```
**********************************************************
   85483 Fri Jul 26 21:08:44 2013
new/usr/src/lib/libzfs/common/libzfs_sendrecv.c
3909 Fix hang when sending dedup stream
**********************************************************
______unchanged_portion_omitted_

1322 /*
1323  * Generate a send stream for the dataset identified by the argument zhp.
1324  *
1325  * The content of the send stream is the snapshot identified by
1326  * 'tosnap'.  Incremental streams are requested in two ways:
1327  *     - from the snapshot identified by "fromsnap" (if non-null) or
1328  *     - from the origin of the dataset identified by zhp, which must
1329  *       be a clone.  In this case, "fromsnap" is null and "fromorigin"
1330  *       is TRUE.
1331  *
1332  * The send stream is recursive (i.e. dumps a hierarchy of snapshots) and
1333  * uses a special header (with a hdrtype field of DMU_COMPOUNDSTREAM)
1334  * if "replicate" is set.  If "doall" is set, dump all the intermediate
1335  * snapshots. The DMU_COMPOUNDSTREAM header is used in the "doall"
1336  * case too. If "props" is set, send properties.
1337  */
1338 int
1339 zfs_send(zfs_handle_t *zhp, const char *fromsnap, const char *tosnap,
1340     sendflags_t *flags, int outfd, snapfilter_cb_t filter_func,
1341     void *cb_arg, nvlist_t **debugnvp)
1342 {
1343         char errbuf[1024];
1344         send_dump_data_t sdd = { 0 };
1345         int err = 0;
1346         nvlist_t *fss = NULL;
1347         avl_tree_t *fsavl = NULL;
1348         static uint64_t holdseq;
1349         int spa_version;
1350         pthread_t tid = 0;
1351         int pipefd[2];
1352         dedup_arg_t dda = { 0 };
1353         int featureflags = 0;

1355         (void) snprintf(errbuf, sizeof (errbuf), dgettext(TEXT_DOMAIN,
1356             "cannot send '%s'"), zhp->zfs_name);

1358         if (fromsnap && fromsnap[0] == '\0') {
1359                 zfs_error_aux(zhp->zfs_hdl, dgettext(TEXT_DOMAIN,
1360                     "zero-length incremental source"));
1361                 return (zfs_error(zhp->zfs_hdl, EZFS_NOENT, errbuf));
1362         }

1364         if (zhp->zfs_type == ZFS_TYPE_FILESYSTEM) {
1365                 uint64_t version;
1366                 version = zfs_prop_get_int(zhp, ZFS_PROP_VERSION);
1367                 if (version >= ZPL_VERSION_SA) {
1368                         featureflags |= DMU_BACKUP_FEATURE_SA_SPILL;
1369                 }
1370         }

1372         if (flags->dedup && !flags->dryrun) {
1373                 featureflags |= (DMU_BACKUP_FEATURE_DEDUP |
1374                     DMU_BACKUP_FEATURE_DEDUPPROPS);
1375                 if (err = pipe(pipefd)) {
1376                         zfs_error_aux(zhp->zfs_hdl, strerror(errno));
1377                         return (zfs_error(zhp->zfs_hdl, EZFS_PIPEFAILED,
1378                             errbuf));
1379                 }
1380                 dda.outputfd = outfd;
```



```
1381                 dda.inputfd = pipefd[1];
1382                 dda.dedup_hdl = zhp->zfs_hdl;
1383                 if (err = pthread_create(&tid, NULL, cksummer, &dda)) {
1384                         (void) close(pipefd[0]);
1385                         (void) close(pipefd[1]);
1386                         zfs_error_aux(zhp->zfs_hdl, strerror(errno));
1387                         return (zfs_error(zhp->zfs_hdl,
1388                             EZFS_THREADCREATEFAILED, errbuf));
1389                 }
1390         }

1392         if (flags->replicate || flags->doall || flags->props) {
1393                 dmu_replay_record_t drr = { 0 };
1394                 char *packbuf = NULL;
1395                 size_t buflen = 0;
1396                 zio_cksum_t zc = { 0 };

1398                 if (flags->replicate || flags->props) {
1399                         nvlist_t *hdrnv;

1401                         VERIFY(0 == nvlist_alloc(&hdrnv, NV_UNIQUE_NAME, 0));
1402                         if (fromsnap) {
1403                                 VERIFY(0 == nvlist_add_string(hdrnv,
1404                                     "fromsnap", fromsnap));
