

```

*****
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);
731     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)
740         rv = zfs_iter_filesystems(zhp, send_iterate_fs, sd);

742     sd->parent_fromsnap_guid = parent_fromsnap_guid_save;

744     zfs_close(zhp);

```

```

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;
1944     char newname[ZFS_MAXNAMELEN], guidname[32];
1945     int error;
1946     boolean_t needagain, progress, recursive;
1947     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);

```

```

1994     snapelem; snapelem = nvlist_next_nvpair(snaps, snapelem) {
1995         uint64_t thisguid;

1997         VERIFY(0 == nvpair_value_uint64(snapelem, &thisguid));
1998         stream_nvfs = fsavl_find(stream_avl, thisguid, NULL);

2000         if (stream_nvfs != NULL)
2001             break;
2002     }

2004     /* check for promote */
2005     (void) nvlist_lookup_uint64(stream_nvfs, "origin",
2006         &stream_oringuid);
2007     if (stream_nvfs && oringuid != stream_oringuid) {
2008         switch (created_before(hdl, local_avl,
2009             stream_oringuid, oringuid)) {
2010             case 1: {
2011                 /* promote it! */
2012                 zfs_cmd_t zc = { 0 };
2013                 nvlist_t *origin_nvfs;
2014                 char *origin_fsname;

2016                 if (flags->verbose)
2017                     (void) printf("promoting %s\n", fsname);

2019                 origin_nvfs = fsavl_find(local_avl, oringuid,
2020                     NULL);
2021                 VERIFY(0 == nvlist_lookup_string(origin_nvfs,
2022                     "name", &origin_fsname));
2023                 (void) strncpy(zc.zc_value, origin_fsname,
2024                     sizeof(zc.zc_value));
2025                 (void) strncpy(zc.zc_name, fsname,
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             }
2032             default:
2033                 break;
2034             case -1:
2035                 fsavl_destroy(local_avl);
2036                 nvlist_free(local_nv);
2037                 nvlist_free(deleted);
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     }

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;

2056         nextsnapelem = nvlist_next_nvpair(snaps, snapelem);

2058         VERIFY(0 == nvpair_value_uint64(snapelem, &thisguid));
2059         found = fsavl_find(stream_avl, thisguid,

```

```

2060         &stream_snapname);

2062         /* check for delete */
2063         if (found == NULL) {
2064             char name[ZFS_MAXNAMELEN];

2066             if (!flags->force)
2067                 continue;

2069             (void) snprintf(name, sizeof(name), "%s%s",
2070                 fsname, nvpair_name(snapelem));

2072             error = recv_destroy(hdl, name,
2073                 strlen(fsname)+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         }

2085         stream_nvfs = found;

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 };

2092             zc.zc_cookie = B_TRUE; /* received */
2093             (void) snprintf(zc.zc_name, sizeof(zc.zc_name),
2094                 "%s%s", fsname, 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         }

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];

2109             (void) snprintf(name, sizeof(name), "%s%s",
2110                 fsname, nvpair_name(snapelem));
2111             (void) snprintf(tryname, sizeof(name), "%s%s",
2112                 fsname, stream_snapname);

2114             error = recv_rename(hdl, name, tryname,
2115                 strlen(fsname)+1, newname, flags);
2116             if (error)
2117                 needagain = B_TRUE;
2118             else
2119                 progress = B_TRUE;
2120         }

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;
2131
2132         error = recv_destroy(hdl, fsname, strlen(tofs)+1,
2133                             newname, flags);
2134         if (error)
2135             needagain = B_TRUE;
2136         else
2137             progress = B_TRUE;
2138         (void) sprintf(guidname, "%llu",
2139                       (u_longlong_t)parent_fromsnap_guid);
2140         fnvlist_add_boolean(deleted, guidname);
2141     #endif /* ! codereview */
2142     }
2143     continue;
2144
2145     if (fromguid == 0) {
2146         if (flags->verbose) {
2147             (void) printf("local fs %s does not have "
2148                          "fromsnap (%s in stream); must have "
2149                          "been deleted locally; ignoring\n",
2150                          fsname, fromsnap);
2151         }
2152     }
2153     continue;
2154
2155     VERIFY(0 == nvlist_lookup_string(stream_nvfs,
2156                                     "name", &stream_fsname));
2157     VERIFY(0 == nvlist_lookup_uint64(stream_nvfs,
2158                                     "parentfromsnap", &stream_parent_fromsnap_guid));
2159
2160     s1 = strrchr(fsname, '/');
2161     s2 = strrchr(stream_fsname, '/');
2162
2163     /*
2164     * Check if we're going to rename based on parent guid change
2165     * and the current parent guid was also deleted. If it was then
2166     * the rename will fail so avoid this and force an early retry
2167     * to determine the new parent_fromsnap_guid.
