Dandelion Reference Implementation

Dandelion transaction support

src/protocol.h

```
namespace NetMsgType {...
  extern const char *DANDELIONTX;
};

enum GetDataMsg
{...
  MSG_DANDELION_TX = 5,...
  MSG_DANDELION_WITNESS_TX =
   MSG_DANDELION_TX | MSG_WITNESS_FLAG,
};
```

Externally define a character string for the Dandelion transaction network message type.

Create a new GETDATA message enum for Dandelion transactions and Dandelion witness transactions.

src/protocol.cpp

```
namespace NetMsgType {...
  const char *DANDELIONTX="dandeliontx";
};

const static std::string allNetMessageTypes[] = {...
  NetMsgType::DANDELIONTX,
};

std::string CInv::GetCommand() const
{
  switch (masked)
  {...
    case MSG_DANDELION_TX:
     return cmd.append(NetMsgType::DANDELIONTX);...
  }
}
```

Set the Dandelion transaction network message string to "dandeliontx"

Include the Dandelion transaction network message type in the array of all network message types.

Return the Dandelion transaction network message string in the Dandelion transaction message case.

Dandelion log support

src/util.h

```
namespace BCLog {
  enum LogFlags : uint32_t {...
    DANDELION = (1 << 21),...
  };
}
```

Add a Dandelion enum log flag.

src/util.cpp

```
const CLogCategoryDesc LogCategories[] =
{...
  {BCLog::DANDELION, "dandelion"},...
};
```

Add a Dandelion log category.

Dandelion routing, data and helper functions

src/net.h

```
static const int DANDELION_MAX_DESTINATIONS = 2;
class CConnman {
public:...
  bool isDandelionInbound(const CNode* const pnode) const;
  bool isLocalDandelionDestinationSet() const;
  bool setLocalDandelionDestination();...
private:...
  std::vector<CNode*> vDandelionInbound;
  std::vector<CNode*> vDandelionOutbound;
  std::vector<CNode*> vDandelionDestination;
  CNode* localDandelionDestination = nullptr;
  std::map<CNode*, CNode*> mDandelionRoutes;
  CNode* SelectFromDandelionDestinations() const;
  void CloseDandelionConnections(const CNode* const pnode);...
};
```

The maximum number of outbound peers allowed in the vDandelionDestination vector.

Public helper functions are used to update Dandelion routing e.g. in wallet functions.

Track all inbound peers, all outbound peers, and an outbound peer shortlist. Dandelion routes map an inbound peer to a shortlisted outbound peer. The local Dandelion destination is selected from the shortlist.

Helper functions for route management.

src/net.cpp

```
bool CConnman::isDandelionInbound(const CNode* const pnode) const
{
  return (std::find(vDandelionInbound.begin(),
                  vDandelionInbound.end(), pnode) !=
          vDandelionInbound.end());
}
```

A peer is considered to be inbound for the purposes of Dandelion if the pointer to its corresponding CNode object can be found in the vDandelionInbound vector.

```
bool CConnman::isLocalDandelionDestinationSet() const {
  return (localDandelionDestination!=nullptr);
}
```

The Dandelion destination for this node is considered to be set if it is not null.

```
bool CConnman::setLocalDandelionDestination() {
  if (!isLocalDandelionDestinationSet()) {
    localDandelionDestination=SelectFromDandelionDestinations();
  }
  return isLocalDandelionDestinationSet();
}
```

To set the Dandelion destination for this node, select a new Dandelion destination if it has not yet been set.

```
CNode* CConnman::SelectFromDandelionDestinations() const {
  std::map<CNode*,uint64_t> mDandelionDestinationCounts;
  for (size_t i=0; i<vDandelionDestination.size(); i++) {
    mDandelionDestinationCounts.insert(
     std::make_pair(vDandelionDestination.at(i),0)
    );
  }
  for (auto& e : mDandelionDestinationCounts) {
    for (auto const& f : mDandelionRoutes) {
      if (e.first == f.second) { e.second+=1; }
    }
  }
  unsigned int minNumConnections = vDandelionInbound.size();
  for (auto const& e : mDandelionDestinationCounts) {
    if (e.second < minNumConnections) {
      minNumConnections = e.second;
    }
  }
  std::vector<CNode*> candidateDestinations;
  for (auto const& e : mDandelionDestinationCounts) {
    if (e.second == minNumConnections) {
      candidateDestinations.push_back(e.first);
    }
  }
  FastRandomContext rng;
  CNode* dandelionDestination = nullptr;
  if (candidateDestinations.size()>0) {
    dandelionDestination = candidateDestinations.at(
     rng.randrange(candidateDestinations.size())
    );
  }
  return dandelionDestination;
}
```

This private helper function selects a new Dandelion destination.

