdl, EZFS_EXISTS, errbuf));
2787     }
2788
2789     if ((zhp = zfs_open(hdl, zc.zc_name,
2790                          ZFS_TYPE_FILESYSTEM | ZFS_TYPE_VOLUME)) == NULL) {
2791         zcmd_free_nvlists(&zc);
2792         return (-1);
2793     }
2794
2795     if (stream_wantsnewfs &&
2796         zhp->zfs_dmustats.dds_origin[0]) {
2797         zcmd_free_nvlists(&zc);
2798         zfs_close(zhp);
2799         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
2800                         "destination '%s' is a clone\n"
2801                         "must destroy it to overwrite it"),
2802                         zc.zc_name);
2803         return (zfs_error(hdl, EZFS_EXISTS, errbuf));
2804     }
2805
2806     if (!flags->dryrun && zhp->zfs_type == ZFS_TYPE_FILESYSTEM &&
2807         stream_wantsnewfs) {
2808         /* We can't do online recv in this case */
2809         clp = changelist_gather(zhp, ZFS_PROP_NAME, 0, 0);
2810         if (clp == NULL) {
2811             zfs_close(zhp);
2812             zcmd_free_nvlists(&zc);
2813             return (-1);
2814         }
2815         if (changelist_prefix(clp) != 0) {
2816             changelist_free(clp);
2817             zfs_close(zhp);
2818             zcmd_free_nvlists(&zc);
2819             return (-1);
2820         }
2821     }
2822     zfs_close(zhp);
2823 } else {
2824     /*
2825      * Destination filesystem does not exist. Therefore we better
2826      * be creating a new filesystem (either from a full backup, or
2827      * a clone). It would therefore be invalid if the user
2828      * specified only the pool name (i.e. if the destination name
2829      * contained no slash character).
2830     */
2831     if (!stream_wantsnewfs ||
2832         (cp = strrchr(zc.zc_name, '/')) == NULL) {
2833         zcmd_free_nvlists(&zc);
2834         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
2835                         "destination '%s' does not exist"), zc.zc_name);
2836         return (zfs_error(hdl, EZFS_NOENT, errbuf));
2837     }
2838
2839     /*
2840      * Trim off the final dataset component so we perform the
2841      * recvbackup ioctl to the filesystem's parent.
2842     */
2843     *cp = '\0';
2844
2845     if (flags->isprefix && !flags->istail && !flags->dryrun &&
2846         create_parents(hdl, zc.zc_value, strlen(tosnap)) != 0) {
2847         zcmd_free_nvlists(&zc);
2848         return (zfs_error(hdl, EZFS_BADRESTORE, errbuf));
2849     }

```

```

2852             newfs = B_TRUE;
2853         }
2854
2855         zc.zc_begin_record = drr_noswap->drr_u.drr_begin;
2856         zc.zc_cookie = infd;
2857         zc.zc_guid = flags->force;
2858         if (flags->verbose) {
2859             (void) printf("%s %s stream of %s into %s\n",
2860                         flags->dryrun ? "would receive" : "receiving",
2861                         drrb->drr_fromguid ? "incremental" : "full",
2862                         drrb->drr_toname, zc.zc_value);
2863             (void) fflush(stdout);
2864         }
2865
2866         if (flags->dryrun) {
2867             zcmd_free_nvlists(&zc);
2868             return (recv_skip(hdl, infd, flags->byteswap));
2869         }
2870
2871         zc.zc_nvlist_dst = (uint64_t)(uintptr_t)prop_errbuf;
2872         zc.zc_nvlist_dst_size = sizeof (prop_errbuf);
2873         zc.zc_cleanup_fd = cleanup_fd;
2874         zc.zc_action_handle = *action_handlep;
2875
2876         err = ioctl_err = zfs_ioctl(hdl, ZFS_IOC_RECV, &zc);
2877         ioctl_errno = errno;
2878         prop_errflags = (zprop_errflags_t)zc.zc_obj;
2879
2880         if (err == 0) {
2881             nvlist_t *prop_errors;
2882             VERIFY(0 == nvlist_unpack((void *)(uintptr_t)zc.zc_nvlist_dst,
2883                                     zc.zc_nvlist_dst_size, &prop_errors, 0));
2884
2885             nvpair_t *prop_err = NULL;
2886
2887             while ((prop_err = nvlist_next_nvpair(prop_errors,
2888                                         prop_err)) != NULL) {
2889                 char tbuf[1024];
2890                 zfs_prop_t prop;
2891                 int intval;
2892
2893                 prop = zfs_name_to_prop(nvpair_name(prop_err));
2894                 (void) nvpair_value_int32(prop_err, &intval);
2895                 if (strcmp(nvpair_name(prop_err),
2896                           ZPROP_N_MORE_ERRORS) == 0) {
2897                     trunc_prop_errs(intval);
2898                     break;
2899                 } else {
2900                     (void) sprintf(tbuf, sizeof (tbuf),
2901                               dgettext(TEXT_DOMAIN,
2902                                   "cannot receive %s property on %s"),
2903                                   nvpair_name(prop_err), zc.zc_name);
2904                     zfs_setprop_error(hdl, prop, intval, tbuf);
2905                 }
2906             }
2907             nvlist_free(prop_errors);
2908         }
2909
2910         zc.zc_nvlist_dst = 0;
2911         zc.zc_nvlist_dst_size = 0;
2912         zcmd_free_nvlists(&zc);
2913
2914         if (err == 0 && snapprops_nvlist) {
2915             zfs_cmd_t zc2 = { 0 };
2916
2917             (void) strcpy(zc2.zc_name, zc.zc_value);

```

