```
**********************************************************
   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_

/*
 * recursively generate nvlists describing datasets.  See comment
 * for the data structure send_data_t above for description of contents
 * of the nvlist.
 */
static int
send_iterate_fs(zfs_handle_t *zhp, void *arg)
{
        send_data_t *sd = arg;
        nvlist_t *nvfs, *nv;
        int rv = 0;
        uint64_t parent_fromsnap_guid_save = sd->parent_fromsnap_guid;
        uint64_t guid = zhp->zfs_dmustats.dds_guid;
        char guidstring[64];

        VERIFY(0 == nvlist_alloc(&nvfs, NV_UNIQUE_NAME, 0));
        VERIFY(0 == nvlist_add_string(nvfs, "name", zhp->zfs_name));
        VERIFY(0 == nvlist_add_uint64(nvfs, "parentfromsnap",
            sd->parent_fromsnap_guid));

        if (zhp->zfs_dmustats.dds_origin[0]) {
                zfs_handle_t *origin = zfs_open(zhp->zfs_hdl,
                    zhp->zfs_dmustats.dds_origin, ZFS_TYPE_SNAPSHOT);
                if (origin == NULL)
                        return (-1);
                VERIFY(0 == nvlist_add_uint64(nvfs, "origin",
                    origin->zfs_dmustats.dds_guid));
        }

        /* iterate over props */
        VERIFY(0 == nvlist_alloc(&nv, NV_UNIQUE_NAME, 0));
        send_iterate_prop(zhp, nv);
        VERIFY(0 == nvlist_add_nvlist(nvfs, "props", nv));
        nvlist_free(nv);

        /* iterate over snaps, and set sd->parent_fromsnap_guid */
        sd->parent_fromsnap_guid = 0;
        VERIFY(0 == nvlist_alloc(&sd->parent_snaps, NV_UNIQUE_NAME, 0));
        VERIFY(0 == nvlist_alloc(&sd->snapprops, NV_UNIQUE_NAME, 0));
        (void) zfs_iter_snapshots_sorted(zhp, send_iterate_snap, sd);
        (void) zfs_iter_snapshots(zhp, send_iterate_snap, sd);
        VERIFY(0 == nvlist_add_nvlist(nvfs, "snaps", sd->parent_snaps));
        VERIFY(0 == nvlist_add_nvlist(nvfs, "snapprops", sd->snapprops));
        nvlist_free(sd->parent_snaps);
        nvlist_free(sd->snapprops);

        /* add this fs to nvlist */
        (void) snprintf(guidstring, sizeof (guidstring),
            "0x%llx", (longlong_t)guid);
        VERIFY(0 == nvlist_add_nvlist(sd->fss, guidstring, nvfs));
        nvlist_free(nvfs);

        /* iterate over children */
        if (sd->recursive)
                rv = zfs_iter_filesystems(zhp, send_iterate_fs, sd);

        sd->parent_fromsnap_guid = parent_fromsnap_guid_save;

        zfs_close(zhp);
```

```
        return (rv);
}
______unchanged_portion_omitted_

static int
recv_incremental_replication(libzfs_handle_t *hdl, const char *tofs,
    recvflags_t *flags, nvlist_t *stream_nv, avl_tree_t *stream_avl,
    nvlist_t *renamed)
{
        nvlist_t *local_nv, *deleted = NULL;
        nvlist_t *local_nv;
        avl_tree_t *local_avl;
        nvpair_t *fselem, *nextfselem;
        char *fromsnap;
        char newname[ZFS_MAXNAMELEN], guidname[32];
        char newname[ZFS_MAXNAMELEN];
        int error;
        boolean_t needagain, progress, recursive;
        char *s1, *s2;

        VERIFY(0 == nvlist_lookup_string(stream_nv, "fromsnap", &fromsnap));

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

        if (flags->dryrun)
                return (0);

again:
        needagain = progress = B_FALSE;

        if ((error = gather_nvlist(hdl, tofs, fromsnap, NULL,
            recursive, &local_nv, &local_avl)) != 0)
                return (error);

        deleted = fnvlist_alloc();

#endif /* ! codereview */
        /*
         * Process deletes and renames
         */
        for (fselem = nvlist_next_nvpair(local_nv, NULL);
            fselem; fselem = nextfselem) {
                nvlist_t *nvfs, *snaps;
                nvlist_t *stream_nvfs = NULL;
                nvpair_t *snapelem, *nextsnapelem;
                uint64_t fromguid = 0;
                uint64_t originguid = 0;
                uint64_t stream_originguid = 0;
                uint64_t parent_fromsnap_guid, stream_parent_fromsnap_guid;
                char *fsname, *stream_fsname;

                nextfselem = nvlist_next_nvpair(local_nv, fselem);

                VERIFY(0 == nvpair_value_nvlist(fselem, &nvfs));
                VERIFY(0 == nvlist_lookup_nvlist(nvfs, "snaps", &snaps));
                VERIFY(0 == nvlist_lookup_string(nvfs, "name", &fsname));
                VERIFY(0 == nvlist_lookup_uint64(nvfs, "parentfromsnap",
                    &parent_fromsnap_guid));
                (void) nvlist_lookup_uint64(nvfs, "origin", &originguid);

                /*
                 * First find the stream's fs, so we can check for
                 * a different origin (due to "zfs promote")
                 */
                for (snapelem = nvlist_next_nvpair(snaps, NULL);
                    snapelem; snapelem = nvlist_next_nvpair(snaps, snapelem)) {
                        uint64_t thisguid;

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

                        if (stream_nvfs != NULL)
                                break;
                }

                /* check for promote */
                (void) nvlist_lookup_uint64(stream_nvfs, "origin",
                    &stream_originguid);
                if (stream_nvfs && originguid != stream_originguid) {
                        switch (created_before(hdl, local_avl,
                            stream_originguid, originguid)) {
                        case 1: {
                                /* promote it! */
                                zfs_cmd_t zc = { 0 };
                                nvlist_t *origin_nvfs;
                                char *origin_fsname;

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

                                origin_nvfs = fsavl_find(local_avl, originguid,
                                    NULL);
                                VERIFY(0 == nvlist_lookup_string(origin_nvfs,
                                    "name", &origin_fsname));
                                (void) strlcpy(zc.zc_value, origin_fsname,
                                    sizeof (zc.zc_value));
                                (void) strlcpy(zc.zc_name, fsname,
                                    sizeof (zc.zc_name));
                                error = zfs_ioctl(hdl, ZFS_IOC_PROMOTE, &zc);
                                if (error == 0)
                                        progress = B_TRUE;
                                break;
                        }
                        default:
                                break;
                        case -1:
                                fsavl_destroy(local_avl);
                                nvlist_free(local_nv);
                                nvlist_free(deleted);
#endif /* ! codereview */
                                return (-1);
                        }
                        /*
                         * We had/have the wrong origin, therefore our
                         * list of snapshots is wrong.  Need to handle
                         * them on the next pass.
                         */
                        needagain = B_TRUE;
                        continue;
                }

                for (snapelem = nvlist_next_nvpair(snaps, NULL);
                    snapelem; snapelem = nextsnapelem) {
                        uint64_t thisguid;
                        char *stream_snapname;
                        nvlist_t *found, *props;

                        nextsnapelem = nvlist_next_nvpair(snaps, snapelem);

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