Start by initializing a map of shortlisted destinations with inbound counts of zero.

Iterate through the Dandelion routes, and count the number of inbound peers routed to each outbound peer.

At least one destination has the smallest number of inbound peers routed to it. In general, there is some set of peers with this smallest number of inbound peers.

Construct a vector of candidate destinations composed of these peers.

Select a peer at random from these candidate destinations.

Return the newly selected Dandelion destination.

Dandelion routing, data and helper functions (continued)

src/net.cpp

```
void CConnman::CloseDandelionConnections(
 const CNode* const pnode
) {
  for (auto iter=vDandelionInbound.begin();
            iter!=vDandelionInbound.end();) {
    if (*iter==pnode) { iter=vDandelionInbound.erase(iter); }
    else { iter++; }
  }
  for (auto iter=vDandelionOutbound.begin();
            iter!=vDandelionOutbound.end();) {
    if (*iter==pnode) { iter=vDandelionOutbound.erase(iter); }
    else { iter++; }
  }
  bool isDandelionDestination = false;
  for (auto iter=vDandelionDestination.begin();
            iter!=vDandelionDestination.end();) {
    if(*iter==pnode) {
      isDandelionDestination = true;
      iter = vDandelionDestination.erase(iter);
    } else { iter++; }
  }
  if (isDandelionDestination) {
    std::vetor<CNode*> candidateReplacements;
    for (auto iteri=vDandelionOutbound.begin();
              iteri!=vDandelionOutbound.end();) {
      bool eligibleCandidate = true;
      for (auto iterj=vDandelionDestination.begin();
                iterj!=vDandelionDestination.end();) {
        if (*iteri==*iterj) {
          eligibleCandidate = false;
          iterj = vDandelionDestination.end();
        } else { iterj++; }
      }
      if (eligibleCandidate) {
        candidateReplacements.push_back(*iteri);
      }
      iteri++;
    }
    FastRandomContext rng;
    CNode* replacementDestination = nullptr;
    if (candidateReplacements.size()>0) {
      replacementDestination = candidateReplacements.at(
       rng.randrange(candidateReplacements.size())
      );
    }
    if (replacementDestination!=nullptr) {
      vDandelionDestination.push_back(replacementDestination);
    }
  }
  CNode* newPto = SelectFromDandelionDestinations();
  for (auto iter=mDandelionRoutes.begin();
            iter!=mDandelionRoutes.end();) {
    if (iter->first==pnode) {iter=mDandelionRoutes.erase(iter);}
    else if (iter->second==pnode) {
      if (newPto==nullptr) { iter=mDandelionRoutes.erase(iter); }
      else { iter->second=newPto; iter++; }
    } else { iter++; }
  }
  if (localDandelionDestination==pnode) {
    localDandelionDestination = newPto;
  }
}
```

When a peer disconnects, call this function to clean up any Dandelion routes with which it may be involved.

Remove the peer from the vDandelionInbound vector if present.

Remove the peer from the vDandelionOutbound vector if present.

Check whether the peer is present in the vDandelionDestination vector.

If the peer was in vDandelionDestination, then replace it.

To do so, first collect a vector of candidate replacements.

Select a replacement node from the candidates.

Add the replacement node to the Dandelion destinations so long as it is not null.

Generate a new Dandelion destination, to be used if needed.

Remove the peer from the Dandelion routing map. Replace it if necessary.

If the peer was the local Dandelion destination, then replace it.