2168     */
2169     if (stream_parent_fromsnap_guid != 0 &&
2170         parent_fromsnap_guid != 0 &&
2171         stream_parent_fromsnap_guid != parent_fromsnap_guid) {
2172         (void) sprintf(guidname, "%llu",
2173                       (u_longlong_t)parent_fromsnap_guid);
2174         if (nvlist_exists(deleted, guidname)) {
2175             progress = B_TRUE;
2176             needagain = B_TRUE;
2177             goto doagain;
2178         }
2179     }
2180
2181     #endif /* ! codereview */
2182     /*
2183     * Check for rename. If the exact receive path is specified, it
2184     * does not count as a rename, but we still need to check the
2185     * datasets beneath it.
2186     */
2187     if ((stream_parent_fromsnap_guid != 0 &&
2188         parent_fromsnap_guid != 0 &&
2189         stream_parent_fromsnap_guid != parent_fromsnap_guid) ||
2190         ((flags->isprefix || strcmp(tofs, fsname) != 0) &&
2191         (s1 != NULL) && (s2 != NULL) && strcmp(s1, s2) != 0)) {
2192         nvlist_t *parent;

```

```

2192     char tryname[ZFS_MAXNAMELEN];
2193
2194     parent = fsavl_find(local_avl,
2195                         stream_parent_fromsnap_guid, NULL);
2196     /*
2197     * NB: parent might not be found if we used the
2198     * tosnap for stream_parent_fromsnap_guid,
2199     * because the parent is a newly-created fs;
2200     * we'll be able to rename it after we recv the
2201     * new fs.
2202     */
2203     if (parent != NULL) {
2204         char *pname;
2205
2206         VERIFY(0 == nvlist_lookup_string(parent, "name",
2207                                         &pname));
2208         (void) snprintf(tryname, sizeof(tryname),
2209                        "%s%s", pname, strrchr(stream_fsname, '/'));
2210     } else {
2211         tryname[0] = '\0';
2212         if (flags->verbose) {
2213             (void) printf("local fs %s new parent "
2214                          "not found\n", fsname);
2215         }
2216     }
2217
2218     newname[0] = '\0';
2219
2220     error = recv_rename(hdl, fsname, tryname,
2221                        strlen(tofs)+1, newname, flags);
2222
2223     if (renamed != NULL && newname[0] != '\0') {
2224         VERIFY(0 == nvlist_add_boolean(renamed,
2225                                       newname));
2226     }
2227
2228     if (error)
2229         needagain = B_TRUE;
2230     else
2231         progress = B_TRUE;
2232 }
2233
2234 doagain:
2235 #endif /* ! codereview */
2236 fsavl_destroy(local_avl);
2237 nvlist_free(local_nv);
2238 nvlist_free(deleted);
2239 #endif /* ! codereview */
2240
2241 if (needagain && progress) {
2242     /* do another pass to fix up temporary names */
2243     if (flags->verbose)
2244         (void) printf("another pass:\n");
2245     goto again;
2246 }
2247
2248 return (needagain);
2249 }
2250
2251 static int
2252 zfs_receive_package(libzfs_handle_t *hdl, int fd, const char *destname,
2253                    recvflags_t *flags, dmuf_replay_record_t *drr, zio_cksum_t *zc,
2254                    char **top_zfs, int cleanup_fd, uint64_t *action_handlep)
2255 {
2256     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     dmuf_replay_record_t drre;
2265     int error;
2266     boolean_t anyerr = B_FALSE;
2267     boolean_t softerr = B_FALSE;
2268     boolean_t recursive;

2270     (void) snprintf(errbuf, sizeof (errbuf), dgettext(TEXT_DOMAIN,
