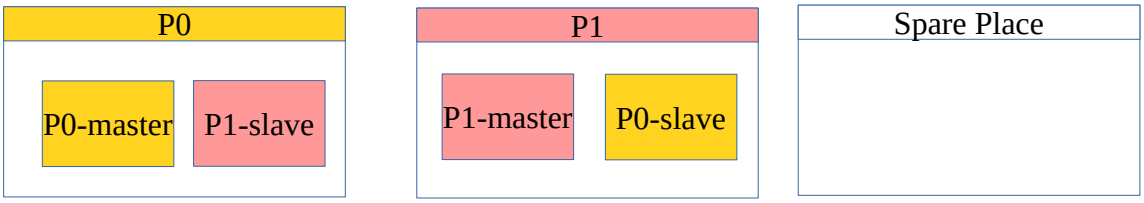
Local Resilient Store

This document describes the design of a resilient store that can survive multiple place failures.

Each place has a *virtual place id* that represents its location in the application's place-group. Spare places can be used to replace main places. In that case, a spare place uses the same virtual place id of the dead place.

Each place has a master store that contains the place resilient data, and a slave store that contains another place's resilient data. Spare places don't have any stores until they are used in the computation.

Each place puts and gets its own data. If P0 needs to put data at place 1, it must shift to place 1 using **at** before putting that data.



I. Resilient Store Coordinator

The coordinator of the resilient store is *a single place* that performs the following:

1. Creates the resilient store data structures at all places (i.e. master stores, slave stores, null stores at spare places)
2. Keeps a map of the location of the slave place of each virtual place.
3. Receives control from the application when a failure occurs to recover dead places.
4. Performs the following recovery steps:
   4.1. Identifies dead places
   4.2. Identifies places that contain slave stores for that dead places
   4.3. *Recovers any pending transactions at a slave who lost its master before committing*. Since the master is dead, the coordinator should be able to decide whether to commit or rollback a pending transaction.
   4.4. Replaces the dead place with a spare place. The spare place copies the 'slave store' of the dead place to be its master store. The dead place must have owned a slave store for another place. The spare place will copy that place's master store to serve as its slave in the future.

*The above functionality is provided to applications through the ResilientStore class, presented in **Fig.1**.*

II. Master-Slave Transaction

To ensure that a slave store can correctly replace a master store, the two stores should be strongly synchronised. In addition, conflicts should be prevented by activities running at the same place and trying to manipulate the same resilient data of the place.

To ensure master-slave consistency and to resolve possible conflicts between local activities, the resilient store requires all read and update operations to be issued within a transaction. To start a transaction at the current place, a call to *ResilientStore.startLocalTransaction()* should be issued. It returns an object of type *LocalTransaction*, shown in **Fig.2**.

Fig.3 shows how the ResilientStore and LocalTransaction APIs can be used by a sample application.

```
public class ResilientStore {

   /*A LocalStore is a container for a SlaveStore and a MasterStore*/
   private val plh:PlaceLocalHandle[LocalStore];

   /* Remote operation: the caller of this method is the "store coordinator" place
    * The function initializes the place-local-handle plh at all places.
    **/
   public static def make(spare:Long):ResilientMap;

   /* Local operation: return the virtual place id of the caller place
    * */
   public def getVirtualPlaceId():Long;

   /* Local operation: return the list of active places that can be used by the computation.
    * It should be called by the application at start-up and after recovery to get
    * the list of places that can be used by the computation.
    */
   public def getActivePlaces():PlaceGroup;

   /* Remote operation: Called at the store coordinator place. The function tries to recover
    * dead places using their slaves. It completes any pending transactions at slaves before
    * using them for recovery.
    * The function returns a map of the newly added places [place-id, place-virtual-id]
    */
   public def recoverDeadPlaces():HashMap[Long,Long];

   /* Local operation:
    * Start a local transaction at the caller place.
    */
   public def startLocalTransaction():LocalTransaction;
}
```

Fig.1: ResilientStore API

```
public class LocalTransaction {
   private log:TransactionLog;

   /* Local operation: updates the transaction log. Master store not impacted*/
   public def put(key:String, newValue:Cloneable):Cloneable;

   /* Local operation: deletes the value of the transaction log. Master store not impacted*/
   public def delete(key:String):Cloneable;

   /* Local operation: gets the value from the transaction log*/
   public def get(key:String):Cloneable;

   /* Remote operation: first check for local conflicts. If no conflicts exist, send the
    * transaction log to the slave store. The slave store will keep the log as a pending
    * transaction.
    *
    * If the master dies before completing prepared transactions,
    * the store coordinator should complete the pending transactions at the slave.
    **/
   public def prepare():Boolean;

   /* Remote operation: notify the slave to commit, then apply the transaction log
    * changes at the local master store.
    *
    * If the slave is dead, ignore it, the application should be able to
    * detect that failure and trigger the recovery process.
    * */
   public def commit():void;

   /*Remote operation:
    * Notify the slave to rollback. The slave will delete the pending transaction log.
    *
    * If the slave is dead, ignore it, the application should be able to
    * detect that failure and trigger the recovery process.
    * */
   public def rollback():void;
}
```

Fig.2: LocalTransaction API

```
val spare = 3;
val map = ResilientMap.make(spare);
var done:Boolean = false;
while (!done) {
    val activePlaces = map.getActivePlaces();
    try {
        finish foreach(p in activePlaces) {
            val tx = map.startLocalTransaction();
            val x = tx.get("X");
            val y = process(x);
            tx.put("Y", y);
            if (tx.prepare())
                tx.commit();
            else
                tx.rollback();
        }
        done = true;
    }
    catch(dpe:DeadPlaceException) {
        map.recoverDeadPlaces();
    }
}
```

Fig.3: Sample program using the ResilientStore APIs

III. Preparing a LocalTransaction

All get/put/delete operations are logged at the master place. When *prepare()* is called, the log is transferred from the master to the slave if it contains any update or delete actions. If all actions are reading actions, the log is not transferred to the slave. Only at that point when the slave becomes aware of the existence of this transaction.