```

2918     zc2.zc_cookie = B_TRUE; /* received */
2919     if (zcmd_write_src_nvlist(hdl, &zc2, snapprops_nvlist) == 0) {
2920         (void) zfs_ioctl(hdl, ZFS_IOC_SET_PROP, &zc2);
2921         zcmd_free_nvlists(&zc2);
2922     }
2923 }
2925 if (err && (ioctl_errno == ENOENT || ioctl_errno == EEXIST)) {
2926     /*
2927      * It may be that this snapshot already exists,
2928      * in which case we want to consume & ignore it
2929      * rather than failing.
2930     */
2931     avi_tree_t *local_avl;
2932     nvlist_t *local_nv, *fs;
2933     cp = strchr(zc.zc_value, '@');
2935     /*
2936      * XXX Do this faster by just iterating over snaps in
2937      * this fs. Also if zc_value does not exist, we will
2938      * get a strange "does not exist" error message.
2939     */
2940     *cp = '\0';
2941     if (gather_nvlist(hdl, zc.zc_value, NULL, NULL, B_FALSE,
2942         &local_nv, &local_avl) == 0) {
2943         *cp = '@';
2944         fs = fsavl_find(local_avl, drrb->drr_toguid, NULL);
2945         fsavl_destroy(local_avl);
2946         nvlist_free(local_nv);
2948         if (fs != NULL) {
2949             if (flags->verbose) {
2950                 (void) printf("snap %s already exists; "
2951                             "ignoring\n", zc.zc_value);
2952             }
2953             err = ioctl_err = recv_skip(hdl, infd,
2954                                         flags->byteswap);
2955         }
2956     }
2957     *cp = '@';
2958 }
2960 if (ioctl_err != 0) {
2961     switch (ioctl_errno) {
2962     case ENODEV:
2963         cp = strchr(zc.zc_value, '@');
2964         *cp = '\0';
2965         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
2966             "most recent snapshot of %s does not\n"
2967             "match incremental source"), zc.zc_value);
2968         (void) zfs_error(hdl, EZFS_BADRESTORE, errbuf);
2969         *cp = '@';
2970         break;
2971     case ETXTBSY:
2972         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
2973             "destination %s has been modified\n"
2974             "since most recent snapshot"), zc.zc_name);
2975         (void) zfs_error(hdl, EZFS_BADRESTORE, errbuf);
2976         break;
2977     case EEXIST:
2978         cp = strchr(zc.zc_value, '@');
2979         if (newfs) {
2980             /* it's the containing fs that exists */
2981             *cp = '\0';
2982         }
2983         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,

```

```

2984         "destination already exists"));
2985         (void) zfs_error_fmt(hdl, EZFS_EXISTS,
2986             dgettext(TEXT_DOMAIN, "cannot restore to %s"),
2987             zc.zc_value);
2988         *cp = '@';
2989         break;
2990     case EINVAL:
2991         (void) zfs_error(hdl, EZFS_BADSTREAM, errbuf);
2992         break;
2993     case ECKSUM:
2994         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
2995             "invalid stream (checksum mismatch)"));
2996         (void) zfs_error(hdl, EZFS_BADSTREAM, errbuf);
2997         break;
2998     case ENOTSUP:
2999         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
3000             "pool must be upgraded to receive this stream."));
3001         (void) zfs_error(hdl, EZFS_BADVERSION, errbuf);
3002         break;
3003     case EDQUOT:
3004         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
3005             "destination %s space quota exceeded"), zc.zc_name);
3006         (void) zfs_error(hdl, EZFS_NOSPC, errbuf);
3007         break;
3008     default:
3009         (void) zfs_standard_error(hdl, ioctl_errno, errbuf);
3010     }
3011 }
3013 /*
3014  * Mount the target filesystem (if created). Also mount any
3015  * children of the target filesystem if we did a replication
3016  * receive (indicated by stream_avl being non-NULL).
3017 */
3018 cp = strchr(zc.zc_value, '@');
3019 if (cp && (ioctl_err == 0 || !newfs)) {
3020     zfs_handle_t *h;
3022     *cp = '\0';
3023     h = zfs_open(hdl, zc.zc_value,
3024         ZFS_TYPE_FILESYSTEM | ZFS_TYPE_VOLUME);
3025     if (h != NULL) {
3026         if (h->zfs_type == ZFS_TYPE_VOLUME) {
3027             *cp = '@';
3028         } else if (newfs || stream_avl) {
3029             /*
3030              * Track the first/top of hierarchy fs,
3031              * for mounting and sharing later.
3032             */
3033             if (top_zfs && *top_zfs == NULL)
3034                 *top_zfs = zfs_strdup(hdl, zc.zc_value);
3035         }
3036         zfs_close(h);
3037     }
3038     *cp = '@';
3039 }
3041 if (clp) {
3042     err |= changelist_postfix(clp);
3043     changelist_free(clp);
3044 }
3046 if (prop_errflags & ZPROP_ERR_NOCLEAR) {
3047     (void) fprintf(stderr, dgettext(TEXT_DOMAIN, "Warning: "
3048         "failed to clear unreceived properties on %s"),
3049         zc.zc_name);

```