```
                            &stream_snapname);

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

                                if (!flags->force)
                                        continue;

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

                                error = recv_destroy(hdl, name,
                                    strlen(fsname)+1, newname, flags);
                                if (error)
                                        needagain = B_TRUE;
                                else
                                        progress = B_TRUE;
                                (void) sprintf(guidname, "%llu",
                                    (u_longlong_t)thisguid);
                                fnvlist_add_boolean(deleted, guidname);
#endif /* ! codereview */
                                continue;
                        }

                        stream_nvfs = found;

                        if (0 == nvlist_lookup_nvlist(stream_nvfs, "snapprops",
                            &props) && 0 == nvlist_lookup_nvlist(props,
                            stream_snapname, &props)) {
                                zfs_cmd_t zc = { 0 };

                                zc.zc_cookie = B_TRUE; /* received */
                                (void) snprintf(zc.zc_name, sizeof (zc.zc_name),
                                    "%s@%s", fsname, nvpair_name(snapelem));
                                if (zcmd_write_src_nvlist(hdl, &zc,
                                    props) == 0) {
                                        (void) zfs_ioctl(hdl,
                                            ZFS_IOC_SET_PROP, &zc);
                                        zcmd_free_nvlists(&zc);
                                }
                        }

                        /* check for different snapname */
                        if (strcmp(nvpair_name(snapelem),
                            stream_snapname) != 0) {
                                char name[ZFS_MAXNAMELEN];
                                char tryname[ZFS_MAXNAMELEN];

                                (void) snprintf(name, sizeof (name), "%s@%s",
                                    fsname, nvpair_name(snapelem));
                                (void) snprintf(tryname, sizeof (name), "%s@%s",
                                    fsname, stream_snapname);

                                error = recv_rename(hdl, name, tryname,
                                    strlen(fsname)+1, newname, flags);
                                if (error)
                                        needagain = B_TRUE;
                                else
                                        progress = B_TRUE;
                        }

                        if (strcmp(stream_snapname, fromsnap) == 0)
                                fromguid = thisguid;
                }
                /* check for delete */
                if (stream_nvfs == NULL) {
                        if (!flags->force)
                                continue;

                        error = recv_destroy(hdl, fsname, strlen(tofs)+1,
                            newname, flags);
                        if (error)
                                needagain = B_TRUE;
                        else
                                progress = B_TRUE;
                        (void) sprintf(guidname, "%llu",
                            (u_longlong_t)parent_fromsnap_guid);
                        fnvlist_add_boolean(deleted, guidname);
#endif /* ! codereview */
                        continue;
                }

                if (fromguid == 0) {
                        if (flags->verbose) {
                                (void) printf("local fs %s does not have "
                                    "fromsnap (%s in stream); must have "
                                    "been deleted locally; ignoring\n",
                                    fsname, fromsnap);
                        }
                        continue;
                }

                VERIFY(0 == nvlist_lookup_string(stream_nvfs,
                    "name", &stream_fsname));
                VERIFY(0 == nvlist_lookup_uint64(stream_nvfs,
                    "parentfromsnap", &stream_parent_fromsnap_guid));

                s1 = strrchr(fsname, '/');
                s2 = strrchr(stream_fsname, '/');

                /*
                 * Check if we're going to rename based on parent guid change
                 * and the current parent guid was also deleted. If it was then
                 * the rename will fail so avoid this and force an early retry
                 * to determine the new parent_fromsnap_guid.
                 */
                if (stream_parent_fromsnap_guid != 0 &&
                    parent_fromsnap_guid != 0 &&
                    stream_parent_fromsnap_guid != parent_fromsnap_guid) {
                        (void) sprintf(guidname, "%llu",
                            (u_longlong_t)parent_fromsnap_guid);
                        if (nvlist_exists(deleted, guidname)) {
                                progress = B_TRUE;
                                needagain = B_TRUE;
                                goto doagain;
                        }
                }

                /*
#endif /* ! codereview */
                 * Check for rename. If the exact receive path is specified, it
                 * does not count as a rename, but we still need to check the
                 * datasets beneath it.
                 */
                if ((stream_parent_fromsnap_guid != 0 &&
                    parent_fromsnap_guid != 0 &&
                    stream_parent_fromsnap_guid != parent_fromsnap_guid) ||
                    ((flags->isprefix || strcmp(tofs, fsname) != 0) &&
                    (s1 != NULL) && (s2 != NULL) && strcmp(s1, s2) != 0)) {
                        nvlist_t *parent;
```