2271     "cannot receive"));

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);

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     }

2290     recursive = (nvlist_lookup_boolean(stream_nv, "not_recursive") ==
2291     ENOENT);

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     }

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         error = zfs_error(hdl, EZFS_BADSTREAM, errbuf);
2325         goto out;
2326     }

2328     (void) nvlist_lookup_string(stream_nv, "fromsnap", &fromsnap);

2330     if (drr->drr_payloadlen != 0) {
2331         nvlist_t *stream_fss;

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         }

2342         if (fromsnap != NULL) {
2343             nvlist_t *renamed;
2344             nvpair_t *pair = NULL;

2346             (void) strlcpy(tofs, destname, ZFS_MAXNAMELEN);
2347             if (flags->isprefix) {
2348                 struct drr_begin *drrb = &drr->drr_u.drr_begin;
2349                 int i;

2351                 if (flags->istail) {
2352                     cp = strrchr(drrb->drr_toname, '/');
2353                     if (cp == NULL) {
2354                         (void) strlcat(tofs, "/",
2355                         ZFS_MAXNAMELEN);
2356                         i = 0;
2357                     } else {
2358                         i = (cp - drrb->drr_toname);
2359                     }
2360                 } else {
2361                     i = strcspn(drrb->drr_toname, "@");
2362                 }
2363                 /* zfs_receive_one() will create_parents() */
2364                 (void) strlcat(tofs, &drrb->drr_toname[i],
2365                 ZFS_MAXNAMELEN);
2366                 *strchr(tofs, '@') = '\0';
2367             }

2369             if (recursive && !flags->dryrun && !flags->nomount) {
2370                 VERIFY(0 == nvlist_alloc(&renamed,
2371                 NV_UNIQUE_NAME, 0));
2372             }

2374             softerr = recv_incremental_replication(hdl, tofs, flags,
2375             stream_nv, stream_avl, renamed);

2377             /* Unmount renamed filesystems before receiving. */
2378             while ((pair = nvlist_next_nvpair(renamed,
2379             pair)) != NULL) {
2380                 zfs_handle_t *zhp;
2381                 prop_changelist_t *clp = NULL;

2383                 zhp = zfs_open(hdl, nvpair_name(pair),
2384                 ZFS_TYPE_FILESYSTEM);
2385                 if (zhp != NULL) {
2386                     clp = changelist_gather(zhp,
2387                     ZFS_PROP_MOUNTPOINT, 0, 0);
2388                     zfs_close(zhp);
2389                     if (clp != NULL) {

```

```

2390         softerr |=
2391             changelist_prefix(clp);
2392         changelist_free(clp);
2393     }
2394 }
2395
2397     nvlist_free(renamed);
2398 }
2399
2401 /*
2402  * Get the fs specified by the first path in the stream (the top level
2403  * specified by 'zfs send') and pass it to each invocation of
2404  * zfs_receive_one().
2405  */
2406 (void) strcpy(sendfs, drr->drr_u.drr_begin.drr_toname,
2407             ZFS_MAXNAMELEN);
2408 if ((cp = strchr(sendfs, '@')) != NULL)
2409     *cp = '\0';
2411 /* Finally, receive each contained stream */
2412 do {
2413     /*
2414      * we should figure out if it has a recoverable
2415      * error, in which case do a recv_skip() and drive on.