Dandelion routing, route management

```
                              src/net.cpp

void CConnman::AcceptConnection(...) {...
  LogPrint(BCLog::NET, "connection from %s accepted\n",...);
  {
    LOCK(cs_vNodes);
    vNodes.push_back(pnode);
    // Dandelion: new inbound connection
    vDandelionInbound.push_back(pnode);
    CNode* pto = SelectFromDandelionDestinations();
    if (pto!=nullptr) {
      mDandelionRoutes.insert(std::make_pair(pnode, pto));
    }
    LogPrint(BCLog::DANDELION, "New Dandelion inbound:\n%s",...);
  }
}

void CConnman::ThreadSocketHandler() {...
  if (fDelete) {
    // Dandelion: closed connection cleanup
    CloseDandelionConnections(pnode);
    LogPrint(BCLog::DANDELION, "Removed Dandelion:\n%s", ...);
    vNodesDisconnected.remove(pnode);
    DeleteNode(pnode);
  }...
}

void CConnman::OpenNetworkConnection(...) {...
  m_msgproc->InitializeNode(pnode);
  {
    LOCK(cs_vNodes);
    vNodes.push_back(pnode);
    // Dandelion: new outbound connection
    vDandelionOutbound.push_back(pnode);
    if(vDandelionDestination.size()<DANDELION_MAX_DESTINATIONS) {
      vDandelionDestination.push_back(pnode);
    }
    LogPrint(BCLog::DANDELION,"New Dandelion outbound:\n%s",...);
  }
}
```

The CConnman AcceptConnection method has been augmented to add new inbound peers to the vDandelionInbound vector.

The new inbound peer is added to a Dandelion route. The corresponding outbound peer is selected from the vector of Dandelion destinations.

The CConnman OpenNetworkConnection method has been augmented to add a new outbound peer to the vDandelionOutbound vector.

If the vDandelionDestination vector is less than its maximum size, add the new peer to this vector as well.

The CConnman ThreadSocketHandler method has been augmented to close Dandelion connections when a peer is removed.

Periodic shuffle of Dandelion routes

```
                              src/net.h

static const int DANDELION_SHUFFLE_INTERVAL = 600;

class CConnman {
private:...
  void ThreadDandelionShuffle();...
  void DandelionShuffle();...
  std::thread threadDandelionShuffle;...
};
```

The Dandelion routes are shuffled every 600 seconds (10 minutes) on average.

The Dandelion shuffle thread uses the private DandelionShuffle method to periodically change the Dandelion routes.

Periodic shuffle of Dandelion routes (continued)

src/net.cpp

```
void CConnman::ThreadDandelionShuffle() {
  int64_t nNextDandelionShuffle=PoissonNextSend(GetTimeMicros(),
   DANDELION_SHUFFLE_INTERVAL);
  while (!interruptNet) {
    nCurrTime = GetTimeMicros();
    if (nCurrTime > nNextDandelionShuffle) {
      DandelionShuffle();
      nNextDandelionShuffle = PoissonNextSend(nCurrTime,
       DANDELION_SHUFFLE_INTERVAL);
      if (!interruptNet.sleep_for(std::chrono::milliseconds(
       (nNextDandelionShuffle-nCurrTime)/1000))) { return; }
    }
  }
}
```

The Dandelion shuffle thread function periodically calls the Dandelion shuffle function, then sleeps until the next shuffle.

```
bool CConnman::Start(...) {...
  threadDandelionShuffle=std::thread(&TraceThread
  <std::function<void()> >,"dandelion",std::function<void()>(
  std::bind(&CConnman::ThreadDandelionShuffle, this)));...
}
```

The CConnman Start method has been augmented to launch the Dandelion shuffle thread.

```
void CConnman::Stop() {...
  if (threadDandelionShuffle.joinable()) {
    threadDandelionShuffle.join();
  }...
}
```

The CConnman Stop method has been augmented to terminate the Dandelion shuffle thread.

```
void CConnman::DandelionShuffle() {
  LogPrint(BCLog::DANDELION,"Before Dandelion shuffle:\n%s",...);
  { LOCK(cs_vNodes);
    for (auto iter=mDandelionRoutes.begin();
              iter!=mDandelionRoutes.end();) {
      iter = mDandelionRoutes.erase(iter);
    }
    if (localDandelionDestination!=nullptr) {
      localDandelionDestination = nullptr;
    } vDandelionDestination.clear();
    while (
     vDandelionDestination.size()<DANDELION_MAX_DESTINATIONS &&
     vDandelionDestination.size()<vDandelionOutbound.size()) {
      std::vector<CNode*> candidateDestinations;
      for (auto iteri=vDandelionOutbound.begin();
                iteri!=vDandelionOutbound.end();) {
        bool eligibleCandidate = true;
        for (auto iterj=vDandelionDestination.begin();
                  iterj!=vDandelionDestination.end();) {
          if (*iteri==*iterj) {
            eligibleCandidate = false;
            iterj = vDandelionDestination.end();
          } else { iterj++; }
        }
        if (eligibleCandidate) {
          candidateDestinations.push_back(*iteri);
        } iteri++;
      } FastRandomContext rng;
      if (candidateDestinations.size()>0) {
        vDandelionDestination.push_back(candidateDestinations.at(
         rng.randrange(candidateDestinations.size())));
      } else { break; }
    }
    for (auto pnode : vDandelionInbound) {
      CNode* pto = SelectFromDandelionDestinations();
      if (pto != nullptr) {
        mDandelionRoutes.insert(std::make_pair(pnode, pto));
      }
    }
    localDandelionDestination=SelectFromDandelionDestinations();
  }LogPrint(BCLog::DANDELION,"After Dandelion shuffle:\n%s",...);
}
```