```
                        char tryname[ZFS_MAXNAMELEN];

                        parent = fsavl_find(local_avl,
                            stream_parent_fromsnap_guid, NULL);
                        /*
                         * NB: parent might not be found if we used the
                         * tosnap for stream_parent_fromsnap_guid,
                         * because the parent is a newly-created fs;
                         * we'll be able to rename it after we recv the
                         * new fs.
                         */
                        if (parent != NULL) {
                                char *pname;

                                VERIFY(0 == nvlist_lookup_string(parent, "name",
                                    &pname));
                                (void) snprintf(tryname, sizeof (tryname),
                                    "%s%s", pname, strrchr(stream_fsname, '/'));
                        } else {
                                tryname[0] = '\0';
                                if (flags->verbose) {
                                        (void) printf("local fs %s new parent "
                                            "not found\n", fsname);
                                }
                        }

                        newname[0] = '\0';

                        error = recv_rename(hdl, fsname, tryname,
                            strlen(tofs)+1, newname, flags);

                        if (renamed != NULL && newname[0] != '\0') {
                                VERIFY(0 == nvlist_add_boolean(renamed,
                                    newname));
                        }

                        if (error)
                                needagain = B_TRUE;
                        else
                                progress = B_TRUE;
                }
        }

doagain:
#endif /* ! codereview */
        fsavl_destroy(local_avl);
        nvlist_free(local_nv);
        nvlist_free(deleted);
#endif /* ! codereview */

        if (needagain && progress) {
                /* do another pass to fix up temporary names */
                if (flags->verbose)
                        (void) printf("another pass:\n");
                goto again;
        }

        return (needagain);
}

static int
zfs_receive_package(libzfs_handle_t *hdl, int fd, const char *destname,
    recvflags_t *flags, dmu_replay_record_t *drr, zio_cksum_t *zc,
    char **top_zfs, int cleanup_fd, uint64_t *action_handlep)
{
        nvlist_t *stream_nv = NULL;
        avl_tree_t *stream_avl = NULL;
        char *fromsnap = NULL;
        char *cp;
        char tofs[ZFS_MAXNAMELEN];
        char sendfs[ZFS_MAXNAMELEN];
        char errbuf[1024];
        dmu_replay_record_t drre;
        int error;
        boolean_t anyerr = B_FALSE;
        boolean_t softerr = B_FALSE;
        boolean_t recursive;

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

        assert(drr->drr_type == DRR_BEGIN);
        assert(drr->drr_u.drr_begin.drr_magic == DMU_BACKUP_MAGIC);
        assert(DMU_GET_STREAM_HDRTYPE(drr->drr_u.drr_begin.drr_versioninfo) ==
            DMU_COMPOUNDSTREAM);

        /*
         * Read in the nvlist from the stream.
         */
        if (drr->drr_payloadlen != 0) {
                error = recv_read_nvlist(hdl, fd, drr->drr_payloadlen,
                    &stream_nv, flags->byteswap, zc);
                if (error) {
                        error = zfs_error(hdl, EZFS_BADSTREAM, errbuf);
                        goto out;
                }
        }

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

        if (recursive && strchr(destname, '@')) {
                zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
                    "cannot specify snapshot name for multi-snapshot stream"));
                error = zfs_error(hdl, EZFS_BADSTREAM, errbuf);
                goto out;
        }