2416      * Note, if we fail due to already having this guid,
2417      * zfs_receive_one() will take care of it (ie,
2418      * recv_skip() and return 0).
2419      */
2420     error = zfs_receive_impl(hdl, destname, flags, fd,
2421                             sendfs, stream_nv, stream_avl, top_zfs, cleanup_fd,
2422                             action_handlep);
2423     if (error == ENODATA) {
2424         error = 0;
2425         break;
2426     }
2427     anyerr |= error;
2428 } while (error == 0);
2430 if (drr->drr_payloadlen != 0 && fromsnap != NULL) {
2431     /*
2432      * Now that we have the fs's they sent us, try the
2433      * renames again.
2434      */
2435     softerr = recv_incremental_replication(hdl, tofs, flags,
2436         stream_nv, stream_avl, NULL);
2437 }
2439 out:
2440     fsavl_destroy(stream_avl);
2441     if (stream_nv)
2442         nvlist_free(stream_nv);
2443     if (softerr)
2444         error = -2;
2445     if (anyerr)
2446         error = -1;
2447     return (error);
2448 }
2450 static void
2451 trunc_prop_errs(int truncated)
2452 {
2453     ASSERT(truncated != 0);
2455     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 }
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];
2470     (void) snprintf(errbuf, sizeof (errbuf), dgettext(TEXT_DOMAIN,
2471         "cannot receive:"));
2473     /* XXX would be great to use lseek if possible... */
2474     drr = buf;
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);
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 substream header"));
2487                 return (zfs_error(hdl, EZFS_BADSTREAM, errbuf));
2488             }
2489             break;
2491         case DRR_END:
2492             free(buf);
2493             return (0);
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;
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         case DRR_SPILL:
2515             if (byteswap) {
2516                 drr->drr_u.drr_write.drr_length =
2517                     BSWAP_64(drr->drr_u.drr_spill.drr_length);
2518             }
2519             (void) recv_read(hdl, fd, buf,
2520                 drr->drr_u.drr_spill.drr_length, B_FALSE, NULL);
2521             break;

```

```

2522         case DRR_WRITE_BYREF:
2523         case DRR_FREEOBJECTS:
2524         case DRR_FREE:
2525             break;

2527         default:
2528             zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
2529                 "invalid record type"));
2530             return (zfs_error(hdl, EZFS_BADSTREAM, errbuf));
2531     }
2532 }

2534     free(buf);
2535     return (-1);
2536 }

2538 /*
2539  * Restores a backup of tosnap from the file descriptor specified by infd.