The Dandelion shuffle method encapsulates route shuffling.

Iterate through the Dandelion routes and erase, bookkeeping as necessary. Do the same for the local Dandelion destination. Clear the Dandelion destinations.

Repopulate the Dandelion destination vector until it contains the maximum number of destinations or all of the outbound peers.

Generate new routes.

Dandelion transactions, data and helper functions

```
                        src/net.h

class CConnman {
public:...
  CNode* getDandelionDestination(CNode* pfrom);...
  bool localDandelionDestinationPushInventory(const CInv& inv)...
};

class CNode {...
public:...
  std::set<uint256> setDandelionInventoryKnown;...
  std::vector<uint256> vInventoryDandelionTxToSend;...
  void PushInventory(const CInv& inv) {
    LOCK(cs_inventory);
    if (inv.type == MSG_TX) {
      if (!filterInventoryKnown.contains(inv.hash)) {
        setInventoryTxToSend.insert(inv.hash);
      }
    }
    else if(inv.type == MSG_DANDELION_TX) {
      if (setDandelionInventoryKnown.count(inv.hash)==0) {
        vInventoryDandelionTxToSend.push_back(inv.hash);
      }
    } else if (inv.type == MSG_BLOCK) {
      vInventoryBlockToSend.push_back(inv.hash);
    }
  }...
};
```

Method getDandelionDestination returns a peer's Dandelion destination.

Public helper functions push to Dandelion inventories e.g. in wallet functions.

Each peer is represented by an instance of the CNode class. Dandelion transactions that should be known to the peer are tracked. There is also a vector of Dandelion inventory advertisements that need to be sent to this peer.

The PushInventory method has been augmented to queue an inventory advertisement if this peer has not been sent the inventory advertisement previously.

```
                        src/net.cpp

CNode* CConnman::getDandelionDestination(CNode* pfrom) {
  for (auto const& e : mDandelionRoutes) {
    if (pfrom==e.first) {
      return e.second;
    }
  }
  CNode* newPto = SelectFromDandelionDestinations();
  if (newPto!=nullptr) {
    mDandelionRoutes.insert(std::make_pair(pfrom, newPto));
    LogPrint(BCLog::DANDELION,"Added Dandelion route:\n%s",...);
  }
  return newPto;
}

bool CConnman::localDandelionDestinationPushInventory(
 const CInv& inv
) {
  if(isLocalDandelionDestinationSet()) {
    localDandelionDestination->PushInventory(inv);
    return true;
  } else if (setLocalDandelionDestination()) {
    localDandelionDestination->PushInventory(inv);
    return true;
  } else {
    return false;
  }
}
```

If the inbound peer is in the map of Dandelion routes, then return the corresponding destination.

Otherwise, select a destination and record this selection in the map of Dandelion routes.

The local Dandelion destination is a private CNode pointer held by a CConnman instance. Call the PushInventory method for this peer.

Dandelion memory pool

src/validation.h

```
extern CTxMemPool stempool;
```

A memory pool for stem-phase Dandelion transactions.

src/validation.cpp

```
CTxMemPool stempool(&feeEstimator);
```

The Dandelion memory pool is constructed in the same scope as mempool.