1405                         }
1406                         VERIFY(0 == nvlist_add_string(hdrnv, "tosnap", tosnap));
1407                         if (!flags->replicate) {
1408                                 VERIFY(0 == nvlist_add_boolean(hdrnv,
1409                                     "not_recursive"));
1410                         }

1412                         err = gather_nvlist(zhp->zfs_hdl, zhp->zfs_name,
1413                             fromsnap, tosnap, flags->replicate, &fss, &fsavl);
1414                         if (err)
1415                                 goto err_out;
1416                         VERIFY(0 == nvlist_add_nvlist(hdrnv, "fss", fss));
1417                         err = nvlist_pack(hdrnv, &packbuf, &buflen,
1418                             NV_ENCODE_XDR, 0);
1419                         if (debugnvp)
1420                                 *debugnvp = hdrnv;
1421                         else
1422                                 nvlist_free(hdrnv);
1423                         if (err)
1424                                 goto stderr_out;
1425                 }

1427                 if (!flags->dryrun) {
1428                         /* write first begin record */
1429                         drr.drr_type = DRR_BEGIN;
1430                         drr.drr_u.drr_begin.drr_magic = DMU_BACKUP_MAGIC;
1431                         DMU_SET_STREAM_HDRTYPE(drr.drr_u.drr_begin.
1432                             drr_versioninfo, DMU_COMPOUNDSTREAM);
1433                         DMU_SET_FEATUREFLAGS(drr.drr_u.drr_begin.
1434                             drr_versioninfo, featureflags);
1435                         (void) snprintf(drr.drr_u.drr_begin.drr_toname,
1436                             sizeof (drr.drr_u.drr_begin.drr_toname),
1437                             "%s@%s", zhp->zfs_name, tosnap);
1438                         drr.drr_payloadlen = buflen;
1439                         err = cksum_and_write(&drr, sizeof (drr), &zc, outfd);

1441                         /* write header nvlist */
1442                         if (err != -1 && packbuf != NULL) {
1443                                 err = cksum_and_write(packbuf, buflen, &zc,
1444                                     outfd);
1445                         }
1446                         free(packbuf);
```

```
1447                        if (err == -1) {
1448                                err = errno;
1449                                goto stderr_out;
1450                        }

1452                        /* write end record */
1453                        bzero(&drr, sizeof (drr));
1454                        drr.drr_type = DRR_END;
1455                        drr.drr_u.drr_end.drr_checksum = zc;
1456                        err = write(outfd, &drr, sizeof (drr));
1457                        if (err == -1) {
1458                                err = errno;
1459                                goto stderr_out;
1460                        }

1462                        err = 0;
1463                }
1464        }

1466        /* dump each stream */
1467        sdd.fromsnap = fromsnap;
1468        sdd.tosnap = tosnap;
1469        if (tid != 0)
1470                sdd.outfd = pipefd[0];
1471        else
1472                sdd.outfd = outfd;
1473        sdd.replicate = flags->replicate;
1474        sdd.doall = flags->doall;
1475        sdd.fromorigin = flags->fromorigin;
1476        sdd.fss = fss;
1477        sdd.fsavl = fsavl;
1478        sdd.verbose = flags->verbose;
1479        sdd.parsable = flags->parsable;
1480        sdd.progress = flags->progress;
1481        sdd.dryrun = flags->dryrun;
1482        sdd.filter_cb = filter_func;
1483        sdd.filter_cb_arg = cb_arg;
1484        if (debugnvp)
1485                sdd.debugnv = *debugnvp;

1487        /*
1488         * Some flags require that we place user holds on the datasets that are
1489         * being sent so they don't get destroyed during the send. We can skip
1490         * this step if the pool is imported read-only since the datasets cannot
1491         * be destroyed.