2540  */
2541 static int
2542 zfs_receive_one(libzfs_handle_t *hdl, int infd, const char *tosnap,
2543     recvflags_t *flags, dmu_replay_record_t *drr,
2544     dmu_replay_record_t *drr_noswap, const char *sendfs,
2545     nvlist_t *stream_nv, avl_tree_t *stream_avl, char **top_zfs, int cleanup_fd,
2546     uint64_t *action_handlep)
2547 {
2548     zfs_cmd_t zc = { 0 };
2549     time_t begin_time;
2550     int ioctl_err, ioctl_errno, err;
2551     char *cp;
2552     struct drr_begin *drrb = &drr->drr_u.drr_begin;
2553     char errbuf[1024];
2554     char prop_errbuf[1024];
2555     const char *chopprefix;
2556     boolean_t newfs = B_FALSE;
2557     boolean_t stream_wantsnewfs;
2558     uint64_t parent_snapguid = 0;
2559     prop_changelist_t *clp = NULL;
2560     nvlist_t *snapprops_nvlist = NULL;
2561     zprop_errflags_t prop_errflags;
2562     boolean_t recursive;

2564     begin_time = time(NULL);

2566     (void) snprintf(errbuf, sizeof (errbuf), dgettext(TEXT_DOMAIN,
2567         "cannot receive"));

2569     recursive = (nvlist_lookup_boolean(stream_nv, "not_recursive") ==
2570         ENOENT);

2572     if (stream_avl != NULL) {
2573         char *snapname;
2574         nvlist_t *fs = fsavl_find(stream_avl, drrb->drr_toguid,
2575             &snapname);
2576         nvlist_t *props;
2577         int ret;

2579         (void) nvlist_lookup_uint64(fs, "parentfromsnap",
2580             &parent_snapguid);
2581         err = nvlist_lookup_nvlist(fs, "props", &props);
2582         if (err)
2583             VERIFY(0 == nvlist_alloc(&props, NV_UNIQUE_NAME, 0));

2585         if (flags->canmountoff) {
2586             VERIFY(0 == nvlist_add_uint64(props,
2587                 zfs_prop_to_name(ZFS_PROP_CANMOUNT), 0));

```

```

2588     }
2589     ret = zcmd_write_src_nvlist(hdl, &zc, props);
2590     if (err)
2591         nvlist_free(props);

2593     if (0 == nvlist_lookup_nvlist(fs, "snapprops", &props)) {
2594         VERIFY(0 == nvlist_lookup_nvlist(props,
2595             snapname, &snapprops_nvlist));
2596     }

2598     if (ret != 0)
2599         return (-1);
2600 }

2602     cp = NULL;

2604     /*
2605     * Determine how much of the snapshot name stored in the stream
2606     * we are going to tack on to the name they specified on the
2607     * command line, and how much we are going to chop off.
2608     *
2609     * If they specified a snapshot, chop the entire name stored in
2610     * the stream.
2611     */
2612     if (flags->istail) {
2613         /*
2614         * A filesystem was specified with -e. We want to tack on only
2615         * the tail of the sent snapshot path.
2616         */
2617         if (strchr(tosnap, '@')) {
2618             zfs_error_aux(hdl, dgettext(TEXT_DOMAIN, "invalid "
2619                 "argument - snapshot not allowed with -e"));
2620             return (zfs_error(hdl, EZFS_INVALIDNAME, errbuf));
2621         }

2623         chopprefix = strrchr(sendfs, '/');

2625         if (chopprefix == NULL) {
2626             /*
2627             * The tail is the poolname, so we need to
2628             * prepend a path separator.
2629             */
2630             int len = strlen(drrb->drr_toname);
2631             cp = malloc(len + 2);
2632             cp[0] = '/';
2633             (void) strcpy(&cp[1], drrb->drr_toname);
2634             chopprefix = cp;
2635         } else {
2636             chopprefix = drrb->drr_toname + (chopprefix - sendfs);
2637         }
2638     } else if (flags->isprefix) {
2639         /*
2640         * A filesystem was specified with -d. We want to tack on
2641         * everything but the first element of the sent snapshot path
2642         * (all but the pool name).