```

3050         (void) fprintf(stderr, "\n");
3051     }
3052     if (prop_errflags & ZPROP_ERR_NORESTORE) {
3053         (void) fprintf(stderr, dgettext(TEXT_DOMAIN, "Warning: "
3054                         "failed to restore original properties on %s"),
3055                         zc.zc_name);
3056         (void) fprintf(stderr, "\n");
3057     }
3058
3059     if (err || ioctl_err)
3060         return (-1);
3061
3062     *action_handlep = zc.zc_action_handle;
3063
3064     if (flags->verbose) {
3065         char buf1[64];
3066         char buf2[64];
3067         uint64_t bytes = zc.zc_cookie;
3068         time_t delta = time(NULL) - begin_time;
3069         if (delta == 0)
3070             delta = 1;
3071         zfs_nicenum(bytes, buf1, sizeof (buf1));
3072         zfs_nicenum(bytes/delta, buf2, sizeof (buf1));
3073
3074         (void) printf("received %sB stream in %lu seconds (%sB/sec)\n",
3075                         buf1, delta, buf2);
3076     }
3077
3078     return (0);
3079 }
3080
3081 static int
3082 zfs_receive_impl(libzfs_handle_t *hdl, const char *tosnap, recvflags_t *flags,
3083     int infd, const char *sendfs, nvlist_t *stream_nv, avl_tree_t *stream_avl,
3084     char **top_zfs, int cleanup_fd, uint64_t *action_handlep)
3085 {
3086     int err;
3087     dmureplay_record_t drr, drr_noswap;
3088     struct drr_begin *drrb = &drr.drr_u.drr_begin;
3089     char errbuf[1024];
3090     zio_cksum_t zcksum = { 0 };
3091     uint64_t featureflags;
3092     int hdrtypes;
3093
3094     (void) sprintf(errbuf, sizeof (errbuf), dgettext(TEXT_DOMAIN,
3095                         "cannot receive"));
3096
3097     if (flags->isprefix &&
3098         !zfs_dataset_exists(hdl, tosnap, ZFS_TYPE_DATASET)) {
3099         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN, "specified fs "
3100                         "(%s) does not exist"), tosnap);
3101         return (zfs_error(hdl, EZFS_NOENT, errbuf));
3102     }
3103
3104     /* read in the BEGIN record */
3105     if (0 != (err = recv_read(hdl, infd, &drr, sizeof (drr), B_FALSE,
3106         &zcksum)))
3107         return (err);
3108
3109     if (drr.drr_type == DRR_END || drr.drr_type == BSWAP_32(DRR_END)) {
3110         /* It's the double end record at the end of a package */
3111         return (ENODATA);
3112     }
3113
3114     /* the kernel needs the non-byteswapped begin record */
3115     drr_noswap = drr;

```