        /*
         * Read in the end record and verify checksum.
         */
        if (0 != (error = recv_read(hdl, fd, &drre, sizeof (drre),
            flags->byteswap, NULL)))
                goto out;
        if (flags->byteswap) {
                drre.drr_type = BSWAP_32(drre.drr_type);
                drre.drr_u.drr_end.drr_checksum.zc_word[0] =
                    BSWAP_64(drre.drr_u.drr_end.drr_checksum.zc_word[0]);
                drre.drr_u.drr_end.drr_checksum.zc_word[1] =
                    BSWAP_64(drre.drr_u.drr_end.drr_checksum.zc_word[1]);
                drre.drr_u.drr_end.drr_checksum.zc_word[2] =
                    BSWAP_64(drre.drr_u.drr_end.drr_checksum.zc_word[2]);
                drre.drr_u.drr_end.drr_checksum.zc_word[3] =
                    BSWAP_64(drre.drr_u.drr_end.drr_checksum.zc_word[3]);
        }
        if (drre.drr_type != DRR_END) {
                error = zfs_error(hdl, EZFS_BADSTREAM, errbuf);
                goto out;
        }
        if (!ZIO_CHECKSUM_EQUAL(drre.drr_u.drr_end.drr_checksum, *zc)) {
                zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
                    "incorrect header checksum"));
```

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

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

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

                VERIFY(0 == nvlist_lookup_nvlist(stream_nv, "fss",
                    &stream_fss));
                if ((stream_avl = fsavl_create(stream_fss)) == NULL) {
                        zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
                            "couldn't allocate avl tree"));
                        error = zfs_error(hdl, EZFS_NOMEM, errbuf);
                        goto out;
                }

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

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

                                if (flags->istail) {
                                        cp = strrchr(drrb->drr_toname, '/');
                                        if (cp == NULL) {
                                                (void) strlcat(tofs, "/",
                                                    ZFS_MAXNAMELEN);
                                                i = 0;
                                        } else {
                                                i = (cp - drrb->drr_toname);
                                        }
                                } else {
                                        i = strcspn(drrb->drr_toname, "/@");
                                }
                                /* zfs_receive_one() will create_parents() */
                                (void) strlcat(tofs, &drrb->drr_toname[i],
                                    ZFS_MAXNAMELEN);
                                *strchr(tofs, '@') = '\0';
                        }

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

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

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

                                zhp = zfs_open(hdl, nvpair_name(pair),
                                    ZFS_TYPE_FILESYSTEM);
                                if (zhp != NULL) {
                                        clp = changelist_gather(zhp,
                                            ZFS_PROP_MOUNTPOINT, 0, 0);
                                        zfs_close(zhp);
                                        if (clp != NULL) {
                                                softerr |=
                                                    changelist_prefix(clp);
                                                changelist_free(clp);
                                        }
                                }
                        }

                        nvlist_free(renamed);
                }
        }

        /*
         * Get the fs specified by the first path in the stream (the top level
         * specified by 'zfs send') and pass it to each invocation of
         * zfs_receive_one().
         */
        (void) strlcpy(sendfs, drr->drr_u.drr_begin.drr_toname,
            ZFS_MAXNAMELEN);
        if ((cp = strchr(sendfs, '@')) != NULL)
                *cp = '\0';

        /* Finally, receive each contained stream */
        do {
                /*
                 * we should figure out if it has a recoverable
                 * error, in which case do a recv_skip() and drive on.
                 * Note, if we fail due to already having this guid,
                 * zfs_receive_one() will take care of it (ie,
                 * recv_skip() and return 0).
                 */
                error = zfs_receive_impl(hdl, destname, flags, fd,
                    sendfs, stream_nv, stream_avl, top_zfs, cleanup_fd,
                    action_handlep);
                if (error == ENODATA) {
                        error = 0;
                        break;
                }
                anyerr |= error;
        } while (error == 0);

        if (drr->drr_payloadlen != 0 && fromsnap != NULL) {
                /*
                 * Now that we have the fs's they sent us, try the
                 * renames again.
                 */
                softerr = recv_incremental_replication(hdl, tofs, flags,
                    stream_nv, stream_avl, NULL);
        }

out:
        fsavl_destroy(stream_avl);
        if (stream_nv)
                nvlist_free(stream_nv);
        if (softerr)
                error = -2;
        if (anyerr)
                error = -1;
        return (error);
}

static void
trunc_prop_errs(int truncated)
{
        ASSERT(truncated != 0);

        if (truncated == 1)
```

```
                (void) fprintf(stderr, dgettext(TEXT_DOMAIN,
                    "1 more property could not be set\n"));
        else
                (void) fprintf(stderr, dgettext(TEXT_DOMAIN,
                    "%d more properties could not be set\n"), truncated);
}

static int
recv_skip(libzfs_handle_t *hdl, int fd, boolean_t byteswap)
{
        dmu_replay_record_t *drr;
        void *buf = malloc(1<<20);
        char errbuf[1024];

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

        /* XXX would be great to use lseek if possible... */
        drr = buf;

        while (recv_read(hdl, fd, drr, sizeof (dmu_replay_record_t),
            byteswap, NULL) == 0) {
                if (byteswap)
                        drr->drr_type = BSWAP_32(drr->drr_type);

                switch (drr->drr_type) {
                case DRR_BEGIN:
                        /* NB: not to be used on v2 stream packages */
                        if (drr->drr_payloadlen != 0) {
                                zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
                                    "invalid substream header"));
                                return (zfs_error(hdl, EZFS_BADSTREAM, errbuf));
                        }
                        break;

                case DRR_END:
                        free(buf);
                        return (0);

                case DRR_OBJECT:
                        if (byteswap) {
                                drr->drr_u.drr_object.drr_bonuslen =
                                    BSWAP_32(drr->drr_u.drr_object.
                                    drr_bonuslen);
                        }
                        (void) recv_read(hdl, fd, buf,
                            P2ROUNDUP(drr->drr_u.drr_object.drr_bonuslen, 8),
                            B_FALSE, NULL);
                        break;

                case DRR_WRITE:
                        if (byteswap) {
                                drr->drr_u.drr_write.drr_length =
                                    BSWAP_64(drr->drr_u.drr_write.drr_length);
                        }
                        (void) recv_read(hdl, fd, buf,
                            drr->drr_u.drr_write.drr_length, B_FALSE, NULL);
                        break;
                case DRR_SPILL:
                        if (byteswap) {
                                drr->drr_u.drr_write.drr_length =
                                    BSWAP_64(drr->drr_u.drr_spill.drr_length);
                        }
                        (void) recv_read(hdl, fd, buf,
                            drr->drr_u.drr_spill.drr_length, B_FALSE, NULL);
                        break;
```

```
		case DRR_WRITE_BYREF:
		case DRR_FREEOBJECTS:
		case DRR_FREE:
			break;

		default:
			zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
			    "invalid record type"));
			return (zfs_error(hdl, EZFS_BADSTREAM, errbuf));
		}
	}

	free(buf);
	return (-1);
}

/*
 * Restores a backup of tosnap from the file descriptor specified by infd.
 */
static int
zfs_receive_one(libzfs_handle_t *hdl, int infd, const char *tosnap,
    recvflags_t *flags, dmu_replay_record_t *drr,
    dmu_replay_record_t *drr_noswap, const char *sendfs,
    nvlist_t *stream_nv, avl_tree_t *stream_avl, char **top_zfs, int cleanup_fd,
    uint64_t *action_handlep)
{
	zfs_cmd_t zc = { 0 };
	time_t begin_time;
	int ioctl_err, ioctl_errno, err;
	char *cp;
	struct drr_begin *drrb = &drr->drr_u.drr_begin;
	char errbuf[1024];
	char prop_errbuf[1024];
	const char *chopprefix;
	boolean_t newfs = B_FALSE;
	boolean_t stream_wantsnewfs;
	uint64_t parent_snapguid = 0;
	prop_changelist_t *clp = NULL;
	nvlist_t *snapprops_nvlist = NULL;
	zprop_errflags_t prop_errflags;
	boolean_t recursive;

	begin_time = time(NULL);

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

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

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

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

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

```
		}
		ret = zcmd_write_src_nvlist(hdl, &zc, props);
		if (err)
			nvlist_free(props);