Dandelion fluff mechanism

src/net_processing.h

```
static const unsigned int DANDELION_FLUFF = 10;
```

At relay, a Dandelion transaction becomes a transaction with 10% probability.

src/net_processing.cpp

```
static void RelayDandelionTransaction(
 const CTransaction& tx, CConnman* connman, CNode* pfrom)
{
  FastRandomContext rng;
  if (rng.randrange(100)<DANDELION_FLUFF) {
    LogPrint(BCLog::DANDELION, "Dandelion fluff: %s\n", ...);
    CValidationState state;
    CTransactionRef ptx = stempool.get(tx.GetHash());
    bool fMissingInputs = false;
    std::list<CTransactionRef> lRemovedTxn;
    AcceptToMemoryPool(
     mempool, state, ptx, &fMissingInputs, &lRemovedTxn, false, 0
    );
    LogPrint(BCLog::MEMPOOL, "AcceptToMemoryPool...", ...);
    RelayTransaction(tx, connman);
  } else {
    CInv inv(MSG_DANDELION_TX, tx.GetHash());
    CNode* destination = connman->getDandelionDestination(pfrom);
    if (destination!=nullptr) {
      destination->PushInventory(inv);
    }
  }
}
```

If the random number is below the Dandelion fluff threshold, then add the transaction to the main memory pool and call the RelayTransaction function.

Otherwise, send an inventory advertisement to the appropriate Dandelion destination.

Dandelion embargo, data and helper functions

src/net.h

```
static const int DANDELION_EMBARGO_MINIMUM = 10;
static const int DANDELION_EMBARGO_AVG_ADD = 20;

class CConnman {
public:...
  std::map<uint256, int64_t> mDandelionEmbargo;...
  bool insertDandelionEmbargo(
   const uint256& hash, const int64_t& embargo);
  bool isTxDandelionEmbargoed(const uint256& hash) const;
  bool removeDandelionEmbargo(const uint256& hash);...
};
```

Dandelion embargoes last for at least 10 seconds. This minimum value is extended by a random value of 20 in expectation.

The embargo map must be public so that it can be iterated over by a static network processing function when checking for expired embargoes.

src/net.cpp

```
bool CConnman::insertDandelionEmbargo(
 const uint256& hash, const int64_t& embargo
) {
  auto pair = mDandelionEmbargo.insert(
   std::make_pair(hash, embargo)
  );
  return pair.second;
}

bool CConnman::isTxDandelionEmbargoed(const uint256& hash) const{
  auto pair = mDandelionEmbargo.find(hash);
  if (pair != mDandelionEmbargo.end()) { return true; }
  else { return false; }
}

bool CConnman::removeDandelionEmbargo(const uint256& hash) {
  bool removed = false;
  for (auto iter=mDandelionEmbargo.begin();
            iter!=mDandelionEmbargo.end();) {
    if (iter->first==hash) {
      iter = mDandelionEmbargo.erase(iter); removed = true;
    } else { iter++; }
  } return removed;
}
```

The insertDandelionEmbargo method takes an identifying hash and an embargo time and inserts the pair into the embargo map.

The isTxDandelionEmbargoed method returns true or false to indicate whether the transaction hash is in the embargo map.

The removeDandelionEmbargo method takes an identifying hash and removes its pair from the embargo map if present.

src/net_processing.cpp

```
static void CheckDandelionEmbargoes(CConnman* connman) {
  int64_t nCurrTime = GetTimeMicros();
  for (auto iter=connman->mDandelionEmbargo.begin();
            iter!=connman->mDandelionEmbargo.end();) {
    if (mempool.exists(iter->first)) {
      LogPrint(BCLog::DANDELION, "Embargoed tx in mempool", ...);
      iter = connman->mDandelionEmbargo.erase(iter);
    } else if (iter->second < nCurrTime) {
      LogPrint(BCLog::DANDELION, "Embargo expired", ...);
      CValidationState state;
      CTransactionRef ptx = stempool.get(iter->first);
      bool fMissingInputs = false;
      std::list<CTransactionRef> lRemovedTxn;
      AcceptToMemoryPool(mempool, state, ptx, &fMissingInputs,
       &lRemovedTxn, false, 0);
      LogPrint(BCLog::MEMPOOL, "AcceptToMemoryPool...", ...);
      RelayTransaction(*ptx, connman);
      iter = connman->mDandelionEmbargo.erase(iter);
    } else { iter++; }
  }
}
```

Get the current time, and iterate through the map of embargoed Dandelion transactions.

If the transaction exists in the memory pool, then remove it from the embargo list.

Otherwise, if the transaction's embargo expiration time is in the past, then move the transaction to the memory pool and relay it to peers.