```

3117     flags->byteswap = B_FALSE;
3118     if (drrb->drr_magic == BSWAP_64(DMU_BACKUP_MAGIC)) {
3119         /*
3120          * We computed the checksum in the wrong byteorder in
3121          * recv_read() above; do it again correctly.
3122         */
3123         bzero(&zcksum, sizeof (zio_cksum_t));
3124         fletcher_4_incremental_byteswap(&drr, sizeof (drr), &zcksum);
3125         flags->byteswap = B_TRUE;
3126
3127         drr.drr_type = BSWAP_32(drr.drr_type);
3128         drr.drr_payloadlen = BSWAP_32(drr.drr_payloadlen);
3129         drrb->drr_magic = BSWAP_64(drrb->drr_magic);
3130         drrb->drr_versioninfo = BSWAP_64(drrb->drr_versioninfo);
3131         drrb->drr_creation_time = BSWAP_64(drrb->drr_creation_time);
3132         drrb->drr_type = BSWAP_32(drrb->drr_type);
3133         drrb->drr_flags = BSWAP_32(drrb->drr_flags);
3134         drrb->drr_toguid = BSWAP_64(drrb->drr_toguid);
3135         drrb->drr_fromguid = BSWAP_64(drrb->drr_fromguid);
3136     }
3137
3138     if (drrb->drr_magic != DMU_BACKUP_MAGIC || drr.drr_type != DRR_BEGIN) {
3139         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN, "invalid "
3140                         "stream (bad magic number)"));
3141         return (zfs_error(hdl, EZFS_BADSTREAM, errbuf));
3142     }
3143
3144     featureflags = DMU_GET_FEATUREFLAGS(drrb->drr_versioninfo);
3145     hdrtypes = DMU_GET_STREAM_HDRTYPE(drrb->drr_versioninfo);
3146
3147     if (!DMU_STREAM_SUPPORTED(featureflags) ||
3148         (hdrtypes != DMU_SUBSTREAM && hdrtypes != DMU_COMPOUNDSTREAM)) {
3149         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
3150                         "stream has unsupported feature, feature flags = %lx"),
3151                         featureflags);
3152         return (zfs_error(hdl, EZFS_BADSTREAM, errbuf));
3153     }
3154
3155     if (strchr(drrb->drr_toname, '@') == NULL) {
3156         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN, "invalid "
3157                         "stream (bad snapshot name)"));
3158         return (zfs_error(hdl, EZFS_BADSTREAM, errbuf));
3159     }
3160
3161     if (DMU_GET_STREAM_HDRTYPE(drrb->drr_versioninfo) == DMU_SUBSTREAM) {
3162         char nonpackage_sendfs[ZFS_MAXNAMELEN];
3163         if (sendfs == NULL) {
3164             /*
3165              * We were not called from zfs_receive_package(). Get
3166              * the fs specified by 'zfs send'.
3167             */
3168             char *cp;
3169             (void) strlcpy(nonpackage_sendfs,
3170                           drr.drr_u.drr_begin.drr_toname, ZFS_MAXNAMELEN);
3171             if ((cp = strchr(nonpackage_sendfs, '@')) != NULL)
3172                 *cp = '\0';
3173             sendfs = nonpackage_sendfs;
3174         }
3175         return (zfs_receive_one(hdl, infd, tosnap, flags,
3176             &drr, &drr_noswap, sendfs, stream_nv, stream_avl,
3177             top_zfs, cleanup_fd, action_handlep));
3178     } else {
3179         assert(DMU_GET_STREAM_HDRTYPE(drrb->drr_versioninfo) ==
3180             DMU_COMPOUNDSTREAM);
3181         return (zfs_receive_package(hdl, infd, tosnap, flags,

```

```
3182             &drr, &zcksum, top_zfs, cleanup_fd, action_handlep));
3183     }
3184 }

3186 /*
3187  * Restores a backup of tosnap from the file descriptor specified by infd.
3188  * Return 0 on total success, -2 if some things couldn't be
3189  * destroyed/rename/promoted, -1 if some things couldn't be received.
3190  * (-1 will override -2).
3191 */
3192 int
3193 zfs_receive(libzfs_handle_t *hdl, const char *tosnap, recvflags_t *flags,
3194     int infd, avl_tree_t *stream_avl)
3195 {
3196     char *top_zfs = NULL;
3197     int err;
3198     int cleanup_fd;
3199     uint64_t action_handle = 0;

3201     cleanup_fd = open(ZFS_DEV, O_RDWR|O_EXCL);
3202     VERIFY(cleanup_fd >= 0);

3204     err = zfs_receive_impl(hdl, tosnap, flags, infd, NULL, NULL,
3205         stream_avl, &top_zfs, cleanup_fd, &action_handle);

3207     VERIFY(0 == close(cleanup_fd));

3209     if (err == 0 && !flags->nomount && top_zfs) {
3210         zfs_handle_t *zhp;
3211         prop_changelist_t *clp;

3213         zhp = zfs_open(hdl, top_zfs, ZFS_TYPE_FILESYSTEM);
3214         if (zhp != NULL) {
3215             clp = changelist_gather(zhp, ZFS_PROP_MOUNTPOINT,
3216                 CL_GATHER_MOUNT_ALWAYS, 0);
3217             zfs_close(zhp);
3218             if (clp != NULL) {
3219                 /* mount and share received datasets */
3220                 err = changelist_postfix(clp);
3221                 changelist_free(clp);
3222             }
3223         }
3224         if (zhp == NULL || clp == NULL || err)
3225             err = -1;
3226     }
3227     if (top_zfs)
3228         free(top_zfs);

3230     return (err);
3231 }
```