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

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

	cp = NULL;

	/*
	 * Determine how much of the snapshot name stored in the stream
	 * we are going to tack on to the name they specified on the
	 * command line, and how much we are going to chop off.
	 *
	 * If they specified a snapshot, chop the entire name stored in
	 * the stream.
	 */
	if (flags->istail) {
		/*
		 * A filesystem was specified with -e. We want to tack on only
		 * the tail of the sent snapshot path.
		 */
		if (strchr(tosnap, '@')) {
			zfs_error_aux(hdl, dgettext(TEXT_DOMAIN, "invalid "
			    "argument - snapshot not allowed with -e"));
			return (zfs_error(hdl, EZFS_INVALIDNAME, errbuf));
		}

		chopprefix = strrchr(sendfs, '/');

		if (chopprefix == NULL) {
			/*
			 * The tail is the poolname, so we need to
			 * prepend a path separator.
			 */
			int len = strlen(drrb->drr_toname);
			cp = malloc(len + 2);
			cp[0] = '/';
			(void) strcpy(&cp[1], drrb->drr_toname);
			chopprefix = cp;
		} else {
			chopprefix = drrb->drr_toname + (chopprefix - sendfs);
		}
	} else if (flags->isprefix) {
		/*
		 * A filesystem was specified with -d. We want to tack on
		 * everything but the first element of the sent snapshot path
		 * (all but the pool name).
		 */
		if (strchr(tosnap, '@')) {
			zfs_error_aux(hdl, dgettext(TEXT_DOMAIN, "invalid "
			    "argument - snapshot not allowed with -d"));
			return (zfs_error(hdl, EZFS_INVALIDNAME, errbuf));
		}