Dandelion transaction handling logic

src/net_processing.cpp

```
bool static AlreadyHave(const CInv& inv) ... {
  switch (inv.type) {
    case MSG_TX:
    case MSG_WITNESS_TX:
      ...
    case MSG_BLOCK:
    case MSG_WITNESS_BLOCK:
      return mapBlockIndex.count(inv.hash);
    case MSG_DANDELION_TX:
    case MSG_DANDELION_WITNESS_TX:
      // Do not use AlreadyHave for Dandelion transactions
      // If accidentally used, returns false so tx is requested
      return false;
  }
  // Don't know what it is, just say we already got one
  return true;
}
```

The AlreadyHave function is generally used to check whether the local node already has the transaction associated with a received inventory advertisement. If the INV is not recognized, the message is said to have already been received in order to reduce network traffic.

However, the "already have" behavior for Dandelion transaction changes per peer. The details are tracked in the CNodes of the CConnman. Thus, a static function should not be used in the Dandelion case.

```
void static ProcessGetData(...) {...
  {
    LOCK(cs_main);
    while (... && it->type==MSG_TX ||
     it->type==MSG_WITNESS_TX || it->type==MSG_DANDELION_TX ||
     it->type==MSG_DANDELION_WITNESS_TX)) {...
      bool push = false;
      if(inv.type==MSG_DANDELION_TX ||
         inv.type==MSG_DANDELION_WITNESS_TX) {
        int nSendFlags = (inv.type == MSG_DANDELION_TX ?
         SERIALIZE_TRANSACTION_NO_WITNESS : 0);
        auto txinfo = stempool.info(inv.hash);
        if(txinfo.tx && !connman->isDandelionInbound(pfrom) &&
           pfrom->setDandelionInventoryKnown.count(inv.hash)!=0){
          connman->PushMessage(pfrom, msgMaker.Make(
           nSendFlags, NetMsgType::DANDELIONTX, *txinfo.tx));
          push = true;
        }
      } else { ... } ...
  }...
}
```

The ProcessGetData method governs the response to a peer's GETDATA request. If the GETDATA request is for a Dandelion transaction, then ensure that the peer is not an inbound peer and that the transaction has been advertised to this peer.

```
bool PeerLogicValidation::SendMessages(...) {...
  // Message: inventory
  std::vector<CInv> vInv;
  {
    LOCK(pto->cs_inventory);
    // Add Dandelion transactions
    for (const uint256& hash : pto->vInventoryDandelionTxToSend){
      pto->setDandelionInventoryKnown.insert(hash);
      vInv.push_back(CInv(MSG_DANDELION_TX, hash));
      if (vInv.size() == MAX_INV_SZ) {
        connman->PushMessage(pto, msgMaker.Make(
         NetMsgType::INV, vInv));
        vInv.clear();
      }
    }
    pto->vInventoryDandelionTxToSend.clear();...
  }
  if (!vInv.empty()) {
    connman->PushMessage(pto, msgMaker.Make(
     NetMsgType::INV, vInv));
  }...
}
```

The SendMessages method is used to push messages to peers.

For Dandelion inventory advertisements, the set of known Dandelion transactions for the receiving peer is updated, and the advertisement is pushed into the queue.

Dandelion transaction handling logic (continued)