2643         */
2644         if (strchr(tosnap, '@')) {
2645             zfs_error_aux(hdl, dgettext(TEXT_DOMAIN, "invalid "
2646                 "argument - snapshot not allowed with -d"));
2647             return (zfs_error(hdl, EZFS_INVALIDNAME, errbuf));
2648         }

2650         chopprefix = strchr(drrb->drr_toname, '/');
2651         if (chopprefix == NULL)
2652             chopprefix = strchr(drrb->drr_toname, '@');
2653     } else if (strchr(tosnap, '@') == NULL) {

```

```

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     choppprefix = 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     choppprefix = drrb->drr_toname + strlen(drrb->drr_toname);
2668 }

2670 ASSERT(strstr(drrb->drr_toname, sendfs) == drrb->drr_toname);
2671 ASSERT(choppprefix > drrb->drr_toname);
2672 ASSERT(choppprefix <= drrb->drr_toname + strlen(drrb->drr_toname));
2673 ASSERT(choppprefix[0] == '/' || choppprefix[0] == '@' ||
2674     choppprefix[0] == '\0');

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, choppprefix, 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 }

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 }

2703 stream_wantsnewfs = (drrb->drr_fromguid == NULL ||
2704     (drrb->drr_flags & DRR_FLAG_CLONE));

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) snprintf(errbuf, sizeof (errbuf), dgettext(TEXT_DOMAIN,
2712         "cannot receive new filesystem stream"));

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) snprintf(errbuf, sizeof (errbuf), dgettext(TEXT_DOMAIN,
2733             "cannot receive incremental stream"));

2734         (void) strcpy(zc.zc_name, zc.zc_value);
2735         *strchr(zc.zc_name, '@') = '\0';

2737     /*
2738      * If the exact receive path was specified and this is the
2739      * topmost path in the stream, then if the fs does not exist we
2740      * should look no further.
2741      */
2742     if ((flags->isprefix || (*choppprefix = drrb->drr_toname +
2743         strlen(sendfs)) != '\0' && *choppprefix != '@') &&
2744         !zfs_dataset_exists(hdl, zc.zc_name, ZFS_TYPE_DATASET)) {
2745         char snap[ZFS_MAXNAMELEN];
2746         (void) strcpy(snap, strrchr(zc.zc_value, '@'));
2747         if (guid_to_name(hdl, zc.zc_name, drrb->drr_fromguid,
2748             zc.zc_value) == 0) {
2749             *strchr(zc.zc_value, '@') = '\0';
2750             (void) strcat(zc.zc_value, snap);
2751         }
2752     }

2754     (void) strcpy(zc.zc_name, zc.zc_value);
2755     *strchr(zc.zc_name, '@') = '\0';

2757     if (zfs_dataset_exists(hdl, zc.zc_name, ZFS_TYPE_DATASET)) {
2758         zfs_handle_t *zhp;

2760         /*
2761          * Destination fs exists. Therefore this should either
2762          * be an incremental, or the stream specifies a new fs
2763          * (full stream or clone) and they want us to blow it
2764          * away (and have therefore specified -F and removed any
2765          * snapshots).
2766          */
2767         if (stream_wantsnewfs) {
2768             if (!flags->force) {
2769                 zcmd_free_nvlists(&zc);
2770                 zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
2771                     "destination '%s' exists\n"
2772                     "must specify -F to overwrite it"),
2773                     zc.zc_name);
2774                 return (zfs_error(hdl, EZFS_EXISTS, errbuf));
2775             }
2776             if (ioctl(hdl->libzfs_fd, ZFS_IOC_SNAPSHOT_LIST_NEXT,
2777                 &zc) == 0) {
2778                 zcmd_free_nvlists(&zc);
2779                 zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
2780                     "destination has snapshots (eg. %s)\n"
2781                     "must destroy them to overwrite it"),
2782                     zc.zc_name);
2783             }
2784         }
2785     }

```

```

2786         return (zfs_error(hdl, EZFS_EXISTS, errbuf));
2787     }
2788 }
2790 if ((zhp = zfs_open(hdl, zc.zc_name,
2791     ZFS_TYPE_FILESYSTEM | ZFS_TYPE_VOLUME)) == NULL) {
2792     zcmd_free_nvlists(&zc);
2793     return (-1);
2794 }
2796 if (stream_wantsnewfs &&
2797     zhp->zfs_dmustats.dds_origin[0] {
2798     zcmd_free_nvlists(&zc);
2799     zfs_close(zhp);
2800     zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
2801         "destination '%s' is a clone\n"
2802         "must destroy it to overwrite it"),
2803         zc.zc_name);
2804     return (zfs_error(hdl, EZFS_EXISTS, errbuf));
2805 }
2807 if (!flags->dryrun && zhp->zfs_type == ZFS_TYPE_FILESYSTEM &&
2808     stream_wantsnewfs) {
2809     /* We can't do online recv in this case */
2810     clp = changelist_gather(zhp, ZFS_PROP_NAME, 0, 0);
2811     if (clp == NULL) {
2812         zfs_close(zhp);
2813         zcmd_free_nvlists(&zc);
2814         return (-1);
2815     }
2816     if (changelist_prefix(clp) != 0) {
2817         changelist_free(clp);
2818         zfs_close(zhp);
2819         zcmd_free_nvlists(&zc);
2820         return (-1);
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 }
2840 /*
2841 * Trim off the final dataset component so we perform the
2842 * recvbackup ioctl to the filesystems's parent.