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

```
                /*
                 * If a filesystem was specified without -d or -e, we want to
                 * tack on everything after the fs specified by 'zfs send'.
                 */
                chopprefix = drrb->drr_toname + strlen(sendfs);
        } else {
                /* A snapshot was specified as an exact path (no -d or -e). */
                if (recursive) {
                        zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
                            "cannot specify snapshot name for multi-snapshot "
                            "stream"));
                        return (zfs_error(hdl, EZFS_BADSTREAM, errbuf));
                }
                chopprefix = drrb->drr_toname + strlen(drrb->drr_toname);
        }

        ASSERT(strstr(drrb->drr_toname, sendfs) == drrb->drr_toname);
        ASSERT(chopprefix > drrb->drr_toname);
        ASSERT(chopprefix <= drrb->drr_toname + strlen(drrb->drr_toname));
        ASSERT(chopprefix[0] == '/' || chopprefix[0] == '@' ||
            chopprefix[0] == '\0');

        /*
         * Determine name of destination snapshot, store in zc_value.
         */
        (void) strcpy(zc.zc_value, tosnap);
        (void) strncat(zc.zc_value, chopprefix, sizeof (zc.zc_value));
        free(cp);
        if (!zfs_name_valid(zc.zc_value, ZFS_TYPE_SNAPSHOT)) {
                zcmd_free_nvlists(&zc);
                return (zfs_error(hdl, EZFS_INVALIDNAME, errbuf));
        }

        /*
         * Determine the name of the origin snapshot, store in zc_string.
         */
        if (drrb->drr_flags & DRR_FLAG_CLONE) {
                if (guid_to_name(hdl, zc.zc_value,
                    drrb->drr_fromguid, zc.zc_string) != 0) {
                        zcmd_free_nvlists(&zc);
                        zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
                            "local origin for clone %s does not exist"),
                            zc.zc_value);
                        return (zfs_error(hdl, EZFS_NOENT, errbuf));
                }
                if (flags->verbose)
                        (void) printf("found clone origin %s\n", zc.zc_string);
        }

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

        if (stream_wantsnewfs) {
                /*
                 * if the parent fs does not exist, look for it based on
                 * the parent snap GUID
                 */
                (void) snprintf(errbuf, sizeof (errbuf), dgettext(TEXT_DOMAIN,
                    "cannot receive new filesystem stream"));

                (void) strcpy(zc.zc_name, zc.zc_value);
                cp = strrchr(zc.zc_name, '/');
                if (cp)
                        *cp = '\0';
                if (cp &&
                    !zfs_dataset_exists(hdl, zc.zc_name, ZFS_TYPE_DATASET)) {
```

```
                        char suffix[ZFS_MAXNAMELEN];
                        (void) strcpy(suffix, strrchr(zc.zc_value, '/'));
                        if (guid_to_name(hdl, zc.zc_name, parent_snapguid,
                            zc.zc_value) == 0) {
                                *strchr(zc.zc_value, '@') = '\0';
                                (void) strcat(zc.zc_value, suffix);
                        }
                }
        } else {
                /*
                 * if the fs does not exist, look for it based on the
                 * fromsnap GUID
                 */
                (void) snprintf(errbuf, sizeof (errbuf), dgettext(TEXT_DOMAIN,
                    "cannot receive incremental stream"));

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

                /*
                 * If the exact receive path was specified and this is the
                 * topmost path in the stream, then if the fs does not exist we
                 * should look no further.
                 */
                if ((flags->isprefix || (*(chopprefix = drrb->drr_toname +
                    strlen(sendfs)) != '\0' && *chopprefix != '@')) &&
                    !zfs_dataset_exists(hdl, zc.zc_name, ZFS_TYPE_DATASET)) {
                        char snap[ZFS_MAXNAMELEN];
                        (void) strcpy(snap, strchr(zc.zc_value, '@'));
                        if (guid_to_name(hdl, zc.zc_name, drrb->drr_fromguid,
                            zc.zc_value) == 0) {
                                *strchr(zc.zc_value, '@') = '\0';
                                (void) strcat(zc.zc_value, snap);
                        }
                }
        }

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

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

                /*
                 * Destination fs exists.  Therefore this should either
                 * be an incremental, or the stream specifies a new fs
                 * (full stream or clone) and they want us to blow it
                 * away (and have therefore specified -F and removed any
                 * snapshots).
                 */
                if (stream_wantsnewfs) {
                        if (!flags->force) {
                                zcmd_free_nvlists(&zc);
                                zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
                                    "destination '%s' exists\n"
                                    "must specify -F to overwrite it"),
                                    zc.zc_name);
                                return (zfs_error(hdl, EZFS_EXISTS, errbuf));
                        }
                        if (ioctl(hdl->libzfs_fd, ZFS_IOC_SNAPSHOT_LIST_NEXT,
                            &zc) == 0) {
                                zcmd_free_nvlists(&zc);
                                zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
                                    "destination has snapshots (eg. %s)\n"
                                    "must destroy them to overwrite it"),
                                    zc.zc_name);
                                return (zfs_error(hdl, EZFS_EXISTS, errbuf));
                        }
                }

                if ((zhp = zfs_open(hdl, zc.zc_name,
                    ZFS_TYPE_FILESYSTEM | ZFS_TYPE_VOLUME)) == NULL) {
                        zcmd_free_nvlists(&zc);
                        return (-1);
                }

                if (stream_wantsnewfs &&
                    zhp->zfs_dmustats.dds_origin[0]) {
                        zcmd_free_nvlists(&zc);
                        zfs_close(zhp);
                        zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
                            "destination '%s' is a clone\n"
                            "must destroy it to overwrite it"),
                            zc.zc_name);
                        return (zfs_error(hdl, EZFS_EXISTS, errbuf));