src/net_processing.cpp

```
bool static ProcessMessage(...) {...
  CheckDandelionEmbargoes(connman);
  ...
  else if (strCommand == NetMsgType::INV) {...
    LOCK(cs_main);...
    for (CInv &inv : vInv) {...
      if (inv.type == MSG_TX || inv.type == MSG_DANDELION_TX) {
        inv.type |= nFetchFlags;
      }
      if (inv.type == MSG_BLOCK) {...}
      else if (inv.type == MSG_DANDELION_TX) {
        auto result =
         pfrom->setDandelionInventoryKnown.insert(inv.hash);
        fAlreadyHave = !result.second;
        if (fBlocksOnly) {...}
        else if (!fAlreadyHave && ... &&
                 connman->isDandelionInbound(pfrom)) {
          pfrom->AskFor(inv);
        }
      } else {...} ...
    }
  }...
  else if (strCommand == NetMsgType::TX) {...
    if (!AlreadyHave(inv) && AcceptToMemoryPool(mempool,...)) {
      if (connman->isTxDandelionEmbargoed(tx.GetHash())) {
        LogPrint(BCLog::DANDELION,"Embargoed tx in mempool",...);
        connman->removeDandelionEmbargo(tx.GetHash());
      }...
    }
  }
  else if (strCommand == NetMsgType::DANDELIONTX) {
    CValidationState state;
    CTransactionRef ptx;
    vRecv >> ptx;
    const CTransaction& tx = *ptx;
    bool fMissingInputs = false;
    std::list<CTransactionRef> lRemovedTxn;
    CInv inv(MSG_DANDELION_TX, tx.GetHash());
    LOCK(cs_main);
    if (connman->isDandelionInbound(pfrom)) {
      if (!stempool.exists(inv.hash)) {
        bool ret = AcceptToMemoryPool(stempool, state, ptx,
         &fMissingInputs, &lRemovedTxn, false, 0);
        if (ret) {
          LogPrint(BCLog::MEMPOOL,"AcceptToStemPool...",...);
          int64_t nEmbargo = 1000000*DANDELION_EMBARGO_MINIMUM+
           PoissonNextSend(
            GetTimeMicros(), DANDELION_EMBARGO_AVG_ADD);
          connman->insertDandelionEmbargo(tx.GetHash(),nEmbargo);
          LogPrint(BCLog::DANDELION, "%s embargoed", ...);
        }
        int nDoS = 0;
        if (state.IsInvalid(nDoS)) {
          LogPrint(BCLog::MEMPOOLREJ, "%s not accepted", ...);
          if(state.GetRejectCode()...){connman->PushMessage(..);}
          if (nDoS > 0) { Misbehaving(pfrom->GetId(), nDoS); }
        }
        if (stempool.exists(inv.hash)) {
          RelayDandelionTransaction(tx, connman, pfrom);
        }
      }
    }
  }
}
```

The ProcessMessage function executes different actions for different messages. Regardless of the message (PING, PONG, etc.), the Dandelion embargoes are checked for expiration.

In the case of an INV message, certain behaviors must occur if it's a Dandelion inventory advertisement.

Specifically, the TX hash is added to the set of known Dandelion transactions for this peer. If the hash is successfully added to the set, then it has not been sent here from that peer before and this node should act like it doesn't already have the TX. This node should only respond to Dandelion INVs from inbound peers.

When a new transaction arrives, check whether it is in the Dandelion embargo.

When a Dandelion transaction arrives, continue only if the message is from an inbound peer.

Try to add it to the stempool; if successful, then add the transaction to the embargo list.

If the Dandelion transaction is in the stempool, then call the RelayDandelionTransaction method.

Dandelion transaction wallet support

```
src/wallet/wallet.cpp

bool CWalletTx::RelayWalletTransaction(CConnman* connman) {...
  LogPrintf("Relaying wtx %s\n", GetHash().ToString());
  if (connman) {
    if (gArgs.GetBoolArg("-dandelion",false)) {
      int64_t nCurrTime = GetTimeMicros();
      int64_t nEmbargo = 1000000*DANDELION_EMBARGO_MINIMUM +
       PoissonNextSend(nCurrTime, DANDELION_EMBARGO_AVG_ADD);
      connman->insertDandelionEmbargo(GetHash(),nEmbargo);
      LogPrint(BCLog::DANDELION,
       "Embargoed for %d secs\n",(nEmbargo-nCurrTime)/1000000);
      CInv inv(MSG_DANDELION_TX, GetHash());
      return
       connman->localDandelionDestinationPushInventory(inv);
    } else {
      CInv inv(MSG_TX, GetHash());
      connman->ForEachNode([&inv](CNode* pnode)
      {
        pnode->PushInventory(inv);
      });
      return true;
    }
  }...
}

bool CWalletTx::AcceptToMemoryPool(...) {
  bool ret;
  if (gArgs.GetBoolArg("-dandelion",false)) {
    ret = ::AcceptToMemoryPool(stempool, state, tx,
     nullptr, nullptr, false, nAbsurdFee);
  } else {
    ret = ::AcceptToMemoryPool(mempool, state, tx,
     nullptr, nullptr, false, nAbsurdFee);
  }
  fInMempool |= ret;
  return ret;
}
```

If the Dandelion feature is turned on, then create a Dandelion transaction and relay it to the local Dandelion destination.

Otherwise, create a transaction and advertise it to all peers.

The wallet AcceptToMemoryPool method has been augmented to add transactions to the appropriate memory pool.