Time slave dies	**Impact on master**
Before prepare() is called	prepare() will return false, which implies that the caller must rollback the transaction
After prepare() was successful, but before commit/rollback is called	The master ignores the slave, and commits/rolls-back locally. The application should be able to detect the failure, and invokes ResilientStore.recoverDeadPlaces()

Time master dies	**Impact on slave**
Before prepare() is called	The transaction has failed. Slave is unaware of the transaction, and its data is consistent
After prepare() was successful, but before commit/rollback is called	The slave now has a pending transaction, that must be resolved by the resilient store coordinator

IV. Group Transaction

In the program of Fig.3, deciding whether to commit or rollback a transaction is based on the state of the current place only. In some applications, that decision may depend on the states of other places as well. Consider, for instance, a SPMD-style application that performs coordinated checkpointing. To ensure the consistency of a checkpoint, all places need to agree whether to commit or rollback the checkpointing transaction.

To support this model, the ResilientStore and LocalTransaction APIs can be combined by a collective agreement algorithm (i.e. Team.agree()). Each place prepares its local transaction, and participates in the agreement algorithm by its *prepare()* outcome. If all places agree that all their transactions are prepared to commit, they all commit. Otherwise, they all rollback. See **Fig.4** for a code example.

Limitation: Team.agree() is available only on the MPI-ULFM transport of X10.

```
val spare = 3;
val map = ResilientMap.make(spare);
var done:Boolean = false;
while (!done) {
    val activePlaces = map.getActivePlaces();
    val team = new Team(activePlaces);
    try {
        finish foreach(p in activePlaces) {
            val tx = map.startLocalTransaction();
            val x = tx.get("X");
            val y = process(x);
            tx.put("Y", y);
            val allPrepared =  team.agree( tx.prepare() );
            if (allPrepared)
                tx.commit();
            else
                tx.rollback();
        }
        done = true;
    }
    catch(dpe:DeadPlaceException) {
        map.recoverDeadPlaces();
    }
}
```

Fig.4: Group Transaction using Team.agree

V. Recovering Slave Pending Transactions

When a master place dies between calling prepare() and commit()/rollback(), the slave place remains with a pending transaction. If this transaction is a single-place transaction, it can be ignored. However, if it is a group transaction, it must be recovered. The slave place must know the decision that other places have reached before the master died. If the group agreed to commit, the slave must also commit the pending transaction. If they agreed to rollback, the slave must rollback.

In order for the resilient store coordinator to be able to recover pending group transactions, it requires the following:

1. A group transaction must use the same id at all group places.
2. Each master place records the ids of the committed transactions, and rolled-back transactions.
3. When a pending transaction is detected, the coordinator checks all active places to know if there was another place participating in that transaction.
4. If another participant was found, it is asked for the status of the transaction. If it was committed, the coordinator issues a commit() at the slave for that pending transaction. Otherwise, it issues a rollback() request.

The above ensures that the data at the slave place is consistent, before using it to recover the master place.

VI. LULESH Performance

No. of spare places: 3
Problem size:30
No. of iterations:50
checkpoining interval: 10 iterations
kill a random place at iteration:5,25,45

Total number of steps included replayed steps after a rollback: 65
Total number of checkpoints: 5
Total number of recovery invocations: 3

Table VI.1: Resilient Execution with the local data store described above

#Places	Init	Step	Single Checkpoint		Single Recovery						Sum of Recovery items
			Put data	agree	Detect	Recover Resilient Store	App Remake	New Team	Get data	Agree	
216	491	123	96	34	97	122	510	100	4	38	871
343	833	123	105	31	104	122	866	129	4	40	1265
512	1369	129	125	34	124	127	1448	157	7	62	1925
729	2105	134	167	37	171	124	2309	181	9	103	2897
1000	3261	141	211	38	215	127	3572	236	10	168	4328

Table VI.2: Non Resilient Execution

#Places	Init	Step
216	422	112
343	434	113
512	500	118
729	582	120
1000	840	128

Table VI.3: Resilient Execution with the old data store that survives only 1 place (X10-16 paper experiments)

#Places	Init	Step	Single Checkpoint		Single Recovery						Sum of Recovery items
			Put data	agree	Detect	Recover Resilient Store	App Remake	New Team	Get data	Agree	
216	459	121	31	28	101	#	523	115	23	48	810
343	764	125	35	27	96	#	868	73	36	61	1134
512	1298	126	41	28	137	#	1417	110	29	63	1756
729	1963	131	56	32	152	#	2294	174	34	69	2723
1000	2884	137	74	43	284	#	3509	266	32	90	4181