                }

                if (!flags->dryrun && zhp->zfs_type == ZFS_TYPE_FILESYSTEM &&
                    stream_wantsnewfs) {
                        /* We can't do online recv in this case */
                        clp = changelist_gather(zhp, ZFS_PROP_NAME, 0, 0);
                        if (clp == NULL) {
                                zfs_close(zhp);
                                zcmd_free_nvlists(&zc);
                                return (-1);
                        }
                        if (changelist_prefix(clp) != 0) {
                                changelist_free(clp);
                                zfs_close(zhp);
                                zcmd_free_nvlists(&zc);
                                return (-1);
                        }
                }
                zfs_close(zhp);
        } else {
                /*
                 * Destination filesystem does not exist.  Therefore we better
                 * be creating a new filesystem (either from a full backup, or
                 * a clone).  It would therefore be invalid if the user
                 * specified only the pool name (i.e. if the destination name
                 * contained no slash character).
                 */
                if (!stream_wantsnewfs ||
                    (cp = strrchr(zc.zc_name, '/')) == NULL) {
                        zcmd_free_nvlists(&zc);
                        zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
                            "destination '%s' does not exist"), zc.zc_name);
                        return (zfs_error(hdl, EZFS_NOENT, errbuf));
                }

                /*
                 * Trim off the final dataset component so we perform the
                 * recvbackup ioctl to the filesystems's parent.
                 */
                *cp = '\0';

                if (flags->isprefix && !flags->istail && !flags->dryrun &&
                    create_parents(hdl, zc.zc_value, strlen(tosnap)) != 0) {
                        zcmd_free_nvlists(&zc);
                        return (zfs_error(hdl, EZFS_BADRESTORE, errbuf));
                }
```

```
                newfs = B_TRUE;
        }

        zc.zc_begin_record = drr_noswap->drr_u.drr_begin;
        zc.zc_cookie = infd;
        zc.zc_guid = flags->force;
        if (flags->verbose) {
                (void) printf("%s %s stream of %s into %s\n",
                    flags->dryrun ? "would receive" : "receiving",
                    drrb->drr_fromguid ? "incremental" : "full",
                    drrb->drr_toname, zc.zc_value);
                (void) fflush(stdout);
        }

        if (flags->dryrun) {
                zcmd_free_nvlists(&zc);
                return (recv_skip(hdl, infd, flags->byteswap));
        }

        zc.zc_nvlist_dst = (uint64_t)(uintptr_t)prop_errbuf;
        zc.zc_nvlist_dst_size = sizeof (prop_errbuf);
        zc.zc_cleanup_fd = cleanup_fd;
        zc.zc_action_handle = *action_handlep;

        err = ioctl_err = zfs_ioctl(hdl, ZFS_IOC_RECV, &zc);
        ioctl_errno = errno;
        prop_errflags = (zprop_errflags_t)zc.zc_obj;

        if (err == 0) {
                nvlist_t *prop_errors;
                VERIFY(0 == nvlist_unpack((void *)(uintptr_t)zc.zc_nvlist_dst,
                    zc.zc_nvlist_dst_size, &prop_errors, 0));

                nvpair_t *prop_err = NULL;

                while ((prop_err = nvlist_next_nvpair(prop_errors,
                    prop_err)) != NULL) {
                        char tbuf[1024];
                        zfs_prop_t prop;
                        int intval;

                        prop = zfs_name_to_prop(nvpair_name(prop_err));
                        (void) nvpair_value_int32(prop_err, &intval);
                        if (strcmp(nvpair_name(prop_err),
                            ZPROP_N_MORE_ERRORS) == 0) {
                                trunc_prop_errs(intval);
                                break;
                        } else {
                                (void) snprintf(tbuf, sizeof (tbuf),
                                    dgettext(TEXT_DOMAIN,
                                    "cannot receive %s property on %s"),
                                    nvpair_name(prop_err), zc.zc_name);
                                zfs_setprop_error(hdl, prop, intval, tbuf);
                        }
                }
                nvlist_free(prop_errors);
        }

        zc.zc_nvlist_dst = 0;
        zc.zc_nvlist_dst_size = 0;
        zcmd_free_nvlists(&zc);

        if (err == 0 && snapprops_nvlist) {
                zfs_cmd_t zc2 = { 0 };

                (void) strcpy(zc2.zc_name, zc.zc_value);
```

```
		zc2.zc_cookie = B_TRUE; /* received */
		if (zcmd_write_src_nvlist(hdl, &zc2, snapprops_nvlist) == 0) {
			(void) zfs_ioctl(hdl, ZFS_IOC_SET_PROP, &zc2);
			zcmd_free_nvlists(&zc2);
		}
	}

	if (err && (ioctl_errno == ENOENT || ioctl_errno == EEXIST)) {
		/*
		 * It may be that this snapshot already exists,
		 * in which case we want to consume & ignore it
		 * rather than failing.
		 */
		avl_tree_t *local_avl;
		nvlist_t *local_nv, *fs;
		cp = strchr(zc.zc_value, '@');

		/*
		 * XXX Do this faster by just iterating over snaps in
		 * this fs.  Also if zc_value does not exist, we will
		 * get a strange "does not exist" error message.
		 */
		*cp = '\0';
		if (gather_nvlist(hdl, zc.zc_value, NULL, NULL, B_FALSE,
		    &local_nv, &local_avl) == 0) {
			*cp = '@';
			fs = fsavl_find(local_avl, drrb->drr_toguid, NULL);
			fsavl_destroy(local_avl);
			nvlist_free(local_nv);

			if (fs != NULL) {
				if (flags->verbose) {
					(void) printf("snap %s already exists; "
					    "ignoring\n", zc.zc_value);
				}
				err = ioctl_err = recv_skip(hdl, infd,
				    flags->byteswap);
			}
		}
		*cp = '@';
	}

	if (ioctl_err != 0) {
		switch (ioctl_errno) {
		case ENODEV:
			cp = strchr(zc.zc_value, '@');
			*cp = '\0';
			zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
			    "most recent snapshot of %s does not\n"
			    "match incremental source"), zc.zc_value);
			(void) zfs_error(hdl, EZFS_BADRESTORE, errbuf);
			*cp = '@';
			break;
		case ETXTBSY:
			zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
			    "destination %s has been modified\n"
			    "since most recent snapshot"), zc.zc_name);
			(void) zfs_error(hdl, EZFS_BADRESTORE, errbuf);
			break;
		case EEXIST:
			cp = strchr(zc.zc_value, '@');
			if (newfs) {
				/* it's the containing fs that exists */
				*cp = '\0';
			}
			zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
```

```
			    "destination already exists"));
			(void) zfs_error_fmt(hdl, EZFS_EXISTS,
			    dgettext(TEXT_DOMAIN, "cannot restore to %s"),
			    zc.zc_value);
			*cp = '@';
			break;
		case EINVAL:
			(void) zfs_error(hdl, EZFS_BADSTREAM, errbuf);
			break;
		case ECKSUM:
			zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
			    "invalid stream (checksum mismatch)"));
			(void) zfs_error(hdl, EZFS_BADSTREAM, errbuf);
			break;
		case ENOTSUP:
			zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
			    "pool must be upgraded to receive this stream."));
			(void) zfs_error(hdl, EZFS_BADVERSION, errbuf);
			break;
		case EDQUOT:
			zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
			    "destination %s space quota exceeded"), zc.zc_name);
			(void) zfs_error(hdl, EZFS_NOSPC, errbuf);
			break;
		default:
			(void) zfs_standard_error(hdl, ioctl_errno, errbuf);
		}
	}

	/*
	 * Mount the target filesystem (if created).  Also mount any
	 * children of the target filesystem if we did a replication
	 * receive (indicated by stream_avl being non-NULL).
	 */
	cp = strchr(zc.zc_value, '@');
	if (cp && (ioctl_err == 0 || !newfs)) {
		zfs_handle_t *h;

		*cp = '\0';
		h = zfs_open(hdl, zc.zc_value,
		    ZFS_TYPE_FILESYSTEM | ZFS_TYPE_VOLUME);
		if (h != NULL) {
			if (h->zfs_type == ZFS_TYPE_VOLUME) {
				*cp = '@';
			} else if (newfs || stream_avl) {
				/*
				 * Track the first/top of hierarchy fs,
				 * for mounting and sharing later.
				 */
				if (top_zfs && *top_zfs == NULL)
					*top_zfs = zfs_strdup(hdl, zc.zc_value);
			}
			zfs_close(h);
		}
		*cp = '@';
	}

	if (clp) {
		err |= changelist_postfix(clp);
		changelist_free(clp);
	}

	if (prop_errflags & ZPROP_ERR_NOCLEAR) {
		(void) fprintf(stderr, dgettext(TEXT_DOMAIN, "Warning: "
		    "failed to clear unreceived properties on %s"),
		    zc.zc_name);
		(void) fprintf(stderr, "\n");
	}
	if (prop_errflags & ZPROP_ERR_NORESTORE) {
		(void) fprintf(stderr, dgettext(TEXT_DOMAIN, "Warning: "
		    "failed to restore original properties on %s"),
		    zc.zc_name);
		(void) fprintf(stderr, "\n");
	}

	if (err || ioctl_err)
		return (-1);

	*action_handlep = zc.zc_action_handle;

	if (flags->verbose) {
		char buf1[64];
		char buf2[64];
		uint64_t bytes = zc.zc_cookie;
		time_t delta = time(NULL) - begin_time;
		if (delta == 0)
			delta = 1;
		zfs_nicenum(bytes, buf1, sizeof (buf1));
		zfs_nicenum(bytes/delta, buf2, sizeof (buf1));

		(void) printf("received %sB stream in %lu seconds (%sB/sec)\n",
		    buf1, delta, buf2);
	}

	return (0);
}

static int
zfs_receive_impl(libzfs_handle_t *hdl, const char *tosnap, recvflags_t *flags,
    int infd, const char *sendfs, nvlist_t *stream_nv, avl_tree_t *stream_avl,
    char **top_zfs, int cleanup_fd, uint64_t *action_handlep)
{
	int err;
	dmu_replay_record_t drr, drr_noswap;
	struct drr_begin *drrb = &drr.drr_u.drr_begin;
	char errbuf[1024];
	zio_cksum_t zcksum = { 0 };
	uint64_t featureflags;
	int hdrtype;

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

	if (flags->isprefix &&
	    !zfs_dataset_exists(hdl, tosnap, ZFS_TYPE_DATASET)) {
		zfs_error_aux(hdl, dgettext(TEXT_DOMAIN, "specified fs "
		    "(%s) does not exist"), tosnap);
		return (zfs_error(hdl, EZFS_NOENT, errbuf));
	}

	/* read in the BEGIN record */
	if (0 != (err = recv_read(hdl, infd, &drr, sizeof (drr), B_FALSE,
	    &zcksum)))
		return (err);

	if (drr.drr_type == DRR_END || drr.drr_type == BSWAP_32(DRR_END)) {
		/* It's the double end record at the end of a package */
		return (ENODATA);
	}

	/* the kernel needs the non-byteswapped begin record */
	drr_noswap = drr;
```

```
	flags->byteswap = B_FALSE;
	if (drrb->drr_magic == BSWAP_64(DMU_BACKUP_MAGIC)) {
		/*
		 * We computed the checksum in the wrong byteorder in
		 * recv_read() above; do it again correctly.
		 */
		bzero(&zcksum, sizeof (zio_cksum_t));
		fletcher_4_incremental_byteswap(&drr, sizeof (drr), &zcksum);
		flags->byteswap = B_TRUE;

		drr.drr_type = BSWAP_32(drr.drr_type);
		drr.drr_payloadlen = BSWAP_32(drr.drr_payloadlen);
		drrb->drr_magic = BSWAP_64(drrb->drr_magic);
		drrb->drr_versioninfo = BSWAP_64(drrb->drr_versioninfo);
		drrb->drr_creation_time = BSWAP_64(drrb->drr_creation_time);
		drrb->drr_type = BSWAP_32(drrb->drr_type);
		drrb->drr_flags = BSWAP_32(drrb->drr_flags);
		drrb->drr_toguid = BSWAP_64(drrb->drr_toguid);
		drrb->drr_fromguid = BSWAP_64(drrb->drr_fromguid);
	}

	if (drrb->drr_magic != DMU_BACKUP_MAGIC || drr.drr_type != DRR_BEGIN) {
		zfs_error_aux(hdl, dgettext(TEXT_DOMAIN, "invalid "
		    "stream (bad magic number)"));
		return (zfs_error(hdl, EZFS_BADSTREAM, errbuf));
	}

	featureflags = DMU_GET_FEATUREFLAGS(drrb->drr_versioninfo);
	hdrtype = DMU_GET_STREAM_HDRTYPE(drrb->drr_versioninfo);

	if (!DMU_STREAM_SUPPORTED(featureflags) ||
	    (hdrtype != DMU_SUBSTREAM && hdrtype != DMU_COMPOUNDSTREAM)) {
		zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
		    "stream has unsupported feature, feature flags = %lx"),
		    featureflags);
		return (zfs_error(hdl, EZFS_BADSTREAM, errbuf));
	}

	if (strchr(drrb->drr_toname, '@') == NULL) {
		zfs_error_aux(hdl, dgettext(TEXT_DOMAIN, "invalid "
		    "stream (bad snapshot name)"));
		return (zfs_error(hdl, EZFS_BADSTREAM, errbuf));
	}

	if (DMU_GET_STREAM_HDRTYPE(drrb->drr_versioninfo) == DMU_SUBSTREAM) {
		char nonpackage_sendfs[ZFS_MAXNAMELEN];
		if (sendfs == NULL) {
			/*
			 * We were not called from zfs_receive_package(). Get
			 * the fs specified by 'zfs send'.
			 */
			char *cp;
			(void) strlcpy(nonpackage_sendfs,
			    drr.drr_u.drr_begin.drr_toname, ZFS_MAXNAMELEN);
			if ((cp = strchr(nonpackage_sendfs, '@')) != NULL)
				*cp = '\0';
			sendfs = nonpackage_sendfs;
		}
		return (zfs_receive_one(hdl, infd, tosnap, flags,
		    &drr, &drr_noswap, sendfs, stream_nv, stream_avl,
		    top_zfs, cleanup_fd, action_handlep));
	} else {
		assert(DMU_GET_STREAM_HDRTYPE(drrb->drr_versioninfo) ==
		    DMU_COMPOUNDSTREAM);
		return (zfs_receive_package(hdl, infd, tosnap, flags,
		    &drr, &zcksum, top_zfs, cleanup_fd, action_handlep));
	}
}

/*
 * Restores a backup of tosnap from the file descriptor specified by infd.
 * Return 0 on total success, -2 if some things couldn't be
 * destroyed/renamed/promoted, -1 if some things couldn't be received.
 * (-1 will override -2).
 */
int
zfs_receive(libzfs_handle_t *hdl, const char *tosnap, recvflags_t *flags,
    int infd, avl_tree_t *stream_avl)
{
	char *top_zfs = NULL;
	int err;
	int cleanup_fd;
	uint64_t action_handle = 0;

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

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

	VERIFY(0 == close(cleanup_fd));

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

		zhp = zfs_open(hdl, top_zfs, ZFS_TYPE_FILESYSTEM);
		if (zhp != NULL) {
			clp = changelist_gather(zhp, ZFS_PROP_MOUNTPOINT,
			    CL_GATHER_MOUNT_ALWAYS, 0);
			zfs_close(zhp);
			if (clp != NULL) {
				/* mount and share received datasets */
				err = changelist_postfix(clp);
				changelist_free(clp);
			}
		}
		if (zhp == NULL || clp == NULL || err)
			err = -1;
	}
	if (top_zfs)
		free(top_zfs);

	return (err);
}
```