1492         */
1493        if (!flags->dryrun && !zpool_get_prop_int(zfs_get_pool_handle(zhp),
1494            ZPOOL_PROP_READONLY, NULL) &&
1495            zfs_spa_version(zhp, &spa_version) == 0 &&
1496            spa_version >= SPA_VERSION_USERREFS &&
1497            (flags->doall || flags->replicate)) {
1498                ++holdseq;
1499                (void) snprintf(sdd.holdtag, sizeof (sdd.holdtag),
1500                    ".send-%d-%llu", getpid(), (u_longlong_t)holdseq);
1501                sdd.cleanup_fd = open(ZFS_DEV, O_RDWR|O_EXCL);
1502                if (sdd.cleanup_fd < 0) {
1503                        err = errno;
1504                        goto stderr_out;
1505                }
1506                sdd.snapholds = fnvlist_alloc();
1507        } else {
1508                sdd.cleanup_fd = -1;
1509                sdd.snapholds = NULL;
1510        }
1511        if (flags->verbose || sdd.snapholds != NULL) {
1512                /*
```



```
1513                 * Do a verbose no-op dry run to get all the verbose output
1514                 * or to gather snapshot hold's before generating any data,
1515                 * then do a non-verbose real run to generate the streams.
1516                 */
1517                sdd.dryrun = B_TRUE;
1518                err = dump_filesystems(zhp, &sdd);

1520                if (err != 0)
1521                        goto stderr_out;

1523                if (flags->verbose) {
1524                        if (flags->parsable) {
1525                                (void) fprintf(stderr, "size\t%llu\n",
1526                                    (longlong_t)sdd.size);
1527                        } else {
1528                                char buf[16];
1529                                zfs_nicenum(sdd.size, buf, sizeof (buf));
1530                                (void) fprintf(stderr, dgettext(TEXT_DOMAIN,
1531                                    "total estimated size is %s\n"), buf);
1532                        }
1533                }

1535                /* Ensure no snaps found is treated as an error. */
1536                if (!sdd.seento) {
1537                        err = ENOENT;
1538                        goto err_out;
1539                }

1541                /* Skip the second run if dryrun was requested. */
1542                if (flags->dryrun)
1543                        goto err_out;

1545                if (sdd.snapholds != NULL) {
1546                        err = zfs_hold_nvl(zhp, sdd.cleanup_fd, sdd.snapholds);
1547                        if (err != 0)
1548                                goto stderr_out;

1550                        fnvlist_free(sdd.snapholds);
1551                        sdd.snapholds = NULL;
1552                }

1554                sdd.dryrun = B_FALSE;
1555                sdd.verbose = B_FALSE;
1556        }

1558        err = dump_filesystems(zhp, &sdd);
1559        fsavl_destroy(fsavl);
1560        nvlist_free(fss);

1562        /* Ensure no snaps found is treated as an error. */
1563        if (err == 0 && !sdd.seento)
1564                err = ENOENT;

1566        if (tid != 0) {
1567                if (err != 0)
1568                        (void) pthread_cancel(tid);
1569                (void) close(pipefd[0]);
1570 #endif /* ! codereview */
1571                (void) pthread_join(tid, NULL);
1569                (void) close(pipefd[0]);
1572        }

1574        if (sdd.cleanup_fd != -1) {
1575                VERIFY(0 == close(sdd.cleanup_fd));
1576                sdd.cleanup_fd = -1;
1577        }
```

```
1579         if (!flags->dryrun && (flags->replicate || flags->doall ||
1580             flags->props)) {
1581                 /*
1582                  * write final end record.  NB: want to do this even if
1583                  * there was some error, because it might not be totally
1584                  * failed.
1585                  */
1586                 dmu_replay_record_t drr = { 0 };
1587                 drr.drr_type = DRR_END;
1588                 if (write(outfd, &drr, sizeof (drr)) == -1) {
1589                         return (zfs_standard_error(zhp->zfs_hdl,
1590                             errno, errbuf));
1591                 }
1592         }

1594         return (err || sdd.err);

1596 stderr_out:
1597         err = zfs_standard_error(zhp->zfs_hdl, err, errbuf);
1598 err_out:
1599         fsavl_destroy(fsavl);
1600         nvlist_free(fss);
1601         fnvlist_free(sdd.snapholds);

1603         if (sdd.cleanup_fd != -1)
1604                 VERIFY(0 == close(sdd.cleanup_fd));
1605         if (tid != 0) {
1606                 (void) pthread_cancel(tid);
1607                 (void) close(pipefd[0]);
1608 #endif /* ! codereview */
1609                 (void) pthread_join(tid, NULL);
1605                 (void) close(pipefd[0]);
1610         }
1611         return (err);
1612 }
```

*unchanged_portion_omitted_*