2843 */
2844 *cp = '\0';
2846 if (flags->isprefix && !flags->istail && !flags->dryrun &&
2847     create_parents(hdl, zc.zc_value, strlen(tosnap)) != 0) {
2848     zcmd_free_nvlists(&zc);
2849     return (zfs_error(hdl, EZFS_BADRESTORE, errbuf));
2850 }

```

```

2852     newfs = B_TRUE;
2853 }
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 }
2866 if (flags->dryrun) {
2867     zcmd_free_nvlists(&zc);
2868     return (recv_skip(hdl, infd, flags->byteswap));
2869 }
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;
2876 err = ioctl_err = zfs_ioctl(hdl, ZFS_IOC_RECV, &zc);
2877 ioctl_errno = errno;
2878 prop_errflags = (zprop_errflags_t)zc.zc_obj;
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));
2885     nvpair_t *prop_err = NULL;
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;
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) snprintf(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 }
2910 zc.zc_nvlist_dst = 0;
2911 zc.zc_nvlist_dst_size = 0;
2912 zcmd_free_nvlists(&zc);
2914 if (err == 0 && snapprops_nvlist) {
2915     zfs_cmd_t zc2 = { 0 };
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 }
2924
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     avl_tree_t *local_avl;
2932     nvlist_t *local_nv, *fs;
2933     cp = strchr(zc.zc_value, '@');
2934
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);
2947
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 = rcv_skip(hdl, infd,
2954                 flags->byteswap);
2955         }
2956     }
2957     *cp = '@';
2958 }
2959
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 }
3012
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;
3021
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 }
3040
3041 if (clp) {
3042     err |= changelist_postfix(clp);
3043     changelist_free(clp);
3044 }
3045
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_NOESTORE) {
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     }
3059     if (err || ioctl_err)
3060         return (-1);
3062     *action_handlep = zc.zc_action_handle;
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));
3074         (void) printf("received %sB stream in %lu seconds (%sB/sec)\n",
3075             buf1, delta, buf2);
3076     }
3078     return (0);
3079 }
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     dmuf_replay_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 hdrtype;
3094     (void) snprintf(errbuf, sizeof (errbuf), dgettext(TEXT_DOMAIN,
3095         "cannot receive"));
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     }
3104     /* read in the BEGIN record */
3105     if (0 != (err = recv_read(hdl, infd, &drr, sizeof (drr), B_FALSE,
3106         &zcksum)))
3107         return (err);
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     }
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;
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     }
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     }
3144     featureflags = DMU_GET_FEATUREFLAGS(drrb->drr_versioninfo);
3145     hdrtype = DMU_GET_STREAM_HDRTYPE(drrb->drr_versioninfo);
3147     if (!DMU_STREAM_SUPPORTED(featureflags) ||
3148         (hdrtype != DMU_SUBSTREAM && hdrtype != 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     }
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     }
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         &addr, &zcksum, top_zfs, cleanup_fd, action_handle));
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/renamed/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
3202     cleanup_fd = open(ZFS_DEV, O_RDWR|O_EXCL);
3203     VERIFY(cleanup_fd >= 0);
3204
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 }
```